TDO-CIM: Transparent Detection and Offloading for Computation In-memory

Kanishkan Vadivel[1]/Lorenzo Chelini[1,2], Ali BanaGozar[1], Gagandeep Singh[1,2], Stefano Corda[1,2], Roel Jordans[1], Henk Corporaal[1]
[1]Eindhoven University of Technology, [2] IBM Research Zürich
{k.vadivel, l.chelini}@tue.nl

***Abstract*—Computation in-memory is a promising non-von Neumann approach aiming at completely diminishing the data transfer to and from the memory subsystem. Although a lot of architectures have been proposed, compiler support for such architectures is still lagging behind. In this paper, we *close* this gap by proposing an end-to-end compilation flow for in-memory computing based on the LLVM compiler infrastructure. Starting from sequential code, our approach automatically detects, optimizes, and offloads kernels suitable for in-memory acceleration. We demonstrate our compiler tool-flow on the PolyBench/C benchmark suite and evaluate the benefits of our proposed in-memory architecture simulated in Gem5 by comparing it with a state-of-the-art von Neumann architecture.**

***Index Terms*—LLVM, compute in memory, memristor, pattern matching, Polly, Loop Tactics**

I. Introduction

As we are moving toward exascale computing, the memory wall [1] is becoming one of the toughest challenges for the traditional von Neumann architecture. Not only does the cost of moving data dwarf the cost of a floating-point operation but also the memory bandwidth is not able to meet the demand of today's applications [2]. Consequently, new architectures with a radical departure from the traditional von Neumann architecture start to arise. Computing in-memory (CIM) is one of them. CIM aims at processing information and storing computation data on the same physical unit using emerging devices referred to as memristive devices. Memristive devices such as phase change memory devices (PCM) can store data within their conductance state, which can be changed by altering the amorphous/crystalline phase within the device [3]. Computation, on the other hand, is carried out through various physical mechanisms such as Ohm's and Kirchhoff's laws. Memristor devices are organized in a computational memory unit—which we refer to as CIM tile. As storing and processing happen in the same physical device, CIM completely diminishes the overhead of data movement between the CPU and the main memory, enabling data-intensive task in an efficient manner [4]. A huge body of research has been done on the architecture side [5]–[7]. However, before CIM can be established as the de-facto solution for HPC and IoT applications a *leap forward need to be done on the compilation toolchain and software stack*, which is the purpose of this paper. Our contributions are:

- An *end-to-end compilation flow* for CIM devices, which allows to *automatically* and *transparently* invoke CIM acceleration, without any user intervention. Therefore, enabling legacy code to exploit in-memory acceleration.
- A *lightweight run-time library* for data allocation, transfer and execution of computational tasks on the CIM device.
- We *evaluate the benefits of CIM computation* in terms of energy and performance by comparing it with a current state-of-the-art von Neumann architecture using the PolyBench/C benchmark suite.

II. Accelerator Organization

In our in-memory accelerator PCM devices are interconnected in a cross-bar like structure (Figure 1 (c)). A matrix can be stored in the crossbar as the conductance state of the PCM devices ($G_{x,y}$ values). Afterward, the input vector is fed as a set of voltages to the crossbar, which multiplies by the conductance values. The resulting current sensed at the columns is the analog dot-product result [8]. The output currents are converted back into digital signals by analog to digital converters (ADC). To further improve the energy efficiency, ADCs are shared amongst multiple columns which are reused using sample and holds (S&H) [9]. In addition, a digital interface is required to hook the CIM tile (Figure 1 (b)) with traditional CMOS logic. The digital interface is composed of row/column buffers, output buffers, and a digital logic block. The row/column buffers act as a data and mask registers for the crossbar [10]. During write operation, the column buffers contain the data that has to be written on the crossbar, and the row buffers supply a row-enable signal. Similarly, during a compute operation, the column buffers supply column-enable signal and the row buffers latch the inputs. The computed result will be stored in the output buffers. The digital logic block implements scalar compute functionality (i.e., reduction functions) to perform post-processing on the crossbar result. A CIM tile, a micro-engine, and a DMA unit for load and store operations make a standalone accelerator.

