

OCC: An Automated End-to-End Machine Learning Optimizing Compiler for Computing-In-Memory

Adam Siemieniuk¹, Lorenzo Chelini², Asif Ali Khan³, Jeronimo Castrillon⁴, *Senior Member, IEEE*,
Andi Drebes, Henk Corporaal⁵, Tobias Grosser⁶, *Member, IEEE*, and Martin Kong

Abstract—Memristive devices promise an alternative approach toward non-Von Neumann architectures, where specific computational tasks are performed within the memory devices. In the machine learning (ML) domain, crossbar arrays of resistive devices have shown great promise for ML inference, as they allow for hardware acceleration of matrix multiplications. But, to enable widespread adoption of these novel architectures, it is critical to have an automatic compilation flow as opposed to relying on a manual mapping of specific kernels on the crossbar arrays. We demonstrate the programmability of memristor-based accelerators using the new compiler design principle of multilevel rewriting, where a hierarchy of abstractions lowers programs level-by-level and perform code transformations at the most suitable abstraction. In particular, we develop a prototype compiler, which progressively lowers a mathematical notation for tensor operations arising in ML workloads, to fixed-function memristor-based hardware blocks.

Index Terms—Computing-in-memory (CIM), machine learning (ML), memristor, MLIR.

I. INTRODUCTION

CMOS-BASED technology benefited from Dennard's scaling for several decades before hitting the energy wall [1]. This trend led to a growing interest in domain-specific architectures, especially in the machine learning (ML) domain, which has witnessed tremendous development in recent years. Examples are tensor cores in graphics processing units (GPUs) and specialized CMOS accelerators, such as Google's TPU [2], Microsoft Brainwave [3], and Intel's Nervana Neural Network Processor [4]. But, the ever-increasing complexity in

applications and data processing poses a quest for the next leap forward in energy efficiency that is anticipated in architectures, which depart from the traditional Von-Neumann model.

Computing-in-memory (CIM) is a promising approach to deliver the next leap in energy efficiency as computation and storage are performed *directly* in memory without incurring frequent long-distance, off-chip data movements [5]. Memristor crossbars, in particular, have attracted significant interest due to their ability to efficiently perform matrix–matrix and matrix–vector multiplications—the dominant computational kernels in deep neural networks [6]. Each memristive device stores a multi-bit value as its conductance state, enabling high-storage density. An application of a voltage at the crossbar input results in an output current at the crossbar columns, which is proportional to the conductance state of each memristive devices following Kirchhoff's law [7]. Organizing memristor in a crossbar-like structure enables execution of parallel matrix–vector multiplications in constant time complexity avoiding data transfers and bottlenecks of current multicores, GPUs, and specialized accelerators.

A key factor in the widespread adoption of these novel architectures will be the software ecosystem [7], [8]. In particular, compilers should be able to efficiently map applications to the crossbars without user intervention. While a significant body of research proposes novel architectures, only a few of them deal with programmability aspects. As a consequence, efficient exploitation of CIM acceleration still relies on the programmer and her understanding of the hardware, thus severely limiting programmability [6], [9], [10]. This is opposed to the recent trend to boost productivity by increasing the level of abstraction in ML frameworks [11]–[14]. In line with this, our work improves programmability by introducing a *fully automatic, end-to-end* compilation flow for CIM accelerators in general and memristor-based architectures in particular. Our compilation flow relies on the novel compiler paradigm of multilevel intermediate representation (IR) rewriting, which is a combination of: 1) IR based on the static single assignment form (SSA); 2) operations with high-level semantic; and 3) progressive lowering, which allows gradual lowering of high-level abstraction to low-level IR across the abstraction levels. The SSA form allows to reuse existing optimizations developed for general-purpose compilers. High-level operations enable encoding instructions specific to our accelerator, while progressive lowering allows to preserve domain-specific information and express high-level transformations as peephole optimizations.

Manuscript received December 15, 2020; revised March 23, 2021 and June 14, 2021; accepted July 7, 2021. Date of publication August 2, 2021; date of current version May 20, 2022. This work was supported in part by the European Commission Horizon2020 Programme through the NeMeCo under Grant 676240; in part by MNEMOSENE under Grant 780215; in part by the German Research Council through the Co4RTM under Grant 450944241; and in part by the Cluster of Excellence “Center for Advancing Electronics Dresden.” This article was recommended by Associate Editor G. Tagliavini. (Corresponding author: Adam Siemieniuk.)

Adam Siemieniuk, Lorenzo Chelini, and Henk Corporaal are with the Department of Electrical Engineering, Technische Universiteit Eindhoven, 5612 AZ Eindhoven, The Netherlands (e-mail: a.i.siemieniuk@student.tue.nl).

Asif Ali Khan and Jeronimo Castrillon are with the Department of Computer Science, Dresden University of Technology, 01069 Dresden, Germany.

Andi Drebes is with the Inria, École Normale Supérieure, 75012 Paris, France.

Tobias Grosser is with the School of Informatics, University of Edinburgh, Edinburgh EH8 9YL, U.K.

Martin Kong is with the School of Computer Science, University of Oklahoma, Norman, OK 73019 USA.

Digital Object Identifier 10.1109/TCAD.2021.3101464

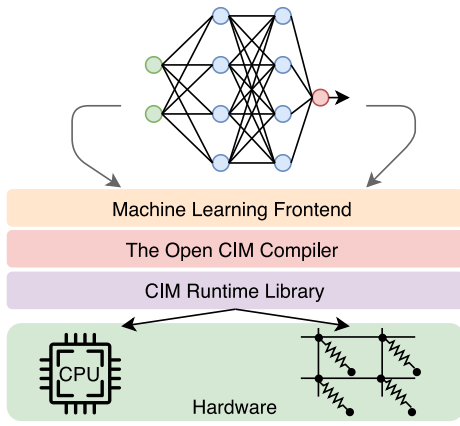


Fig. 1. OCC enables mapping tensor operations described in a productive-oriented language to fixed-function memristor-based hardware blocks exposed by the CIM runtime library.

The open CIM compiler (OCC)¹ in Fig. 1 together with our ML frontend and CIM runtime library is, to the best of our knowledge, the first end-to-end compilation flow for in-memory computing that leverages multilevel IR rewriting for reliable mapping of computational primitives to crossbar arrays. OCC leverages the recently introduced MLIR compiler infrastructure, which allows compiler developers to match the right abstraction levels for the problem at hand by adding custom abstraction or dialects—namespace of custom types, operations, and attributes [15].

We make the following contributions.

- 1) The OCC, a fully automatic end-to-end compilation framework based on multilevel IR rewriting, which allows reliable mapping of computational motifs to the crossbar in a transparent way, without any user intervention (Section IV).
- 2) An OCC frontend for expressing tensor computations arising in the ML domain based on tensor comprehensions (TCs) (Section IV-A).
- 3) A set of high-level hardware-agnostic passes to rewrite computational motifs using the matrix–matrix product as basic block primitive, thus enabling efficient execution on the crossbar (Section IV-B1).
- 4) A set of hardware-specific compiler passes to ensure the computation fit the CIM crossbar and reduce the number of write operations, hence increasing the crossbar’s lifetime (Section IV-B2).
- 5) A runtime library for transparent data allocation and data transfer management to and from the CIM accelerator (Section IV-B3).
- 6) An experimental evaluation with a full system comprising a CIM accelerator as a co-processor based on Gem5, demonstrating that tensor operations frequently occurring in ML kernels can be reliably identified and mapped on the crossbar (Section V).

II. MLIR INFRASTRUCTURE

MLIR is a new compiler infrastructure under the LLVM umbrella [15]. It aims at simplifying the compilation for

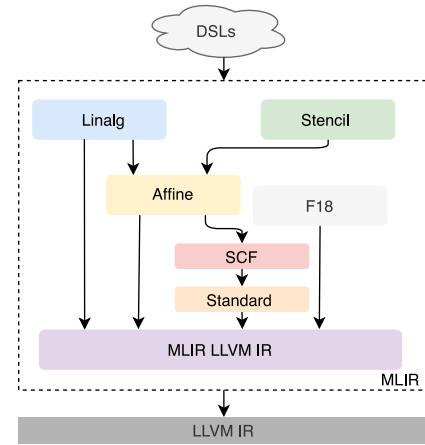


Fig. 2. Some of the available dialects in MLIR. The higher a dialect, the higher is its level of abstraction.

heterogeneous hardware and reducing the burden of developing domain-specific compilers. For this, MLIR provides a nonopinionated IR, i.e., little builtins leaving most of the IR customizable. Having a customizable IR allows compiler developers to match the right abstraction level for their problem at hand by introducing custom types, operations, and attributes. Operations are the essential atomic constituents of the IR; each operation uses and produces new values. A value represents data at runtime, and it is associated with a type known at compile time, whereas types model compile-time information about values. Attributes, on the other hand, allow attaching compile-time information to operations. Custom types, operations, and attributes are logically grouped into dialects. A dialect is a basic ingredient that enables the MLIR infrastructure to implement a stack of reusable abstractions. Each abstraction encodes and preserves transformation validity preconditions directly in its IR, reducing the complexity and the cost of analysis passes.

