A Compilation Tool for Computation Offloading in ReRAM-based CIM Architectures

HAI JIN, BO LEI, HAIKUN LIU, XIAOFEI LIAO, ZHUOHUI DUAN, CHENCHENG YE, and YU ZHANG, Huazhong University of Science and Technology, China

Computing-in-Memory (CIM) architectures using *Non-volatile Memories* (NVMs) have emerged as a promising way to address the "memory wall" problem in traditional Von Neumann architectures. CIM accelerators can perform arithmetic or Boolean logic operations in NVMs by fully exploiting their high parallelism for bit-wise operations. These accelerators are often used in cooperation with general-purpose processors to speed up a wide variety of artificial neural network applications. In such a heterogeneous computing architecture, the legacy software should be redesigned and re-engineered to utilize new CIM accelerators. In this article, we propose a compilation tool to automatically migrate legacy programs to such heterogeneous architectures based on the *low-level virtual machine* (LLVM) compiler infrastructure. To accelerate some computations such as vector-matrix multiplication in CIM accelerators, we identify several typical computing patterns from LLVM *intermediate representations*, which are oblivious to high-level programming paradigms. Our compilation tool can modify accelerable LLVM IRs to offload them to CIM accelerators automatically, without re-engineering legacy software. Experimental results show that our compilation tool can translate many legacy programs to CIM-supported binary executables effectively, and improve application performance and energy efficiency by up to 51× and 309×, respectively, compared with general-purpose x86 processors.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Computer systems organization** → **Analog computers;**

Additional Key Words and Phrases: ReRAM, accelerator, LLVM-IR, compilation

ACM Reference format:
Hai Jin, Bo Lei, Haikun Liu, Xiaofei Liao, Zhuohui Duan, Chencheng Ye, and Yu Zhang. 2023. A Compilation Tool for Computation Offloading in ReRAM-based CIM Architectures. *ACM Trans. Arch. Code Optim.* 20, 4, Article 47 (October 2023), 25 pages.
https://doi.org/10.1145/3617686

1 INTRODUCTION

Recently, ***Computing-in-Memory* (CIM)** architectures [14, 25, 40] have emerged as a new computing paradigm to solve the memory-wall problem of traditional Von Neumann computer

This work is supported jointly by National Key Research and Development Program of China under Grant No. 2022YFB4500303, and National Natural Science Foundation of China (NSFC) under Grants No. 62072198, No. 61825202, and No. 61929103.

Author's address: H. Jin, B. Lei, H. Liu (Corresponding author), X. Liao, Z. Duan, C. Ye, and Y. Zhang, National Engineering Research Center for Big Data Technology and System, Service Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China; e-mails: {hjin, leibo, hkliu, xfliao, zhduan, yecc, zhyu}@hust.edu.cn.

1544-3566/2023/10-ART47
https://doi.org/10.1145/3617686

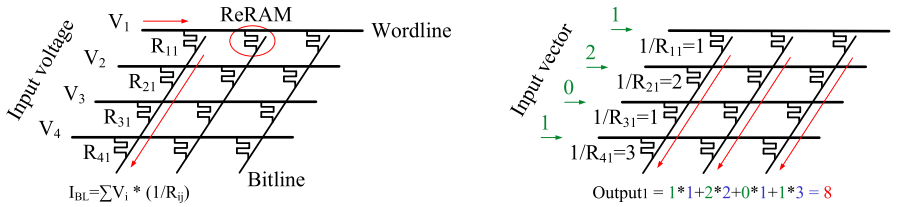
architectures. Most CIM accelerators use ***Non-volatile memories* (NVM)** technologies such as ***Resistive Random-access Memory* (ReRAM)** to architect crossbar arrays, and perform *in situ* analog computing based on the Kirchhoff's circuit laws [33]. ReRAM crossbars offer high parallelism for bitwise computation, and avoid data movement between storage and processing units [25, 40]. Particularly, ReRAM crossbar structures can perform ***Matrix-Vector Multiplication* (MVM)** operations in the analog domain with a constant time ($O(1)$ complexity). They have demonstrated a significant potential to improve performance and energy efficiency for domain-specific applications, such as neuromorphic computing [39], graph computing [37, 45], database applications [29, 34, 35], and image processing [24].

Many neural network and graph computing applications contain a large portion of computing-intensive MVM operations [39, 55]. To accelerate these applications with ReRAM crossbars, they have to be re-engineered to offload MVM operations to CIM accelerators. However, because the physical feature of ReRAM is different from traditional CMOS circuits, and the CIM computing paradigm is also different from the traditional Von Neumann's, many legacy applications are infeasible for acceleration in CIM architectures. Thus, it is essential to redesign new programming models, APIs, and compilation tools to program ReRAM-based CIM accelerators.

There have been some preliminary studies on programming models for CIM architectures. Yu et al. [54] provide a specific language and ***application programming interfaces* (APIs)** for ReRAM-based accelerators. Similarly, Ankit et al. [10] design a new ***instruction set architecture* (ISA)** and a compiler for ReRAM-based accelerators. However, programmers need to modify the source code of legacy software according to new library functions. Ambrosi et al. [8] develop a software stack to run existing applications in ReRAM accelerators without modifying source codes. However, it only supports neural network applications developed in a specific framework. Chakraborty et al. [13] propose a programming framework to convert applications' source codes into ***Boolean Decision Diagram* (BDD)**, and then map a portion of computation to ReRAM crossbars. However, it only supports very limited computing patterns. A state-of-the-art work [47] exploits LLVM ***intermediate representations* (IRs)** to identify accelerable parts from the ***Control Flow Graph* (CFG)**, and provides a compiler [21] for the proposed CIM architecture. However, it only supports program with source codes, and can not migrate binary executables to CIM accelerators. These proposals all need to redesign, recompile application source codes to adapt to CPU/CIM heterogeneous architectures, impeding the application of CIM accelerators for a wide range of legacy software.

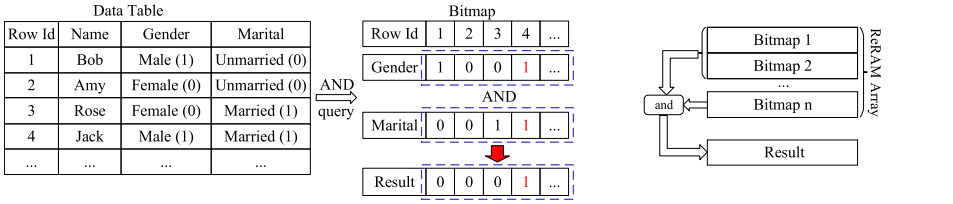
In this article, we propose a new compilation tool to migrate legacy programs to CPU/CIM heterogeneous architectures automatically. Our compilation tool relies on the LLVM compiler infrastructure to compile the application's source codes into LLVM IR. For binary executables without source codes, we exploit a decompiling tool [3] to generate the programs' IR. We define and identify several typical computing patterns that can be accelerated by ReRAM-based accelerators from the perspective of LLVM IR. Our compilation tool can recognize the accelerable computing patterns such as MVM and Boolean logical operations, and automatically offload those computations to CIM accelerators using very simple APIs. We make the following contributions:

- For domain-specific applications, we define and abstract several computing patterns that can be accelerated by ReRAM-based crossbars from the perspective of LLVM IR, so that our compilation tool can recognize them and offload these computations to ReRAM-based crossbars effectively.
- We further propose a performance model to quantify the performance gain of offloading these accelerable computing patterns to CIM accelerators, and thus can improve the efficiency of computation offloading.



(a) ReRAM crossbar structure (b) An example of analog computing for MVM

Fig. 1. ReRAM crossbar array used for matrix-vector multiplication.



(a) Data queries using Boolean logic operations (b) ReRAM architecture for Boolean logic operations

Fig. 2. Example of data querying using ReRAM-based Boolean logic operations.

- We develop a compilation tool to migrate legacy programs to CPU/CIM heterogeneous architectures automatically, without modifying application source codes. It can also transparently convert binary executables without source codes by exploiting a decompilation tool.
- With our compilation tool, experimental results show that the performance of domain-specific programs can be significantly accelerated by up to 51×, and the energy consumption can be reduced by up to 309×, compared with general-purpose x86 processors.

The remainder of this article is organized as follows. Section 2 introduces the background and motivation of this work. Section 3 shows the architecture of our system and the design of the compilation tool. Section 4 introduces four kinds of ReRAM-accelerable LLVM IR patterns. Section 5 describes how the compilation tool identifies the accelerable patterns and performs the code translation. Section 6 shows how binary executables generated by our compilation tool are executed at runtime. Section 7 presents experimental results. We discuss the related work in Section 8 and conclude in Section 9.

2 BACKGROUND AND MOTIVATION

2.1 CIM Accelerators

ReRAM is a emerging NVM technology that stores data by adjusting its resistance. This feature also enables highly parallel computation based on the Kirchhoff's circuit laws. Figure 1 shows that a typical ReRAM crossbar array can complete matrix-vector multiplication in a constant time ($O(1)$ complexity), while a typical CPU often takes a quadratic time ($O(n^2)$ complexity). Therefore, the ReRAM crossbar-based CIM accelerators have been widely studied to accelerate domain-specific applications that contains a large number of MVM operations.

Moreover, ReRAM can be also used in other architectures to accelerate bitwise Boolean logic operations, which are widely used in applications such as databases [29, 35, 46]. Figure 2 shows an example of data querying via ReRAM-based Boolean logic operations. To retrieve all married man from the data table, only an AND operation is needed for two bitmaps generated from columns "Gender" and "Marital."

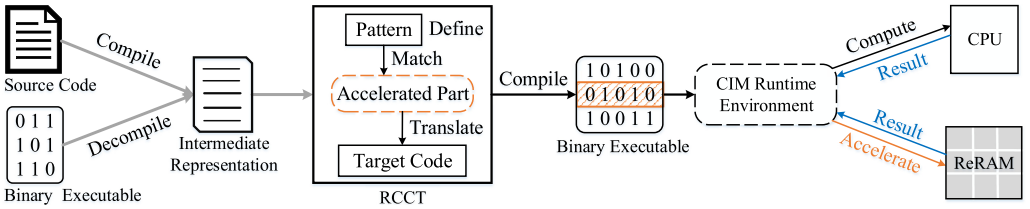


Fig. 3. Sketch of migrating software to CPU/CIM heterogeneous architectures using our compilation tool.

Besides the above examples, ReRAM-based CIM architectures can be applied to many other scenarios for higher performance and energy efficiency. Therefore, there have been many studies on the adaption of specific applications to CIM architectures.

2.2 Motivation

To accelerate applications with CIM accelerators, most previous proposals usually provide specific APIs for programmers, and have to modify the source codes for legacy applications. These proposals have several limitations to adopt CIM accelerators for a large amount of applications.

