Compiling Neural Networks for a Computational Memory Accelerator

Kornilios Kourtis*
Independent researcher

Martino Dazzi
IBM Research

Nikolas Ioannou
IBM Research

Tobias Grosser
ETH Zurich

Abu Sebastian
IBM Research

Evangelos Eleftheriou
IBM Research

Abstract

Computational memory (CM) is a promising approach for accelerating inference on neural networks (NN) by using enhanced memories that, in addition to storing data, allow computations on them. One of the main challenges of this approach is defining a hardware/software interface that allows a compiler to map NN models for efficient execution on the underlying CM accelerator. This is a non-trivial task because efficiency dictates that the CM accelerator is explicitly programmed as a dataflow engine where the execution of the different NN layers form a pipeline.

In this paper, we present our work towards a software stack for executing ML models on such a multi-core CM accelerator. We describe an architecture for the hardware and software, and focus on the problem of implementing the appropriate control logic so that data dependencies are respected. We propose a solution to the latter that is based on polyhedral compilation.

1 Introduction

As general-purpose architectures hit scalability limits, important applications such as machine learning (ML) turn to specialized hardware to meet their energy and performance requirements [1–9].

Computational memory (CM) contrasts traditional Von Neumann architectures, which separate computation and memory, by enabling memory to perform computations on the data it holds. Specifically, technologies such as PCM or Flash can be used to build crossbar arrays that, using Kirchhoff's laws, implement an analog Matrix-Vector multiplication (MxV), where the matrix data are stored in the crossbar memory cells [10–12]. Such a crossbar can execute an MxV in a single step, whereas digital logic typically requires multiple steps. At the same time, combining compute and storage in a single unit reduces communication which constitutes the main challenge of data-intensive workloads. This makes CM an attractive alternative for accelerating ML workloads [12–22].

CM technologies such as PCM or Flash require significant time to program (write). This renders the common practice of using accelerators to execute one NN layer at a time [1, 4, 6, 7, 23] impractical for these technologies due to the high overhead of reconfiguring the crossbars, which might take as long as a minute [11]. Instead, efficiency dictates that the CM accelerator is configured once to implement a given NN. After configuration, inference is performed by streaming input data to the accelerator.

In this work, we discuss our approach towards a software stack that targets such a multi-core CM chip for accelerating deep learning inference at the edge [10, 24, 25], where each core includes a crossbar that implements an analog MxV operation. The accelerator follows a dataflow processing model, where inference is executed as a pipeline formed by the different NN layers.

Our ultimate goal is to build a software stack that enables transparent use of the CM accelerator, and, at the same time, guide hardware design so the accelerator can be better utilized by software. Hence, we design the compiler and the rest of the software stack in tandem with the accelerator. Specifically, we prototype a Computational Memory Neural Network Compiler (cmnnc) that aims to compile NN models to be executed on the CM accelerator, and a simulator that models such hardware and acts as the target platform.

A key problem we faced, that stems from the fact that the NN network needs to be fitted into the accelerator, is generating the control logic between the CM cores that execute different layers of the NN so that data dependencies are respected. Because traditional accelerators do not require this feature, existing ML compilers [26–28] do not have facilities for tackling it. We address this challenge by using polyhedral compilation techniques [29, 30] to represent data dependencies and generate code for state machines that implement the desired control.

In summary, the *contributions* of this work are: **1)** We propose an architecture for an inference CM accelerator and the software stack for driving it. **2)** We discuss an approach for compiling NN models for the proposed CM accelerator, focusing on using polyhedral compilation to express the control logic between the CM cores so that data dependencies are respected. To the best of our knowledge this is an open problem, and we are the first to use polyhedral compilation to tackle it.

While we believe that our approach is generally applicable to dataflow architectures, for the sake of brevity and clarity

*Majority of work done while at IBM Research

our discussion assumes a specific architecture which is described in §2. In §3 we present the proposed compiler, and a solution for dealing with data dependencies. Finally, we discuss related work in §4, and conclude in §5.

2 The Computational Memory Accelerator

We start by discussing a functional model of the hardware. This model is meant as a vehicle for co-designing the hardware interface with the software stack, and hence omits many details about the hardware implementation.