Fig. 2 shows a subset of the available dialects in MLIR. The Linalg dialect models linear-algebra operation on tensor or buffer operands. The Stencil dialect represents iterative kernels that update an array element according to a given stencil pattern [16]. At a lower level, the affine dialect models a simplified polyhedral representation while F18 does it for Fortran-specific constructs. SCF and standard represent control flow and a collection of miscellaneous operations, respectively. MLIR LLVM IR models LLVM-IR constructs.

Progressive lowering (black arrows in Fig. 2) enables converting from higher level domain-specific dialects to lower level platform-specific ones. We integrate OCC within the MLIR infrastructure, carefully considering and exploiting the existing abstraction and introducing new ones.

III. PCM MEMRISTOR BASIC

Multiple devices, such as resistive random access memory (RRAM) or phase change memory (PCM), can be used as memristors, practically a passive device with two-port components with a variable resistance state. We focus on PCM devices in this work, as they are a strong candidate for

¹<https://github.com/adam-smnk/Open-CIM-Compiler>

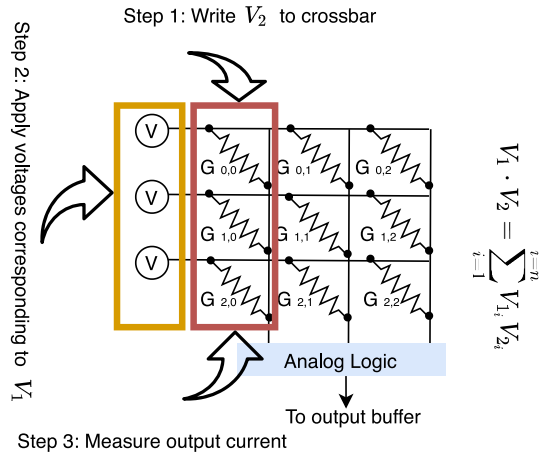


Fig. 3. Mapping a dot product on the CIM crossbar.

future non-CMOS and beyond von-Neumann computing solutions [17]. PCM devices store information by changing the cell resistance, switching between amorphous and crystalline states. The transition between the two states happens due to the application of external voltages that exceed the threshold voltage of a device. The phase change material is sandwiched between two electrodes, and its state can be changed by applying a current burst through one of the electrodes. A short, but intense, pulse—known as reset pulse—is used to bring the material back to the amorphous phase (high resistance). Contrarily, to switch to low resistance, the set pulse—a lower and longer pulse—is applied. An even lower pulse than the set one is used to read the device’s state. Assembling memristive devices into crossbar tiles allows for the in-place computation of fixed-size tensor operations in constant time. For example, the dot product of two fixed-size vectors v_1 and v_2 can be accomplished by applying voltages corresponding to the values of v_1 to a column of memristive devices whose conductance corresponds to the values v_2 and by measuring the resulting current for the entire column (Fig. 3). This can be extended to fixed-size matrix–vector in constant time by adding one column of memristive devices for each row of the input matrix and measuring each column’s current.

IV. OCC LOWERING PIPELINE

In our compilation flow (Fig. 4), the entry point is a collection of computational motifs defined in a productive-oriented language for tensor operations, TCs. We use Teckyl [18] to enter the OCC lowering pipeline at the Linalg abstraction. Our familiarity with Teckyl drives the choice of using TC; however, this does not preclude OCC usage with other frontends. OCC makes use of the Linalg and the CIM dialect; thus, as long as a frontend can raise/lower to such dialects, OCC can apply his transformations. At the Linalg level, OCC performs a set of hardware-agnostic passes to rewrite computational motifs for a CIM-friendly execution. All the passes serve to rewrite each motif (i.e., contraction) using the matrix–matrix multiplication as a basic building block to execute them efficiently on the crossbar. The Linalg dialect is then lowered to CIM. The CIM dialect acts as an interface to our accelerator. It performs

hardware-specific optimizations to ensure the computation fits in the crossbar array and to reduce the number of write operations to increase the crossbar lifetime. Besides, it orchestrates the data movement to and from the device. During lowering from CIM to SCF and then to Standard, the operations amenable for CIM execution are mapped to function calls to our accelerator runtime library. The remaining operations that are not being offloaded to CIM follow the route toward the CPU code generation path.

A. Teckyl: Frontend for Tensor Operations

Teckyl is our entry point in the lowering pipeline [18]. It allows expressing tensor computations arising in ML using TCs syntax [14]. Contrary to conventional ML notation, TCs are not limited to predefined operators but allows users to specify custom operations on tensors using index expressions. Its syntax derives from the ubiquitous Einstein notation, where universal quantifiers are introduced explicitly. The listing below demonstrates how a matrix–vector multiplication is expressed in TC syntax, where a matrix of $M \times K$ size is multiplied by a K -size vector, resulting in a vector c .

```
def mv(float(M, K) A, float(K) x) -> (c) {
    c(i) +=! A(i, k) * x(k)
}
```

The function signature provides the shape and type of the inputs, while the index variables define the shape of the output tensor. The index variable i iterates over the first dimension of matrix A ; thus, its range is $[0, M-1]$. Since the variable i indexes also the c vector, the size of c will be M . Similarly, the range for the k variable can be derived. “!” after the addition assignment operator denotes a default initialization, meaning that the c vector will be default initialized with zeros of the appropriate type.

B. OCC Transformations

We distinguish between three types of transformations that work in symbiosis across our entire lowering pipeline: hardware-agnostic rewriting passes ① in Fig. 4, hardware-specific passes to adapt the computational motif to the hardware features ②, and the actual lowering to CIM library ③.

1) *Hardware-Agnostic Rewriting Passes*: All hardware-agnostic passes work at the Linalg level and aim at rewriting motifs using the matrix–matrix multiplication as building block. OCC supports two rewriting passes: 1) transpose transpose GEMM transpose (TTGT) [19] for contractions and 2) Im2Col [20] for convolutions.

The *TTGT* pass automatically detects and applies the TTGT transformation scheme on each detected contraction. Algorithm 1 shows how contractions are detected at the Linalg level. The operation must contain exactly three operands: two inputs A and B and one output C . The operation body must contain a multiply accumulate computation, which is verified by `HasMultiplyAddBody` callback by analyzing the innermost loop’s operations. Next, the dimensions of the operands are evaluated. For the operation to represent a contraction,

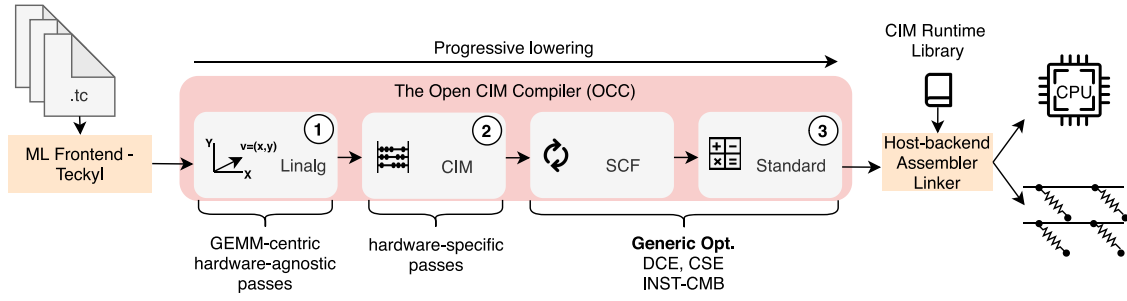


Fig. 4. Different building blocks of the OCC.

```

def contr(int16(K, L, M) A, int16(L, K, N) B)
  -> (int16(M, N) C)
{
  C(m, n) += A(k, l, m) * B(l, k, n)
}

    ↓ lowers to
%0 = linalg.transpose(%A, {2, 0, 1})
%1 = linalg.transpose(%B, {1, 0, 2})
%2 = linalg.reshape(%0, {0, {1, 2}})
%3 = linalg.reshape(%1, {{0, 1}, 2})
// eligible for offloading to CIM
linalg.matmul(%2, %3, %C)

```

```

def conv2d(int16(B, IP, H, W) Img,
  int16(OP, IP, KH, KW) Filt)
  -> (int16(B, OP, H, W) Out)
{
  Out(b, op, h, w) +=! Img(b, ip, h + kh, w + kw)
    * Filt(op, ip, kh, kw)
}

    ↓ lowers to
%0 = linalg.im2Col(%Img)
%1 = linalg.im2Col(%Filt)
// eligible for offloading to CIM
linalg.matmul(%0, %1, %OutTmp)
linalg.Col2Im(%OutTmp, %Out)

```

Listing 1. TTGT rewriting rule on the left reduces each contraction to a sequence of transpose, reshape, and GEMM operations. Complementary, the Im2Col rule (right) transforms each detected 2-D contraction in a GEMM operation. GEMM operations will be executed efficiently on the CIM crossbar.