The accelerator uses a shared global memory interface for data sharing and exposes a set of context registers to the system via a memory-mapped IO interface. Context registers are used for control and offloading, and are read or written by the host. The micro-engine translates the high level-parameters stored in the context registers into a series of circuit-level operations such as loading the data from shared memory to row/column buffers, configuring the mask values, triggering the computation on CIM tile, and writing back the results

978-3-9819263-4-7/DATE20/©2020 EDAA

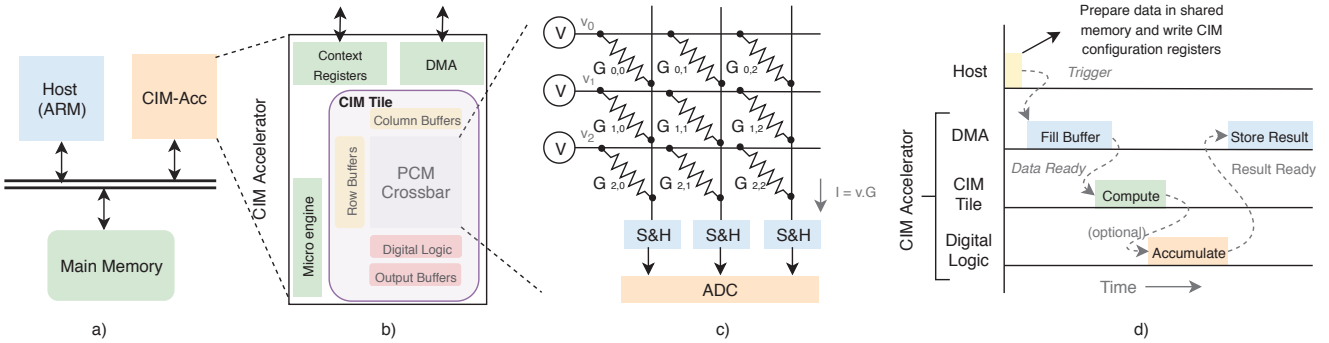

Fig. 1: Overview of the emulated system (a). A more detailed view of our CIM accelerator (b). A memristor-based crossbar (c). Timeline of a kernel execution on our CIM accelerator (d).

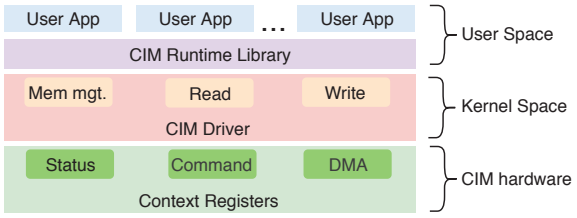

Fig. 2: Overview of the CIM's software stack.

from the output buffers to the shared memory. Additionally, it supports double buffering for all the registers to hide the data latency of the memory accesses. Figure 1 (d) shows a timeline of the events that happen after the host trigger the CIM accelerator. Figure 1 (a) shows our emulated system with a host, main memory, and a CIM accelerator connected through a system bus. We implement the CIM accelerator as a cycle-accurate model integrated into the Gem5 simulator [11]. The accelerator is based on a port-mapped IO (PMIO) and a DMA interface. The PMIO interface exposes context registers to the system, and the DMA provides a memory interface to the accelerator. The host mimics a dual-core Arm-A7 processor based on [12]. For the experiments in the paper, we run the simulator in full-system mode to capture the effects of the operating system, device drivers, and hardware interactions.

A. Software Stack

Our developed software stack (Figure 2) allows applications running in the user space to interact directly with the hardware. The software stack is divided into kernel-space and user-space. At the lowest level of the stack, the kernel-space CIM driver reads and writes to the context registers of the accelerator through a `ioctl` system call. Besides, the driver translates the virtual address used by the host processor to a physical address as the accelerator can work only with physical addresses. On the other hand, the user-space CIM API is responsible for encoding CIM runtime library calls into context register parameters. Furthermore, with the help of the CIM driver, it implements the support for allocating and releasing the physically-contiguous pages in shared memory via the contiguous memory allocator (CMA) APIs exposed by the Linux kernel [13]. The use of CMA offers two main benefits compared to the traditional malloc-based approach: 1) the size of the shared memory region is not limited by the page boundary; 2) there is no need for explicit memory management in the driver routines, which diminishes overhead in the host. To enforce memory coherence in the shared memory region, the kernel driver triggers a cache flush on the host side before invoking the accelerator. The accelerator, on his part, uses only un-cachable requests for memory accesses which automatically enforces memory coherence. Once the accelerator completes its execution, it updates the status in a specific context register. The host can either wait on spin-lock or continue with other tasks and check the status of such register periodically.

III. Overview of the CIM compiler

The high-level design of our compilation flow is shown in Figure 3. It follows a classical compiler design with a front-end, a mid-level optimizer and targets specific back-ends. We extend this flow by introducing 1) Loop Tactics [14], [15]—a state-of-the-art declarative optimizer—in the mid-level. Loop Tactics enables automatic detection and offload of specific computational patterns; 2) A lightweight runtime library that provides optimized performance and memory usage for the CIM device. The library has been designed to be used by the application programmer, or an optimizer (i.e., Loop Tactics). It exposes a host-callable C API, similar to what cuBLAS or MKL offers for Nvidia GPU and Intel CPU, respectively.