The left side of Fig. 1 shows the various components of the CM core. In addition to the crossbar (XBAR), the core also includes a lightweight digital processing unit (DPU), and local memory (MEM), which is, typically, a few kilobytes of SRAM. The crossbar array implements an analog MxV operation, where the matrix M data (typically, weights) are stored directly in the crossbar's memory cells, while vector V (typically, activations) is loaded from the local memory. It is worth noting that the crossbar has certain dimensions reflected in the size of the MxV operations. We call this dimension the *width* of the unit, i.e., a width of 64 means that M contains 64×64 elements, while input and output vectors contain 64 elements.

The motivation behind this design is to run inference on NNs by executing each different layer on a separate core, thus forming a pipeline between the cores that resembles the structure of the NN. The operation that we target to accelerate (e.g., the convolution operation on convolutional NNs) executes on the crossbar, while everything else (e.g., activation functions, pooling layers) executes on the DPU.

In Fig. 1, thick red lines and labels with black background denote data operations while dashed lines and labels with white background denote control operations. Execution proceeds in cycles. During a cycle, the local control unit (LCU) loads the data of the input vector from the local SRAM to the crossbar (②, ❶) so that the analog MxV is performed. When the MxV operation completes, its output vector is made available to the DPU (❷), which then executes a sequence of instructions. During the execution, the DPU may load and store data to the local SRAM (❸), and schedule data transfers from the local SRAM to other cores (④, ❹). The data will become available on the remote core's local SRAM on the next cycle (❺).

The cores are organized in a pipeline, where the input of one is the output of another. Hence, in many cases one core has to wait until all necessary data from remote cores become available before executing. To this end, the LCU "snoops" the remote writes to SRAM (①, ❺) and implements a state machine that controls execution. Depending on this state machine, the LCU may either do nothing or execute a computation step. In the latter case, specific SRAM values are loaded to the crossbar input (②), the MxV operation is executed, and then DPU executes its instructions.

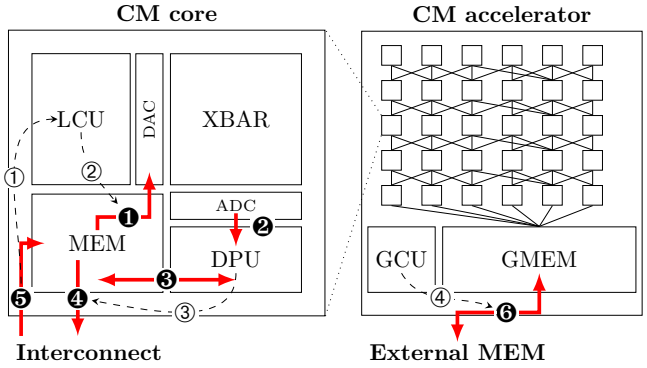

Figure 1. CM core and CM accelerator architecture.

The CM accelerator chip (right side of Fig. 1) includes a number of interconnected cores, a global input/output buffer (GMEM), and a global control unit (GCU). The GCU issues DMA operations for transferring data between the external (e.g., host) memory and the chip's GMEM (④, ❻). The GCU also transfers data from (to) the GMEM to (from) cores that act as input (output) nodes in the dataflow graph.

The hardware model of the CM accelerator also includes the topology of the interconnect. A simple approach would be to assume that all cores are connected with each other, either via an all-to-all topology or by having a routing scheme that enables all-to-all communication via message forwarding. Both of these approaches, however, have inefficiencies [31], and we, instead, decide to expose the interconnect topology to the compiler so that it can optimize the mapping of NN layers to CM cores accordingly. We represent the chip topology as a directed graph, where an edge from one unit to another means that the first can send data to the second. An example of an effective interconnect topology for a CM accelerator is described by Dazzi et al. [32].

3 Compiling NNs for the CM accelerator

Machine learning frameworks such as Tensorflow or PyTorch represent NNs as dataflow graphs where nodes are computational operators (e.g., a matrix multiplication), and edges are data dependencies between the operators. This approach is well-suited for mapping these networks onto the CM accelerator that also follows a dataflow execution model. Inference is performed on pre-trained models that bundle two types of data: the dataflow graph, and initialization data for the operators (e.g., model weights) as produced by a separate training phase. Abstractly, the role of the compiler is to map such a model (in our prototype we use the ONNX format [33]) to the CM accelerator for execution.