- **Poor Programmability.** CIM accelerators usually offer an ISA and new APIs for application programmers [22]. Programmers have to spend a lot of time to learn new APIs for using CIM accelerators. A rookie programmer may wrongly utilize CIM accelerators, and even causes performance degradation for applications.
- **Poor Compatibility.** Most previous CIM architectures and the corresponding optimizations are usually targeted at a given application or a special goal [27, 28, 44]. The poor compatibility limits the adoption of those proposals for a wide range of programs and application scenarios.
- **Poor Portability.** All previous proposals have to modify source codes of legacy applications. For applications with only binary executables, it is difficult to migrate them to CIM architectures.

Although there have been some previous studies [13, 47] on migrating legacy applications to CIM architectures, they have to modify applications' source codes, and can only accelerate very limited computing patterns. In this work, our goal is to facilitate the adoption of CIM accelerators for diverse legacy software. We analyze and identify typical computing patterns that can be accelerated by CIM accelerators at the LLVM IR layer, because LLVM IR has many good features. First, since LLVM IR offers a low-level and universal abstract model for many programming languages, different implementations in source codes for a given task (e.g., different types of loops) can be represented by the same LLVM IR. Second, the binary executables of applications can be decompiled into LLVM IRs via reverse engineering tools [1, 3–5, 53] even if the source codes of applications are unavailable. Third, LLVM IR is portable, and platform/language independent. It offers vast opportunities to analyze/optimize the program codes with a variety of transformations at compile-time, link-time, or run-time. The low-level abstraction of LLVM IR offers an efficient and simple way to recognize special computing patterns that can be accelerated by ReRAM crossbar arrays.

3 SYSTEM OVERVIEW

Figure 3 shows the flow diagram of legacy software migration using our ***ReRAM-based CIM Compilation Tool*** **(RCCT)**. At first, we should generate program's LLVM IR. If the source codes of applications are available, then we compile source codes into IRs. Otherwise, we exploit reverse engineering tools to decompile binary executables into IRs. In our system, we mainly compile

C/C++ source codes to LLVM IR (.ll) via Clang/Clang++. For binary executables, we choose a high-maturity decompiler [3] to generate the LLVM IR. Second, our compilation tool identifies typical code snippets according to predefined patterns, and it determines whether they can be accelerated by CIM accelerators based on a simple performance model. For accelerable code snippets, our compilation tool replaces the codes with APIs specified by CIM accelerators. The modified LLVM IR can be recompiled into binary executables again. Finally, when the binary executable is executed, the ***CIM runtime environment*** **(CRE)** determines whether each instruction should be executed on the CPU or ReRAM arrays. The CIM runtime also should fetch data into ReRAM arrays from main memory according to the API type and the memory address.

4 RERAM-ACCELERABLE LLVM IR PATTERNS

Current ReRAM-based CIM architectures only support **multiply-add (MAD)** operations in an analog computing paradigm [39, 55], or bitwise Boolean logic operations in a digital computing paradigm [29, 35, 46]. They can be adopted in very limited scenarios to accelerate MVM, MMM, and bitwise operations for Boolean vectors. In the following, we elaborate the detailed methodologies of recognizing four typical accelerable patterns for both compiled IRs and decompiled IRs.

4.1 Accelerable Patterns

To accelerate the execution of applications, we should find out typical computing patterns in LLVM IR that can be accelerated by ReRAM. Although there are an increasing number of application scenarios for ReRAM, the fundamental operations that can be accelerated by ReRAM-based CIMs are very limited. According to the physical characteristics of ReRAM, we recognize four typical accelerable computing patterns, i.e., ***matrix-vector multiplications*** **(MVMs)**, ***matrix-matrix multiplications*** **(MMMs)**, library functions for MVM and MMM operations, and bitmap logical operations. Among these patterns, library functions are relatively simple and easy to identify, while the other three patterns are all implemented with more complex loop tactics. Therefore, to identify these accelerable patterns, we have to analyze the loop structure in the IR. For example, a MVM is typically implemented by a two-level nested loop, and the iteration statements in the inner loop body are related to vectors and matrices. Thus, to identify a MVM, we have to identify the pattern of the 2-level nested loop at first, and then recognize the MAD operation of two vectors in the inner loop.

Fortunately, since the LLVM IR has the ability of low-level abstraction for different programming languages, it can be exploited to recognize computing patterns that can be accelerated by ReRAM crossbar arrays. Their structural and critical information can be found in different blocks of the LLVM IR. One IR file usually contains many blocks and each block consists of multiple statements. Figure 4 shows the logic blocks of the two-level nested loop in the LLVM IR. The circles with different colors represent different types of blocks. The green, orange, and purple circles represent the logic blocks of the loop condition, the loop body, and the loop end. These blocks are connected by jump labels (statements), including unconditional jump and conditional jump. Therefore, we analyze these blocks in LLVM to identify accelerable MVM operations, and convert them into APIs defined by CIM accelerators.

These blocks can be easily identified in the LLVM IR, because it maintains some useful information during the compiling. For example, we can infer the type of the block from its name directly (e.g., *for.cond*, *for.body*, and *for.end*). However, it is hard to identify loop structures in an IR decompiled from an binary executable, because the decompilation tool does not provide sufficient semantic information about blocks (all blocks have similar names like "block_4006f 7"). Furthermore, the number of instructions generated from reverse engineering often increases by several times, and instructions also become more complex compared with the IR compiled from source

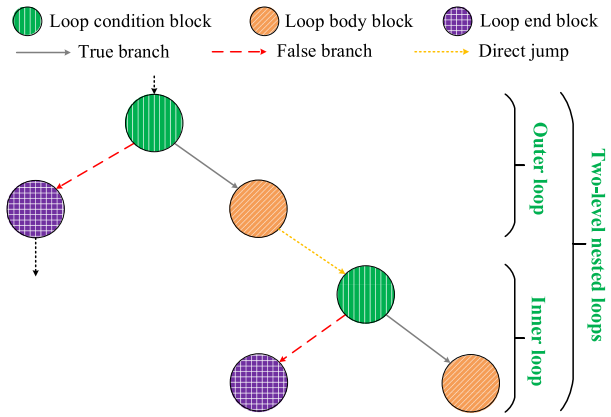


Fig. 4. Logic tree of blocks in a MVM operation represented by a two-level nested loop.

Pattern 1: A typical MVM pattern in LLVM IR generated from source codes

```
loop.condA:                          ; "loop" can be "for" or "while"
  %range_1 = load <i_ty>, <i_ty>* [i]
  %cmp1 = icmp <cmp> <i_ty> %range_1, [r_1]
  br i1 %cmp1, label %loop.bodyB, label %loop.endX
loop.bodyB:
  br label %loop.condC
loop.condC:
  %range_2 = load <i_ty>, <i_ty>* [j]
  %cmp2 = icmp <cmp> <i_ty> %range_2, [r_2]
  br i1 %cmp2, label %loop.bodyD, label %loop.endY
loop.bodyD:
  %arrayidx1 = getelementptr inbounds <ty>, <ty>* addr, <i_ty> idx, <i_ty> idx1
  %val_1 = load <ty>, <ty>* %arrayidx1          ; %val_2 and %val_3 are loaded similarly
  %mul = mul/fmul <ty> %val_1, %val_2
  %add = add/fadd <ty> %val_3, %mul_e
```

codes. Moreover, blocks generated by decompiling tools usually are out of order, and often lose the program semantic in the high-level programming language. To address this problem, we conduct massive experiments to get IRs from both compiling and decompiling for the same program, and then reorder the blocks in the decompiled IR. At last, we analyze the type of blocks according to some critical instructions, and determine whether these blocks match the accelerable computing patterns.

4.2 Matrix-Vector Multiplication

MVM is a typical computing pattern that can be accelerated by CIM accelerators. For most neural networks and graph processing applications, massive MVM operations often have a significant impact on the performance and energy consumption. Since ReRAM crossbar arrays can perform a MVM operation in a constant time ($O(1)$ complexity) via the analog computing model, they have been widely exploited to accelerate these applications.

At first, we analyze a program's IR generated from its source codes. Pattern 1 shows the MVM pattern in the LLVM IR. We can find the same logical structure in Figure 4. `Loop.condA`

Pattern 2: An MVM pattern in the IR decompiled from an executable binary

```
block_A:
    %a = phi %struct.Memory* [%n1, block_M], [%n2, ...]
    %b = load i32 i32* [r_1]
    %d = icmp <cmp> i32 %c, i32 0
    %f = select i1 %e, i64 num1, i64 6
    %br i1 %g, label %block_B, label %block_X
block_B:
    store i32 [num], i32 * %r_2
    br label %block_C
block_C:...
block_D:
    %o_1 = mul i32 %a_1, %col              ; %col represents the column size of the matrix
    %add_mid = add i64 %o_1, %m_addr
    %v1_addr = add i64 %a_2, %add_mid
    %v2_addr = add i64 %input_addr, %a_1
    %mul = mul/fmul <ty> %val_1, %val_2
    %v3_addr = add i64 %a_2, %output_addr
    %add = add/fadd <ty> %val_3, %mul_e
```

and `loop.condC` are loop condition blocks of outer and inner loops, while `loop.bodyB` and `loop.bodyD` are loop body blocks of outer and inner loops. We note that the block for updating the loop control variable is not presented in Pattern 1. The prefix `loop.` represents loops generated from "for" or "while" statements. The label `A-D` can be letters or numbers. `loop.bodyB` is mainly used to initialize the second layer of loop variables, while `loop.bodyD` is used to complete the innermost loop where the application logic is implemented. Moreover, some critical instructions and dependencies are constrained in `loop.bodyD`. For example, *arrayidx#* denotes that the data involved in the MVM operation comes from a matrix or a vector. In the statements of the IR, [*r_#*] represents the loop condition variable, including local variables and global variables, and *%range_#* represents the corresponding values. [*v_#*] represents a variable that stores the range of the loop. <*cmp*> represents the type of a comparison, such as *less* and *less than or equal to*. The *icmp* instruction is used to compare two variables (e.g., *%range_1* and [*v_1*]) and stores the output in a boolean variable (e.g., *cmp1*). This critical instruction is used to determine whether the two-layer loop can continue. <*ty*> represents the data type and <*i_ty*> refers to the integer type specifically. The result of vector multiplication is assigned to the output vector and then accumulated. Thus, the dependence between *mul* and *add* should be constrained in `loop.bodyD`, where *mul*/*add* and *fmul*/*fadd* represent operations for integers and floating-point numbers, respectively. We note that *%mul_e* is just an alias of *%mul*, or *%mul_e* is a multiple of *%mul*. Overall, to recognize the MVM pattern, we have to identify the two-level nested loop structure at first, and then identify the multiplication and addition instructions in the inner loop.