Algorithm 1 Contraction Detection

```

1: procedure ISCONTRACTION(op)
2:   numInputs ← op.getNumInputs()
3:   numOutputs ← op.getNumOutputs()
4:
5:   if (numInputs ≠ 2) ∨ (numOutputs ≠ 1) then
6:     return false
7:   end if
8:   if ¬HasMultiplyAddBody(op) then
9:     return false
10:  end if
11:
12:  (A, B, C) ← op.getOperands()
13:  dimsA ← {A.getDims()}           ▷ Set of dimensions
14:  dimsB ← {B.getDims()}
15:  dimsC ← {C.getDims()}
16:
17:  reductionDims ← dimsA ∩ dimsB
18:  outDimsA ← dimsA \ reductionDims
19:  outDimsB ← dimsB \ reductionDims
20:  outputDims ← outDimsA ∪ outDimsB
21:
22:  return
    (|reductionDims| > 0) ∧
    (dimsC = outputDims)
23: end procedure

```

the following criteria have to be fulfilled: 1) the dimensions common to the both inputs A and B are the reduction dimensions and shall not be present in the output C; 2) the

input dimensions other than the reduction dimensions shall be present in the output C; and 3) the output C shall not contain any dimensions that are not present in the inputs A and B. If all the above conditions are met, the pass applies the TTGT transformation.

TTGT stands for TTGT and enables rewriting a contraction as a composition of transpose, reshape, and GEMM operations. More specifically, the main idea is first to flatten the tensors into matrices via direct tensor transposition and reshape operations, then execute a single GEMM operation, and finally, fold back the resulting matrix into the original tensor layout. Only the GEMM operation will be offloaded to the CIM accelerator, while the rest of the operations will follow the CPU code generation path. More concretely, consider how the TTGT pass rewrites the contraction in Listing 1 (left) as a sequence of Linalg operations. Two `linalg.transpose` operations are emitted to rearrange the dimensions for tensor A and B [i.e., $(k, l, m) \rightarrow (m, k, l)$ for A]. The transposes' outputs are then fed to `linalg.reshape` operations that collapse the second and third dimensions for A and the first and second dimensions for B, reshaping the tensors to matrices. Finally, a `linalg.matmul` is emitted. For our running example, no additional reshape and transpose operations are needed for the output tensor C.

Complementarily, the *Im2Col* pass automatically detects and applies the Im2Col transformation to every detected convolution. Algorithm 2 shows how convolutions are detected. Currently, only spatial convolutions using N_C data format [21] are supported. Specifically, for an operation to represent a convolution, the access pattern for the input A must represent a sliding window. For example, suppose A contains an image of

Algorithm 2 Convolution Detection (N_C Data Format)

```

1: procedure ISCONVOLUTION(op)
2:   numInputs  $\leftarrow$  op.getNumInputs()
3:   numOutputs  $\leftarrow$  op.getNumOutputs()
4:
5:   if (numInputs  $\neq$  2)  $\vee$  (numOutputs  $\neq$  1) then
6:     return false
7:   end if
8:   if  $\neg$ HasMultiplyAddBody(op) then
9:     return false
10:  end if
11:
12:  (A, B, C)  $\leftarrow$  op.getOperands()
13:  indexesA  $\leftarrow$  A.getIndexes()
14:  indexesB  $\leftarrow$  B.getIndexes()
15:  indexesC  $\leftarrow$  C.getIndexes()
16:
17:  isN  $\leftarrow$  indexesC[0] = indexesA[0]
18:  isK  $\leftarrow$  indexesC[1] = indexesB[0]
19:  isC  $\leftarrow$  indexesA[1] = indexesB[1]
20:  if  $\neg$ (isN  $\wedge$  isK  $\wedge$  isC) then
21:    return false
22:  end if
23:
24:  for i  $\leftarrow$  2 to indexesA.size() do
25:    if indexesA[i]  $\neq$  (indexesB[i] + indexesC[i]) then
26:      return false
27:    end if
28:  end for
29:
30:  return true
31: end procedure

```

height H and width W and the input B is a kernel with sizes K_H and K_W ; in that case, the mapping for the height of the image shall take the form of $h + k_h$ where h and k_h are the iterators of the H and K_H dimensions, respectively.

The Im2Col transformation is based on the idea that convolution is no more than a dot product between the kernel filters and the moving window's local regions. If we take each window and stack them in a columnwise order and we do the same for the filters, but in a rowwise order, we obtain two matrices, and the output of multiplication between them will be equivalent to the original convolution's output. Similar to the TTGT transformation, only the GEMM computation will be offloaded to the CIM device, while the remaining operations will follow the CPU code-generation path. Listing 1 (right) shows how a 2D-convolution is rewritten at the Linalg level by applying the Im2Col transformation.

2) *Hardware-Specific Passes*: The CIM dialect focuses on interfacing high-level Linalg GEMM operations with the underlying accelerator hardware. Table I shows a subset of operations exposed by the CIM dialect. The optimizations performed at this level focus primarily on data layouts and computation reorder. The former maximizes hardware

```

// GEMM in the Linalg dialect
linalg.matmul(%A, %B, %C)
      ↓ lowers to
// tiled GEMM in the CIM dialect
%c0 = constant 0 : i32
%c1 = constant 1 : i32
%id = constant 0 : i32 // tile id
scf.for %i = %c0 to %tiledRows step %c1 {
  scf.for %j = %c0 to %tiledCols step %c1 {
    %tileC = cim.copyTile(%C, %i, %j)
    %tempTile = cim.allocDuplicate(%tileC)
    scf.for %k = %c0 to %numTiles step %c1 {
      %tileA = cim.copyTile(%A, %i, %k)
      %tileB = cim.copyTile(%B, %k, %j)
      cim.write(%id, %tileB)
      cim.matmul(%id, %tileA, %tempTile)
      cim.barrier(%id)
      // tileC += tempTile
      cim.accumulate(%tileC, %tempTile)
    }
    cim.storeTile(%tileC, %C, %i, %j)
  }
}

```

Listing 2. Lowering from a Linalg GEMM to a tiled version that fits into a CIM accelerator crossbar. Bold operations are executed on the CIM accelerator.

```

// original tiled GEMM
scf.for %i = %c0 to %tiledRows step %c1 {
  scf.for %j = %c0 to %tiledCols step %c1 {
    %tileC = cim.copyTile(%C, %i, %j)
    scf.for %k = %c0 to %numTiles step %c1 {
      ...
      %tileB = cim.copyTile(%B, %k, %j)
      cim.write(%id, %tileB)
      ...
    }
    cim.storeTile(%tileC, %C, %i, %j)
  }
}

      ↓ transforms to
// loop interchanged GEMM
scf.for %k = %c0 to %numTiles step %c1 {
  scf.for %j = %c0 to %tiledCols step %c1 {
    %tileB = cim.copyTile(%B, %k, %j)
    cim.write(%id, %tileB)
    scf.for %i = %c0 to %tiledRows step %c1 {
      %tileC = cim.copyTile(%C, %i, %j)
      ...
      cim.storeTile(%tileC, %C, %i, %j)
    }
  }
}

```

Listing 3. Loop interchange to reduce the number of writes to the CIM crossbar during a tiled GEMM computation.

utilization and transfer bandwidth, and enables the computation on the crossbar. The latter aims to extend the crossbar lifetime.

The *tiling pass*, the output of which is shown in Listing 2, allows performing matrix multiplication on data that exceeds a CIM device's capacity. The data are split into multiple smaller GEMM computations. The input buffers are divided into tiled rows (*tiledRows*) and columns (*tiledCols*) equal to $\lceil M/\text{tileSize} \rceil$ and $\lceil N/\text{tileSize} \rceil$ where the *tileSize*

TABLE I
CIM DIALECT OPERATIONS

Operation	Datatype	Target	Description
<code>cim.write(%id, %matB)</code>	integer, memref	CIM	Transfer and write the 2D buffer <code>%matB</code> to the crossbar of the CIM tile <code>%id</code> , synchronously (or blocking).
<code>cim.matmul(%id, %matA, %matC)</code>	integer, memref, memref	CIM	Transfer the 2D buffer <code>%matA</code> to the CIM tile <code>%id</code> , perform GEMM and store the results in the 2D buffer <code>%matC</code> , asynchronously (or non-blocking).
<code>cim.barrier(%id)</code>	integer	HOST	Wait for work completion on the CIM tile <code>%id</code> .
<code>%tile = cim.copyTile(%mat, %row, %col)</code>	memref, memref, index, index	HOST	Allocate a contiguous 2D buffer <code>%tile</code> and copy a tile at the position <code>(%row, %col)</code> from the 2D buffer <code>%mat</code> .
<code>cim.storeTile(%tile, %mat, %row, %col)</code>	memref, memref, index, index	HOST	Copy the tile <code>%tile</code> to the 2D buffer <code>%mat</code> at the position <code>(%row, %col)</code> .
<code>cim.accumulate(%lhs, %rhs)</code>	memref, memref	HOST	Add corresponding elements of the <code>%rhs</code> to the <code>%lhs</code> .
<code>%output = cim.allocDuplicate(%input)</code>	memref, memref	HOST	Allocate an empty buffer <code>%output</code> which has the same dimensions as the input buffer <code>%input</code> .