We focus on convolutional NNs (CNNs), NNs based on convolutions, which is where most of their execution time is spent [34]. We accelerate these networks by mapping the convolution to the crossbar's MxV operation (see Listing 1), and use the DPU for the remaining operations.

```
# input (inp) shape: (D,IH,IW)
# filters (flt) shape: (FL,D,FH,FW)
# output (out) shape: (D,OH,OW)
def conv2d_mxv(inp,flt,out):
  N = FD*FH*FW
  m = flt.reshape((FL,N))
  for oh in range(OH):
    for ow in range(OW):
      v = inp[:,oh:oh+FH,ow:ow+FW].reshape(N)
      out[:,oh,ow] = matmul(m,v)
```

Listing 1. Convolution using an MXV operation (implemented via Python NumPy).

From a software perspective, there are three phases in using the CM accelerator: compilation, initialization, and execution. Compilation uses the NN dataflow graph and a hardware description of the accelerator (number of cores and their properties, interconnect topology, etc.) as input, and produces a configuration for each individual unit of the accelerator (GCU, DPUs, LCUs). These configurations, bundled together and serialized, initialize the accelerator. After initialization, inference on the compiled model is executed by streaming data to the accelerator.

Compilation is performed in two steps: *partitioning*, where the dataflow graph is partitioned so that each partition is mapped to a different CM core, and *lowering*, where the compiler processes each partition separately and produces the configuration for each of the units of the corresponding core. Note that the partitioning step must adhere to the constrains imposed by the hardware. For example, local objects have to fit into local memory, and if two dataflow nodes connected via an edge are mapped to different cores, the cores should also be connected in the hardware interconnect graph.

After compilation, the serialized configurations are used together with the model weights to initialize the CM accelerator. Model weights are used for programming the crossbar arrays, but also to initialize objects that reside in accelerator-local memories. Once the accelerator is initialized, the execution phase may start where an external process (e.g., a driver running on the host) passes descriptors for the input and output data to the GCU.

Next, we discuss our approach for building such a compiler. Our ideas are realized in a prototype implementation called cmnnc (Computational Memory Neural Network Compiler).

3.1 Partitioning and Mapping

As discussed previously, the dataflow graph of the NN is partitioned during the first phase of the compilation, and each partition is mapped onto a different CM core on the accelerator. We perform these two steps, namely partitioning and mapping, separately. For the first step, we enforce two invariants: that each partition has *at most* one convolution operator (or more generally speaking, one operator executed using the crossbar), and that there are no cycles

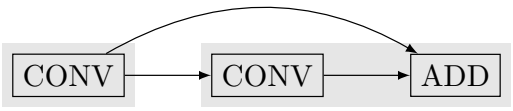

Figure 2. Dataflow graph example with two convolutions and an addition node.

in the partition graph. These invariants are not necessary, but they simplify the compilation problem without limiting applicability in practice. Consider, for example, the dataflow graph of Fig. 2, with two convolution and one addition operators. Following the two invariants presented above the graph is partitioned as shown in the figure: based on the first invariant we must create two partitions, one for each convolution operation. Based on the second invariant, we must bundle the addition operation with the right hand-side partition; if we bundle it with the left hand-side partition, a cycle is formed and the invariant breaks.

We perform the partitioning by iterating the dataflow nodes in their topological order, and creating a new partition whenever we encounter a convolution node. (This assumes that there are no cycles in the dataflow graph, but this is a common assumption, e.g., the ONNX format also disallows cycles.) Given a set of partitions, edges of the dataflow graph are either within the same partition or span multiple partitions. The latter type of edges define the partition graph. We map the partition graph to the CM accelerator, i.e., mapping each partition to a CM core and each edge to a connection in the interconnect topology, by expressing the problem as a set of constrains in the Z3 SMT solver [35].

3.2 Lowering

Once the partitions are defined and mapped to CM cores, the compiler produces the configurations for the GCU, LCUs, and DPUs (lowering phase).

The problem of configuring the DPU, i.e., defining the set of instructions to execute after the MxV operation, is very similar to the problem solved by existing ML compiler frameworks [26–28], and in practice we expect to address it by developing an appropriate backend for such a framework that targets the DPU instruction set.

Configuring the GCU and LCUs, however, cannot, to the best of our knowledge, be addressed by existing ML compiler frameworks, even the ones that target specialized ML accelerator hardware. The reason is that existing ML accelerators typically provide offloading of certain operations, but are not explicitly programmed as a dataflow engine, so such functionality is not needed. As we discussed previously, however, explicitly mapping the NN execution into a dataflow graph on the CM accelerator is necessary due to the high cost of reconfiguring the crossbars. In the next paragraphs, we describe the problem of configuring the LCU, and propose a solution that utilizes polyhedral compilation. A similar approach can be used for the GCU.

