TC-CIM: Empowering Tensor Comprehensions for Computing-In-Memory

Andi Drebes
Inria and École Normale Supérieure
Paris, France
andi.drebes@inria.fr

Lorenzo Chelini
TU Eindhoven
Eindhoven, The Netherlands
IBM Research Zurich
Zurich, Switzerland
l.chelini@tue.nl

Oleksandr Zinenko
Google
Paris, France
zinenko@google.com

Albert Cohen
Google
Paris, France
albertcohen@google.com

Henk Corporaal
TU Eindhoven
Eindhoven, The Netherlands
h.corporaal@tue.nl

Tobias Grosser
ETH Zurich
Zurich, Switzerland
tobias.grosser@inf.ethz.ch

Kanishkan Vadivel
TU Eindhoven
Eindhoven, The Netherlands
k.vadivel@tue.nl

Nicolas Vasilache
Google
New York, United States
ntv@google.com

Abstract

Memristor-based, non-von-Neumann architectures performing tensor operations directly in memory are a promising approach to address the ever-increasing demand for energy-efficient, high-throughput hardware accelerators for Machine Learning (ML) inference. A major challenge for the programmability and exploitation of such Computing-In-Memory (CIM) architectures consists in the efficient mapping of tensor operations from high-level ML frameworks to fixed-function hardware blocks implementing in-memory computations.

We demonstrate the programmability of memristor-based accelerators with TC-CIM, a fully-automatic, end-to-end compilation flow from Tensor Comprehensions, a mathematical notation for tensor operations, to fixed-function memristor-based hardware blocks. Operations suitable for acceleration are identified using *Loop Tactics*, a declarative framework to describe computational patterns in a polyhedral representation. We evaluate our compilation flow on a system-level simulator based on Gem5, incorporating crossbar arrays of memristive devices. Our results show that TC-CIM reliably recognizes tensor operations commonly used in ML workloads across multiple benchmarks in order to offload these operations to the accelerator.

Keywords Machine Learning, Computing-In-Memory, Tensor Comprehensions, Loop Tactics, Schedule Trees.

IMPACT 2020, January 22, 2020, Bologna, Italy

1 Introduction

CMOS-based technology has hit the power wall and nears the end of decades-long aggressive scaling. The energy consumption and delay of data movement compared to arithmetic operations [10] have never been higher. Additionally, the available memory bandwidth in today's systems is not able to keep up with the demand of modern, data-intensive workloads [37].

This has led to a growing interest in domain-specific hardware accelerators, abandoning the conventional separation of memory and computing units of the von-Neumann architecture. Such *Computing-In-Memory* (CIM) systems are composed of combined units, performing computation directly in memory and without frequent long-distance, off-chip data movement.

A promising approach for the implementation of such accelerators consists in combining a set of *memristor*-based device into crossbar arrays [35]. Each individual memristive device is capable of storing a multi-bit value as its conductance state. Upon application of an input voltage, the conductance value is multiplied by the input, and the result can be measured at the output of the device.

Assembling many memristive devices into arrays allows for in-place computation of fixed-size tensor operations in constant time. For example, the dot product of two fixed-size vectors v_1 and v_2 can be accomplished by applying voltages corresponding to the values of v_1 to a column of memristive devices whose conductance correspond to the values v_2 and by measuring the resulting current for the entire column. This can further be extended to fixed-size matrix-vector multiplications in constant time by adding one column of memristive devices for each row of the input matrix and by measuring the current at each column. Since memristive devices combine storage and computation, data movement only occurs when new input needs to be loaded into the array or when output values need to be moved out.

This solution is particularly appealing for Machine Learning (ML) applications [22], which commonly rely on tensor operations that can be broken down into the above-mentioned matrix-vector operations.

A key factor for efficient acceleration of ML workloads is the detection, extraction and efficient mapping of tensor operations to the crossbars. While a significant body of work uses memristor crossbars to build special-purpose accelerators, the mapping of operations is often left to the programmer [22, 26, 36]. As a consequence, efficient exploitation requires manual intervention and a deep understanding of technical details of the hardware and thus severely limits programmability. This opposes to the recent trend to increase productivity through high-level abstractions [12, 32, 40, 41] for ML.

The goal of this work is to demonstrate the programmability of CIM accelerators in general and memristor-based architectures in particular. To this end, we present an approach to fully automatically detect tensor operations eligible for offloading to memristor-based accelerators in high-level, abstract representations of machine-learning applications and to map these operations efficiently to the hardware. Our implementation, called *TC-CIM*, integrates *Loop Tactics*, a technique for matching patterns in schedule trees of the polyhedral framework, into *Tensor Comprehensions*, a framework generating highly optimized kernels for accelerators from an abstract, mathematical notation for tensor operations. The flow performs a set of dedicated optimizations aiming at enabling the reliable detection of computational patterns and their efficient mapping to the accelerator. Our approach is based on the observation that despite the large variety of ML workloads, most of them share common linear algebra primitives suitable for CIM architectures.

We make the following contributions:

- TC-CIM, a fully automatic, end-to-end compilation framework based on *Tensor Comprehensions* and *Loop Tactics* which enables the users to exploit in-memory acceleration transparently.
- An experimental evaluation, demonstrating that tensor operations frequently occurring in ML kernels can be reliably identified and extracted with TC-CIM and executed on a Gem5-based simulator for memristor-based accelerators.

The paper is organized as follows. Section 2 introduces *Tensor Comprehensions* and *Loop Tactics*. The integration of these approaches into TC-CIM, as well as dedicated extensions to extract, offload and map tensor operations to the accelerator are presented in Section 3. Section 4 describes our experiments on tensor operations commonly found in ML kernels. Related work is presented in Section 5, before the concluding remarks of Section 6.

2 Tensor Comprehensions and Loop Tactics

Since our approach is based on *Tensor Comprehensions* and *Loop Tactics*, we first provide an overview of both projects in Sections 2.1 and 2.3. An explanation of schedule trees, necessary for the discussion of *Loop Tactics*, is given in Section 2.2.

2.1 Tensor Comprehensions

Tensor Comprehensions [41] is an integrated productivity-oriented system for expressing machine learning workloads embedded into PyTorch, providing a concise input notation with range inference capabilities and leveraging a polyhedral compiler for GPU code generation. In contrast to conventional ML systems, *Tensor Comprehensions* is not limited to a predefined set of operations or layers (such as convolutions or matrix multiplications), but allows the user to specify custom computations using index expressions on tensors. Types and shapes of intermediate tensors only need to be provided explicitly where automatic inference fails due to ambiguity. *Tensor Comprehensions* generates efficient GPU code that fits into a single GPU kernel or, alternatively, into an ML framework *operator*.

The automation of the compilation flow abstracts the implementation details away from programmers, allowing them to reason in mathematical terms. It also enables the compiler to perform advanced program transformations using precise analysis from the polyhedral model, such as operator fusion, eliminating the requirement for temporary storage and spurious serialization by combining multiple ML operations.

2.1.1 Specifying Tensor Operations

The syntax of *Tensor Comprehensions* borrows from Einstein Notation, where universal quantifiers (becoming loops in a program) are introduced implicitly. The listing below demonstrates a simple matrix-vector multiplication between an $M \times K$ matrix A and a K-vector x, resulting in a vector c, all composed of single-precision floating-point values.

