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
- specialized (not general purpose) computations, expressed as dataflow graphs

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
- specialized (not general purpose) computations, expressed as dataflow graphs

As a result, many attempts to accelerate their performance

- GPUs (e.g., NVIDIA's cuDNN), ASICs (e.g., Google TPU), FPGAs (e.g., Microsoft Brainwave)

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
- specialized (not general purpose) computations, expressed as dataflow graphs

As a result, many attempts to accelerate their performance

- GPUs (e.g., NVIDIA's cuDNN), ASICs (e.g., Google TPU), FPGAs (e.g., Microsoft Brainwave)
- We use **Computational Memory**

Computational Memory

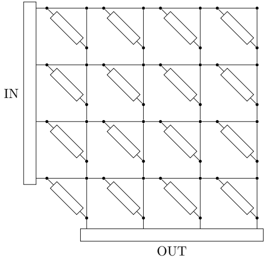
Exploit the physical attributes of the memory devices to perform computations at the place where data are stored.

(In contrast with traditional designs where computation and memory are separate.)

Computational memory (CM) crossbar

Basic unit is a memristive crossbar array that can:

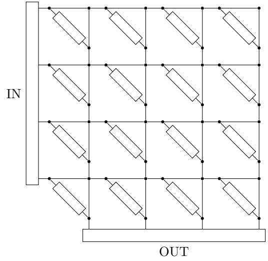
- store a matrix M
- perform an analog matrix vector multiplication ($M \times v$) operation (input: v, output: $M \times v$)



Computational memory (CM) crossbar

Basic unit is a memristive crossbar array that can:

- store a matrix M
- perform an analog matrix vector multiplication ($M \times v$) operation (input: v, output: $M \times v$)



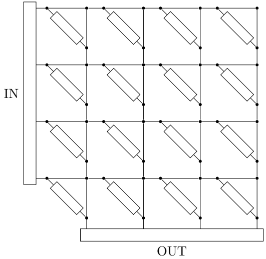
Benefits:

- $M \times v$ can be executed in a single step (while digital logic typically requires multiple steps)
- reduced communication (main challenge for data-intensive workloads)

Computational memory (CM) crossbar

Basic unit is a memristive crossbar array that can:

- store a matrix M
- perform an analog matrix vector multiplication ($M \times v$) operation (input: v, output: $M \times v$)



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)

Our CM accelerator comprises multiple cores with such crossbars

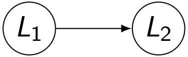
What about software?

- Traditional accelerators use data parallelism

What about software?

- Traditional accelerators use data parallelism

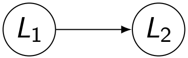
NN dataflow graph



What about software?

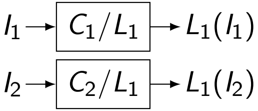
- Traditional accelerators use <u>data parallelism</u>

NN dataflow graph



Data parallel execution on two cores

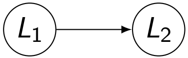
step 1:



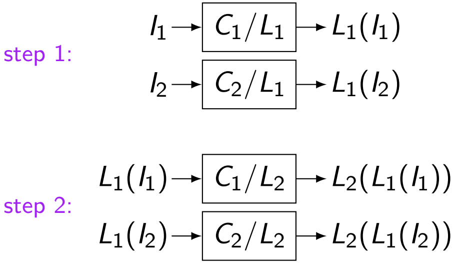
What about software?

- Traditional accelerators use data parallelism

NN dataflow graph



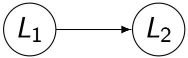
Data parallel execution on two cores



What about software?

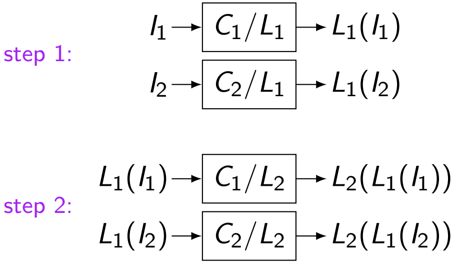
- Traditional accelerators use data parallelism
- This will not work for the CM accelerator
  - built with PCM (or Flash)
  - reprogramming crossbars takes too long

NN dataflow graph



$L_1 \rightarrow L_2$

Data parallel execution on two cores



step 1:

$I_1 \rightarrow \boxed{C_1/L_1} \rightarrow L_1(I_1)$

$I_2 \rightarrow \boxed{C_2/L_1} \rightarrow L_1(I_2)$

step 2:

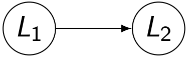
$L_1(I_1) \rightarrow \boxed{C_1/L_2} \rightarrow L_2(L_1(I_1))$

$L_1(I_2) \rightarrow \boxed{C_2/L_2} \rightarrow L_2(L_1(I_2))$

What about software?

- Traditional accelerators use data parallelism
- This will not work for the CM accelerator
  - built with PCM (or Flash)
  - reprogramming crossbars takes too long
- Instead, we use pipeline parallelism

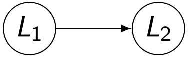
NN dataflow graph



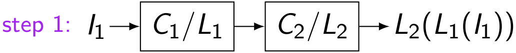
What about software?