Second, we decompile the same program's binary executable to generate its IR, and illustrate the MVM pattern of the IR in Pattern 2. We can find the same logical structure shown in Figure 4, but the name and key instructions of each block are significantly different from the IR in Pattern 1. We note that these key instructions in the blocks are formalized by a high-level abstraction. We can find that the critical instructions in the loop condition block `block_A` is completely different from the `loop.condA` block in Pattern 1. The significant difference is mainly caused by the decompiling tool. However, we can still infer that this block is the loop condition based on the

Pattern 3: An MMM pattern in the IR generated from source codes

```
1: loop.condA:   ...
2: loop.bodyB:   ...
3: loop.condC:   ...
4: loop.bodyD:   ...
5: loop.condE:   ...
6: loop.bodyF:
7:     ...
8:     %mul = mul/fmul <ty> %val_1, %val_2
9:     %add = add/fadd <ty> %val_3, %mul_e
```

icmp instruction and the pattern of following blocks. Also, similar to Pattern 1, [*%r_#*] represents loop condition variables. Moreover, the instructions in the loop body block (`block_D`) have also changed significantly. Unlike the key variable `arrayidx_n` in Pattern 1, no instruction can directly indicate vectors or matrices in the innermost loop body block. However, we find there are total four `add` instructions involved in memory addressing, and two other instructions are used to perform multiplication and addition operations. The first three `add` instructions (lines 14–16) are used to calculate the element's address of the matrix and the input vector, and one more `add` instruction (line 18) is used to address the element of the output vector, where *%m_addr*, *input_addr*, and *output_addr* represent the starting address of the matrix, the input vector, and the output vector, respectively. *a_#* represents the index of the element in vectors or matrices. Their elements can be accessed directly by the index from the starting address of *a*. The value of the variable *%val_#* corresponds to the virtual address *%v#_addr*. Since these instructions are tightly coupled with a high dependency, we can infer that these block actually performs an MVM operation by recognizing the pattern of these seven instructions.

Overall, although instructions in the compiled IR is different from that in the decompiled IR, the MAC operations and constraints among the input vector, the input matrix, and the output vector are immutable. To recognize this pattern, we should first identify the two-level nested loop structure, and then recognize the key instructions and their dependencies.

4.3 Matrix-Matrix Multiplication

MMM is a common operation in neural network applications. In traditional computer architectures, MMM is costly in terms of execution time and energy consumption. It can be also accelerated by ReRAM [26] by decomposing an MMM into multiple MVMs. Thus, we also recognize the MMM pattern in the LLVM IR.

The logical structure of MMM is similar to MVM. An MMM operation should be performed in a three-level nested loop where the key instructions are executed in the innermost loop body. Pattern 3 shows the MMM pattern of the IR compiled from source codes. Similar to the structure of MVM in Pattern 1, `loop.condX` and `loop.bodyX` represent the loop condition block and the loop body block in each level, respectively. Except for `loop.bodyF`, they have the same roles as that in Pattern 1, so that we omit the instructions in these blocks. The key instructions in `loop.bodyF` are also similar to that in the MVM, but *val_#* are all numbers in matrices. To recognize the pattern of an MMM, we only have to identify the three-level loop structure and the multiplication/addition operations in the innermost loop.

The MMM pattern of the IR decompiled from a binary executable is shown in Pattern 4. It has the same structure as Pattern 3. However, the key instructions in different blocks are similar to the MVM pattern (Pattern 2), so that we omit blocks A–E. In *block_F*, the memory addressing of each

Pattern 4: An MMM pattern in the IR decompiled from a binary executable

```
1:  block_A:
2:      ...
3:  block_F:
4:      %addr_mid1 = add i64 %o_1, %m1_addr
5:      %v1_addr = add i64 %a_2, %addr_mid1
6:      %addr_mid2 = add i64 %o_2, %m2_addr
7:      %v2_addr = add i64 %a_3, %addr_mid2
8:      %mul = mul/fmul <ty> %val_1, %val_2
9:      %addr_mid3 = add i64 %a_4, %m3_addr
10:     %v3_addr = add i64 %a_3, %addr_mid3
11:     %add = add/fadd <ty> %val_3, %mul_e
```

Pattern 5: A library function pattern in the IR

```
1:  call ret_type @function_name(<ty> p_1, ..., <ty> p_n)
```

element in a matrix is also similar to that in an MVM. The difference is that total six *add* instructions are used for memory addressing in the innermost loop body. According to these tightly coupled instructions, we can easily recognize the MMM pattern in the IR.

4.4 Library Functions of MVM and MMM

Many programming libraries provide MVM and MMM functions to improve developer's productivity, such as *CBLAs* library functions in the Caffe framework, *GEMV* and *GEMM* in *OpenBL* e[52], and other library functions [41, 49]. We can simply convert these library functions into APIs provided by CIM accelerators. Therefore, we can directly recognize this accelerable pattern by hooking these function calls. Pattern 5 shows the pattern of library functions in the IR. Both compiling and decompiling generate the same name of function calls in the IR. *p_n* denotes the *n*th argument of the library function. Our compilation tool translates those accelerable library functions into predefined CIM APIs according to the function's name and arguments.

4.5 Bitmap Logical Operations

Besides arithmetic operations, a number of studies have demonstrated that ReRAM can perform logical operations efficiently [12, 31, 32]. Particularly, ReRAM crossbar arrays are well-suited to accelerating Boolean logical operations, which are commonly used for data querying in databases. As described in Section 2.1, a quarry in a data table can be converted into a Boolean logical operation for two bitmaps [35]. In a ReRAM crossbar array, this bitwise operation can be performed in parallel with only one step.

Pattern 6 shows the IR pattern of Boolean logical operations for two bitmaps generated from source codes. %val_# is an element of a bitmap, and *logic_op* represents a logical operator, such as `OR`, `AND`. In traditional computing model without using SIMD instructions, the Boolean logical operation for two bitmaps is often performed sequentially in a single-level loop. For the IR decompiled from a binary executable, the pattern of Boolean logical operation shows the same loop structure and logical operator as that in Pattern 6. The only difference is the instructions for data addressing, which do not affect the pattern recognition. Thus, to recognize this accelerable pattern, our tool only needs to identify the single-level loop structure and the Boolean logical operator.

Pattern 6: A pattern of Boolean logical operations in the IR compiled from source codes

```
loop.condA: ...
loop.bodyB:
    %val_1 = load <ty>, <ty>* %arrayidx_1
    %val_2 = load <ty>, <ty>* %arrayidx_2
    %logic = logic_op <ty> %val_1, %val_2
```

4.6 Performance Model

Although we can recognize four typical accelerable patterns, we still have to check whether these patterns can be really accelerated by ReRAM crossbar arrays, because the operation of data mapping to ReRAM is usually costly and may offset the performance gain of CIM accelerators. For example, only when the size of matrix is large enough, an MVM operation can be accelerated by ReRAM crossbar arrays [9, 38]. Thus, we propose performance models to estimate the net benefit of computation offloading based on the data size of matrices and vectors [38].

We note that the capacity of each ReRAM cell, the size of each crossbar array, the total number of crossbar arrays, and various approaches of titling/blocking matrices/vectors all affect the latency of ReRAM writes and computation. For example, since each ReRAM cell can only store limited bits, a value is usually split into multiple bits and mapped to multiple columns in ReRAM crossbar arrays. This affects the number of writes to ReRAM crossbars and the number of crossbar computations, and also induces additional shift-and-add latency. Moreover, a large matrix can be partitioned into multiple sub-matrices and mapped into multiple crossbar arrays simultaneously. Although different rows in a single sub-matrix should be written to a crossbar array sequentially, the rows of different sub-matrices can be written to multiple crossbar arrays concurrently. The size of the CPU cache, the cache hierarchy, and the size of accelerator buffer also have a significant impact on the latency of loading a large matrix. To simplify the performance model, we assume that the size of the accelerator buffer is the same as that of the CPU cache, and thus the time spent in loading a matrix to the CPU cache and to the buffer of the ReRAM-based accelerator are similar. In this way, we do not consider these latencies of data loading in the performance model, because they can be counteracted.

Taking the MVM operation as an example, Equation (1) shows the performance model. The MVM operation can be offloaded to ReRAM crossbar arrays only when the net benefit $Benefit_{MVM}$ is positive. T_{CPU} and T_{ReRAM} represent the total execution times of an MVM operation on CPU and ReRAM, respectively. T_{ReRAM} mainly consists of the time for mapping an matrix to ReRAM crossbar arrays ($L_{DataMap}$), and the execution time of an MVM operation on ReRAM crossbar arrays (including analog computing and ADC operations, i.e., $L_{com\&ADC}$):

$$\begin{aligned} Benefit_{MVM} &= T_{CPU} - T_{ReRAM} \\ &= T_{CPU} - (L_{DataMap} + L_{com\&ADC}) \\ &= m \times n \times L_{mul} + (m-1) \times n \times L_{add} - (r \times L_{write-a-row} + L_{com} + k \times L_{ADC}). \end{aligned} \tag{1}$$

Assume the size of a crossbar array is $r \times r$, an $m \times n$ matrix (m rows and n columns) should be mapped into $\lceil m/r \rceil$ ReRAM crossbar arrays. We can write a vector to one row of the crossbar array at a time, and the write latency ($L_{write-a-row}$) can be deemed as a constant. Since we can map $\lceil m/r \rceil$ sub-matrices to $\lceil m/r \rceil$ ReRAM crossbar arrays simultaneously, total r rows in the whole matrix should be written to crossbar arrays sequentially, and the latency of data mapping becomes $r \times L_{write-a-row}$. For matrices with different sizes, the latency of analog MVM operation in the ReRAM array (L_{com}) is usually the same (O(1) complexity), and is much lower than the latency

of data mapping. After the analog MVM operation is completed, the analog values in the output vector should be converted into digital values through ADCs. Since the die area of an ADC device is often too large relative to the ReRAM array, we assume k columns share one ADC to convert the elements of the output vector sequentially, and the latency of ADC is L_{ADC}. Thus, the total latency of ADC operations becomes $k \times L_{ADC}$, which is mainly determined by the number of columns in the matrix.

For the $m \times n$ matrix, total $m \times n$ multiplications and $(m-1) \times n$ additions are required for an MVM operation on CPU, respectively. Assume L_{add} and L_{mul} represent the clock cycles required for each addition and multiplication instruction [18], respectively. The total execution time of an MVM on CPU T_{CPU} can be estimated by $m \times n \times L_{mul} + (m-1) \times n \times L_{add}$.

Since $L_{mul}, L_{add}, L_{write-a-row}, L_{com}, L_{ADC}$ can be all deemed as constants, the benefit of MVM offloading is mainly determined by the matrix size (i.e., m, n). Thus, we can exploit the simple performance model in Equation (1) to determine whether an MVM operation can be really accelerated by the CIM accelerator.