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

The shape and the type of inputs are provided explicitly in the function signature, while those of the output tensor are inferred from the index variables. The index variable i iterates over the first dimension of A, its range is therefore $[0, M-1]$. Since i also indexes the output vector c, the size of c is M. Similarly, *Tensor Comprehensions* deduces the range for k from the tensors on the right-hand side of the expression. The algorithm proceeds iteratively if more than one iterator is involved in a subscript, using the ranges computed on the previous step together with the known sizes of the tensors on the right-hand side to infer the shape of the tensor on the left-hand side.

The iterator k only appears on the right-hand side, which indicates that the entire expression is a reduction over k. The "!" mark in the operator denotes a default-initialized reduction. For sum-reductions, as in the example, the left-hand side is initialized with zeros of an appropriate type.

The element type of the output matrix is inferred from the result type of the multiplication and the reduction. As the sum of multiplications of single-precision floating-point values has the same type, c is a vector of single-precision floating-point elements.

2.1.2 Compilation to GPU kernels

To translate the high-level expressions to GPU kernels, *Tensor Comprehensions* successively translates and transforms the input in multiple steps. Figure 1 illustrates the compilation flow when targeting CUDA GPUs.

The input notation is first parsed using a recursive descent method, producing an abstract syntax tree. This tree is converted into the (high-level) Halide [33] intermediate representation (IR), involving tensors and mathematical expressions. Shape inference is performed at this level using Halide's mathematical toolkit to identify the shape of all intermediate and output tensors. The IR is then expanded to expressions with loops around them, which are further converted into a representation based on schedule trees [44]. *Tensor Comprehensions* applies customized affine scheduling derived from *isl* [42] to this representation to expose parallelism and exploit locality, while producing at least one outermost parallel loop. It then maps the computations to a GPU device, associating outermost parallel loops with GPU blocks and innermost parallel loops with GPU threads, and copying data to shared and private memory and inserting relevant synchronizations based on fine-grain dependence analysis. Finally, a customized *isl* code generation algorithm is applied to produce a GPU kernel, which is JIT-compiled through CUDA drivers and executed immediately. The compilation artifacts are stored in an internal cache, indexed by the canonicalized form of input program and compilation options. Additionally, some scheduling, tiling and mapping choices are exposed to the user and can be used to auto-tune the compiler.

Schedule trees are the cornerstone intermediate representation for affine transformations and mapping in *Tensor Comprehensions*. Although implementation details differ, they are also the basis for *Loop Tactics*' pattern-matching scheme. The following section provides an overview on the schedule tree representation.

2.2 Schedule Trees

Within the polyhedral model, the computations to be carried out at execution are defined by the *iteration domain*, a set of statement instances resulting from executions of a statement in a loop nest. In *Tensor Comprehensions*, the iteration domain of each statement is a defined as a Cartesian product of all integer values in the range of all iterators. The schedule defines the order in which statement instances are executed. A *schedule tree* [44] is a representation of affine schedules, designed to eliminate redundancy and provide flexibility for code generation. Among others, the *isl* library, *Tensor Comprehensions* and *Loop Tactics* have adopted schedule trees as the representation for schedules. In this section, we provide a brief description of the schedule tree components relevant to the further discussion. For a detailed description, we refer to the schedule tree paper [44].

A schedule tree is composed of nodes, which are instances of multiple types: *domain nodes*, *context nodes*, *sequence nodes*, *set nodes*, *filter nodes*, *band nodes*, *extension nodes*, and *mark nodes*.

Domain nodes define the iteration domain of the program. In this paper, we only consider trees with a single domain node as the root.

Optional *context* nodes provide information about program parameters, i.e., values that are known to be constant during the execution, but whose values are unknown at compile time. Each context node may introduce new program parameters or provide additional constraints on parameter values. For example, in the context of code generation for GPUs, context nodes are used to introduce parameters for block and thread identifiers, as these must be emitted symbolically.

The schedule represented by a schedule tree is structured by *sequence* nodes and *set* nodes. The former imposes sequential execution of their children, while the latter specifies that their children can be executed concurrently. Sequence and set nodes are the only nodes that can have multiple children; all other nodes can have at most one child. Their children are always *filter* nodes, which restrict the schedule defined by their descendants to a subset of the iteration domain.

Partial affine schedules are encoded in *band* nodes, which also indicate if the band dimensions are parallel and whether they are permutable. *Extension* nodes introduce auxiliary statements into the computation. They are primarily used for synchronization primitives and data transfer loops when targeting accelerators. Finally, *mark* nodes represent annotations of a subtree with arbitrary additional data that is preserved for the code generation phase.

Figure 2a illustrates the schedule tree corresponding to the C code in Listing 1. The tree consists of a domain node, followed by a sequence node with two children, the first of which has a two-dimensional band, encoding the schedule of statement $S1$, while the second has a three-dimensional band encoding the schedule of $S2$. This tree corresponds to the loop-distributed version of matrix multiplication. Fusing the outer loops would create a band node above the sequence

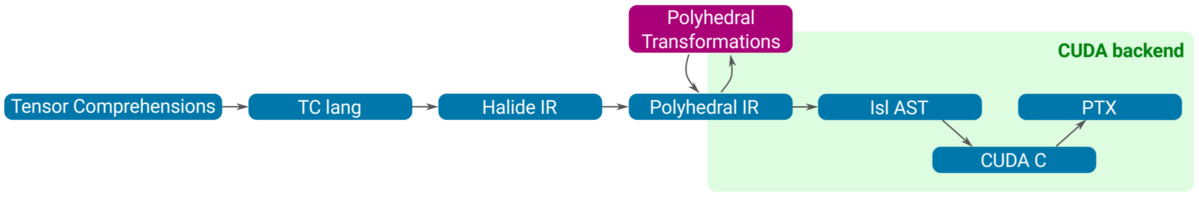


Figure 1. Compilation flow of *Tensor Comprehensions*.

```
    for (int i = 0; i < N; ++i)
      for (int j = 0; j < N; ++j)
S1:      C[i][j] = beta * C[i][j];
    for (int i = 0; i < N; ++i)
      for (int j = 0; j < N; ++j)
         for (int k = 0; k < N; ++k)
S2:        C[i][j] += alpha * A[i][k] * B[k][j];
```

Listing 1. Generalized matrix multiplication (GEMM).

node, common to both statements, as shown in Figure 2b. The new band node makes the band node of the branch for S_1 unnecessary, but the band node for S_2 is still required due to ordering on the k dimension.

2.3 Loop Tactics

Loop Tactics [11, 47] is a framework based on the *isl* library supporting a declarative specification of affine transformations. The three main concepts in *Loop Tactics* are *Schedule Tree Matchers*, recognizing patterns in schedule trees, *Relation Matchers*, recognizing access patterns, and *Tree Builders*, which allow for the construction of sub-trees implementing loop transformations.