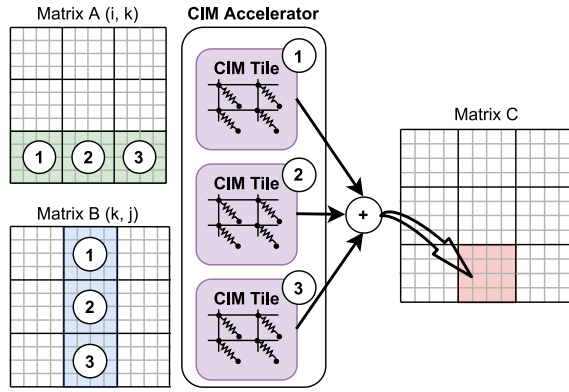


Fig. 5. Loop unrolling used to parallelize the GEMM computation across multiple CIM tiles.

depends on the size of a crossbar tile. Along the inner dimension K , the number of tiles (`numTiles`) is defined as $\lceil K/\text{tileSize} \rceil$. At the buffer boundaries, the tile sizes are trimmed to stay within the matrix dimensions. Because the host and the accelerator communicate through DMA, the transferred data must be contiguous in the memory. This is ensured by `cim.copyTile` operation that copies elements from an input matrix that resides on a specific tile to a temporary buffer. Then, a partial result is computed and placed into a temporary, tile-sized output buffer allocated using `cim.allocDuplicate`. These buffers are then accumulated on the host by performing elementwise addition, represented by `cim.accumulate`. Finally, the tile is written back to the provided output matrix C by a `cim.storeTile` operation.

Listing 3 shows the application of the op unrolling used to parallelize the GEMM computation across multiple CIM tiles. *Loop interchange pass*, which reduces the number of crossbars writes performed during a tiled GEMM computation. In the original tiled GEMM computation, the `cim.copyTile` and the `cim.write` operations in the innermost k -loop depend only on two out of the three iterators: k and j . By interchanging the innermost loop k with the outermost one i , the `cim.copyTile` and the `cim.write` operations can be moved one level higher. Thus, reducing the number of crossbars writes by a factor of `tiledRows`.

Fig. 5 shows the *loop unrolling pass*, which parallelizes computation across multiple CIM tiles by unrolling the inner dimension loop k of the tiled GEMM. The crossbars are first populated with data and then all the computations are performed in parallel. Once the first partial result is available, it is accumulated before waiting for the next CIM tile, which helps improving overall latency.

3) *Lowering to CIM Library Calls*: Similar to existing works, our CIM accelerator exposes an application program interface (API). Thus, the last step in our compilation flow is to lower high-level CIM operations to function calls exposed by the CIM runtime library. Each CIM-dialect operation (Table I) that is accelerator specific (`cim.write`, `cim.matmul`, and `cim.barrier`) has a one-to-one mapping with a function call exposed by the CIM runtime library. The rest of the operations get lowered to the other existing MLIR dialects.

V. EVALUATION

In this section, we demonstrate the efficacy of OCC in transparently detecting and offloading matrix–matrix and matrix–vector multiplications. We first explain our experimental setup and the set of benchmarks used, and then show the impact of OCC offloading and transformations on both performance and energy consumption.

A. Experimental Setup

Fig. 6 shows the overview of our emulated SoC in Gem5. We use the full-system simulation environment to simulate a bare-metal machine. The SoC consists of a single high-performance in-order (HPI) ARM core (a representative model of the modern ARMv8-A implementation, see Table II) with on-chip instruction and data caches (32 and 64 kB, respectively) and a unified L2-cache of size 2 MB. The core operates at a clock frequency of 2 GHz, uses a main memory of size 4 GB (single channel DRAM DDR3_1600_8×8), and is connected to the CIM accelerator via the system bus. For the memory system, we use the classic memory model in Gem5. The CIM accelerator acts as a co-processor, is memory-mapped, and accesses the shared main-memory using DMA operations.

The accelerator’s brain is the control unit responsible for orchestrating and steering the internal circuitry. The execution

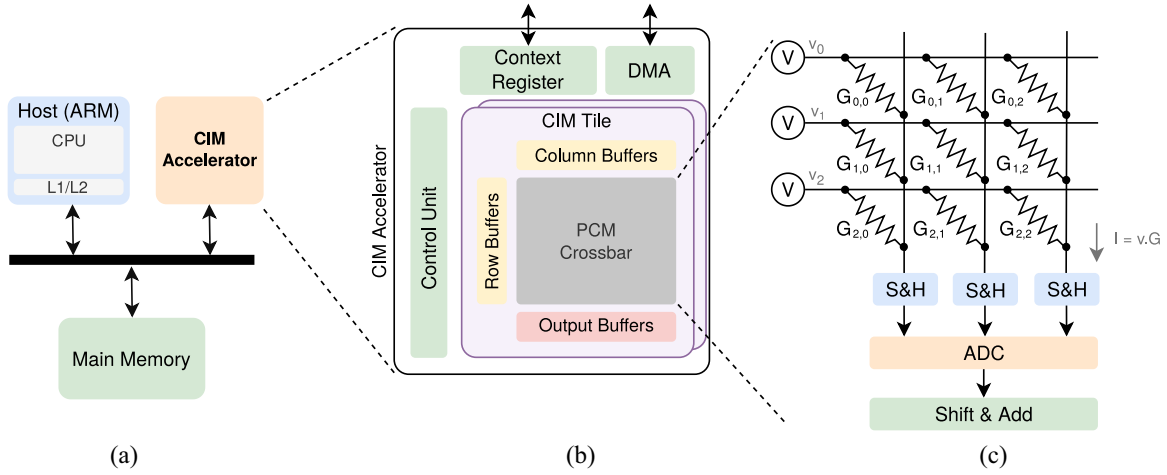


Fig. 6. Overview of the emulated SoC. (a) Overview of the architecture. (b) Architecture of the CIM accelerator. (c) Memristive crossbar array.

pipeline is represented by an array of CIM tiles. Within a CIM-tile, the memristor crossbar executes the analog vector–matrix multiplication. We model a 4-tile PCM crossbar with a tile size of 64×64 and 8-bit precision. To accomplish 8-bit precision, we rely on the bit-slicing technique, which allows increasing accuracy by combining modules of smaller bit width [7]. To be precise, we implement bit slicing by distributing the computation over multiple columns, where each column represents a single bit slice. The bits are then weighted at the column output using a shift and add block. The physical properties of each PCM device, such as write and read latency, are taken from [9] and [22], and reported in Table II. Additional CMOS peripheral logic is required to interface the PCM crossbar with the rest of the CIM-tile. Sample-and-Holds and ADC converters serve such purpose. Finally, row, columns input buffers, and output buffer in each CIM-tile follow the purpose of storing temporary data that will be read (output) or written (row and column buffer). ADC converters introduce quantization loss, which reduces inference accuracy [23]. This aspect is not addressed in this article.

Typical Offloading Scenario: In a typical offloading scenario, the host prepares the data on shared memory and triggers the accelerator execution by writing to special memory-mapped registers. The CIM accelerator then reads the data in the shared memory via DMA transactions (`cim.write` or `read`). Once done, the accelerator writes back the results in the shared memory. The host monitors the status of CIM execution by polling the status register (`cim.barrier`), and upon completion, it can safely resume execution.

We evaluate the following configurations.

- 1) *Arm*: Benchmarks compiled with the LLVM static compiler after MLIR code generation (LLVM git commit fc2199d), with no-parallelization enabled. The kernels are executed on the host (ARM) processor with no-call to the CIM accelerator. This provides a baseline for comparison.
- 2) *Tile*: Program generated by the OCC where the compute kernel is tiled and offloaded to the accelerator. Tiling is required to fit the kernel on the crossbar if the kernel's size exceeds the crossbar's size.