As a crossbar array can only perform an MVM operation or multiple MAD operations for vectors, it cannot perform MMM operations directly. Fortunately, we can decompose an MMM into multiple MVM operations. Assume the size of two matrices are $l \times m$ and $m \times n$, this MMM can be decomposed to l MVM operations. According to Equation (1), we can simply estimate the benefit of offloading an MMM operation to the CIM accelerator, i.e.,

$$Benefit_{MMM} = l \times Benefit_{MVM}. \tag{2}$$

We also offer a simple performance model for bitmap-based bitwise operations. Assume a bitmap contains s bits, the size of a crossbar array is $r \times r$, and there are total t crossbar arrays in a CIM accelerator. Thus, we can concurrently write total $t \times r$ bits to t crossbar arrays at a time. The total latency of ReRAM writes for the bitmap is $\frac{s}{t \times r} L_{write-a-row}$. We note that all kinds of logical operations can be converted into a series of NOR operations. The latency of bitwise operation ($L_{bitwise}$) for two rows in a crossbar array can be deemed as a constant. At the host side, we assume the CPU uses a 256-bit SIMD unit with SSE/AVX for bitwise operation acceleration, and the SIMD latency is L_{SIMD}. We can model the benefit of offloading a bitmap-based bitwise operation in Equation (3):

$$Benefit_{bitwise} = T_{CPU} - T_{ReRAM} = \frac{s}{256} \times L_{SIMD} - \frac{s}{t \times r}(L_{write-a-row} + L_{bitwise}). \tag{3}$$

5 AUTOMATIC CODE CONVERSION IN THE IR

When an accelerable pattern is found, the IR code should be replaced with the corresponding API provided by CIM accelerators. In this section, we describe the details of code conversion, including finding key variables, resolving data dependencies, and API construction and replacement.

5.1 IR Preprocessing

As mentioned in Section 4.1, unlike the compiled IR, blocks in the decompiled IR are often out of order, and they often have a different code layout of the control flow graph relative to the compiled IR code. Since we should find out key variables that indicate the loop structure, we have to reorder all blocks of each function call in the IR to construct a control flow graph for loop blocks.

To reorder blocks and find out key variables at the same time, we design a special data structure to describe blocks in the IR. As shown in Figure 5, a block can be divided into four parts: the block name, predecessors, the block content, and the branch statement. Each block has an unique name. The predecessors indicate which blocks (e.g., `block1` and `block2`) can jump to this block.

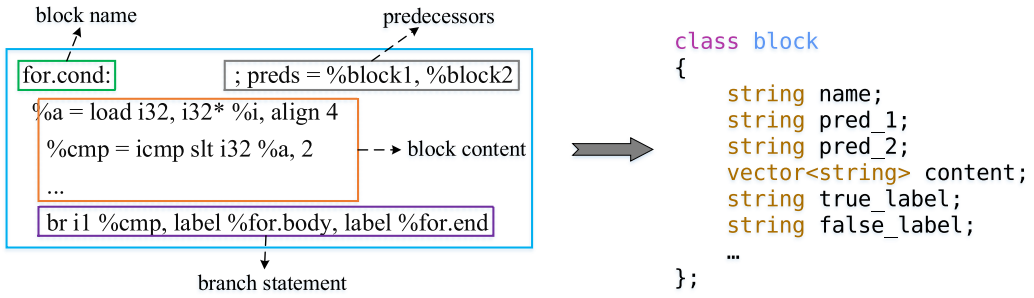


Fig. 5. Self-defined data structure of IR blocks.

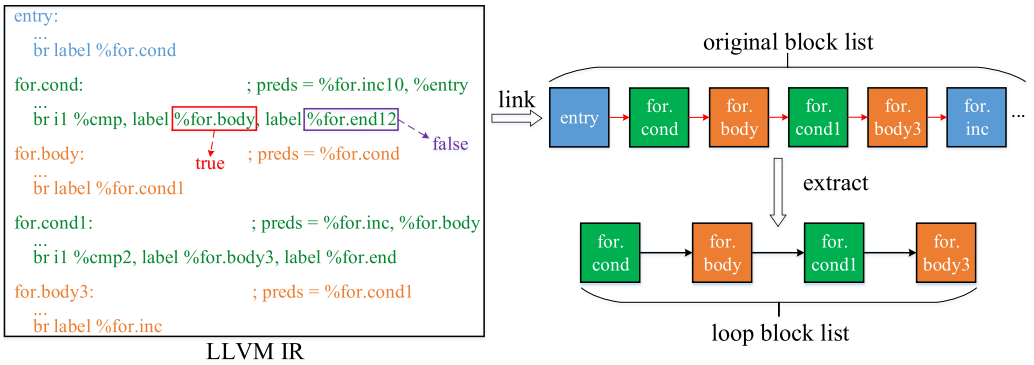


Fig. 6. Extracting key blocks in a single-level loop structure.

The block content contains a majority instructions of a program. The branch statement indicates which blocks (e.g., `for.body` and `for.end` blocks) can be reached from this block.

Figure 6 illustrates an example of extracting loop blocks in a single-level loop structure. We use a list to record all blocks involved in a function. The linked list maintains all blocks of a loop in order assuming all jump conditions are true. As shown in Figure 6, the successor of *for.cond* is *for.body*, while the successor of *for.cond1* is *for.body3*. We first link all condition blocks and body blocks of a loop in a list if all jump conditions are true, and then extract these loop blocks to construct a control flow graph.

5.2 Recognizing Key Variables

When an accelerable pattern is found, we have to find out some key variables within it to complete the code conversion. These variables are related to the interface provided by the CIM accelerator, including the starting address and the size of the matrix or the vector involved in the accelerable pattern. Then, the CRE can read the data from main memory and map it to ReRAM crossbar arrays according to the starting address and the data size. When the CIM accelerator completes the computation, the CRE also writes the output result to main memory.

For matrix-vector multiplications and matrix-matrix multiplications, we can format these operations uniformly as follows:

$$M_1[m \times n] \times M_2[n \times k] = M_3[m \times k]. \tag{4}$$

For an MVM, the variable m is 1, and thus we only need to find two other variables n and k. The MVM is done in a two-level nested loop while the MMM is performed in a three-level nested

loop. For an MMM operation, the variable m can be found in the condition block of the outer loop.

The dimension of a vector (V_{dim}) can be easily found in loop condition blocks. In general, we can perform a head-to-tail traversal or a tail-to-head traversal to infer the dimension of a vector. In the loop control statement, the loop control variable (i) is initialized, tested, and changed when the loop executes. We denote its initial value as *LoopVarInit* and the range control value as *RangeVar*. All of them appear in the `loop.cond` block definitely.

For the head-to-tail traversal, the initial value of i (*LoopVarInit*) is usually 0, and should be *no more* than *RangeVar*. When the comparison symbol is $<$ or $\leq$, the dimension of the vectors should be $RangeVar$ or $RangeVar + 1$, respectively. For the tail-to-head traversal, the initial value of *LoopVarInit* is usually larger than or equal to the *RangeVar*. Thus, the dimension of the vector becomes $LoopVarInit + 1$ regardless of the comparison symbol ($>$ or $\geq$).

Fortunately, the the loop condition block can be easily recognized in the IR. Although the IR can be generated from source codes or a binary executable, the IR blocks corresponding to the loop control statement are almost the same. By searching the *"icmp"* instruction, we can recognize the jump condition and the *RangeVar*. For example, in Pattern 1, *"<cmp_1>"* in the block `loop.condA` represents the comparison statement, and [*v_1*] represents the *RangeVar*. For *LoopVarInit*, we can find it in the statement containing a *load* instruction, which use *LoopVarInit* to initialize the *i*, i.e., [*r_1*] represents *LoopVarInit*. Generally, the dimension calculated in the first loop condition block (e.g., `loop.condA` in Pattern 1) corresponds to the size of the input vector, while the dimension calculated in the second loop condition block (e.g., `loop.condC` in Pattern 1) corresponds to the number of columns or rows in the matrix.

To complete an MVM, the element of the input vector and a row of the matrix should be accessed alternately. For convenience, we use the starting address of the vector/matrix as its identifier. For a single element of a vector, its address can be represented by *a[i]* and calculated by $starting\_add + offset$, where *starting_add* is the starting address of the vector *a*, and *offset* equals to the loop control variable (i) multiplied by the size of data.

As shown in Pattern 2, the variable *%input_addr* in the line 15 represents the starting address of the input vector, and *%a_1* represents the offset. However, the position of these two variables may be not fixed in the IR. Fortunately, we find that *%input_addr* only appears in the innermost loop body block, while the loop control variable *%a_1* is also involved in the loop control block. Thus, we can still distinguish these two variables by tracking their positions in different blocks. Due to the ***static single assignment form* (SSA)** property of the IR, a variable may correspond to several variants with different names, but they have the same value. Thus, we construct an one-to-many mapping between the original loop control variable and its variants, and associate the starting address of the vector with its dimension, which corresponds to the loop control variable.

For each element in a matrix, e.g., *m[i][j]*, its address can be also calculated by $starting\_add + offset$. However, unlike the offset in an one-dimensional vector, an additional add operation is required to calculate the offset here, i.e., $addr = first\_addr + i \times col\_size \times data\_size + j \times data\_size$. Since only two variables can be added in each addition operation, total two *add* instructions are required, as shown in the *block_D* of Pattern 2. Therefore, to find the starting address and the dimension of the matrix, we also need to identify and distinguish these three variables from each other by checking whether the variable appears in the loop control block. Since the loop control variable (i) corresponding to the rows of the matrix is involved in one more multiplication than the loop control variable (j), it is possible to distinguish these two variables that correspond to the rows and columns of a matrix. Once the dimensions of the input vector and the matrix are determined, we can easily deduce the dimension of the output vector. However, its starting address still should be determined in the same way as the above.

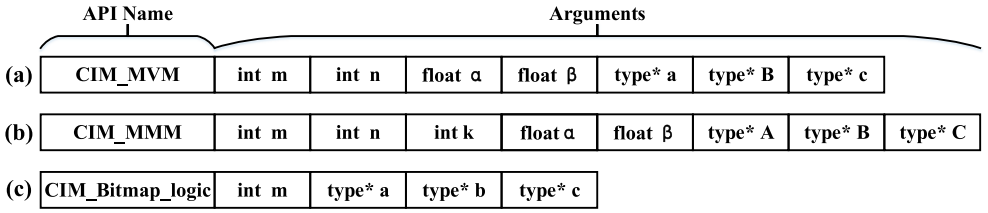


Fig. 7. APIs designed for an ReRAM-based CIM accelerator [38].

To support general GEMV and GEMM, we need to further identify coefficients before matrices and vectors, such $\alpha \times a[m] \times B[m \times n] + \beta \times c[n]$. The β appears in the penultimate loop body block and multiplies each element of the output vector, and the α appears in the innermost loop body block and multiplies the result of the MAC.

For a bitmap logical operation, the variable of two bitmaps can be easily found from the loop body of a single-level loop, as shown in Pattern 6. Then, we can infer the dimension of the bitmap from the loop condition block (i.e., the *RangeVar*).

In summary, we can recognize the dimensions of the input vector and the matrix from the loop control block, and recognize their starting addresses from the innermost loop body block. Moreover, the whole loop structure is also used to establish the mapping between the starting address and the dimension of the input vector or the matrix. Although these variables can be easily identified from the IR code, the dimension of matrix or vector is usually only known at runtime. Thus, we still need code instrumentation techniques to track these variables at runtime, and then determine whether the MVM/MMM operation should be run on the CIM accelerator.