A. A Bird's-eye View of TDO-CIM

The entry point in our compilation flow is an application written in a high-level language (i.e., C++). To handle a variety of languages front-ends lower the high-level language to an intermediate representation (IR) on which all the subsequent optimizations are spelled. For our work we use the LLVM compiler infrastructure, hence adopting its intermediate representation LLVM-IR. Given an application, we can use any of the LLVM-based front-ends (i.e., Clang) to lower the high-level language to LLVM-IR. At LLVM-IR level we rely on the polyhedral optimizer Polly [16] to detect, extract and model compute kernels. Internally Polly represents the schedule of each detected kernel as a tree, which we refer to as schedule tree. Schedule tree [17] is the way of representing the execution strategy of each kernel by

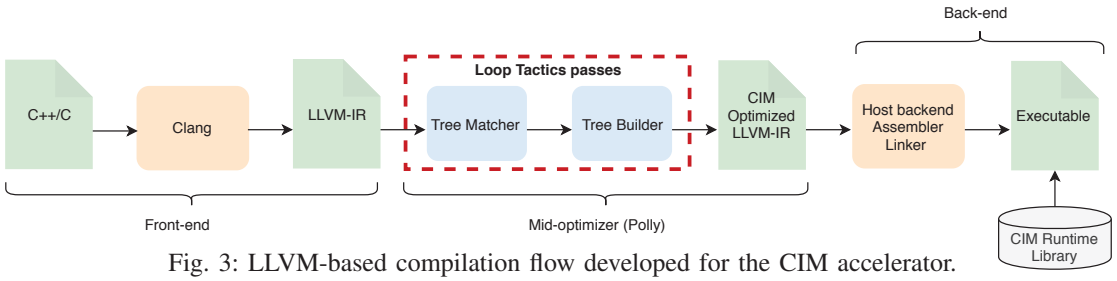

Fig. 3: LLVM-based compilation flow developed for the CIM accelerator.

```
// initialize CIM device 0
polly_cimInit(0);
// allocate data on CIM device
polly_cimMalloc((void**)&C, M*N*4);
polly_cimMalloc((void**)&A, M*K*4);
polly_cimMalloc((void**)&B, K*N*4);
// execute GEMM kernel on CIM device
polly_cimBlasSGemm(transA, transB, M, N, K,
  &alpha, A, lda, B, ldb, &beta, C, ldc);
// copy C back to host
polly_cimDevToHost(C, host_C, M*N*4);
```

Listing 1: High-level code for a generalize matrix multiplication (GEMM) kernel (top). Loop Tactics generated code to offload GEMM kernel to the CIM accelerator (bottom).

mapping each dynamic statement instance with its execution order. This mapping is implicitly defined by the node parent-child relation within the tree. Loop optimizations and device mapping are expressed as tree modifications and carried out by Loop Tactics, which works as additional passes within Polly. Loop Tactics' passes consume schedule trees and output a CIM-optimized schedule. The modified tree is then passed back to Polly which lowers it back to an imperative AST and then further down to LLVM-IR. In the back-end LLVM-IR is lowered to final executable. It is at this stage of the compilation pipeline where we link our executable with the CIM runtime library. Listing 1 shows the mid-level optimizer output as pseudo-C++ for a GEMM kernel assuming single-precision operands. The GEMM kernel has been detected and swapped by Loop Tactics with a function call to the CIM runtime library (`polly_cimBlasSGemm`). Blas parameters (i.e., values of alpha or leading dimensions) are automatically collected or computed by Loop Tactics. In addition, Loop Tactics inserts an initialization call to configure the CIM hardware (`polly_cimInit`) as well as all the function calls to orchestrate the data movement to and from the device (i.e., `polly_cimMalloc` and `polly_cimDevToHost`).

IV. Demonstration and Evaluation

In this section, we quantify the benefits of CIM computation for a set of linear-algebra kernels from the Polybench/C benchmark suite compiled with TDO-CIM.

a) Experimental Setup: We use the system shown in Figure 1 (a). We select an energy efficient dual-core Arm-A7 with a shared L2 cache. The accelerator is a cycle-accurate model that imitates the functionality of the memristor computations and surrounding digital blocks [10]. The memristor crossbar is an 8-bit 256x256 PCM crossbar based on IBM's 4-bit PCM [4]. To mimic an 8-bit cell with a 4-bit cell, two adjacent columns are used, one for 4 MSBs and the other for 4 LSBs. The final result is computed by a weighted sum of MSB and LSB columns in the digital logic block. The energy and latency model for the crossbar and mixed-signal circuitry is from [4] and [9] respectively. The energy model for the rest of the digital blocks is based on a synthesis report of commercial 40nm finFET technology. Table I summarises our system configuration and energy model.