TABLE II
SoC CONFIGURATION

CIM Parameter	Value
Technology(64×64 @8-bit)	IBM PCM $4 \times (64 \times 64$ @8-bit)
Compute and write latency/8-bit	1 μ s and 2.5 μ s
Compute/Read energy / 8-bit	200 fJ (2×100 fJ / 8-bit PCM)
Write energy / 8-bit	200 pJ (2×100 pJ / 8-bit PCM)
Cell endurance	3.2×10^7
Mixed signal circuit energy	3.9 nJ (@1.2 GHz)
Input/Output buffer energy (1.5 kB)	5.4 pJ/byte-access
Digital logic energy	40 pJ / GEMV for weighted sum and 2.11 pJ / extra ALU operation
DMA and control unit energy	<0.78 nJ
Host CPU Spec	Value
ARM-A53, 28 nm	2 GHz
L1-I, L1-D	32 kB, 64 kB
L2, Main-memory	2 MB, 4 GB (DDR3_1600_8x8)

- 3) *Tile+interchange*: Program generated by the OCC where the compute kernel is tiled and the tiled loops are interchanged to reduce write operations to the crossbar. The cim-optimized code is offloaded to the accelerator.
- 4) *Tile+parallel*: Program generated by the OCC where the compute kernel is tiled, parallelized, and offloaded across multiple tiles by unrolling the inner loop dimension.
- 5) *Tile+interchange+parallel*: All optimizations are enabled.

The selection of the ARM core as a baseline is justified by the fact that we focus on developing a new compiler infrastructure for CIM. The evaluation thereof stresses demonstrating the effectiveness of OCC and not the CIM device itself. Comparison of the CIM computational paradigm to state-of-the-art multi- and many-cores machines as well as hardware accelerators has already shown in previous works [7], [9], [24].

As for workloads, we use kernels from the ML domain and kernels from previous studies on tensor contractions [14], [19]. For the ML domain, we include applications ranging from simple matrix multiplication kernels (e.g., mm) to a full WaveNet cell [19]. While for the contractions, we included tensors with different dimensionality used in coupled-cluster methods [26] and chemistry calculations [27]. All the workloads

are expressed in 8-bit integers to match the precision of the CIM crossbar.

Matrix multiplication in the context of deep learning is ubiquitous. We consider different flavors of matrix multiplications: `mm` single matrix multiplication, `2 mm` two consecutive independent matrix multiplications, and `3 mm` two matrix-matrix multiplications and the multiplication of their results. We consider also a transposed version with `trmm`.

Contraction generalizes matrix-multiplication to N -dimensional tensors. We select three relevant contractions. For example, `abcd-aebf-dfce` is widely used in chemistry computation performing a reduction over the `ef` dimensions and resulting in an output tensor `abcd`. Another use of contractions is in Kronecker recurrent units, which reduces the size of neural network models. We consider an example of a Kronecker product of three matrices with `kronecker3`.

Convolution is essential in modern neural network-based visual processing. For the evaluation, we consider three representative shapes: 1) `conv1d` convolution between 1-D data such as two signal time series; 2) `conv2d` convolution between two 2-D inputs commonly used with images and feature extraction kernels; and 3) `conv3d` convolution between data such as video and 3-D kernels.

Fused Multilayer Perceptron or `MLP3` is a component of a production model, which consists of three MLP layers that feed into the binary classifier. Each MLP layer consists of matrix multiplication followed by a pointwise operation.

Long Short-Term Memory (`lstm`) is widely used in the recurrent neural network that uses temporal dependencies in data sequence (i.e., speech recognition) [28].

WaveNet is a full neural network model that generates raw audio waveforms. The model consists of causal and dilated convolutions, gated activation units, and residual and skip connections. The latter two components are represented by tensor contractions that can benefit from CIM acceleration.

Unsupported Kernels: Specific workloads are not offloaded due to limitations in the current operation set and mapping pipeline. Batch matrix multiplication and group convolution are not processed as they contain extra dimensions (batch and group dimension, respectively), which require additional, currently unsupported, preprocessing steps to map them into CIM primitives. In case of 2-D moments computation, a potential candidate for offloading is omitted as it consists of an input matrix I being multiplied with itself $I \times I$. This computation does not fit the current contraction detection model, which assumes that all not reduced dimensions are unique. Finally, some workloads, e.g., a gather operation, are ineligible for CIM acceleration as they cannot be rewritten into equivalent GEMM representation.

B. Effect of Kernel Size on CIM Performance

Fig. 7 shows the effect of varying the data size for the `mm` kernel. For smaller data sizes, programming the crossbar dominates the benefits in the execution time, resulting in 32% better performance for the baseline. But, as the data size increases, the CIM accelerator outperforms the host (ARM) processor by as much as 512 \times . The performance speedup mainly comes

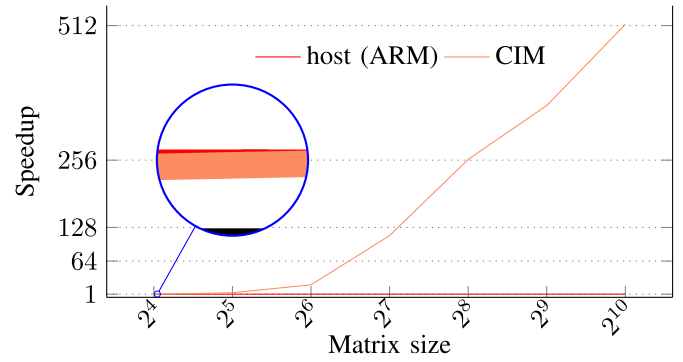


Fig. 7. Effect of matrix size on the CIM accelerator performance. The results are normalized to the host processor performance. The matrices always fit in the crossbar.

from the parallel nature of the accelerator and the slow down of the ARM processor due to limited data reuse in the small caches as well as its relatively longer execution time.

For the results in the following sections, we fix the size for majority of kernels to 256 to show the impact of tiling and loop interchange transformations. A few exceptions are made to retain representative workload shapes. The convolutions are performed on a single dataset with three input feature maps using the default size of 256 in every dimension. The inputs are convolved with three output feature maps using a dimension size of 3. The waveNet uses batch size of 1 and 32 residual and dilation channels, and 64 skip channels. The `lstm` is configured with 64 hidden states. Another exception is made for the two largest contractions, which dimensions are reduced to fit them in the main memory. The `abcd-aebf-dfce` is limited to a dimension size of 128 and the `kronecker3` uses size of 64×128 for its weights.

C. Effect on Performance

Fig. 8 (top) shows the performance comparison of the ARM core and the CIM accelerator for various workloads. We gather Gem5 statistics only for our region of interest, which consists of the offloaded kernel accounting for its execution and the data transfer time to and from the device. The latter is negligible and less than 3%. We neglect initialization statements as they are always executed on the CPU side in all the considered configurations.

Overall, the CIM accelerator improves performance in the majority of benchmarks and configurations. The tile configuration improves performance by 6.6 \times . Since a write in PCM is 2.5 \times more expensive compared to a read/compute operation (cf. Table II), the overall execution time in the tile configuration is dominated by the write operation. Adding the interchange pass on top of the tile pass reduces the number of write operations to the crossbar by 7.4 \times and gives us extra performance benefits. By parallelizing, we obtain the highest gains 15.5 \times for the `tile+parallel` configuration and 17 \times for the `tile+parallel+interchange` configuration.

The effective utilization of the crossbar also has a significant impact on the benchmarks' performance gain. Once mapped, the GEMM-like kernels utilize the CIM accelerator 100%. On the contrary, the kernel size in convolution benchmarks is