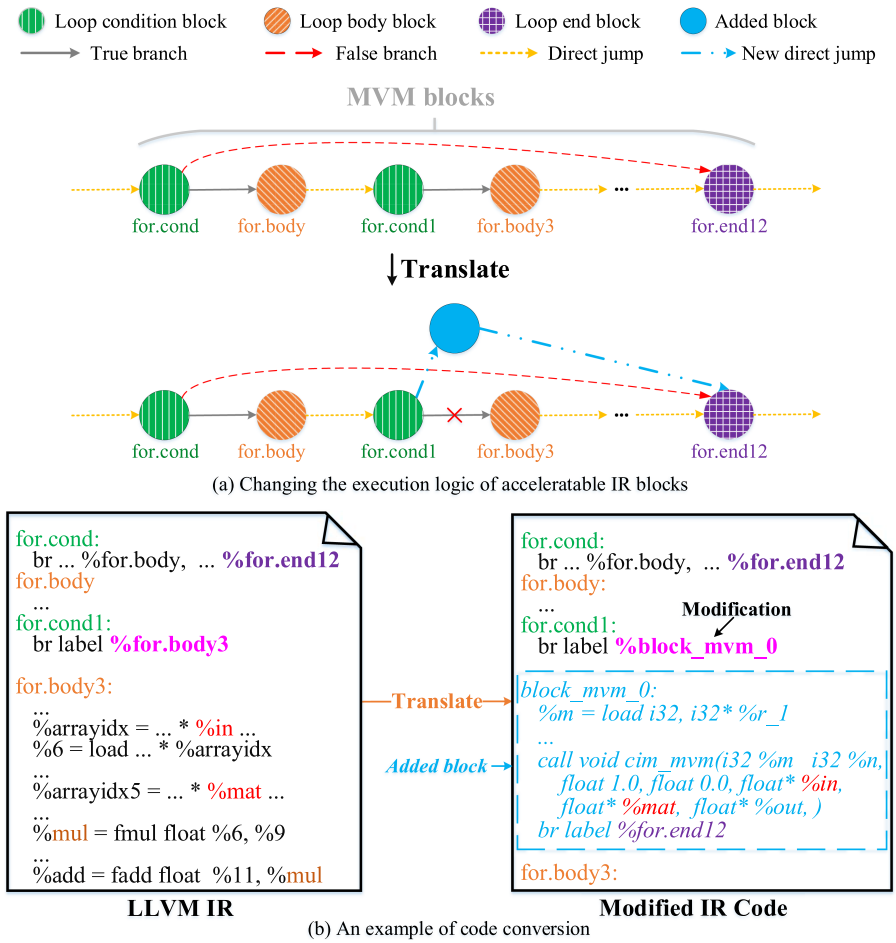
5.3 Constructing CIM Function Calls

We provide three APIs for a typical ReRAM-based CIM simulator [38], as shown in Figure 7. `CIM_MVM` performs $\alpha \times a[m] \times B[m \times n] + \beta \times c[n]$, where α and β are coefficients, and m and n specify the numbers of rows and columns in the matrix, respectively. Also, m and n reflect the sizes of the input vector and the output vector, respectively. The symbols a, B, and c represent the starting address of the input vector, the matrix, and the output vector, respectively. The notation *type* represents the type of data in the MVM operation. Similarly, `CIM_MMM` performs $\alpha \times A[m \times n] \times B[n \times k] + \beta \times C[m \times k]$, where m, n, and k represent the numbers of rows and columns in the input/output matrices. α and β are also coefficients. A, B, and C represent the starting address of the matrices. `CIM_Bitmap_logic` performs $a[m] \otimes b[m] = c[m]$, where $\otimes$ represents Boolean logic operators such as $\vee$ and $\wedge$, and a, b, and c represent the starting address of the three vectors.

When the sizes of matrices and input vectors are determined, we can construct new function calls according to these APIs defined for CIM accelerators, and then replace the original accelerable patterns in the IR. Moreover, we also need to add the declaration of the newly inserted functions in the IR file.

5.4 Code Instrumentation

To accelerate IR patterns described in Section 4, we need to replace these code snippets with newly generated CIM function calls. For library functions of MVM and MMM, we can simply replace the original function call. However, we can not do this for other accelerable patterns. Since the LLVM IR is based on the SSA form, some variables defined in an IR block may be referenced by instructions in other blocks. Therefore, if we directly delete the accelerable IR blocks, then the program may not be compiled or gets incorrect results. In this case, to guarantee the correctness



(a) Changing the execution logic of acceleratable IR blocks

(b) An example of code conversion

Fig. 8. Example of code instrumentation in RCCT.

of the program, we have to modify all instructions related to the deleted variables. This is a time-consuming work and is difficult for automatic code conversion in the IR.

Our compilation tool replaces the accelerable IR blocks using a more smart approach. Figure 8 illustrates an example of code conversion in the IR. We construct and add a new block, and change the original execution logic seamlessly. We do not delete any instructions in the accelerable IR blocks so that the following involved instructions can still execute without errors. However, we change the execution flow to skip the original accelerable block (i.e., *for.body3*), and jump to the newly added block (i.e., *block_mvm_0*) in which the MVM operation is offloaded to the CIM accelerator. We place the newly added block just before the innermost loop body block (i.e., *for.body3*), because the loop control variables have been defined and initialized at this time. Thus, we can fully utilize these original variables to determine the arguments of the *cim_mvm* function. When the function call returns, it directly jumps to the *for. end12* block and the loop is finished.

Since there may be more than one accelerable pattern of the same type in a single function, we give each newly added block a unique name to avoid conflicts. Moreover, since some patterns may be not accelerated by the CIM accelerator, we add instructions in the beginning of the newly added block to calculate the net benefit according to the sizes of input vectors and matrices, and

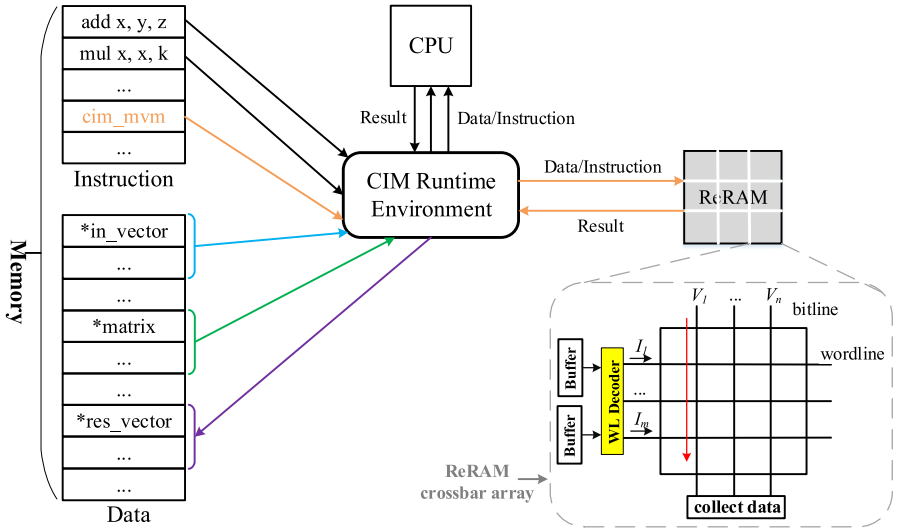


Fig. 9. Execution flow of the new binary executable.

then determine whether the program continues to execute the CIM function call or it jumps back to execute the loop on CPUs. In this way, we can achieve conditional computation offloading based on the performance model.

Limitation. Although our compilation tool can automatically transfer legacy programs to binary executables that can offload some accelerable codes to the CIM accelerator, it still has some limitations. For example, for several application domains such as graph processing and genome analysis, the original algorithm usually needs some additional transformation to transfer to MVM, MMM, or bitmap logical operations. These cases cannot be easily supported by our compilation tool. However, since matrices and vectors are widely used in most deep learning and database applications, our compilation tool naturally support these applications for automatic code migration.

6 EXECUTION OF THE BINARY CODE

The modified IR can be further compiled to a binary executable and then is executed in the CRE. Figure 9 shows the execution flow of the regenerated binary executable. In the beginning, all instructions and data are stored in main memory. When the program begins to run, the CPU is in charge of dispatching instructions. A CIM instruction is offloaded to the ReRAM accelerator, and other instructions are still executed by the CPU. When the CIM instruction is finished, the result is returned to the CRE and finally is sent back to main memory. We use a synchronous execution model to synchronize the computation on CPU and ReRAM. The application calls a CIM API just like a local function and does not continue until the result returns. Thus, there is no data race between the CPU and ReRAM-based CIM accelerator. We note that IR instructions associated with the CIM function should be compiled according to the ISA of ReRAM. In our system, the CIM APIs are simulated by the ReRAM simulator. Here, we only validate the functional correctness of those APIs. The simulator can recognize those CIM functions and simulate the execution of these APIs.

Here, we take the *CIM_MVM* function as an example to illustrate the simulation of CIM APIs. When a CIM instruction is recognized by the CRE, it is offloaded to the ReRAM-based CIM accelerator. According to the arguments of *CIM_MVM*, such as the starting address and the size of the data, the CIM accelerator loads the required data from main memory accordingly. At first, it should load the matrix into the ReRAM crossbar array row by row. Because the write latency of ReRAM is

Table 1. Configuration of the Host CPU and the CIM Accelerator

CIM Simulator Parameter	**Configuration**
Technology (128×128 @16-bit), 128 arrays per tile	ReRAM 8× (128×128 @2-bit)
Compute and Write latency/32-bit	1.8 μs and 1.0 μs
Load/Store Latency (memory-to-buffer)	0.1 μs
Compute Energy/32-bit	100 fJ
Write Energy/32-bit	200 pJ
Host CPU Specification	**Configuration**
Intel-Xeon @2.3 GHz	20 Cores
Cache	L1-d/L1-i 32KB, L2 256KB, L3 25MB
Memory	64 GB DDR4
CPU Power (TDP)	105 W

rather high, we use two input buffers to hide the latency of loading data from main memory. The two input buffers can be read and written alternately so that one buffer can be used to store the data from main memory while another buffer can be used to write the data to the ReRAM array.

We note that a large matrix may be not perfectly fit for the size of crossbar arrays. They should be partitioned into multiple sub-matrices and mapped to different crossbar arrays. The CRE performs the titling/blocking of matrices/vectors explicitly to fit the available capacity of crossbar arrays. When the matrix has been mapped, the CRE loads the input vector from main memory to an input buffer, and then the digital-to-analog converter converts the input vector into the input voltage. The MVM operation can be completed in only one step. At last, the peripheral devices (ADC, S&A) are responsible for collecting and converting the output result to digital data. Finally, CRE writes back the final result to main memory.