TABLE I: CIM and Host System Configuration.

CIM Parameter	**Value**
Technology(256x256 @8-bit)	IBM PCM 2x(256x256 @4-bit)
Compute and Write Latency/8-bit	1μs and 2.5μs
Compute Energy/8-bit	200fJ (2x 100fJ/4-bit PCM)
Write Energy/8-bit	200pJ (2x 100pJ/4-bit PCM)
Energy for Mixed signal circuit	3.9nJ (@1.2GHz)
Input/Output buffer Energy (1.5KB)	5.4pJ/byte-access
Digital Logic	40pJ/GEMV for weighted sum and 2.11pJ/extra ALU operation
Energy for DMA and microEngine	<0.78nJ
Host CPU Spec	
2xArm-A7 @1.2GHz	128pJ/inst[1](including cache)
L1-I/D-32KB, L2-2MB	2GB LDDDR3 @933MHz

b) Performance Evaluation: We use `clang -O3 -march-native` and `clang -O3 -march-native -enable-loop-tactics` for the host and the host+CIM, respectively. Dynamic instruction count and run-time are profiled in Gem5 by inserting ROI markers. For energy estimates, we use the numbers shown in Table I. We do not include DRAM energy numbers in the estimates. Figure 4 (left) shows the energy numbers obtained for the reference platform (Arm-A7), and for the Arm-A7+CIM where the kernel execution is performed on the in-memory accelerator. For the host, the energy numbers include the energy spent on computation and in the memory hierarchy. For the CIM, the energy numbers incorporate the energy spent on the driver (host side) and in the accelerator. GEMM-like kernels: `2mm`, `3mm`, `gemm`, and `conv` were able to achieve good energy improvements over the reference system. This is not the case for GEMV-like kernels (`bicg`, `mvt`, `gesummv`) due to their low compute intensity. From the CIM perspective, the compute intensity for a given kernel can be formulated as $\frac{Number-of-MAC-operations}{Number-of-CIM-writes}$ which is very low for GEMV-like kernels as can be seen in Figure 4 (left). With such low compute intensity the energy is dominated by the overhead in host for offloading computations to accelerator

[1]Based on *Ara: Energy-Efficient RISC-V*, Matheus et al. 2019.

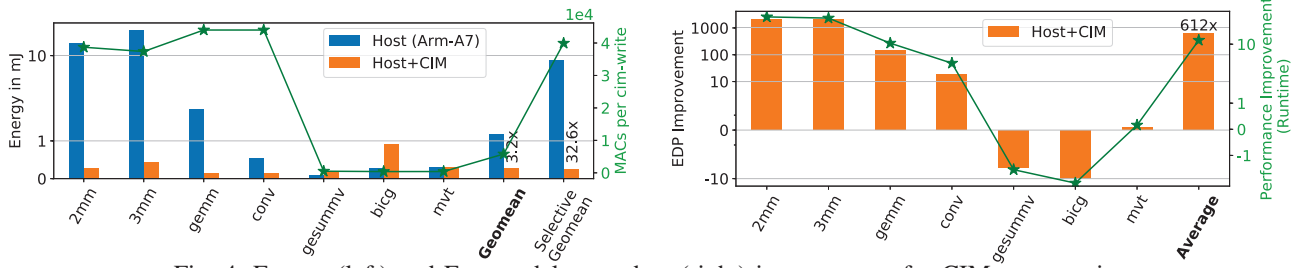

Fig. 4: Energy (left) and Energy-delay-product (right) improvement for CIM computation.

and the number of writes which are costly for the CIM device 200pJ/byte (see Table I). Figure 4 (right) shows the energy-delay-product (EDP). It follows the same trend as the energy plot. We gain for GEMM-like kernels (up to 612x) while we lose for GEMV-like.

