Compiling Neural Networks for a Computational Memory Accelerator

Kornilios Kourtis ¹  Martino Dazzi ²  Nikolas Ioannou ²  Tobias Grosser ³
Abu Sebastian ²  Evangelos Eleftheriou ²

¹ Independent  ² IBM Research  ³ ETH Zurich

April 27, 2020
Introduction

- Traditional HW designs have reached their limits
- Applications that require improved performance, turn to specialized HW
Introduction

- Traditional HW designs have reached their limits
- Applications that require improved performance, turn to specialized HW

A notable application domain where above applies is Neural Networks (NNs)
- widely used
- specific (not general purpose) computations
Introduction

- Traditional HW designs have reached their limits
- Applications that require improved performance, turn to specialized HW

A notable application domain where above applies is Neural Networks (NNs)
- widely used
- specific (not general purpose) computations

As a result, many attempts to accelerate their performance
- GPUs, ASICs, FPGAs
- Computational Memory: exploit the physical attributes of the memory devices to perform computations at the place where data are stored.
Computational Memory (CM)

Basic unit is a meristive crossbar array that can:

- store a matrix $M$
- perform an analog matrix vector ($M \times v$) operation

Benefits:
- $M \times v$ can be executed in a single step (while digital logic typically requires multiple steps)
- reduced communication (main challenge for data-intensive workloads)
What's unique about a CM accelerator?

- Traditional NN accelerators work "one layer at a time"
- CM accelerators are built with technologies such as PCM or Flash
  - reprogramming the crossbars might take as long as one minute
- NN should be fully mapped onto the CM accelerator
What's unique about a CM accelerator?

- Traditional NN accelerators work "one layer at a time"
- CM accelerators are built with technologies such as PCM or Flash
  - reprogramming the crossbars might take as long as one minute
- NN should be fully mapped onto the CM accelerator

Our goal is to co-design the hardware and the software stack for a CM accelerator for NNs.
Outline

1. Hardware: CM accelerator
   ▶ Chip comprising multiple cores, each including a crossbar
   ▶ (explicit) Dataflow engine
Outline

1. Hardware: CM accelerator
   - Chip comprising multiple cores, each including a crossbar
   - (explicit) Dataflow engine

2. Software: Compiler for mapping arbitrary NNs onto the chip
   - software architecture
   - implementing dependency control between the cores
Outline

1. Hardware: CM accelerator
   ▶ Chip comprising multiple cores, each including a crossbar
   ▶ (explicit) Dataflow engine

2. Software: Compiler for mapping arbitrary NNs onto the chip
   ▶ software architecture
   ▶ implementing dependency control between the cores

Scope:
   ▶ Inference, specifically on the edge
   ▶ Convolutional NNs (CNNs)
CM core

Interconnect
CM core

- XBAR: analog crossbar, MxV

Interconnect
CM core

- XBAR: analog crossbar, MxV
- DPU: digital processing unit

Interconnect
- XBAR: analog crossbar, MxV
- DPU: digital processing unit
- MEM: local memory
CM core

- XBAR: analog crossbar, MxV
- DPU: digital processing unit
- MEM: local memory
- LCU: local control unit

Interconnect
LCU transfers data from MEM to XBAR, and initiates crossbar operation.
CM core

1. LCU transfers data from MEM to XBAR, and initiates crossbar operation.
2. XBAR output is made available to DPU, which executes its instructions. (non MxV operations.)
1. LCU transfers data from MEM to XBAR, and initiates crossbar operation.
2. XBAR output is made available to DPU, which executes its instructions. (non MxV operations.)
3. DPU may load and store data to local memory.
CM core

1. LCU transfers data from MEM to XBAR, and initiates crossbar operation.
2. XBAR output is made available to DPU, which executes its instructions. (non MxV operations.)
3. DPU may load and store data to local memory
4. Data from local memory may be transferred to other cores via the interconnect.
LCU transfers data from MEM to XBAR, and initiates crossbar operation.

XBAR output is made available to DPU, which executes its instructions. (Non MxV operations.)