7 SIMULATION AND EVALUATION

In this section, we use the Polybench/C benchmark [42] and other micro-benchmarks to verify the effectiveness of our complication tool–RCCT. We also compare RCCT with the state-of-the-art TDO-CIM [47].

7.1 Experimental Setup

We use a simulation framework MHSim [38] to set up a heterogeneous computing system composed of CPUs and CIM accelerators. We simulate two Intel Xeon processors with total 20 cores based on a cycle-accurate simulator ZSim [43]. The ReRAM accelerator is simulated by NeuroSim [16] using a circuit-level macro model. Since Zsim can track specific instructions for function calls, we can directly offload library functions to ReRAM crossbar arrays.

Table 1 summarizes the configuration of CPUs and CIM accelerators. We use 128×128 ReRAM crossbar arrays with 16-bit data accuracy. Each ReRAM cell can store a 2-bit number [51] with low latency and high accuracy. Thus, 8 adjacent columns are used to represent a 16-bit number according to IEEE-754 half-precision floating-point format. To reduce the die area of ReRAM, 8 columns share a 9-bit ADC to convert the analog value to a digital value. The final result is computed by a weighted sum of 8 columns. For bitwise Boolean logic operations, we refer to a PIM architecture called Pinatubo [35] to simulate its computation latency and energy consumption.

The energy and latency models of the crossbar array and peripheral circuits are based on Neurosim, as shown in Table 1. The simulator also takes into account the latency and energy consumption of data movement between main memory and CIM accelerators. We use *thermal design power* (TDP) to estimate the energy consumption of CPUs [30].

Table 2. Benchmarks

Category	Program	Description	Data Size
Polybench/C	mvt	Matrix-Vector Product and Transpose	4,000×4,000
	3mm	3 Matrix Multiplications (A×B)×(C×D)	1,024×1,024
	gemm	Matrix-multiply (C=α ×A×B+β ×C)	1,024×1,024
	gemver	Vector-multiply and Matrix-Vector Product	1,024×1,024
	gesummv	Scalar, Vector and Matrix Multiplication	1,024×1,024
Cblas	sgemm	Matrix-multiply using multi-threads	1,024×1,024
Bitmap	bitmap_and	logical AND operation for two bitmaps	2^{20} bits
MLP	MLP_sort	handwritten digit classification	MNIST
Cryptography	Mceliece	asymmetric encryption algorithm	1 MB
SPEC CPU 2017 (FP)	povray	Ray tracing with matrix/vector operations	test
	lbm	Fluid dynamics with matrix/vector operations	test
Real Bitwise Operations	BFS [11]	bitmap-based BFS for graph processing	Amazon-2008
	Fastbit [50]	bitmap-based database operations	STAR [6]

We use several benchmarks to evaluate the benefit gained from CIM accelerators, as shown in Table 2. The Polybench/C benchmarks contain different combinations of MVM, MMM, and vector multiplication operations. `Cblas_sgemm` and `Bitmap_and` are particularly developed to verify the effectiveness of accelerating library functions, bitmap logical operations, respectively. `Cblas_sgemm` performs the MMM operation using the CBLAS library. `Bitmap_and` performs the logical "AND" operation for two bitmaps. Apart from the above micro-benchmarks, we also evaluate several real-world macro-benchmarks. `MNIST_mlp_sort` is a handwritten digit classification application based on three-layer ***multi-layer perceptron*** **(MLP)** neural network. `Mceliece_decrypt` is an asymmetric encryption algorithm involving in heavy matrix multiplication operations. `Povray` and `lbm` are selected from SPEC CPU 2017 (floating point) benchmark suite, and they all contain heavy matrix/vector operations. `BFS` [11] and `Fastbit` [50] are bitmap-based graph processing and database applications, respectively. These real-world applications are used to verify the effectiveness of our compilation tool for computation offloading in ReRAM-based CIM architectures.

7.2 Accuracy of RCCT

In this section, we evaluate the recognition accuracy of RCCT and TDO-CIM [47] for both compiled and decompiled IR patterns in different benchmarks. At first, we analyze the source codes of programs in Table 2 manually, and count the total number of four accelerable IR patterns and use it as a baseline. For programs with source codes, we count the total number of accelerable patterns recognized by our compilation tool in the compiled IR, and then calculate the recognition accuracy, i.e., the ratio of recognized patterns in the compiled IR to the total number of accelerable patterns in the program. To evaluate the recognition accuracy of decompiled IR patterns, we first compile the source code into a binary executable, and then decompile this binary executable into the LLVM IR. At last, we calculate the recognition accuracy of decompiled IR patterns using the same way similar to the compiled IR. Since a higher level optimization of the GCC compiler leads to a lower recognition accuracy for the decompiled IR, we adopt the O1 optimization to generate all executable binaries.

Table 3 shows the recognition accuracy of RCCT and TDO-CIM. Both TDO-CIM and RCCT achieve 100% recognition accuracy for compiled IRs of Polybench/C benchmarks, because there are only a few and very simple computation kernels in each program's source code. However,

Table 3. Recognition Accuracy of Accelerable IR Patterns

Benchmarks	Compiled IR		Decompiled IR	
	TDO-CIM	RCCT	TDO-CIM	RCCT
Polybench_mvt	100%	100%	N.A.	100%
Polybench_3mm	100%	100%	N.A.	100%
Polybench_gemm	100%	100%	N.A.	100%
Polybench_gemver	100%	100%	N.A.	100%
Polybench_gesummv	100%	100%	N.A.	100%
Cblas_sgemm	N.A.	100%	N.A.	100%
Bitmap_and	N.A.	100%	N.A.	100%
BFS	N.A.	100%	N.A.	100%
Fastbit	N.A.	100%	N.A.	100%
MNIST_mlp_sort	82.4%	97.8%	N.A.	52.7%
Mceliece_decrypt	87.5%	98.5%	N.A.	68.6%
SPECfp_povary	72.6%	95.7%	N.A.	64.8%
SPECfp_lbm	62.6%	96.4%	N.A.	54.3%

TDO-CIM does not support the recognition of library functions and bitmap logical operations, such as Cblas_sgemm, bitmap_and, bitmap-based BFS and Fastbit, while RCCT can recognize these accelerable patterns with 100% accuracy. For more complex real-world applications such as MNIST_mlp_sort, Mceliece_decrypt, SPECfp_povary, and SPECfp_lbm, TDO-CIM can only achieve 62.6%~87.5% accuracy, because it omits some library functions and bitmap logical operations that can be offloaded to CIM accelerators. RCCT can recognize most accelerable patterns except that MVM or MMM operations are tightly coupled with other complex operations with high data dependency. Since the decompiling technology is not full-blown, the decompiled IR generated by the decompiling tool–McSema [3] is usually significantly different with the compiled IR, increasing the complexity of pattern recognition. Thus, RCCT can only achieve 52.7%~68.6% accuracy for decompiled IR of these real-world applications. Overall, TDO-CIM can only accelerate MVM and MMM operations from the source code, and cannot recognize the accelerable patterns from decompiled LLVM IR of executable binary. In contrast, RCCT is able to accelerate other typical patterns such as library functions and bitmap logic operations from both source codes and binary executables.

7.3 Validation of Performance Models

We also evaluate the effectiveness of the performance model proposed in Section 4.6. We measure the performance speedup of RCCT on MVM operations for different matrix sizes with/without the computation offloading model, as shown in Figure 10. Since all matrices are square arrays, we only use the number of rows to represent their sizes. Without our performance model, we can find that the computation offloading even lead to a performance slowdown for small matrices compared with the CPU-only system. For a larger matrix, our computation offloading model can achieve higher performance speedup.

To verify the effectiveness of our performance models, we evaluate the performance speedup of RCCT (without models), RCCT (with models), and TDO-CIM for both micro and macro benchmarks, as shown in Figure 11. Since the Polybench/C benchmark suite contains only very simple computation kernels such as MVM, MMM, and vector multiplications, and all involved matrices are very large, the CIM accelerators are well-suited to offload these kernels that involve heavy matrix/vector operations. Thus, both TDO-CIM and RCCT achieve similar performance speedup

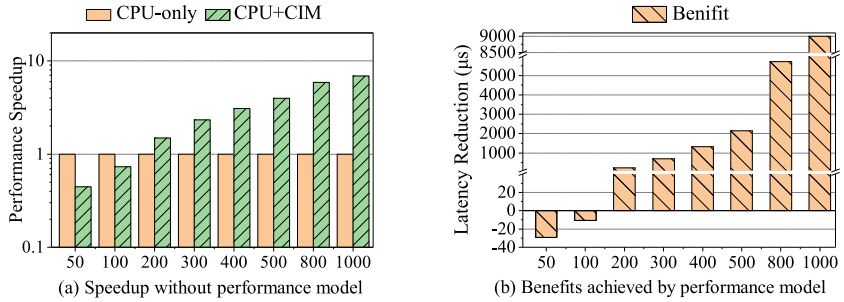


Fig. 10. Performance speedup of MVMs sensitive to the size of MVM.

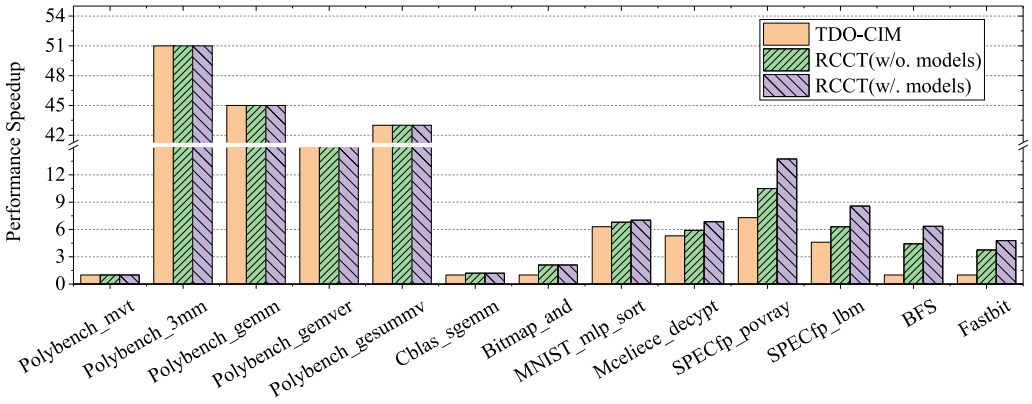


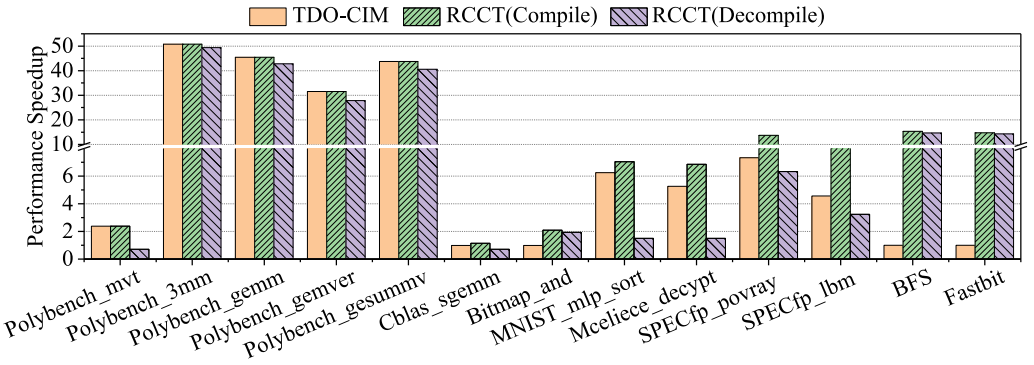
Fig. 11. Performance speedup with/without performance models, all normalized to the CPU-only system.

even without using our performance models. However, for other real-world applications such as SPECfp_povary, and SPECfp_lbm, RCCT with performance models deliver higher-performance speedup, because it only offloads large-scale MVM/MMM operations to the CIM accelerators. Without our performance models, TDO-CIM and RCCT (w/o models) performs computation offloading for all MVM, MMM, and bitmap logical operations indiscriminately, and thus may partially offset the benefit of computation offloading for large-sized matrices/vectors.