Schedule Tree Matchers and Builders A *Schedule Tree Matcher* is a declarative description of a tree pattern whose occurrences are to be located in a schedule tree. Each matcher is represented as a tree that corresponds to the structure of sub-trees of interest. In addition to the node types for schedule trees, matchers may also include wildcard nodes to specify arbitrary sub-patterns and further support filter predicates (implemented as callback functions) to restrict matches to properties that involve more than just the shape of the tree. For example, a matcher may only accept permutable bands if it is used to find tiling opportunities, or only accept outermost bands with parallel loops if it is used for device mapping.

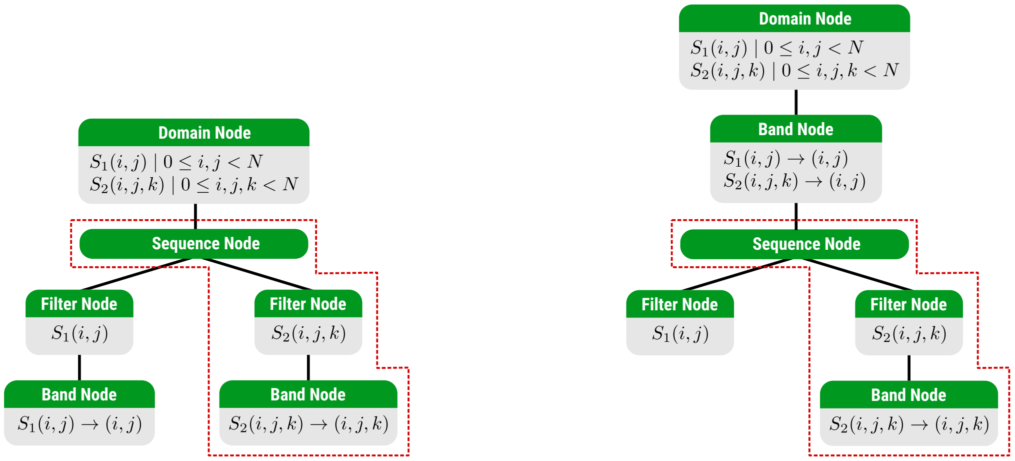
The matchers can be used to *capture* specific schedule tree nodes that serve as pointers into the matched sub-tree. These nodes can then be used to implement transformations declaratively using *Schedule Tree Builders*. Builders use a syntax similar to matchers to describe the structure of the schedule tree to be constructed. Each call to a builder takes as arguments the properties of the node to be constructed, e.g., the partial schedule for the band nodes or a subset of the iteration domain for filter nodes. In addition, builders support the insertion of arbitrary sub-trees rooted at a given node, provided that the final tree respects the schedule tree invariants.

In summary, a loop transformation can be expressed as a matcher pattern that captures a set of nodes and a builder that reorganizes the captured nodes into a new structure or modifies their properties. The schedule tree is essentially recreated by each builder, which aligns well with functional-style declarative APIs.

The matcher in Listing 2 (Lines 25–30) shows how a sub-tree starting at a sequence node with at least one filter node followed by a band as children can be detected. The band node is matched only if the callback `hasGemmPattern` returns true. In case of a match, the node corresponding to the GEMM pattern (body) is captured (Line 29), and the sub-tree is rebuilt by splitting the body node into two nested bands by taking the integer division and the modulo parts of the schedule (Lines 32–35). The integer division and the modulo part of the schedule are obtained via *Loop Tactics*' functions `tileSchedule` and `pointSchedule`.

Access Relation Matchers *Access relation matchers* allow the caller to identify memory accesses that have certain properties in a union of relations. The matching mechanism operates through placeholders (`placeholder` and `arrayPlaceholder`). Each placeholder has two data components: a constant *pattern* and a variable *candidate*. Each relation is checked against a pattern and may yield one or more candidates. Currently, *Loop Tactics* provides pattern and candidate descriptions for affine expressions of the form $\omega = k * \iota + c$, where k and c represent a pattern and ω and ι define a candidate by matching one of the output and input dimensions, respectively. A match is an *assignment* of candidates to placeholders.

As an example, consider the access relation matchers shown in Listing 2 (lines 1–21), locating GEMM-specific access patterns. Such patterns have at least three two-dimensional reads to different arrays (lines 12–15), one write access (line 17), and a permutation of indexes that satisfies the placeholder pattern $[i, j] \longrightarrow [i, k][k, j]$. The latter condition is enforced by using the same placeholders in `_i`, `_j` and `_k` at the respective positions in the access matchers and by checking the number of matches in lines 19 and 20.



(a) With distributed *i* and *j* loops **(b)** With fused *i* and *j* loops

Figure 2. Schedule trees for Listing 1. Dotted lines: sub-tree matched by the matcher from Listing 2, Lines 25–30.

```
1  auto hasGemmPattern = [&](schedule_node node) {
2    auto _i = placeholder();
3    auto _j = placeholder();
4    auto _k = placeholder();
5    auto _A = arrayPlaceholder();
6    auto _B = arrayPlaceholder();
7    auto _C = arrayPlaceholder();
8
9    auto reads = /* get read accesses */;
10   auto writes = /* get write accesses */;
11
12   auto mRead = allOf(
13     access(_C, _i, _j),
14     access(_A, _i, _k),
15     access(_B, _k, _j));
16
17   auto mWrite = allOf(access(_C, _i, _j));
18
19   return match(reads, mRead).size() == 1 &&
20          match(writes, mWrite).size() == 1;
21 };
22
23 schedule_node body, continuation;
24
25 auto matcher =
26   sequence(
27     hasDescendant(
28       filter(
29         band(body, hasGemmPattern,  // filter function
30            anyTree(continuation)())))); // wildcard
31
32 auto builder =
33   band([&]() { return tileSchedule(body, tileSize); },
34     band([&]() { return pointSchedule(body, tileSize); },
35       subtree(body)));
```

Listing 2. Tree and Access Relation matchers for GEMM.

3 TC-CIM

We may now describe how TC-CIM integrates *Loop Tactics* into Tensor Comprehensions and how matchers are used to recognize patterns for offloading to CIM accelerators.

The accelerator architectures targeted by TC-CIM differ substantially from GPUs targeted by *Tensor Comprehensions*: they are not generally programmable and only support fixed functions. This means that parallelism can only be exploited within the offloaded patterns and there is no concept of threads executing in parallel. All instructions that are not part of a fixed pattern must therefore be executed by a general-purpose host CPU and the fixed patterns are offloaded via specialized instructions. From the programmer's perspective, this is similar to sequential code for general-purpose CPUs with interlaced calls to specialized libraries (e.g., BLAS [8]).

3.1 Outline of the modified compilation flow

In order to support CIM accelerators, we first added a new backend for the generation of sequential C code. This backend is based on the existing CUDA backend, but omits all work partitioning for blocks and threads, synchronization between instructions executing in parallel, memory promotion and any CUDA-specific primitives and library calls. We then added an optimization pass invoking *Loop Tactics* with appropriate matchers and builders.