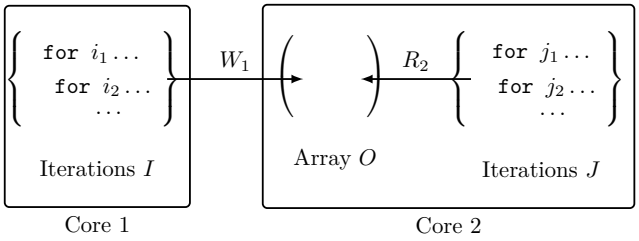

Figure 3. Modeling dependencies between CM cores.

As is the case with existing frameworks, we represent the inputs and outputs of operations as tensors, i.e., multi-dimensional arrays. These arrays reside on the local memory of the CM core that reads them. In the context of the LCU, we are concerned with arrays that are written to and read by different cores, i.e., the arrays that are defined by cross-partition edges in the partition dataflow graph. For example, in Fig. 2, the output of the first `CONV` operator on the first partition is read by the operators of the second partition.

The LCU executes a state machine that snoops the remote writes from other cores to the local objects and decides when and how (e.g., what data to load to the crossbar) to trigger the local computation. For example, if we consider Listing 1, the first iteration can be performed only when the data in `inp[:,0:FH,0:FW]` have been written (either by the GCU or by another core). In other words, there is a read after write (RAW) dependency that needs to be respected to ensure correct execution. Next, we discuss how we can generate the LCU state machine using the polyhedral model.

3.3 Enforcing dependencies via the polyhedral model

To formally reason about data dependencies across operations and partitions, we use the polyhedral model, where computations are represented as nested loops that access multi-dimensional arrays. Since most nodes on NN dataflows are linear algebra operators with tensor operands, the polyhedral model works well for such applications. Specifically, we use the integer set library (ISL) [36, 37] to represent the properties of computations as Presburger sets and relations. ISL allows representing sets of integer tuples, and relations that map elements of one set to another, efficiently, without enumerating all their elements. ISL sets are used to model iteration spaces, where each tuple consists of the values of the induction variables of a loop nest, as well as array locations, where each tuple is a multi-dimensional index into the array elements. ISL relations are used to expressed mappings between such sets (e.g., data dependencies).

To illustrate these concepts, we consider two cores, each executing its own loop nest and an array O that the first core writes to and that the second reads from (Fig. 3). (For simplicity, we ignore the outermost loop, which is iterating over input data and is generally unbounded.) Each core's loop nest defines an instance set: $(i_1, i_2, \ldots) \in I$ for core 1, and $(j_1, j_2, \ldots) \in J$ for core 2. The data each iteration reads (writes) from (to) arrays defines the read (write) *access relation*, which maps each iteration instance to one or more array locations. For example, if R_2 is the read relation of J to O ($J \rightarrow O$), R_2 includes a $(j \rightarrow o)$ pair *iff* iteration $j \in J$ reads from location $o \in O$. Listing 2 shows the read access relation for array `inp` of Listing 1, where tuples of values for the induction variables (`CONV_MXV[oh,ow]`) are mapped to locations on the input array (`inp[id,ih,iw]`) using inequality constrains. Similarly, if W_1 is the write relation of I to O ($I \rightarrow O$), then W_1 includes a $(i \rightarrow o)$ pair *iff* iteration $i \in I$ writes to location $o \in O$. We assume that object locations are written to at most once, i.e., write relations are injective. Note that we do not (indeed cannot) make the same assumption for read relations.

```
1  { CONV_MXV[oh,ow] -> inp[id,ih,iw] :
2        0 <= oh < OH
3    and 0 <= ow < OW
4    and 0 <= id < D
5    and oh <= ih < oh + FH
6    and ow <= iw < ow + FW }
```

Listing 2. Read access relation for Listing 1

We can enforce dependencies by executing all iterations of I on core 1 before all iterations of J on core 2, but this is inefficient. Instead, we want to allow cores to execute iterations in parallel as much as possible, forming a pipeline. Hence, our goal is to compile a state machine that observes writes from iterations in I, and advances iterations in J so that they are only executed if all the data they read have already been written. To this end, we use the ISL algebra to compute relation $\mathcal{S}$ $(O \rightarrow J)$ that maps observed writes in O, to the maximum (based on execution order) iteration in J that can be executed. We present the steps for computing $\mathcal{S}$ in ISL in Appendix A.