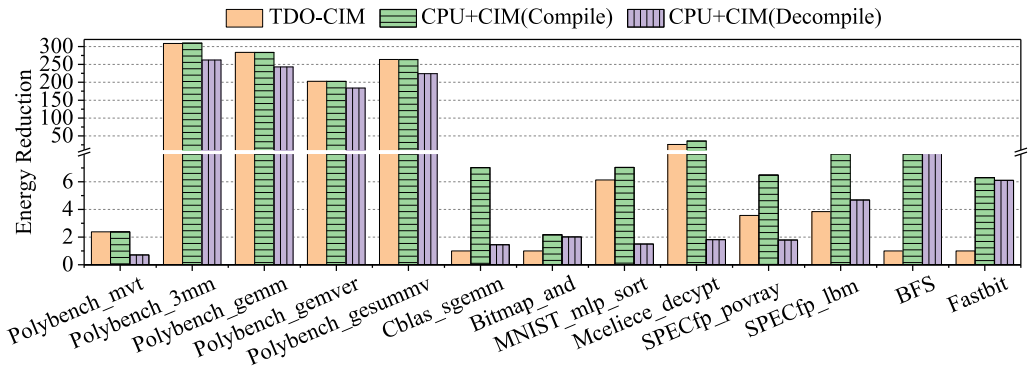
7.4 Performance Evaluation

We port all legacy programs in Table 2 to the simulated heterogeneous computing system using our compilation tool. Both source codes and binary files of these programs are converted into the final binary execuatables that can be run in CIM accelerators. All experimental results are normalized with a baseline in which all original binary files execute in a CPU-only system.

Figure 12(a) shows that the CIM accelerator can improve the application performance by up to 51× compared with the CPU-only system. Polybench_3mm shows the highest speedup, because MMM operations spend the most of its execution time. Moreover, the decompilation case shows similar performance speedup to the compilation case, implying that our compilation tool can effectively identify those MMM operations and offload them to CIM accelerators. The Cblas_sgemm shows the lowest speedup, because Clbas library functions can use multiple threads to take full advantage of the high-performance CPUs. For bitmap-based graph and database applications (BFS and Fastbit), because their codes are manually optimized to fully leverage the bitwise logical APIs of the CIM accelerator, RCCT can achieve similar performance speedup



(a) Performance speedup of CIM accelerators



(b) Energy consumption reduction of CIM accelerators

Fig. 12. Performance speedups and energy efficiency improved by CIM accelerators, all normalized to the CPU-only system.

for both compiled and decompiled IRs. Other programs show moderate performance speedup, because the number of accelerable patterns in those program are very limited. We also find that the performance speedup of the decompiled IR is not as high as the compiled IR. The reason mainly stems from two-folds. First, the reverse engineering tool—McSema [3]—can significantly increase the number of instructions in the decompiled IR by several times and thus offset the benefit from the CIM accelerator. Second, the recognition accuracy of accelerable patterns in the decompiled IR also becomes low, especially for real-world applications, and thus cannot offload sufficient accelerable patterns to the CIM accelerator. We believe our approach can still achieve sufficient performance improvement when the reverse engineering tool is fully developed and optimized [2].

Figure 12(b) shows the energy efficiency of the CIM accelerator can be improved by up to 87× on average compared with the CPU-only system. For Polybench_3mm, we can find the energy efficiency is significantly improved by 309 times, because the ReRAM-based CIM accelerator is particular energy efficient for massive MMM operations. Overall, the energy efficiency shows a similar trend relative to the performance speedup. The programs converted from executable binary codes generally show lower energy efficiency than programs generated from source codes, because the reverse engineering using McSema [3] significantly increases the number of instructions in the decompiled IR and degrades the recognition accuracy of accelerable patterns, and thus it degrades the energy efficiency.

8 RELATED WORK

There have been only a few studies on the compilation tool for ReRAM-based CIM accelerators. TDO-CIM [47] uses the polyhedron tool to identify MVM and MMM operations from compiled LLVM IRs, and offloads them to CIM accelerators. However, it cannot handle the IR generated from decompilation. It also does not recognize/offload bitmap logical operations and library functions. Chakraborty et al. propose a similar work based on LLVM IR to migrate legacy programs to CIM accelerators [13]. They transform accelerable computation kernels into a ***Boolean Decision Diagram*** **(BDD)**, and then map the BDD to ReRAM crossbar array. However, the supported kernels are simple linear operations and can only be written in C language. Also, it cannot support nonlinear arithmetic operations such as multiplication and division. Ambrosi et al., implement an end-to-end software stack for a ReRAM-based accelerator [8]. The software stack consists of interpreter, compiler, and driver, etc. The software stack interprets and compiles computation models of neural network applications into a customized ReRAM instruction set. However, it can only be used for applications developed with neural network frameworks such as TensorFlow. Moreover, it only supports neural network inference except neural network training. In contrast, RCCT does not have the above constraints. It supports a wide range of applications no matter the source codes are available or not, and can recognize more computing patterns to fully exploit the advantages of CIM accelerators.

There have been also a few proposals focusing on the programming interface for CIM accelerators. PRIME [17] is a ReRAM-based crossbar architecture designed for accelerating neural network inference applications. It provides software/hardware APIs for developers to implement various neural network applications on CIM accelerators. PUMA [10] is also a ReRAM-based accelerator designed for accelerating neural network inference applications. It provides a customized ISA, an optimized compiler and high-level APIs for programmers to develop neural network applications on the PUMA accelerator. However, PUMA does not support the migration of existing legacy applications to the PUMA accelerator. Pinatubo [35] is a ReRAM-based crossbar architecture. It exploits sensor amplifiers for efficient bitmap Boolean logic operations, and also provides high-level APIs for developers. However, since it only supports bitmap logic operations, its application scenario is very limited. Yu et al., design a domain specific language and APIs for ReRAM-based accelerators [54], and offer a special compiler to translate computing patterns into highly optimized CIM-executable circuits. However, the language and compiler cannot be applied to migrate legacy applications, and developers still need to write codes from scratch to accelerate these specific computing patterns. Unlike these proposals, RCCT is a complication tool to automatically translate various legacy applications into CIM accelerable binary executables. RCCT is complementary to these existing CIM programming frameworks, and can efficiently migrate various legacy applications to these CIM accelerators using their specialized APIs.

Pattern recognition has been comprehensively studied for decades. There have many proposals for code intelligence using machine learning techniques, e.g., detecting parallel patterns of multi-threaded applications [19], binary code matching [23]. There are also many other studies on retrieving design patterns with programs in embedded multicore systems and energy-efficient scenarios [36]. Recently, there have been many studies on computation offloading for CMOS-based PIM accelerators [7, 15, 20, 48]. Although these schemes are not proposed for ReRAM-based PIM architectures, they offer other angles to model computation offloading in heterogeneous computing architectures.

9 CONCLUSION

In this article, we first formulate four computing patterns that can be accelerated by ReRAM crossbar arrays from the perspective of LLVM IR. We also propose a compilation tool to automat-

ically identify and translate these patterns in the form of LLVM IR into special APIs that can be accelerated by ReRAM, without modifying the source codes. Moreover, our compilation tool also support the migration of legacy programs to CIM accelerators when the programs' source codes are not available. Experimental result shows that our compilation tool can effectively convert legacy programs to CIM-accelerable binary executables, and the performance of programs can be significantly improved by 10× on average, and energy consumption is reduced by 27× on average.

REFERENCES

[1] Dagger. 2018. A Binary Translator to LLVM IR. Retrieved from https://github.com/repzret/dagger

[2] Anvill. 2022. A Reverse Engineering Tool. Retrieved from https://github.com/lifting-bits/anvill

[3] McSema. 2022. A Framework for Lifting Binaries to LLVM Bitcode. Retrieved from https://github.com/lifting-bits/mcsema

[4] Reopt. 2022. A Tool for Analyzing X86-64 Binaries. Retrieved from https://github.com/GaloisInc/reopt

[5] RetDec. 2022. A Retargetable Machine-code Decompiler Based on LLVM. Retrieved from https://github.com/avast/retdec

[6] STAR. 2023. The STAR Experiment. Retrieved from https://www.star.bnl.gov/

[7] Hameeza Ahmed, Paulo C. Santos, João P. C. Lima, Rafael F. Moura, Marco A. Z. Alves, Antônio C. S. Beck, and Luigi Carro. 2019. A compiler for automatic selection of suitable processing-in-memory instructions. In *Proceedings of Design, Automation, and Test in Europe Conference and Exhibition*. 564–569.

[8] Joao Ambrosi, Aayush Ankit, Rodrigo Antunes, Sai Rahul Chalamalasetti, Soumitra Chatterjee, Izzat El Hajj, Guilherme Fachini, Paolo Faraboschi, Martin Foltin, Sitao Huang, Wen-Mei Hwu, Gustavo Knuppe, Sunil Vishwanathpur Lakshminarasimha, Dejan Milojicic, Mohan Parthasarathy, Filipe Ribeiro, Lucas Rosa, Kaushik Roy, Plinio Silveira, and John Paul Strachan. 2018. Hardware-software co-design for an analog-digital accelerator for machine learning. In *Proceedings of the IEEE International Conference on Rebooting Computing*. 1–13.

[9] Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Sapan Agarwal, Matthew Marinella, Martin Foltin, John Paul Strachan, Dejan Milojicic, Wen-Mei Hwu, and Kaushik Roy. 2020. PANTHER: A programmable architecture for neural network training harnessing energy-efficient ReRAM. *IEEE Trans. Comput.* 69, 8 (2020), 1128–1142.

[10] Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Geoffrey Ndu, Martin Foltin, R. Stanley Williams, Paolo Faraboschi, Wen-mei W. Hwu, John Paul Strachan, Kaushik Roy, and Dejan S. Milojicic. 2019. PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. 715–731.

[11] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage, and Analysis (SC'12)*. 1–10.