Figure 3 shows the modified compilation flow for TC-CIM. Up until the generation of the polyhedral IR, this flow is identical with the default flow of *Tensor Comprehensions*. The major modification occurs before generation of the *isl* AST, when pattern detection based on *Loop Tactics* is carried out. In this step, patterns are searched in the schedule tree and matches are recorded using mark nodes. These marks are preserved in the *isl* AST and processed by the custom

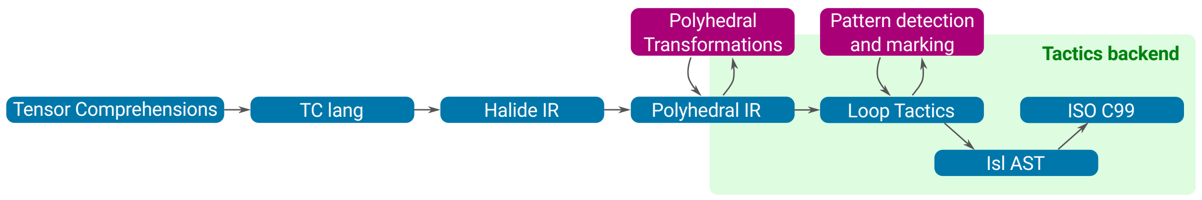


Figure 3. Modified compilation flow for TC-CIM.

printer generating C Code, which finally emits function calls to a CIM accelerator library.

The flow currently focuses on the detection and offloading of individual operations with hardware support and does not yet include inter-operation optimizations (e.g., minimizing data movement between the host and the accelerator, operator fusion for combined instructions, etc.). For this reason, we chose to match patterns on the schedule tree rather than on TC expressions, to benefit from affine scheduling and its ability to expose parallel dimensions, tilable bands, distribute computations across different bands amenable to pattern matching. We will discuss this choice and the potential for more global optimization in Section 4.4.

3.2 Matching

TC-CIM currently recognizes patterns for three operations: plain matrix-vector multiplication, plain matrix multiplication and batched matrix multiplication; these operations are further extended to handle any loop permutation and transposed accesses.

The patterns in the schedule tree for all three operations are sufficiently similar such that recognition can be implemented with a *single* matcher and appropriate functions distinguishing between the operations when processing a match. In addition, the matcher is provided with a callback function that checks the access pattern and operations of a candidate match to exclude matches on structurally compliant sub-trees featuring scalar operations that are not supported on the target. The former involves the use of appropriate access relation matchers, while the latter requires custom procedures analyzing the Halide expressions associated to the polyhedral statements.

In case of a successful match and positive checks, the root node of the matched sub-tree is marked with a mark node whose name indicates the matched operation (i.e., `_mvt`, `_gemm` or `_batched_gemm`) and whose pointer for user-defined data receives the address of a structure with all meta-information that is needed to emit the call to the specialized library function with the correct parameters during code generation.

Upon generation of the *isl* AST, the information of mark nodes is preserved as AST mark nodes, which are processed by the printer generating the C code.

Figure 4 illustrates this procedure for a plain matrix multiplication. The schedule tree initially contains a sub-tree formed of a band node, a sequence node, and its children. The children of the sequence corresponds to the zero-initialization of the output matrix and the reduction operation of the matrix multiplication. The pattern matching procedure inserts a mark node at the root of the sub-tree, which points to a GEMM Info structure whose payload holds the operands, sizes of the dimensions and flags for the transposition of the operands. When generating the *isl* AST, the mark is interpreted by the printer to emit a call to `cimblas_gemm`.

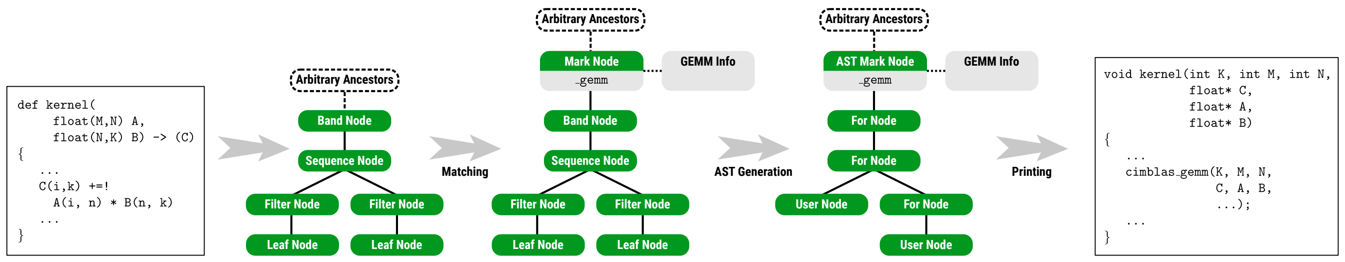
Although the AST contains an entire sub-tree for the operation below the mark node (e.g., for nodes and user nodes for the initialization and reduction in the matrix multiplication example), as of now, this part is simply ignored in our TC-CIM prototype. For future implementations, we plan to replace the marking-based scheme with actual transformations of the schedule tree, such that no custom handling is needed in the printer.

We also plan to perform tiling on the bands of matched operations in order to overcome the limitations for the maximum size of operands when offloading to the accelerator. Currently, this issue can be addressed by applying tiling when generating the schedule before matching. While this leads to the desired effects on the size of offloaded operands, this scheme is overly eager and also tiles bands that do not belong to any offloaded operation. By performing the tiling only on bands of matched operations in future implementations, no unnecessary tiling will be performed.

Stepping back, one should notice that most of these adaptations of the matchers are not specific to CIM. Many optimization and hardware acceleration schemes involve shape and memory capacity limitations that TC-CIM addresses, from hardwired tensor core shapes on Nvidia GPUs to fixed matrix sizes and alignment constraints on numerical libraries [39]. The future work highlighted in the previous paragraph makes even more sense when considering this broader context.

4 Evaluation

In this section, we show that TC-CIM is able to identify and extract matrix and matrix-vector multiplications from multiple benchmarks and that the matching is robust against prior transformations. We evaluate both the static impact of



```
def kernel(
     float(M,N) A,
     float(N,K) B) -> (C)
{
   ...
   C(i,k) +=!
     A(i, n) * B(n, k)
   ...
}
```

```
void kernel(int K, int M, int N,
            float* C,
            float* A,
            float* B)
{
   ...
   cimblas_gemm(K, M, N,
                C, A, B,
                ...);
   ...
}
```

Figure 4. Pattern matching, marking and code generation for GEMM.

TC-CIM on the generated source code and its impact on the number of dynamic operations offloaded to the accelerator.

4.1 CIM Architecture

Figure 5 shows an overview of our SoC emulated in Gem5 [7]. The SoC consists of a single Arm high-performance in-order core (HPI core[1]), main memory (256 MiB), and a CIM accelerator interconnected via the system bus. The accelerator is memory-mapped and accesses the shared memory via DMA operations. The central part of the accelerator is the control unit, while the execution pipeline is a single CIM-tile. Within the CIM-tile, the memristor crossbar executes analog vector-matrix multiplications. We model a 256 × 256 PCM-crossbar with 8 bit precision using 4 bit PCMs. We accomplish this by distributing the computation over multiple columns and perform a weighted sum at the columns' output using additional digital logic (Shift & Add block in Figure 5c) [13]. This allows the accelerator to perform operations on tensors composed of 8 bit integer elements. The physical properties of a single PCM device such as read and write latency are taken from literature [24]. Being an analog device, the memristor crossbar requires additional mixed-signal circuitry to interact with the digital system. The voltage source (V), sample and hold (S&H), and the analog-to-digital converter ADCs in the design (Figure 5c) serve this purpose. Row and column buffers in Figure 5b act as temporary data registers for the PCM crossbar.