Using relation $\mathcal{S}$, we can generate code for the LCU state machines. Each cross-partition edge in the dataflow graph defines an array shared by two partitions: its writer (source) and its reader (destination). We compute the access relations (read and write) for each array based on the operator type (e.g., convolution) and parameters (e.g., convolution kernel size, padding, etc.). Note that we can combine edges with the same source and destination, so that only one array is used. For example, for the graph of Fig. 2 we use a single array by combining the read access relations of the `CONV` and `ADD` node. For every partition, we compute the relation $\mathcal{S}$ for every object read. Using these relations, we use the ISL AST facilities to generate code that implements the LCU state machine.

3.4 Prototype

We realize our approach in a prototype implemented in Python, which aims to compile ONNX models and execute

them in a CM accelerator simulator that acts as the target platform. While still work in progress, our implementation is available as open source in https://github.com/IBM/cmnnc.

3.5 Further Challenges

While we believe our approach shows promise, it is still at the early stages, with missing functionality before we can claim a full-fledged solution. Here, we highlight some of the main challenges, which we plan to address in future work.

In our prototype, we program the LCU state machine by generating arbitrary Python code (specifically, we generate a Python AST using the ISL AST facilities, which we then compile to Python bytecode). This works well for our simulation, because it enables flexibility to experiment, but a hardware implementation might not be able to afford the ability to execute arbitrary code on the LCU. Hence, it might be necessary to implement a more restrictive interface for configuring the LCU (and also the GCU) so that the dependency tracking state machine can be efficiently implemented in hardware, while allowing the compiler to generate the appropriate configuration for arbitrary NN dataflow graphs.

Most of the challenges of a software stack for a CM accelerator stems from the high cost of reprogramming the crossbar arrays. In a traditional accelerator, if the width of a hardware unit (e.g., GEMM) cannot support the full operation, the software can just break the operation in sub-operations and issue them separately. In contrast, the CM accelerator has to be configured so that the implemented dataflow execution graph deals with this limitation at initialization time. Hence, the NN dataflow graph needs to be transformed so that it is compatible with the CM accelerator properties [38, 39] (e.g., performing quantization [40] or breaking up operations that do not fit into individual CM cores), while, at the same time, ensuring that the inference performance does not take a hit. As a first step to address these challenges, we plan to quantify their effect by executing and evaluating existing CNNs in our simulator.

4 Related work

Digital NN accelerators Most hardware NN accelerators work by considering one NN layer at a time, and splitting its execution among multiple blocks that can be offloaded to the device [1, 4, 6, 7, 23]. Frequently, these accelerators follow a dataflow (spatial) architecture, e.g., by building a matrix multiply unit as a systolic array [1], or by executing a 2D convolution into an array of interconnected processing elements [4]. Our approach, instead, fits multiple NN layers to the accelerator and implements a dataflow execution model at a coarser granularity (inter-layer instead of intra-layer). One of the key challenges that recent NN accelerators try to address is how to organize the computation to maximize data reuse within a given memory hierarchy across the different options of reusing input data, reusing weights, and reusing intermediate results [2, 5, 41, 42]. On that spectrum, the CM accelerator described here is an extreme because the weights are directly encoded into the crossbars.

CM NN accelerators There is extensive work in accelerating NNs using CM [13–19]. Prime [22] is a CM accelerator based on resistive memory with a software/hardware interface similar to the one described here, as is ISAAC [20], where control vectors for driving state machines are briefly mentioned but no precise description is given. These works identify the problem of implementing logic for respecting dependencies, but provide no solution on how the compiler can generate this logic which is the main focus of our work. Indeed, we are not aware of any works that tackle this problem. PUMA [12] implements a CM accelerator using memristor crossbars, and defines an ISA [21] for programming the accelerator. In this case, the dependencies are enforced not by a state machine, but by respecting the order of the generated instruction sequence.

ML compiler frameworks Initially, ML frameworks used hand-crafted specialized routines for implementing ML operators for each different hardware target (CPUs, GPUs, accelerators). A number of ML compiler frameworks have since been develop that aim to automatically generate code for these operators, but also perform cross-operator optimizations [26–28]. These frameworks target traditional accelerators, and focus on offloading parts of the NN dataflow graph and exploiting data parallelism rather than pipeline parallelism which our work focuses on.