[12] Debjyoti Bhattacharjee, Rajeswari Devadoss, and Anupam Chattopadhyay. 2017. ReVAMP: ReRAM based VLIW architecture for in-memory computing. In *Proceedings of Design, Automation, and Test in Europe Conference and Exhibition*. 782–787.

[13] Dwaipayan Chakraborty, Sunny Raj, Julio Cesar Gutierrez, Troyle Thomas, and Sumit Kumar Jha. 2017. In-memory execution of compute kernels using flow-based memristive crossbar computing. In *Proceedings of the IEEE International Conference on Rebooting Computing*. 1–6.

[14] Liang Chang, Chenglong Li, Zhaomin Zhang, Jianbiao Xiao, Qingsong Liu, Zhen Zhu, Weihang Li, Zixuan Zhu, Siqi Yang, and Jun Zhou. 2021. Energy-efficient computing-in-memory architecture for AI processor: Device, circuit, architecture perspective. *Sci. China Info. Sci.* 64, 6 (2021), 1–15.

[15] Dan Chen, Hai Jin, Long Zheng, Yu Huang, Pengcheng Yao, Chuangyi Gui, Qinggang Wang, Haifeng Liu, Haiheng He, Xiaofei Liao, and Ran Zheng. 2022. A general offloading approach for near-DRAM processing-in-memory architectures. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*. 246–257.

[16] Pai-Yu Chen, Xiaochen Peng, and Shimeng Yu. 2018. NeuroSim: A circuit-level macro model for benchmarking neuro-inspired architectures in online learning. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 37, 12 (2018), 3067–3080.

[17] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture*. 27–39.

[18] Intel Corporation. 2009. Intel 64 and IA-32 architectures optimization reference manual. Retrieved from https://cdrdv2.intel.com/v1/dl/getContent/671488?fileName=248966-046A-software-optimization-manual.pdf

[19] Etem Deniz and Alper Sen. 2016. Using machine learning techniques to detect parallel patterns of multi-threaded applications. *Int. J. Parallel Program.* 44 (2016), 867–900.

[20] Alexandar Devic, Siddhartha Balakrishna Rai, Anand Sivasubramaniam, Ameen Akel, Sean Eilert, and Justin Eno. 2022. To PIM or not for emerging general purpose processing in DDR memory systems. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 231–244.

[21] Andi Drebes, Lorenzo Chelini, Oleksandr Zinenko, Albert Cohen, Henk Corporaal, Tobias Grosser, Kanishkan Vadivel, and Nicolas Vasilache. 2020. TC-CIM: Empowering tensor comprehensions for computing-in-memory. In *Proceedings of the 10th International Workshop on Polyhedral Compilation Techniques*. 1–12.

[22] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. 2018. In-memory data parallel processor. *SIGPLAN Not.* 53, 2 (Mar. 2018), 1–14.

[23] Yi Gui, Yao Wan, Hongyu Zhang, Huifang Huang, Yulei Sui, Guandong Xu, Zhiyuan Shao, and Hai Jin. 2022. Cross-language binary-source code matching with intermediate representations. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering*. 601–612.

[24] Said Hamdioui, Hoang Anh Du Nguyen, Mottaqiallah Taouil, Abu Sebastian, Manuel Le Gallo, Sandeep Pande, Siebren Schaafsma, Francky Catthoor, Shidhartha Das, Fernando G. Redondo, G. Karunaratne, Abbas Rahimi, and Luca Benini. 2019. Applications of computation-in-memory architectures based on memristive devices. In *Proceedings of the Design, Automation, and Test in Europe Conference and Exhibition*. 486–491.

[25] Said Hamdioui, Lei Xie, Hoang Anh Du Nguyen, Mottaqiallah Taouil, Koen Bertels, Henk Corporaal, Hailong Jiao, Francky Catthoor, Dirk Wouters, Linn Eike, and Jan van Lunteren. 2015. Memristor based computation-in-memory architecture for data-intensive applications. In *Proceedings of the Design, Automation, and Test in Europe Conference and Exhibition*. 1718–1725.

[26] Adib Haron, Jintao Yu, Razvan Nane, Mottaqiallah Taouil, Said Hamdioui, and Koen Bertels. 2016. Parallel matrix multiplication on memristor-based computation-in-memory architecture. In *Proceedings of the International Conference on High Performance Computing Simulation*. 759–766.

[27] Miao Hu, Hai Li, Qing Wu, and Garrett S. Rose. 2012. Hardware realization of BSB recall function using memristor crossbar arrays. In *Proceedings of the 49th Annual Design Automation Conference*. 498–503.

[28] Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. 2019. FloatPIM: In-memory acceleration of deep neural network training with high precision. In *Proceedings of the ACM/IEEE 46th Annual International Symposium on Computer Architecture*. 802–815.

[29] Mohsen Imani, Saransh Gupta, Sahil Sharma, and Tajana Simunic Rosing. 2019. NVQuery: Efficient query processing in nonvolatile memory. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 38, 4 (2019), 628–639.

[30] Intel. 2014. Intel Xeon Processor E5-2650 v3. Retrieved from https://ark.intel.com/content/www/us/en/ark/products/81705/intel-xeon-processor-e5-2650-v3-25m-cache-2-30-ghz.html

[31] Hai Jin, Cong Liu, Haikun Liu, Ruikun Luo, Jiahong Xu, Fubing Mao, and Xiaofei Liao. 2021. ReHy: A ReRAM-based digital/analog hybrid PIM architecture for accelerating CNN training. *IEEE Trans. Parallel Distrib. Syst.* 33, 11 (2021), 2872–2884.

[32] Shahar Kvatinsky, Dmitry Belousov, Slavik Liman, Guy Satat, Nimrod Wald, Eby G. Friedman, Avinoam Kolodny, and Uri C. Weiser. 2014. MAGIC–Memristor-Aided logic. *IEEE Trans. Circ. Syst. II: Express Briefs* 61, 11 (2014), 895–899.

[33] Boxun Li, Yi Shan, Miao Hu, Yu Wang, Yiran Chen, and Huazhong Yang. 2013. Memristor-based approximated computation. In *Proceedings of the International Symposium on Low Power Electronics and Design*. 242–247.

[34] Huize Li, Hai Jin, Long Zheng, Yu Huang, and Xiaofei Liao. 2023. ReCSA: A dedicated sort accelerator using ReRAM-based content addressable memory. *Front. Comput. Sci.* 17, 2 (2023), 1–13.

[35] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. 2016. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *Proceedings of the 53rd ACM/IEEE Design Automation Conference*. 1–6.

[36] Cheng-Yen Lin, Chi-Bang Kuan, Wen-Li Shih, and Jenq Kuen Lee. 2015. Compilers for low power with design patterns on embedded multicore systems. *J. Signal Process. Syst.* 80 (2015), 277–293.

[37] Cong Liu, Haikun Liu, Hai Jin, Xiaofei Liao, Yu Zhang, Zhuohui Duan, Jiahong Xu, and Huize Li. 2022. ReGNN: A ReRAM-based heterogeneous architecture for general graph neural networks. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 469–474.

[38] Haikun Liu, Jiahong Xu, Xiaofei Liao, Hai Jin, Yu Zhang, and Fubing Mao. 2022. A simulation framework for memristor-based heterogeneous computing architectures. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 41, 12 (2022), 5476–5488.

[39] Xiaoxiao Liu, Mengjie Mao, Beiye Liu, Hai Li, Yiran Chen, Boxun Li, Yu Wang, Hao Jiang, Mark Barnell, Qing Wu, and Jianhua Yang. 2015. RENO: A high-efficient reconfigurable neuromorphic computing accelerator design. In *Proceedings of the 52nd ACM/EDAC/IEEE Design Automation Conference*. 1–6.

[40] Dejan Milojicic, Kirk Bresniker, Gary Campbell, Paolo Faraboschi, John Paul Strachan, and Stan Williams. 2018. Computing in-memory, revisited. In *Proceedings of the 38th International Conference on Distributed Computing Systems*. 1300–1309.

[41] Cedric Nugteren. 2018. CLBlast: A tuned OpenCL BLAS library. In *Proceedings of the International Workshop on OpenCL*. 1–10.
[42] Louis-Noël Pouchet and Tomofumi Yuki. 2016. Polybench/C. Retrieved from http://web.cs.ucla.edu/~pouchet/software/polybench/
[43] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*. 475–486.
[44] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture*. 14–26.
[45] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2018. GraphR: Accelerating graph processing using ReRAM. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*. 531–543.
[46] Yuliang Sun, Yu Wang, and Huazhong Yang. 2017. Energy-efficient SQL query exploiting RRAM-based process-in-memory structure. In *Proceedings of the IEEE 6th Non-Volatile Memory Systems and Applications Symposium*. 1–6.
[47] Kanishkan Vadivel, Lorenzo Chelini, Ali BanaGozar, Gagandeep Singh, Stefano Corda, Roel Jordans, and Henk Corporaal. 2020. TDO-CIM: Transparent detection and offloading for computation in-memory. In *Proceedings of the Design, Automation, and Test in Europe Conference and Exhibition*. 1602–1605.
[48] Yizhou Wei, Minxuan Zhou, Sihang Liu, Korakit Seemakhupt, Tajana Rosing, and Samira Khan. 2022. PIMProf: An automated program profiler for processing-in-memory offloading decisions. In *Proceedings of Design, Automation, and Test in Europe Conference and Exhibition*. 855–860.
[49] R. Clint Whaley and Jack J. Dongarra. 1998. Automatically tuned linear algebra software. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. 38–38.
[50] Kesheng Wu. 2005. FastBit: An efficient indexing technology for accelerating data-intensive science. *J. Phys.: Conf. Ser.* 16, 1 (2005), 556–560.
[51] Wei Wu, Huaqiang Wu, Bin Gao, Peng Yao, Xiang Zhang, Xiaochen Peng, Shimeng Yu, and He Qian. 2018. A methodology to improve linearity of analog RRAM for neuromorphic computing. In *Proceedings of the IEEE Symposium on VLSI Technology*. 103–104.
[52] Zhang Xianyi, Wang Qian, and Zaheer Chothia. 2012. OpenBLAS. Retrieved from http://xianyi.github.io/OpenBLAS
[53] S. Bharadwaj Yadavalli and Aaron Smith. 2019. Raising binaries to LLVM IR with MCTOLL (WIP paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. 213–218.
[54] Jintao Yu, Tom Hogervorst, and Razvan Nane. 2017. A domain-specific language and compiler for computation-in-memory skeletons. In *Proceedings of the Great Lakes Symposium on VLSI*. 71–76.
[55] Long Zheng, Jieshan Zhao, Yu Huang, Qinggang Wang, Zhen Zeng, Jingling Xue, Xiaofei Liao, and Hai Jin. 2020. Spara: An energy-efficient ReRAM-based accelerator for sparse graph analytics applications. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*. 696–707.

Received 8 November 2022; revised 3 June 2023; accepted 20 July 2023