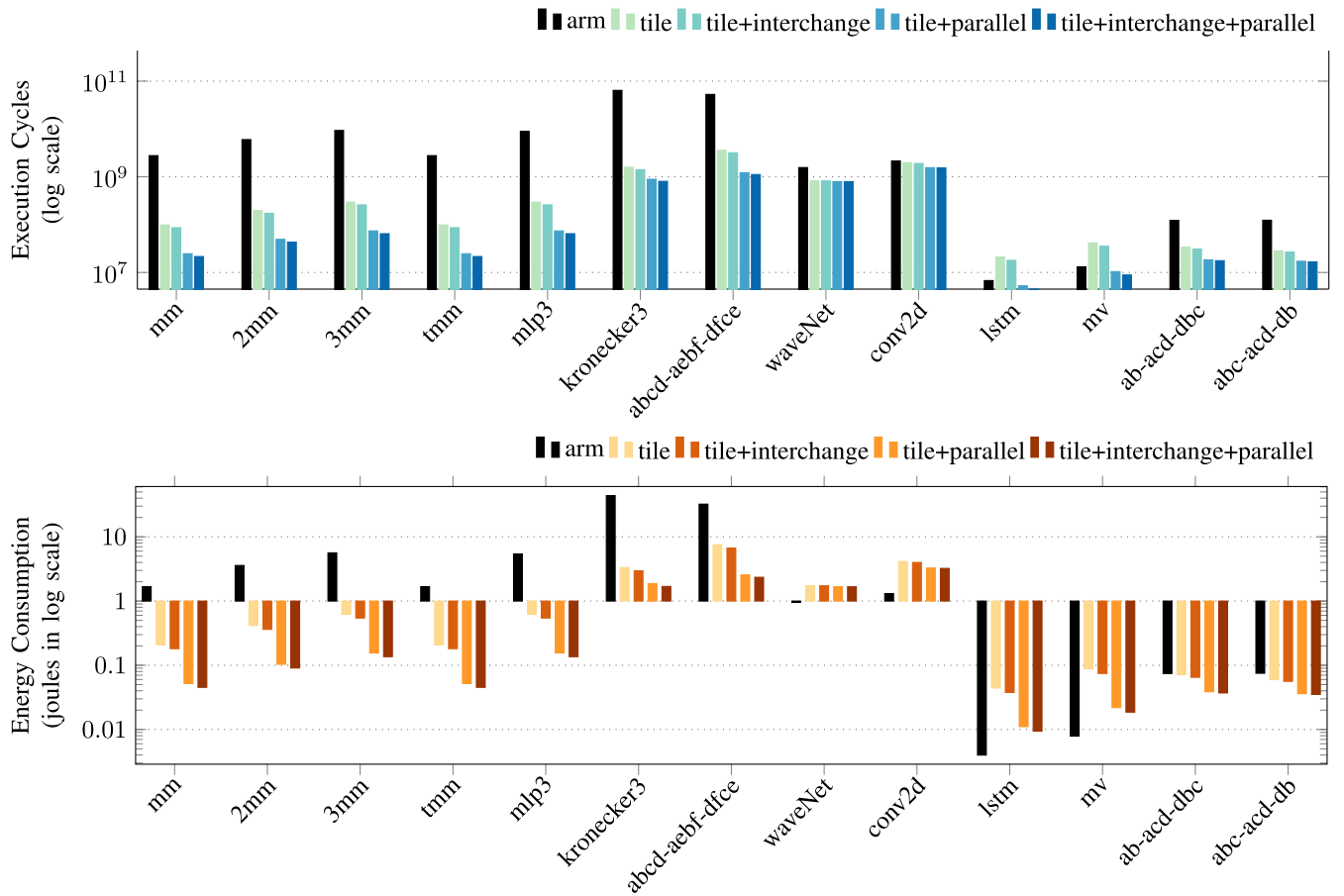


Fig. 8. Performance (top) and energy consumption results (bottom) of various configurations.

much smaller compared to the tile size and utilizes only around 2% of the crossbar accelerator. In WaveNet, the utilization of the CIM accelerator is around 6.5%. Similarly, a portion of the computations still has to be performed on the host (Fig. 9), which further limits the maximal achievable speedup.

Kernels with the highest potential for reuse benefit the most from the CIM accelerator (i.e., mm). Contrary kernels with low data reuse (i.e., mv and lstm) achieve better performance only in the parallelized configuration. This is reasonable as in the nonparallelized configurations, the overhead introduced by the expensive write operations is not amortized due to the low data reuse, which makes the ARM core faster than the CIM.

The convolution kernels get correctly detected and offloaded by the OCC. However, as the obtained results are similar in all three cases, only the most common ML kernel conv2d is shown. The convolution uses a box moving window of size 3, thus the workload does not fully utilize the CIM crossbars nor can it be parallelized. The execution time is dominated by the Im2Col transformation overhead, which limits the potential acceleration to the 34% of the kernel computations.

Similarly, the offloaded contractions in waveNet perform reduction over the dilation channels of size 32, which underutilize the crossbar and prevent parallelization. The overall speedup is also limited by the significant part of the kernel, which remains on the ARM CPU. For comparison, in waveNet, only 50% of the total computations can be offloaded, while this fraction increases to 99.5% for mlp3.

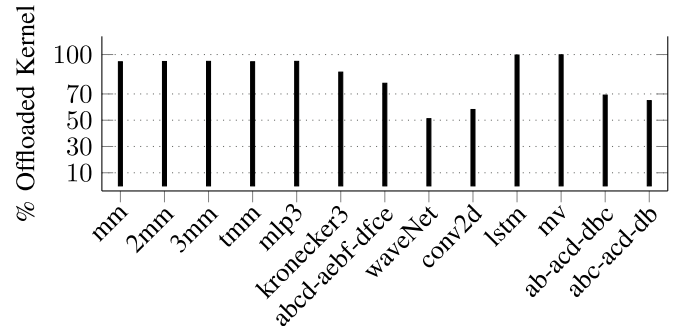


Fig. 9. Percentage of kernel workload performed on the CIM accelerator.

Looking at the contractions, the abcd-aebf-dfce achieves higher speedup than ab-acd-dbc and abc-acd-db as the baseline performance drops for abcd-aebf-dfce due to the high control-flow overhead and high-stride accesses. Fig. 10 shows the percentage of execution cycles spent on the host and on the CIM accelerator.

D. Effect on Energy Consumption

To measure the energy consumption, we feed McPact [29] with the Gem5 statistics. The energy estimate of the CIM module is not integrated into McPAT and is computed independently and added to the total energy based on the parameters provided in Table II.

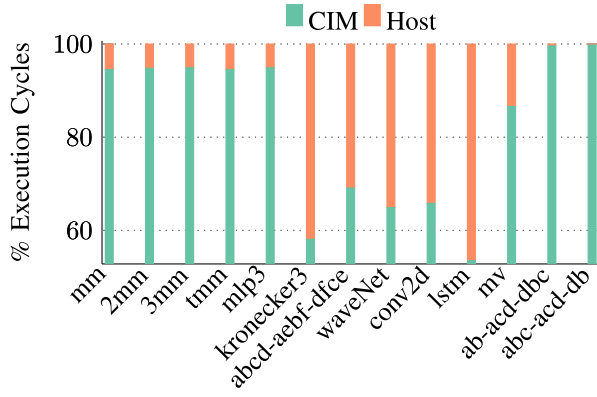


Fig. 10. Percentage of execution cycles spent on the host CPU and the CIM accelerator.

The CIM accelerator reduces the energy consumption by an average of $1.9\times$ – $5.0\times$ (Fig. 8). While the memory and compute operation in the accelerator are extremely cheap, the control unit, i.e., in/out/buffer registers, DAC/ADC, and other peripheral circuitry contribute a significant amount to the total energy consumption of the accelerator. The reduction in energy consumption of the CIM accelerator is mainly attributed to the lower compute energy of the device and reduced data movement, compared to the ARM processor.

Compared to the baseline ARM, the highest energy reduction of $4.5\times$ and $5.0\times$ is achieved for `tile+parallel` and `tile+parallel+interchange` configurations, respectively. They are also $2.6\times$ and $2.3\times$ better compared to the `tile` and `tile+interchange` configurations, respectively. The reduction in energy consumption comes from the shorter runtime attained by parallelizing the computations over multiple tiles.

Similar to the performance results, the energy consumption in some benchmarks also increases compared to the baseline arm configuration. In the `tile+interchange+parallel` configuration, `waveNet`, `conv2d`, `lstm`, and `mv` notice 60%, 30%, 10%, and 40% more energy consumption compared to the baseline arm configuration, respectively.

E. Effect on Endurance

One of the main challenges in the PCM-based architecture is the limited write endurance, currently projected between 10^7 and 10^9 [30], [31]. Once the endurance limit is reached, the PCM cell loses its ability to transition between state, potentially giving data errors. Multiple works provide wear-leveling techniques as hardware mechanisms [30]; OCC does that on the software side. OCC tackles the problem by maximizing the reuse of written tiles to the crossbar to reduce the overall number of write operations. We compare our default `tile` strategy with the `tile+interchange` where tile reuse is maximized by interchanging the point loops. To estimate the system lifetime, we adopt the formula as follows [30]:

$$\text{SystemLifetime} = \frac{\text{CellEndurance} * S}{B} \quad (1)$$

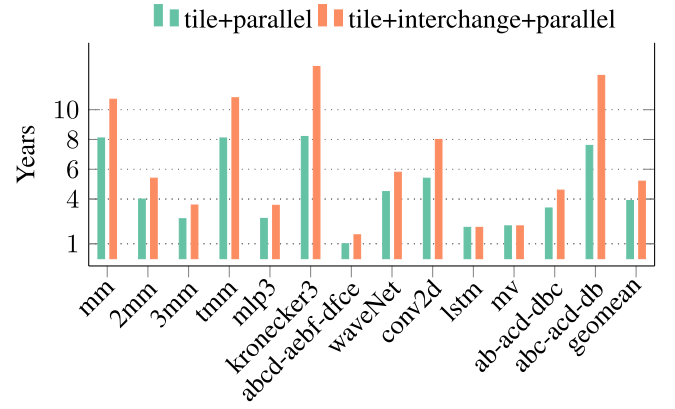


Fig. 11. OCC increases the crossbar's lifetime by minimizing the number of write operations to the crossbar.

where S is the crossbar's size ($4 \times (64 \times 64)$), while B is the write traffic in GB/s estimated with Gem5. We assume a cell endurance of 3.2×10^7 , as in [30], and a uniform write distribution on the crossbar. Fig. 11 shows the expected lifetime for the two mapping strategies. By exploiting data reuse OCC increases the device's lifetime by almost two years.