In a typical offloading scenario, the host prepares the data on shared memory and triggers the accelerator execution by writing to special memory-mapped registers. The CIM accelerator then reads the data in the shared-memory via DMA transactions. Once done, the accelerator writes back the results in the shared memory. The host monitors the status of CIM execution by polling the status register, and upon completion, it can safely resume execution. Cache coherency is ensured by flushing the CPU cache before acceleration. All steps necessary for offloading (i.e., flushing the CPU cache) are encapsulated by a system library that exposes high-level BLAS-like API. To reduce simulation time in our experiments, we used a lightweight, bare-metal implementation of the library directly interacting with the hardware instead of a full-system simulation with a complete operating system kernel and user-space.

[1]The Arm Research Starter Kit, Sep 2017

4.2 Benchmarks and Workloads

To evaluate TC-CIM on simple kernels composed of a single tensor operation, we used *mv* (matrix-vector multiplication), *mm* (matrix-matrix multiplication), and *batchmm* (batched matrix-matrix multiplication). As a representative for kernels with multiple tensor operations, we have added *3mm* (two matrix-matrix multiplications and the multiplication of their results) and *4cmm*, which performs four consecutive, independent matrix-matrix multiplications.

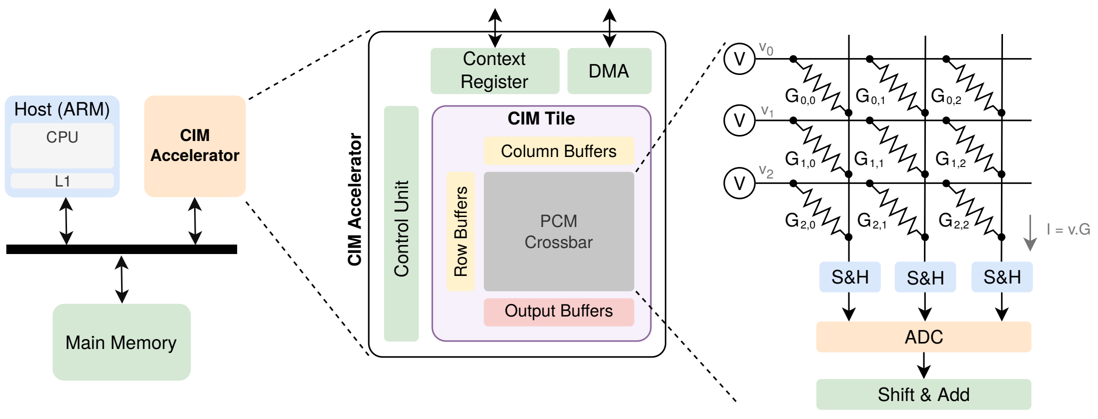
As a more advanced use case, we experimented with the multi-layer perceptron component of a production model reported in the *Tensor Comprehensions* paper [41]. This model has trailing fully-connected layers followed by non-linearities, which corresponds to matrix-matrix multiplications and pointwise operations. In *Tensor Comprehensions*, these layers were split into two parts: MLP1 and MLP3, with the former containing a single matrix multiplication and the latter containing three matrix multiplications using each other's result. Since the single matrix multiplication of MLP1 is already covered by *mm*, we have only added *mlp3* to the evaluation.

In order to match the element size of the CIM accelerator, we used tensor elements with a size of 8 bit in all benchmarks.

4.3 Kernel Experiments

We evaluate the effects of TC-CIM on the program in two ways: static impact on the generated source code and dynamic impact on execution.

Static impact For each combination of benchmarks and workloads, we have generated specialized code with TC-CIM by setting the parameters corresponding to the workload sizes statically. As a first metric for the success of the matching, we have determined the number of callsites for CIM library functions statically from the source code and compared it to the maximum number of callsites expected for perfect matching (`Oracle`). Figure 6 shows the number of callsites for each of the benchmarks, under two different conditions: in `TC-CIM-not tiled`, we carried out the matching right after running the affine scheduler, while for `TC-CIM`



(a) Overview of the architecture **(b)** Architecture of the CIM accelerator **(c)** Memristive crossbar array

Figure 5. Overview of the emulated system.

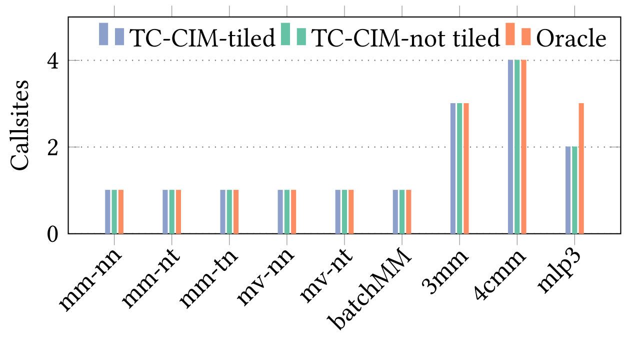


Figure 6. Number of callsites for CIM library functions inserted by TC-CIM without (TC-CIM-not tiled) and with prior tiling (TC-CIM-tiled) compared to perfect matching (Oracle).

`tiled`, we additionally forced tiling on the three outermost loops with tile sizes of 32 before the matching. The suffixes for *mm* and *mv* indicate whether the operands are transposed (*nn*: no transposition, *tn*: first operand transposed, *nt*: second operand transposed).

For the non-tiled version, TC-CIM detects all but one matrix multiplication in *mlp3*. This multiplication is missed, as it does not match the structure expected by the matcher: the matcher expects the initialization statement and the core-computation statement to be filter nodes under the same sequence, while in this case, the filters are children of different sequences.

For the tiled version, TC-CIM is still capable of detecting all operations for all benchmarks. Indeed, tiling does not affect detection at all since TC-CIM runs a canonicalization pass which squashes together point-loop and tile-loop bands.

At this stage, TC-CIM reliably detects the relevant patterns for most cases, even in the presence of prior tiling transformations and for transposed accesses. The remaining mismatches motivate further work to better coordinate affine transformations and matchers/builders.

Dynamic Impact To measure the *impact* of the matching on the execution, we have instrumented the simulator to report the number of dynamic arithmetic instructions and memory accesses on the host CPU and CIM accelerator. We recorded these numbers for the code generated by TC-CIM without tiling and for a sequential baseline without library calls that has been generated with the matchers disabled.

Figure 7 shows the breakdown of host instructions for the baseline without CIM offloading. The breakdown for host instructions with offloading, normalized to the total number of dynamic host instructions of the corresponding benchmark from the baseline from Figure 7, is shown in Figure 8.

Several key observations can be made from the differences between the figures:

- Almost all ALU operations are offloaded to CIM. This is coherent with the observations from the previous section: the eligible operations represent the majority of the instructions of each kernel and all of them could be offloaded.
- The CIM runtime library overhead is very small, showing that offloading itself is not an issue.
- As the storage and computations happen in the same place, the CIM offloading is able to reduce the number of external memory accesses compared to execution on the host. In the baseline, each multiplication requires two operands to be loaded from memory, whereas CIM offloading requires the input data but not the (constant) weights in a neural network layer.