Polyhedral compilation Polyhedral compilation enables reasoning about properties of code in the form of nested loops with affine bounds, and have been used for many applications including optimizations [43, 44], accelerator mapping [45], program verification [46], modeling caches [47, 48], and computing bounds on IO complexity [49]. Indeed, many compiler frameworks that target NN workloads utilize polyhedral compilation techniques [50–53]. These works are complementary to ours in that they deal with transforming and scheduling loop nests so that they can be executed efficiently, while approach targets generating control state machines for respecting data dependencies.

5 Conclusion

In this paper, we presented an initial approach towards a CM accelerator and its corresponding software stack that targets efficient inference on NN models. We focus on the problem of explicitly programming the accelerator as a dataflow engine and discuss how the compiler can generate control state machines that ensure that data dependencies during pipelined execution are respected.

A Computing $\mathcal{S}$ using ISL

Here, we present the details of how to compute the relation $\mathcal{S}$ using ISL. Assuming an iteration space I that writes an object O, and an iteration space J that reads it (§3.3), we want to compute relation $\mathcal{S}$ $(O \rightarrow J)$ that maps observed writes in O, to the maximum iteration in J that can be executed.

A.1 Background

For completeness, we briefly present the ISL operators that we use. We make some simplifications for brevity, and we refer the reader to the ISL documentation [36, 54] for a complete discussion.

We represent iteration spaces (I, J) and object locations (O) as **sets** of integer tuples. We also consider **relations** which map elements of one set to another. For example, a relation $I \rightarrow O$ consists of pairs $(i \rightarrow o)$ such that $i \in I$ and $o \in O$.

The domain $dom(R)$ of a relation R is the set defined by the first element of the pairs.

$$dom(R) = i : \exists j : (i \rightarrow j) \in R$$

The inverse of a relation R, $\mathbf{R^{-1}}$ includes the same tuple pairs as R, but with their order reversed:

$$R^{-1} = \{\ \ (j \rightarrow i) : (i \rightarrow j) \in R\ \ \}$$

Relations A,B can be composed as $\mathbf{B(A)}$ (or B *after* A):

$$B(A) = \{i \rightarrow j : \exists k : (i \rightarrow k) \in A\ \wedge\ (k \rightarrow j) \in B\}$$

Integer tuples can be ordered lexicographically. We use $\prec$, $\succ$, $\preceq$, $\succeq$ to represent this order.

The lexicographical maximum $\mathbf{lexmax(R)}$ of a relation R is a subset of R, where for pairs in R with the same first element, it only keeps the pair with the lexicographically maximal second element.

$$\text{lexmax}\,(R) = \{(i \rightarrow j) : (i \rightarrow j) \in R \\ \wedge \forall (i' \rightarrow k) \in R : i = i' \Rightarrow k \preceq j\}$$

A.2 Computing S

Relation W_1^{-1} $(O \rightarrow I)$ maps each array location to the iteration in I that writes it. Relation $\mathcal{K}$ pairs read iterations $j \in J$ write iterations $i \in I$ based on RAW dependencies. That is, if $(j \rightarrow i) \in \mathcal{K}$, iteration i writes locations read by iteration j:

$$\mathcal{K} := W_1^{-1}(R_2) \qquad (J \rightarrow I)$$

Next, we compute relation $\mathcal{L}$ that pairs every read iteration $j \in J$ to the last write iteration $i \in I$ that satisfies all dependencies for all iterations up to and including j. In other words, after i is executed, the reader can safely execute all iterations until j. Note that the above assumes an order between iterations. We assume iterations are executed in their lexicographical order, as is normally the case. We compute $\mathcal{L}$ as follows. First, we compute the domain D of $\mathcal{K}$.

$$D := \text{dom}\,(\mathcal{K}) \qquad (J)$$

Next, we use the *lexicographically-greater-than-or-equal* relation on sets operations (>>=) to compute relation D' such that each iteration j is mapped to all iterations ζ that do not succeed it ($\zeta \preceq j$).

$$D' := D >>= D \qquad (J \rightarrow J)$$

We can apply relation D' after $\mathcal{K}$, which results in a relation that pairs each read iteration j to every write iteration i that writes data read by iterations $\zeta \preceq j$. We compute $\mathcal{L}$ using the *lexmax* operator, which will only keep the last write iteration after which all iterations $\zeta \preceq j$ can be safely executed.