F. Comparison With Previous CIM Compilers

To show OCC's ability to identify and extract matrix and matrix–vector multiplications from multiple benchmarks and its robustness, we compare it with the work of Vadivel *et al.* [32] (TDO-CIM) and Drebes *et al.* [33] (TC-CIM). To the best of our knowledge, these are the only works on CIM compilation that aim at providing an end-to-end compilation for automatically and transparently invoking CIM acceleration. All the other works we are aware of require the users to rewrite the application to explicit CIM acceleration, reducing application readiness. As a metric for success, we count the number of identified matrix–matrix or matrix–vector operations identified by inspecting the generated code and comparing it with the maximum number of callsites expected for perfect matching. A callsite is thus an expected kernel identified and offloaded to the CIM accelerator. Fig. 12 shows the number of callsites for each benchmark. `conv` refers to either a 1d, 2d, or 3d convolution.

OCC behaves like the Oracle for the considered benchmarks, enabling to map each kernel on the crossbar efficiently. TDO-CIM and TC-CIM, on the other hand, miss some opportunities. Both frameworks are not able to identify and map contractions. Additionally, in `mlp3` and `waveNet`, both frameworks miss to identify one matrix–vector multiplication. The main culprit is how both TDO-CIM and TC-CIM operate. Both frameworks rely on loop tactics to identify computational motifs in low-level code. Loop tactics relies on canonicalization passes to provide robust detection; however, sometimes, it may fail. On the other hand, OCC relies on the progressive lowering preserving semantic information till when they are needed.

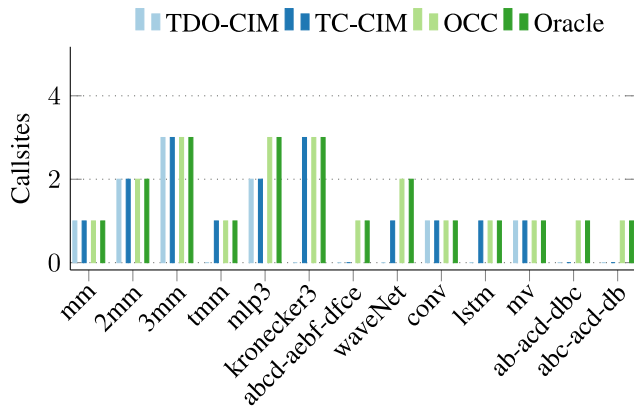


Fig. 12. Number of callsites for CIM library functions inserted by OCC compared to previous works and a perfect mapping (Oracle).

VI. RELATED WORK

There is a considerable body of work that proposes new architectures for in-memory computing, but few of them deal with the programmability issue. Ambrosi *et al.* [34] proposed a software stack with an ONNX backend targeting ISA-programmable memristor accelerators. The compiler generates ISA code from a graph representation of a neural network model constructed via the ONNX backend. In the first phase of the compilation, the graph is partitioned and the different operators are distributed to different virtual crossbars. In the second phase, data communication operations between producer and consumer are inserted by the compiler. Finally, the virtual tiles are placed on the physical tiles, in such a way that tiles that communicate frequently are placed closer together to minimize latency. The authors present a limited evaluation of the compiler. Building on their effort, Ankit *et al.* [24] developed a runtime compiler implemented as a C++ library, which requires the users to rewrite the application with the proposed API. On the contrary, our work enables transparent acceleration as computational idioms amendable for CIM acceleration are lowered to API calls directly without user intervention. A similar approach is used in other works [10], [35]–[37] where the in-memory accelerator exposes an API. During compilation, the API is lowered to data-path configurations and control-flow commands. Again, it is up to the programmer to rewrite the application using the provided API, thus reducing application readiness. Fujiki *et al.* [38] proposed a compiler framework that lowers Google’s TensorFlow DFG into simpler instructions supported by the in-memory accelerator. During compilation, complex instructions, such as division, exponents, and transcendental functions, are broken down into a set of LUTs, additions, and multiplications that can be executed efficiently on the crossbar array. Besides, a set of scheduling optimizations to expose instruction and block-level parallelism is also implemented. Software pipelining is used to overlap computation and storage in the CIM crossbar. Vadivel *et al.* [32] proposed TDO-CIM, an LLVM-based compiler to transparently detect and offload computational motifs suitable for in-memory execution starting from C or C++ code. Their approach relies on identifying computational patterns from the LLVM-IR, which is low level and

close to machine instructions, thus in some cases the detection fails. On the other hand, our approach exploits the progressive lowering provided by multilevel IR, making the discovery of motifs not only easier but also more robust. In a follow-up paper, Drebes *et al.* [33] proposed TC-CIM, a fully automatic, end-to-end compilation flow from TCs to fixed-function memristor-based hardware block. Also, in this case, motifs recognition is based on detection on the schedule tree—an internal representation of a polyhedral compiler. Although the detection has been proven to be robust to prior code transformations in some cases, the detection still fails. Tang *et al.* proposed to co-design accelerator with program synthesis frameworks to overcome the programmability challenges. Their approach is orthogonal to ours as they focus on exploiting the structured computation resulting from synthesis to enhance programmability, while OCC addresses the same issue from the software side [39].

VII. CONCLUSION

We presented the OCC, which is, to the best of our knowledge, the first end-to-end compilation flow for in-memory computing that takes advantage of multilevel IR rewriting for reliable mapping and transparent offloading for CIM computation. We illustrated how progressive lowering enables easy mapping of ML applications to the CIM crossbar, and how a small yet self-contained set of rewriting passes enables a CIM-friendly execution.

We evaluated our compilation flow using simulations based on statistically accurate PCM models capturing the behavior of devices fabricated in 90-nm CMOS technology and show the soundness of our approach.

In the future, we would like to extend OCC to: 1) detect and offload a broader set of operations (both logical and arithmetic) that can be accelerated using the in-memory programming model and 2) to provide, over time, a catalog of portable optimizations to benefit multiple in-memory technologies.

REFERENCES

- [1] S. Borkar, “Exascale computing—A fact or a fiction?” in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, May 2013, p. 3.
- [2] N. P. Jouppi *et al.*, “In-datacenter performance analysis of a tensor processing unit,” in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 1–12.
- [3] E. Chung *et al.*, “Serving DNNs in real time at datacenter scale with project brainwave,” *IEEE Micro*, vol. 38, no. 2, pp. 8–20, Mar./Apr. 2018.
- [4] A. Yang, “Deep learning training at scale spring crest deep learning accelerator (Intel®-Nervana™ NNP-T),” in *Proc. IEEE Hot Chips 31 Symp. (HCS)*, 2019, pp. 1–20.
- [5] A. Sebastian, M. L. Gallo, and E. Eleftheriou, “Computational phase-change memory: Beyond von-Neumann computing,” *J. Phys. D Appl. Phys.*, vol. 52, no. 44, Aug. 2019, Art. no. 443002.
- [6] V. Joshi *et al.*, “Accurate deep neural network inference using computational phase-change memory,” *Nat. Commun.*, vol. 11, p. 2473, May 2020.
- [7] A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh, and E. Eleftheriou, “Memory devices and applications for in-memory computing,” *Nat. Nanotechnol.*, vol. 15, pp. 529–544, Mar. 2020.
- [8] J. Castrillon *et al.*, “A hardware/software stack for heterogeneous systems,” *IEEE Trans. Multi-Scale Comput. Syst.*, vol. 4, no. 3, pp. 243–259, Jul.–Sep. 2018.