Due to the lack of sufficient experience with physical implementations of the selected memristor crossbar array, the simulation model is not yet accurate enough to estimate the energy savings and performance improvements for offloading. However, similar approaches report significant energy savings for kernel offloading, e.g., the 8-bit proj-PCM [16], which requires 6 fJ per 8 bit multiplication (assuming 100 ns

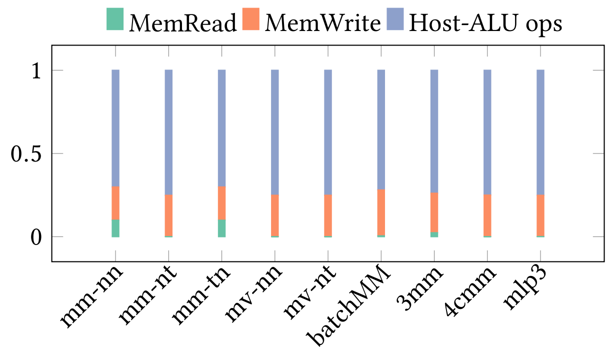


Figure 7. Instruction breakdown for the baseline without offloading.

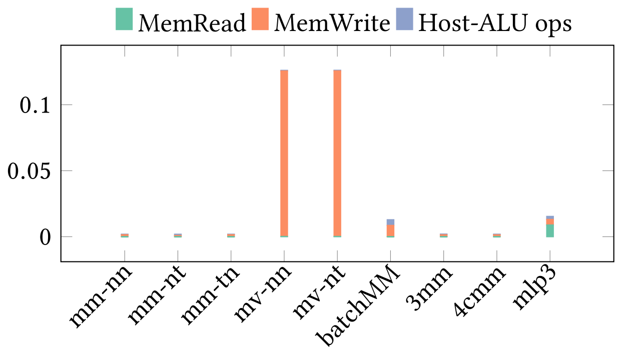


Figure 8. Instruction breakdown with offloading.

read time provided by an integrated circuit), compared to 0.2 pJ per multiplication for 45 nm CMOS logic (33× higher).

4.4 Discussion

We chose to apply loop tactics *after* affine scheduling. This allowed us to use the scheduler to canonicalize the schedule tree: discover permutability and parallelism properties, coalesce input bands into a single band when possible, all to make the matching patterns simpler. On the downside, the scheduling algorithm in this case is unaware of the higher-level information extracted by matchers. As itentified in the previous experiments, the interplay between *Loop Tactics* and the scheduler deserves to be explored further, in particular, the effects of (re-)scheduling after matching to re-evaluate fusion decisions or using the matched primitives to guide tile size selection. Since machine instructions may expect specific data layout, matched patterns can also guide data layout transformations.

Exploring this interplay extends to design-space questions, such as whether affine transformations should be made smarter or more aggressive to reduce the effort in implementing domain- and target-specific matchers and builders (e.g., matchers with fixed tile size, dimension ordering, layout and alignment constraints, with builders in control of the innermost levels of computation and data movement only); or on the contrary, whether more effective tool flows will result from limiting the expectations on generic/enabling affine transformations, communicating with versatile matchers and builders capable of structured mapping decisions (e.g., tiling, permutation, orchestrating complex internal control flow and data transfers).

Based on this analysis, we believe that future work will blur the boundaries between advanced affine scheduling heuristics [48], matchers [11, 47], and explicit metaprogramming like URUK [17], CHiLL [46], or Halide [33]. Our work advocates for a declarative, constraint-based approach, where target-specific constraints help guide affine transformations together with domain- and expert-provided metaprogramming sketches.

5 Related Work

In-memory computing Fujiki et. al. proposed a compiler framework that lowers Google's TensorFlow DFG into simpler instructions supported by the in-memory accelerator [15]. During lowering, complex instructions (e.g., division, exponents, and transcendental functions) are broken down into a set of LUTs, additions, and multiplications that can be executed on the crossbar array. The approach also includes a set of scheduling optimizations to expose instruction and block-level parallelism. Software pipelining is used to overlap computation and storage in the CIM crossbar. Ambrosi et. al. proposed end-to-end software stack to support hybrid analog accelerators for ML that are ISA programmable [1]. The software stack includes an ONNX [30] back-end for importing models from popular deep-learning frameworks and a compiler which lower the ONNX description to a custom ISA. During the lowering phase, the compiler performs a set of optimizations, such as graph partitioning and tile placement. The authors present a limited evaluation of their compiler. Building upon the work of Ambrosi et. al., Ankit et. al. developed a runtime compiler implemented as C++ library. The programmer needs to write the application using the library provided constructs, and the compiler will lower the high-level code to assembly targeting their own custom ISA [3]. Contrary to previously mentioned works [1, 3, 15] we decided not implement an ISA for our accelerator, the main reasons are: a) the high non-recurring engineering cost of developing an ISA, 2) the loss in crossbar density due to the addition of an hardware decoder. Crossbar density is one of the workhorse for PCM-based devices and we want to preserve it [24].

Similar to our adopted approach, other works expose an API for their accelerator [9, 13, 26, 38]. In the compilation stage, the API is lowered to data-path configurations, as well as executing commands with data dependencies and control flow. However, this approach requires the programmer to write the entire application using the proposed API, hence reducing application readiness. On the other hand, in our approach the API invocation is handled transparently in the compiler; thus, no changes in the application are needed.

Near-memory computing Another computational paradigm that is promising to overcome the memory wall problem is *near-memory computing*, which aims at processing close to where the data resides. One of the main challenges in near-memory compilation is to decide which code portion should be offloaded to the accelerator. Our pattern matching can be seen as an explicit way of performing code offloading. Other works propose cost-based analysis. Hsieh et. al. proposed to statically identify code portions with the highest potential in bandwidth saving using simple cost functions [21]. Pattanik et. al. proposed an affinity prediction model relying on memory-related metrics [31]. Hadidi et. al. identified code region to be offloaded in the context of the Hybrid Memory Cube using a cache profiler, a compile-time analysis phase and benefit analysis models [20]. Differentiating from previous work, Nair et. al. rely on the user to offload code regions that should be marked with OpenMP 4.0 directives [29]. Cost-based analysis can be easily embedded in our approach as callback functions during matching. One of the future works will be the integration of a fast analytical model for associative caches [19] such that we can have a more sophisticated analysis for offloading profitability.

Broader applications of CIM-tile Clearly, there exist a wide CIM design space exploring different technological and architecture tradeoffs. All of them have in common the need to automatically decompose the computation, data, and communication patterns to suit the hardware constraints. Our approach offers a portable abstraction to program any such device, providing automatic tiling and enabling loop transformations specifically suited to the target. Coupling affine transformations with Tactics provides additional performance and specialization through the embedding of target-specific software or hardware blocks. Moreover, many CIM designs—including all finite state and analog-based ones—have in common to expose a limited-functionality API rather than a programmable architecture. In addition to enabling transformations, such architectures strongly depend on Tactics to lower a portable tensor programming layer to target-specific API calls.