- Traditional accelerators use data parallelism
- This will not work for the CM accelerator
  - built with PCM (or Flash)
  - reprogramming crossbars takes too long
- Instead, we use pipeline parallelism

NN dataflow graph



Pipeline execution on two cores



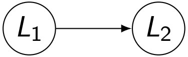
What about software?

- Traditional accelerators use data parallelism

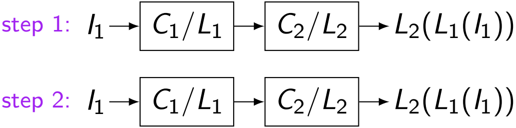
- This will not work for the CM accelerator
  - built with PCM (or Flash)
  - reprogramming crossbars takes too long

- Instead, we use pipeline parallelism

NN dataflow graph



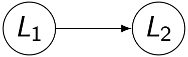
Pipeline execution on two cores



What about software?

- Traditional accelerators use data parallelism
- This will not work for the CM accelerator
  - built with PCM (or Flash)
  - reprogramming crossbars takes too long
- Instead, we use pipeline parallelism

NN dataflow graph



Pipeline execution on two cores

step 1: $I_1 \rightarrow \boxed{C_1/L_1} \rightarrow \boxed{C_2/L_2} \rightarrow L_2(L_1(I_1))$

step 2: $I_1 \rightarrow \boxed{C_1/L_1} \rightarrow \boxed{C_2/L_2} \rightarrow L_2(L_1(I_1))$

- Existing compilers (e.g., Glow, TVM, XLA) offer no help for pipeline parallelism

Outline

Our goal is build a SW stack for a CM accelerator for NNs

- co-design SW with HW

Outline

Our goal is build a SW stack for a CM accelerator for NNs

- co-design SW with HW

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

Outline

Our goal is build a SW stack for a CM accelerator for NNs

- co-design SW with HW

1. Hardware: CM accelerator
   - Chip comprising multiple cores, each including a crossbar
   - (explicit) Dataflow engine
2. Software: Compiler for mapping aribtrary NNs onto the chip
   - software architecture
   - **implementing dependency control between the cores**

Outline

Our goal is build a SW stack for a CM accelerator for NNs

- co-design SW with HW

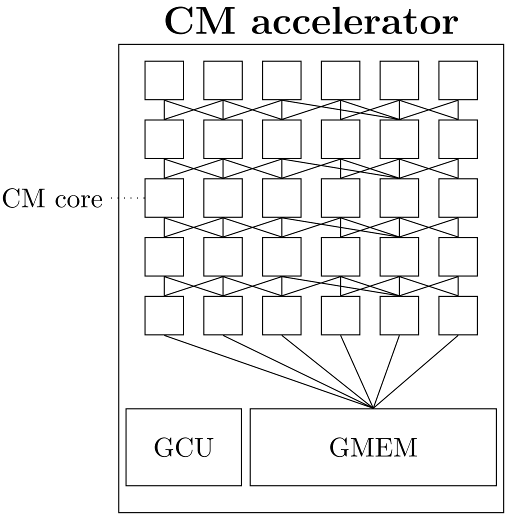
1. Hardware: CM accelerator
   - Chip comprising multiple cores, each including a crossbar
   - (explicit) Dataflow engine
2. Software: Compiler for mapping aribtrary NNs onto the chip
   - software architecture
   - **implementing dependency control between the cores**

Scope:

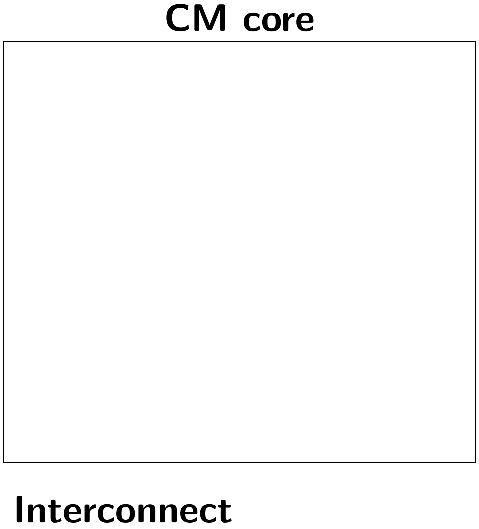
- Convolutional NNs (CNNs)
- Inference, specifically on the edge

CM accelerator (chip)