$$\mathcal{L} := \text{lexmax}\,(\mathcal{K}(D')) \qquad (J \rightarrow I)$$

Applying W_1 *after* $\mathcal{K}$, results in relation $\mathcal{M}$, which if reversed maps locations written by the writer loop, to the maximum iteration we can execute on the reader loop. Because a single write might be mapped to multiple read iterations, we use the *lexmax* operator to keep the maximal (i.e., latest).

$$\mathcal{M} := W_1(\mathcal{L}) \qquad (J \rightarrow O)$$
$$\mathcal{S} := \text{lexmax}\left(\mathcal{M}^{-1}\right) \qquad (O \rightarrow J)$$

References

[1] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, 2017.

[2] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[3] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. A hardware-software blueprint for flexible deep learning specialization. https://arxiv.org/abs/1807.04188, 2018.

[4] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 44(3):367–379, 2016.

[5] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. *ACM SIGPLAN Notices*, 53(2):461–475, 2018.

[6] NVDLA deep learning accelerator. http://nvdla.org/, 2017.

[7] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGARCH Computer Architecture News*, 42(1):269–284, 2014.

[8] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 45(2):27–40, 2017.

[9] I. Bratt. Arm's first-generation machine learning processor. https://www.hotchips.org/hc30/2conf/2.07_ARM_ML_Processor_HC30_ARM_2018_08_17.pdf, 2018. HotChips 30.

[10] E. Eleftheriou, M. L. Gallo, S. R. Nandakumar, C. Piveteau, I. Boybat, V. Joshi, R. Khaddam-Aljameh, M. Dazzi, I. Giannopoulos, G. Karunaratne, B. Kersting, M. Stanisavljevic, V. P. Jonnalagadda, N. Ioannou, K. Kourtis, P. A. Francese, and A. Sebastian. Deep learning acceleration based on in-memory computing. *IBM Journal of Research and Development*, 63(6):7:1–7:16, Nov 2019.

[11] D. Fick and M. Henry. Analog computation in Flash memory for datacenter-scale AI inference in a small chip. https://www.hotchips.org/hc30/2conf/2.05_Mythic_Mythic_Hot_Chips_2018_V5.pdf, 2018. HotChips 30.

[12] 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, et al. 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*, pages 715–731, 2019.

[13] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. PipeLayer: A pipelined reram-based accelerator for deep learning. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 541–552. IEEE, 2017.

[14] Xiaoxiao Liu, Mengjie Mao, Beiye Liu, Hai Li, Yiran Chen, Boxun Li, Yu Wang, Hao Jiang, Mark Barnell, Qing Wu, et al. RENO: A high-efficient reconfigurable neuromorphic computing accelerator design. In *Proceedings of the 52nd Annual Design Automation Conference*, pages 1–6, 2015.

[15] Shubham Jain, Abhronil Sengupta, Kaushik Roy, and Anand Raghunathan. RxNN: A framework for evaluating deep neural networks on resistive crossbars. https://arxiv.org/abs/1809.00072, 2018.

[16] Ming Cheng, Lixue Xia, Zhenhua Zhu, Yi Cai, Yuan Xie, Yu Wang, and Huazhong Yang. TIME: A training-in-memory architecture for memristor-based deep neural networks. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2017.

[17] Shimeng Yu. Neuro-inspired computing with emerging nonvolatile memorys. *Proceedings of the IEEE*, 106(2):260–285, 2018.

[18] Shuangchen Li, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. DRISA: A dram-based reconfigurable in-situ accelerator. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 288–301. IEEE, 2017.

[19] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. In-memory data parallel processor. *ACM SIGPLAN Notices*, 53(2):1–14, 2018.

[20] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Computer Architecture News*, 44(3):14–26, 2016.

[21] Joao Ambrosi, Aayush Ankit, Rodrigo Antunes, Sai Rahul Chalamalasetti, Soumitra Chatterjee, Izzat El Hajj, Guilherme Fachini, Paolo Faraboschi, Martin Foltin, Sitao Huang, et al. Hardware-software co-design for an analog-digital accelerator for machine learning. In *2018 IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–13. IEEE, 2018.

[22] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. PRIME: a novel processing-in-memory architecture for neural network computation in reram-based main memory. *ACM SIGARCH Computer Architecture News*, 44(3):27–39, 2016.