Beyond CIM, and thanks to its declarative nature, the matcher-based approach extends easily to other kinds of emerging accelerators with matrix- or tensor-level operations (e.g., GPU tensor cores). Such devices are often programmable through driver library calls, featuring the same interface as numerical libraries, e.g., BLAS. Our approach facilitates the porting of standard tool flows and programming models to new hardware accelerators, and also helps improving performance on existing hardware when aggressively optimized library implementations are available.

General-purpose (Polyhedral) accelerator mapping There are a variety of approaches for automatic accelerator programming. At the source level, Par4All [2] uses a non-polyhedral approach based on abstract interpretation which enables powerful inter-procedural analyses. Polyhedral compilation techniques have first been used for GPU code generation by Baskaran [6] and have later been improved as part of the R-Stream compiler [25]. An alternative mapping approach that relies on the counting of integer points to tightly fill shared memory caches has been proposed by Baghdadi et. al. [5], but the resulting memory accesses have been shown to be too costly in practice. With CUDA-CHiLL[34] generating GPU codes based on user provided scripts has been proposed. The state-of-the-art in general-purpose polyhedral source-to-source compilation is ppcg [43, 45], which provides effective GPU mappings that exploit shared and private memory. The main focus of these tools is the generation of GPU kernel code from code following strict programming rules [4].

Offloading from within a compiler has been first proposed by GRAPHITE-OpenCL [23] which allowed for the static mapping of parallel loops, but did not considering inter SCoP data reuse. In the context of Polly [18], Kernelgen [28] proposed a new approach in which it aims to push as much execution as possible on the GPU, using the CPU only for system calls and other program parts not suitable for the GPU. The final executables are shipped with a sophisticated run-time system that supports just-in-time accelerator mapping, parameter specialization and provides a page-locking based run-time system to move data between devices. Damschen et. al. [14] introduce a client-server system to automatically offload compute kernels to a Xeon-Phi system. These approaches are based on an early version of Polly (or GRAPHITE), without support for non-affine subregions, modulo expressions, schedule trees or delinearization and are consequently limited in the kind of SCoPs they can detect. Finally, with Hexe [27] a modular data management and kernel offloading system was proposed which does to our understanding not take advantage of polyhedral device mapping strategies. The presented approaches aim for general-purpose accelerator mapping and do not consider the identification and transformation of algorithm specific constructs.

6 Conclusion

We presented TC-CIM, a fully automatic compilation flow based on *Tensor Comprehensions* and *Loop Tactics* dedicated to in-memory computation. TC-CIM offers a unique combination of domain-specific optimizations, affine transformations and target-specific matchers, leveraging schedule tree abstractions. Our preliminary evaluation on tensor operations frequently occurring in ML kernels confirms the expressive power and reliable extraction of computational building blocks offloaded to a simulated memristor array. We believe these results will support future research and applications to in-memory computing and beyond, some of which have been discussed in the paper.

Acknowledgments

This work was partially supported by the European Commission Horizon 2020 programme through the MNEMOSENE grant agreement, id. 780215 and the NeMeCo grant agreement, id. 676240 as well as through Polly Labs (Xilinx Inc, Facebook Inc, and ARM Holdings) and the Swiss National Science Foundation through the Ambizione program.

References

[1] J. Ambrosi, A. Ankit, R. Antunes, S. R. Chalamalasetti, S. Chatterjee, I. E. Hajj, G. Fachini, P. Faraboschi, M. Foltin, S. Huang, W. Hwu, G. Knuppe, S. V. Lakshminarasimha, D. Milojicic, M. Parthasarathy, F. Ribeiro, L. Rosa, K. Roy, P. Silveira, and J. P. Strachan. 2018. Hardware-Software Co-Design for an Analog-Digital Accelerator for Machine Learning. In *2018 IEEE International Conference on Rebooting Computing (ICRC)*. 1–13. https://doi.org/10.1109/ICRC.2018.8638612

[2] Mehdi Amini, Béatrice Creusillet, Stéphanie Even, Ronan Keryell, Onig Goubier, Serge Guelton, Janice Onanian McMahon, François-Xavier Pasquier, Grégoire Péan, and Pierre Villalon. [n.d.]. Par4all: From convex array regions to heterogeneous computing. In *IMPACT: Second Int. Workshop on Polyhedral Compilation Techniques HiPEAC 2012.*

[3] 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 Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, New York, NY, USA, 715–731. https://doi.org/10.1145/3297858.3304049

[4] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F. Donaldson, Jeroen Ketema, Javed Absar, Sven Van Haastregt, Alexey Kravets, Anton Lokhmotov, Robert David, and Elmar Hajiyev. 2015. PENCIL: a Platform-Neutral Compute Intermediate Language for Accelerator Programming. In *International Conference on Parallel Architectures and Compilation Techniques*. San Francisco, US.

[5] Soufiane Baghdadi, Armin Größlinger, and Albert Cohen. 2010. Putting Automatic Polyhedral Compilation for GPGPU to Work. In *Proceedings of the 15th Workshop on Compilers for Parallel Computers (CPC'10)*. Vienna, Austria. https://hal.inria.fr/inria-00551517

[6] Muthu Manikandan Baskaran, J Ramanujam, and P Sadayappan. 2010. Automatic C-to-CUDA code generation for affine programs. In *Compiler Construction*. Springer.

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[8] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. 2001. An Updated Set of Basic Linear Algebra Subprograms (BLAS). 28 (2001), 135–151.

[9] M. N. Bojnordi and E. Ipek. 2016. Memristive Boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 1–13. https://doi.org/10.1109/HPCA.2016.7446049

[10] S. Borkar. 2013. Exascale computing - A fact or a fiction?. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 3–3. https://doi.org/10.1109/IPDPS.2013.121

[11] Oleksandr; Grosser Tobias; Corporaal Henk Chelini, Lorenzo; Zinenko. 2019. Declarative Loop Tactics for Domain-Specific Optimization. *ACM TACO* 16, 4 (Dec. 2019). https://doi.org/10.1145/3372266

[12] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symp. on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[13] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie. 2016. PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 27–39. https://doi.org/10.1109/ISCA.2016.13

[14] Marvin Damschen, Heinrich Riebler, Gavin Vaz, and Christian Plessl. 2015. Transparent Offloading of Computational Hotspots from Binary Code to Xeon Phi. In *Proc. of the 2015 Design, Automation & Test in Europe Conf. & Exh (DATE '15)*. EDA Consortium, 1078–1083. http://dl.acm.org/citation.cfm?id=2757012.2757063

[15] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. 2018. In-Memory Data Parallel Processor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/3173162.3173171

[16] I. Giannopoulos, A. Sebastian, M. Le Gallo, V. P. Jonnalagadda, M. Sousa, M. N. Boon, and E. Eleftheriou. 2018. 8-bit Precision In-Memory Multiplication with Projected Phase-Change Memory. In *2018 IEEE International Electron Devices Meeting (IEDM)*. 27.7.1–27.7.4. https://doi.org/10.1109/IEDM.2018.8614558

[17] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies. *International Journal of Parallel Programming* 34, 3 (2006), 261–317. https://doi.org/10.1007/s10766-006-0012-3

[18] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly – performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters* 22, 04 (2012), 1250010.