DPU may load and store data to local memory.

Data from local memory may be transferred to other cores via the interconnect.

Data via the interconnect arrive at local memory, and act as input to LCU’s state machine (①) which may trigger the next operation (②).
CM chip

- GMEM: chip memory
- GCU: Global Control Unit orchestrates data transfers between external (e.g., host) memory and GMEM, as well as between GMEM and cores-local memory.
- Interconnect network
Executing CNNs on the CM accelerator

- Convolutions are mapped to the crossbar’s MxV operation
- Everything else (e.g., activation functions) is executed on the DPU
- CNN layers are assigned to CM cores, forming a pipeline
Compiling NNs for the CM accelerator

**Compilation:**
- Input: an NN model
  - a dataflow graph of operators (e.g., convolution, ReLU, etc.)
  - values for the weights
- Output:
  - configuration for the LCUs, GCU
  - instructions for the DPU
Compilation steps

- Partitioning and Mapping
  partition the NN dataflow graph and map each partition to a CM core, respecting interconnect constrains.
Compilation steps

- Partitioning and Mapping
  partition the NN dataflow graph and map each partition to a CM core, respecting interconnect constrains.

- Lowering
  For each partition, produce the corresponding configurations for LCUs and DPUs
  - DPU configuration: a set of instructions
  - LCU configuration: a state machine
LCU state machine

- snoops remote writes from other cores (or GCU)
- loads necessary data to crossbar
- triggers local computations
  (only when dependencies are satisfied)

How do we configure it?
Modeling dependencies

- We need to model the dependencies of the local computation

\[ \{ \text{for } i_1 \ldots \} \] \{ \text{for } i_2 \ldots \} \ldots

\text{Iterations } I

\[ \text{Array } O \]

\{ \text{for } j_1 \ldots \} \} \{ \text{for } j_2 \ldots \} \ldots

\text{Iterations } J

\begin{align*}
\text{Core 1} & \quad W_1 \quad R_2 \quad \text{Core 2}
\end{align*}

ISL Example:
\[ \{ \text{CONV_MXV}[oh,ow] \rightarrow \text{inp}[id,ih,iw] : \\
0 \leq oh < OH \quad \text{and} \\
0 \leq ow < OW \quad \text{and} \\
0 \leq id < D \quad \text{and} \\
oh \leq ih < oh + FH \quad \text{and} \\
ow \leq iw < ow + FW \} \]
Modeling dependencies

- We need to model the dependencies of the local computation

Polyhedral model:
- allows reasoning about nested loops computations that access multi-dimensional arrays
- works well with NN operations
Modeling dependencies

- We need to model the dependencies of the local computation

Polyhedral model:
- allows reasoning about nested loops computations that access multi-dimensional arrays
- works well with NN operations
- We use ISL, which represents computations as Presburger sets and relations

ISL Example:
\[
\{ \text{CONV\_MXV}[oh, ow] \rightarrow \text{inp}[id, ih, iw] : \\
\quad 0 \leq oh < OH \\
\quad \text{and } 0 \leq ow < OW \\
\quad \text{and } 0 \leq id < D \\
\quad \text{and } oh \leq ih < oh + FH \\
\quad \text{and } ow \leq iw < ow + FW \} 
\]
we use ISL to compute relation $S$

$S$ maps observed writes in array $O$ to the maximum iteration in $J$ that can be executed.

we use $S$ to generate code for the LCU state machine
LCU state machine with polyhedral model

- we use ISL to compute relation $S$
- $S$ maps observed writes in array $O$ to the maximum iteration in $J$ that can be executed.
- we use $S$ to generate code for the LCU state machine

(more details can be found on the paper and https://github.com/IBM/cmnnc.)
Conclusion

- A first step towards compiling NNs for a CM accelerator.
- SW / HW architecture
- tracking dependencies using polyhedral compilation

Open questions / challenges
- What is the HW/SW interface?
- What happens if the NN does not fit the accelerator?
- Quantization
- Breaking up operations that do not fit into a single CM core

Our prototype can be found at https://github.com/IBM/cmnnc.