- [9] A. Shafiee *et al.*, “ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars,” in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 14–26.
- [10] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, “Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories,” in *Proc. 53rd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2016, pp. 1–6.
- [11] L. Truong *et al.*, “Latte: A language, compiler, and runtime for elegant and efficient deep neural networks,” in *Proc. 37th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, 2016, pp. 209–223.
- [12] T. Chen *et al.*, “TVM: An automated end-to-end optimizing compiler for deep learning,” in *Proc. 13th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2018, pp. 578–594.
- [13] (2018). *PlaidML*. [Online]. Available: <https://www.intel.ai/plaidml>
- [14] N. Vasilache *et al.*, “The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated GPU kernels, automatically,” *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, p. 38, 2020.
- [15] C. Lattner *et al.*, “MLIR: Scaling compiler infrastructure for domain specific computation,” in *Proc. CGO*, 2021, pp. 2–14.
- [16] T. Gysi *et al.*, “Domain-specific multi-level IR rewriting for GPU,” 2020. [Online]. Available: [arXiv:2005.13014](https://arxiv.org/abs/2005.13014).
- [17] A. Mehonic, A. Sebastian, B. Rajendran, O. Simeone, E. Vasilaki, and A. J. Kenyon, “Memristors—From in-memory computing, deep learning acceleration, spiking neural networks, to the future of neuromorphic and bio-inspired computing,” 2020. [Online]. Available: [arXiv:2004.14942](https://arxiv.org/abs/2004.14942).
- [18] A. Drebes. (2020). *Teckyl: An MLIR Frontend for Tensor Operations*. [Online]. Available: <https://github.com/andidr/teckyl>
- [19] P. Springer and P. Bientinesi, “Design of a high-performance GEMM-like tensor-tensor multiplication,” *ACM Trans. Math. Softw.*, vol. 44, no. 3, pp. 1–29, 2018.
- [20] A. Vasudevan, A. Anderson, and D. Gregg, “Parallel multi channel convolution using general matrix multiplication,” in *Proc. IEEE 28th Int. Conf. Appl. Spec. Syst. Archit. Processors (ASAP)*, 2017, pp. 19–24.
- [21] S. Chetlur *et al.*, “cuDNN: Efficient primitives for deep learning,” 2014. [Online]. Available: [arXiv:1410.0759](https://arxiv.org/abs/1410.0759).
- [22] M. Le Gallo, A. Sebastian, G. Cherubini, H. Giefers, and E. Eleftheriou, “Compressed sensing with approximate message passing using in-memory computing,” *IEEE Trans. Electron Devices*, vol. 65, no. 10, pp. 4304–4312, Oct. 2018.
- [23] H. Jiang, W. Li, S. Huang, S. Cosemans, F. Catthoor, and S. Yu, “Analog-to-digital converter design exploration for compute-in-memory accelerators,” *IEEE Design Test*, early access, Jan. 11, 2021, doi: [10.1109/MDAT.2021.3050715](https://doi.org/10.1109/MDAT.2021.3050715).
- [24] A. Ankit *et al.*, “PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference,” in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2019, pp. 715–731.
- [25] A. V. D. Oord *et al.*, “WaveNet: A generative model for raw audio,” 2016. [Online]. Available: [arXiv:1609.03499](https://arxiv.org/abs/1609.03499).
- [26] K. Stock, T. Henretty, I. Murugandi, P. Sadayappan, and R. Harrison, “Model-driven SIMD code generation for a multi-resolution tensor kernel,” in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2011, pp. 1058–1067.
- [27] G. Baumgartner *et al.*, “Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models,” *Proc. IEEE*, vol. 93, no. 2, pp. 276–292, Feb. 2005.
- [28] M. Zhang, S. Rajbhandari, W. Wang, and Y. He, “DeepCPU: Serving RNN-based deep learning models 10x faster,” in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 951–965.
- [29] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, “McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures,” in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2009, pp. 469–480.
- [30] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, “Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling,” in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2009, pp. 14–23.
- [31] G. W. Burr *et al.*, “Phase change memory technology,” *J. Vacuum Sci. Tech. B, Nanotechnol. Microelectron. Mater. Process. Meas. Phenomena*, vol. 28, no. 2, pp. 223–262, 2010.
- [32] K. Vadivel *et al.*, “TDO-CIM: Transparent detection and offloading for computation in-memory,” in *Proc. Design Autom. Test Europe Conf. Exhibition (DATE)*, 2020, pp. 1602–1605.
- [33] A. Drebes *et al.*, “TC-CIM: Empowering tensor comprehensions for computing-in-memory,” in *Proc. 10th Int. Workshop Polyhedral Compilation Techn.*, Jan. 2020, pp. 1–12.
- [34] J. Ambrosi *et al.*, “Hardware-software co-design for an analog-digital accelerator for machine learning,” in *Proc. IEEE Int. Conf. Rebooting Comput. (ICRC)*, Nov. 2018, pp. 1–13.
- [35] P. Chi *et al.*, “PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory,” in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 27–39.
- [36] L. Song, X. Qian, H. Li, and Y. Chen, “PipeLayer: A pipelined ReRAM-based accelerator for deep learning,” in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2017, pp. 541–552.
- [37] M. N. Bojnordi and E. Ipek, “Memristive Boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning,” in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Mar. 2016, pp. 1–13.
- [38] D. Fujiki, S. Mahlke, and R. Das, “In-memory data parallel processor,” in *Proc. 23rd Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2018, pp. 1–14.
- [39] Y. Tang, H. Jia, and N. Verma, “Reducing energy of approximate feature extraction in heterogeneous architectures for sensor inference via energy-aware genetic programming,” *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 67, no. 5, pp. 1576–1587, May 2020.



Adam Siemieniuk received the B.Sc. degree in electrical engineering and the M.Sc. degree in embedded systems from the Eindhoven University of Technology, Eindhoven, The Netherlands, in 2017 and 2021, respectively.

His research interest includes compilation and optimizations for heterogeneous architectures.



Lorenzo Chelini received the B.S. degree in computer engineering from the University of Pisa, Pisa, Italy, in 2015, and the M.S. degree in computer engineering from the Politecnico di Torino, Turin, Italy, in 2017. He is currently pursuing the Ph.D. degree with the Eindhoven University of Technology, Eindhoven, The Netherlands.

His research interest includes heterogeneous compilation and multilevel intermediate representations.



Asif Ali Khan is currently pursuing the Ph.D. degree with the Chair for Compiler Construction, Computer Science Department, Dresden University of Technology, Dresden, Germany.

His research interests include computer architecture, heterogeneous memories, and compiler support for the memory system. His research mainly focuses on exploring the emerging nonvolatile memory technologies in the memory subsystems and their optimizations for various metrics.



Jeronimo Castrillon (Senior Member, IEEE) received the Electronics Engineering degree from the Pontificia Bolivariana University, Antioquia, Colombia, in 2004, the master's degree from the Advanced Learning and Research Institute, Lugano, Switzerland, in 2006, and the Ph.D. (Dr.-Ing.) degree (Hons.) from the RWTH Aachen University, Aachen, Germany, in 2013.

He is a Professor with the Department of Computer Science, Dresden University of Technology, Dresden, Germany, where he is also affiliated with the Center for Advancing Electronics Dresden. He is the Head of the Chair for Compiler Construction, with research focus on methodologies, languages, tools and algorithms for programming complex computing systems. In 2014, he co-founded Silexica GmbH/Inc., Cologne, Germany, a company that provides programming tools for embedded multicore architectures.

Andi Drebes received the Ph.D. degree from Université Pierre et Marie Curie (Paris VI), Paris, France, in 2015.

After Postdoctoral Research with the University of Manchester and Inria, he is currently working as a Compiler Engineer. His research focuses on compilation, dynamic optimization, and performance analysis.



Henk Corporaal received the M.Sc. degree in theoretical physics from the University of Groningen, Groningen, The Netherlands, in 1982, and the Ph.D. degree in electrical engineering, in the area of computer architecture, from the Delft University of Technology, Delft, The Netherlands, in 1995.

He is a Professor of Embedded System Architectures with the Eindhoven University of Technology, Eindhoven, The Netherlands. He has coauthored over 500 journal and conference papers.

Furthermore, he invented a new class of VLIW architectures, the Transport Triggered Architectures, which is used in several commercial products, and by many research groups. His research is on low power multiprocessor, heterogenous processing architectures, their programmability, and the predictable design of soft- and hard real-time systems. This includes research and design of embedded system architectures, including CGRAs, SIMD, VLIW, and GPUs, on accelerators, the exploitation of all kinds of parallelism, fault-tolerance, approximate computing, architectures for machine and deep learning, optimizations and mapping of deep learning networks, and the (semi-)automated mapping of applications to these architectures. For further details see corporaal.org.



Tobias Grosser (Member, IEEE) received the Diploma degree in computer science from the University of Passau, Passau, Germany, in 2011, and the Ph.D. degree in computer science from Université Pierre et Marie Curie (Paris), Paris, France, in 2014, and was an SNSF Ambizione Postdoctoral Fellow until 2020.

He is an Associate Professor (Reader) with the University of Edinburgh, Edinburgh, U.K. His research advances programming languages, compilers, and hardware design languages to facilitate the development of and reasoning about high-performance software and hardware.



Martin Kong received the Ph.D. degree in computer science from the Ohio State University, Columbus, OH, USA, in 2016.

He is an Assistant Professor with the School of Computer Science, University of Oklahoma, Norman, OK, USA. His research interests include high-performance computing, programming languages, optimizing compilers, domain specific computing, data-flow and task parallelism, and programming models.