[19] Tobias Gysi, Tobias Grosser, Laurin Brandner, and Torsten Hoefler. 2019. A fast analytical model of fully associative caches. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 816–829.

[20] Ramyad Hadidi, Lifeng Nai, Hyojong Kim, and Hyesoon Kim. 2017. CAIRO: A Compiler-Assisted Technique for Enabling Instruction-Level Offloading of Processing-In-Memory. *ACM Trans. Archit. Code Optim.* 14, 4, Article 48 (Dec. 2017), 25 pages. https://doi.org/10.1145/3155287

[21] K. Hsieh, E. Ebrahim, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler. 2016. Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 204–216. https://doi.org/10.1109/ISCA.2016.27

[22] Vinay Joshi, Manuel Le Gallo, Irem Boybat, Simon Haefeli, Christophe Piveteau, Martino Dazzi, Bipin Rajendran, Abu Sebastian, and Evangelos Eleftheriou. 2019. Accurate deep neural network inference using computational phase-change memory. arXiv:cs.ET/1906.03138

[23] A Kravets, A Monakov, and A Belevantsev. 2010. GRAPHITE-OpenCL: Automatic parallelization of some loops in polyhedra representation. *GCC Developers' Summit, GCC Developers' Summit* (2010).

[24] M. Le Gallo, A. Sebastian, G. Cherubini, H. Giefers, and E. Eleftheriou. 2018. Compressed Sensing With Approximate Message Passing Using In-Memory Computing. *IEEE Transactions on Electron Devices* 65, 10 (Oct 2018), 4304–4312. https://doi.org/10.1109/TED.2018.2865352

[25] Allen Leung, Nicolas Vasilache, Benoît Meister, Muthu Baskaran, David Wohlford, Cédric Bastoul, and Richard Lethin. 2010. A Mapping Path for multi-GPGPU Accelerated Computers from a Portable High Level Programming Abstraction. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU-3)*. ACM, New York, NY, USA, 51–61. https://doi.org/10.1145/1735688.1735698

[26] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie. 2016. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/2897937.2898064

[27] Christos Margiolas. 2015. A Heterogeneous Execution Engine for LLVM. *LLVM Developers Meeting* (2015).

[28] Dmitry Mikushin, Nikolay Likhogrud, Zheng (Eddy) Zhang, and Christopher Bergstrom. 2014. KernelGen – The Design and Implementation of a Next Generation Compiler Platform for Accelerating Numerical Models on GPUs. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW)*.

[29] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C. . Cher, C. H. A. Costa, J. Doi, C. Evangelinos, B. M. Fleischer, T. W. Fox, D. S. Gallo, L. Grinberg, J. A. Gunnels, A. C. Jacob, P. Jacob, H. M. Jacobson, T. Karkhanis, C. Kim, J. H. Moreno, J. K. O'Brien, M. Ohmacht, Y. Park, D. A. Prener, B. S. Rosenburg, K. D. Ryu, O. Sallenave, M. J. Serrano, P. D. M. Siegl, K. Sugavanam, and Z. Sura. 2015. Active Memory Cube: A processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development* 59, 2/3 (March 2015), 17:1–17:14. https://doi.org/10.1147/JRD.2015.2409732

[30] ONN. [n.d.]. Open Neural Network Exchange Home Page. `https://onnx.ai/` (Nov. 2019).

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

[32] PlaidML 2018. PlaidML. https://www.intel.ai/plaidml.

[33] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[34] Gabe Rudy, MalikMurtaza Khan, Mary Hall, Chun Chen, and Jacqueline Chame. 2011. A Programming Language Interface to Describe Transformations and Code Generation. In *Languages and Compilers for Parallel Computing*, Keith Cooper, John Mellor-Crummey, and Vivek Sarkar (Eds.). Lecture Notes in Computer Science, Vol. 6548. Springer Berlin Heidelberg, 136–150. https://doi.org/10.1007/978-3-642-19595-2_10

[35] Abu Sebastian, Manuel Le Gallo, and Evangelos Eleftheriou. 2019. Computational phase-change memory: beyond von Neumann computing. *Journal of Physics D: Applied Physics* 52, 44 (aug 2019), 443002. https://doi.org/10.1088/1361-6463/ab37b6

[36] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar. 2016. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 14–26. https://doi.org/10.1109/ISCA.2016.12

[37] Gagandeep Singh, Lorenzo Chelini, Stefano Corda, Ahsan Javed Awan, Sander Stuijk, Roel Jordans, Henk Corporaal, and Albert-Jan Boonstra. 2019. Near-memory computing: Past, present, and future. *Microprocessors and Microsystems* 71 (2019), 102868.

[38] L. Song, X. Qian, H. Li, and Y. Chen. 2017. PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 541–552. https://doi.org/10.1109/HPCA.2017.55

[39] Daniele G. Spampinato and Markus Püschel. 2014. A Basic Linear Algebra Compiler. In *International Symposium on Code Generation and Optimization (CGO)*. 23–32.

[40] Leonard Truong, Rajkishore Barik, Ehsan Totoni, Hai Liu, Chick Markley, Armando Fox, and Tatiana Shpeisman. 2016. Latte: A Language, Compiler, and Runtime for Elegant and Efficient Deep Neural Networks. In *Proc. of the 37th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'16)*. ACM, New York, NY, 209–223. https://doi.org/10.1145/2908080.2908105

[41] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2019. The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically. *ACM Trans. Archit. Code Optim.* 16, 4, Article 38 (Oct. 2019), 26 pages. https://doi.org/10.1145/3355606

[42] Sven Verdoolaege. 2010. isl: An Integer Set Library for the Polyhedral Model. In *Mathematical Software - ICMS 2010*, Komei Fukuda, Joris Hoeven, Michael Joswig, and Nobuki Takayama (Eds.). Lecture Notes in Computer Science, Vol. 6327. Springer, 299–302.

[43] Sven Verdoolaege. 2015. *PENCIL support in pet and PPCG.* Technical Report RT-0457. INRIA Paris-Rocquencourt. https://hal.inria.fr/hal-01133962

[44] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. 2014. Schedule trees. In *International Workshop on Polyhedral Compilation Techniques, Date: 2014/01/20-2014/01/20, Location: Vienna, Austria.*

[45] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization* 9, 4 (Jan. 2013), 54:1–54:23. https://doi.org/10.1145/2400682.2400713

[46] Huihui Zhang, Anand Venkat, Protonu Basu, and Mary Hall. 2016. Combining Polyhedral and AST Transformations in CHiLL. In *IMPACT Workshop*.

[47] Oleksandr Zinenko, Lorenzo Chelini, and Tobias Grosser. 2018. *Declarative Transformations in the Polyhedral Model.* Research Report RR-9243. Inria ; ENS Paris - Ecole Normale Supérieure de Paris ; ETH Zurich ; TU Eindhoven ; IBM Zürich. https://hal.inria.fr/hal-01965599

[48] Oleksandr Zinenko, Sven Verdoolaege, Chandan Reddy, Jun Shirako, Tobias Grosser, Vivek Sarkar, and Albert Cohen. 2018. Modeling the conflicting demands of parallelism and Temporal/Spatial locality in affine scheduling. In *Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria.* 3–13. https://doi.org/10.1145/3178372.3179507