V. Related Work

Several works address the issue of code offloading, TOM [18] being perhaps the very first of them. TOM proposes an offloading decision based on a simple cost function. The idea is to statically identify the code section with the highest potential in bandwidth saving. Similarly, Pattanik et. al. propose an affinity prediction model based on memory-related metrics to decide where a given kernel should be executed (i.e., main CPU or in-memory accelerator) [19]. Previously mentioned works target GPU as an in-memory accelerator. On the other hand, in our case, we are targeting a memristor crossbar which means that only specific kernels must be offloaded as the accelerator is capable of executing only GEMM and GEMVs-like kernels. Nair et. al. propose a code offloading based on OpenMP 4.0 user annotation [20]. Contrary, our approach is completely transparent to the application and does not require any user intervention to exploit CIM acceleration. CAIRO relies on an LLC cache profiler and analytical models to decide potential offloading candidates [21]. The LLC profiler is not integrated into the compilation flow and requires to characterize the behavior of each kernel offline. Other works expose CIM acceleration via API [6], [7], which requires significant changes in the application, reducing application readiness, and hurdling widespread adoption.

VI. Conclusion

We present an end-to-end compilation flow for in-memory computing. Our approach automatically identifies, optimizes, and offloads computing kernels to our in-memory accelerator. For a set of linear-algebra kernels from the Polybench/C benchmark suit we obtain an average energy reduction of 32.6x and energy-delay-product improvement of 612x.

Acknowledgments

This research is supported by EC Horizon 2020 Research and Innovation Program through MNEMOSENE project under Grant 780215 and the NeMeCo grant agreement, id. 676240.

References

[1] W. A. Wulf *et al.*, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995.

[2] Singh *et al.*, "Near-memory computing: Past, present, and future," *Microprocessors and Microsystems*, vol. 71, p. 102868, 2019.

[3] M. Le Gallo *et al.*, "Mixed-precision in-memory computing," *Nature Electronics*, vol. 1, no. 4, p. 246, 2018.

[4] M. Le Gallo *et al.*, "Compressed sensing with approximate message passing using in-memory computing," *IEEE Transactions on Electron Devices*, vol. 65, no. 10, pp. 4304–4312, Oct 2018.

[5] M. N. Bojnordi *et al.*, "Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning," in *2016 HPCA*, March 2016, pp. 1–13.

[6] P. Chi *et al.*, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *2016 ISCA*, June 2016, pp. 27–39.

[7] S. Li *et al.*, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *2016 DAC*, June 2016, pp. 1–6.

[8] Q. Xia *et al.*, "Memristive crossbar arrays for brain-inspired computing," *Nature materials*, vol. 18, no. 4, p. 309, 2019.

[9] A. Shafiee *et al.*, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *2016 ISCA*, June 2016, pp. 14–26.

[10] A. BanaGozar *et al.*, "Cim-sim: computation in memory simuiator," in *Proceedings of the 22nd International Workshop on Software and Compilers for Embedded Systems*. ACM, 2019, pp. 1–4.

[11] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[12] F. A. Endo *et al.*, "Micro-architectural simulation of embedded core heterogeneity with gem5 and mcpat," in *RAPIDO 2015*, ser. RAPIDO '15. New York, NY, USA: ACM, 2015, pp. 7:1–7:6.

[13] M. Nazarewicz. (2012) A deep dive into cma. [Online]. Available: https://lwn.net/Articles/486301/

[14] O. Zinenko *et al.*, "Declarative Transformations in the Polyhedral Model," Inria ; ENS Paris - Ecole Normale Supérieure de Paris ; ETH Zurich ; TU Delft ; IBM Zürich, Research Report RR-9243, Dec. 2018. [Online]. Available: https://hal.inria.fr/hal-01965599

[15] L. Chelini *et al.*, "Declarative loop tactics for domain-specific optimization," *ACM TACO*, vol. 16, no. 4, Nov. 2019. [Online]. Available: http://doi.acm.org/10.1145/3372266

[16] T. Grosser *et al.*, "Polly—performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, p. 1250010, 2012.

[17] S. Verdoolaege, *et al.*, "Schedule trees," in *IMPACT, Date: 2014/01/20-2014/01/20, Location: Vienna, Austria*, 2014.

[18] K. Hsieh *et al.*, "Transparent offloading and mapping (tom): Enabling programmer-transparent near-data processing in gpu systems," in *2016 ISCA*, June 2016, pp. 204–216.

[19] A. Pattnaik *et al.*, "Scheduling techniques for gpu architectures with processing-in-memory capabilities," in *2016 PACT*, Sep. 2016, pp. 31–44.

[20] R. Nair *et al.*, "Active memory cube: A processing-in-memory architecture for exascale systems," *IBM Journal of Research and Development*, vol. 59, no. 2/3, pp. 17:1–17:14, March 2015.

[21] R. Hadidi *et al.*, "Cairo: A compiler-assisted technique for enabling instruction-level offloading of processing-in-memory," *ACM TACO*, vol. 14, no. 4, pp. 48:1–48:25, Dec. 2017. [Online]. Available: http://doi.acm.org/10.1145/3155287