[23] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE, 2014.

[24] Abu Sebastian, Manuel Le Gallo, and Evangelos Eleftheriou. Computational phase-change memory: beyond von neumann computing. *Journal of Physics D: Applied Physics*, 52(44):443002, 2019.

[25] Abu Sebastian, I Boybat, Martino Dazzi, Iason Giannopoulos, V Jonnalagadda, Vikram Joshi, Geethan Karunaratne, Benedikt Kersting, Riduan Khaddam-Aljameh, SR Nandakumar, et al. Computational memory-based inference and training of deep neural networks. In *2019 Symposium on VLSI Technology*, pages T168–T169. IEEE, 2019.

[26] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, Jack Montgomery, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, Misha Smelyanskiy, and Man Wang. Glow: Graph lowering compiler techniques for neural networks. https://arxiv.org/abs/1805.00907, 2018.

[27] 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. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI 18*, 2018.

[28] Chris Leary and Todd Wang. XLA: TensorFlow, compiled. TensorFlow Dev Summit, 2017.

[29] Paul Feautrier. Some efficient solutions to the affine scheduling problem. i. one-dimensional time. *International journal of parallel programming*, 21(5):313–347, 1992.

[30] Paul Feautrier. Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. *International journal of parallel programming*, 21(6):389–420, 1992.

[31] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. Rethinking NOCs for spatial neural network accelerators. In *2017 Eleventh IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 1–8. IEEE, 2017.

[32] Martino Dazzi, Abu Sebastian, Pier Andrea Francese, Thomas Parnell, Luca Benini, and Evangelos Eleftheriou. 5 Parallel Prism: A topology for pipelined implementations of convolutional neural networks using computational memory. https://arxiv.org/abs/1906.03474, 2019.

[33] Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: Open neural network exchange. https://github.com/onnx/onnx, 2019.

[34] Jason Cong and Bingjun Xiao. Minimizing computation in convolutional neural networks. In *International conference on artificial neural networks*, pages 281–290, 2014.

[35] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[36] Sven Verdoolaege. Presburger formulas and polyhedral compilation. https://lirias.kuleuven.be/retrieve/361209, 2016.

[37] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In *International Congress on Mathematical Software*, pages 299–302, 2010.

[38] Yu Ji, Youhui Zhang, Wenguang Chen, and Yuan Xie. Bridge the gap between neural networks and neuromorphic hardware with a neural network compiler. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 448–460, 2018.

[39] Yu Ji, YouHui Zhang, ShuangChen Li, Ping Chi, CiHang Jiang, Peng Qu, Yuan Xie, and WenGuang Chen. NEUTRAMS: neural network transformation and co-design under neuromorphic hardware constraints. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[40] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018.

[41] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 754–768, New York, NY, USA, 2019. Association for Computing Machinery.

[42] M. Alwani, H. Chen, M. Ferdman, and P. Milder. Fused-layer cnn accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, Oct 2016.

[43] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 101–113, 2008.

[44] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. When polyhedral transformations meet simd code generation. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 127–138, 2013.

[45] Tobias Grosser and Torsten Hoefler. Polly-acc transparent compilation to heterogeneous hardware. In *Proceedings of the 2016 International Conference on Supercomputing*, pages 1–13, 2016.

[46] Kedar S Namjoshi and Nimit Singhania. Loopy: Programmable and formally verified loop transformations. In *International Static Analysis Symposium*, pages 383–402. Springer, 2016.

[47] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noel Pouchet, and Ponnuswamy Sadayappan. Analytical modeling of cache behavior for affine programs. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–26, 2017.

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

[49] Auguste Olivry, Julien Langou, Louis-Noël Pouchet, P Sadayappan, and Fabrice Rastello. Automated derivation of parametric data movement lower bounds for affine programs. *arXiv preprint arXiv:1911.06664*, 2019.

[50] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–205. IEEE, 2019.

[51] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. https://arxiv.org/abs/1802.04730, 2018.

[52] Tim Zerrell and Jeremy Bruestle. Stripe: Tensor compilation via the nested polyhedral model. 2019.

[53] MLIR affine dialect. https://mlir.llvm.org/docs/Dialects/Affine/. Retrieved Feb 2020.

[54] Sven Verdoolaege. Integer set library: Manual. *Tech. Rep.*, 2011.