- CM Cores
- 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



CM core

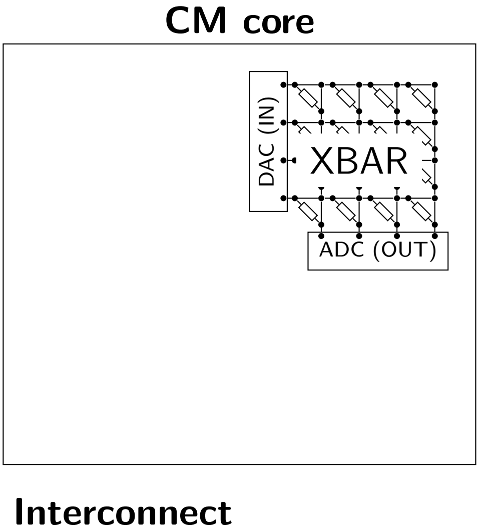


CM core

Interconnect

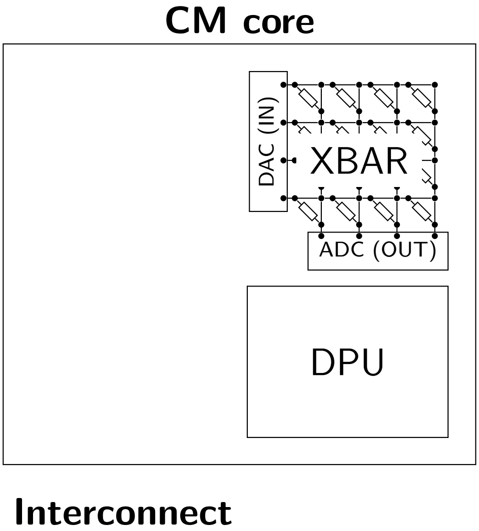
CM core

- XBAR: analog crossbar, MxV



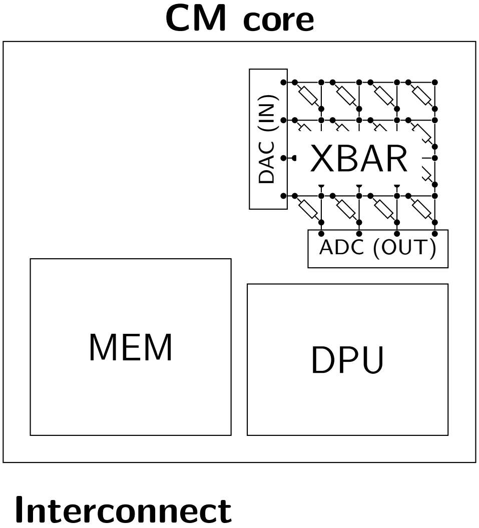
CM core

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



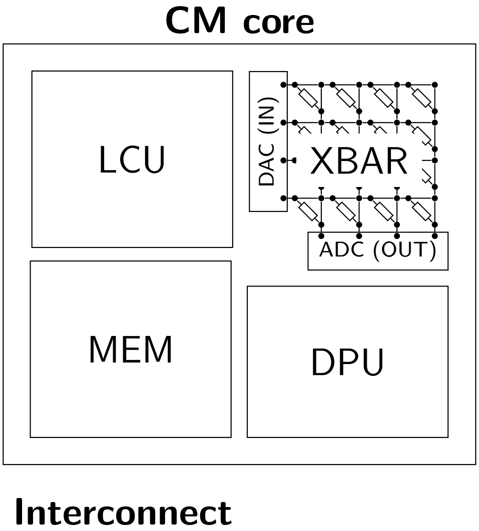
CM core

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



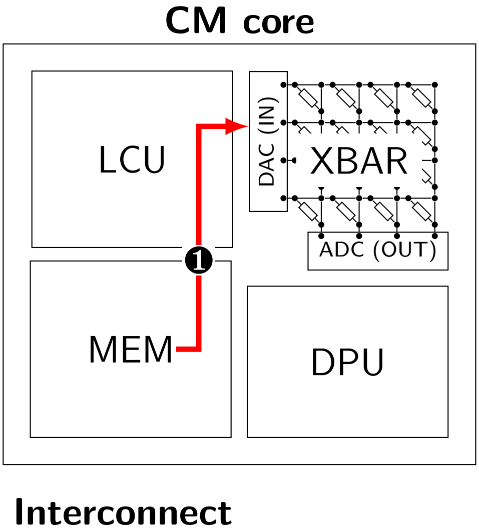
CM core

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



CM core

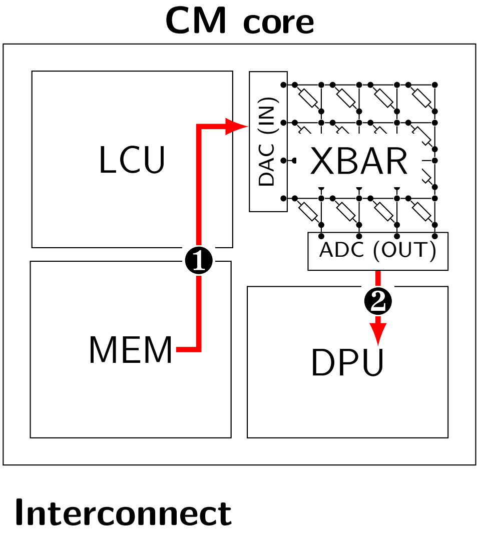
❶ LCU transfers data from MEM to XBAR, and initiates crossbar operation.



CM core

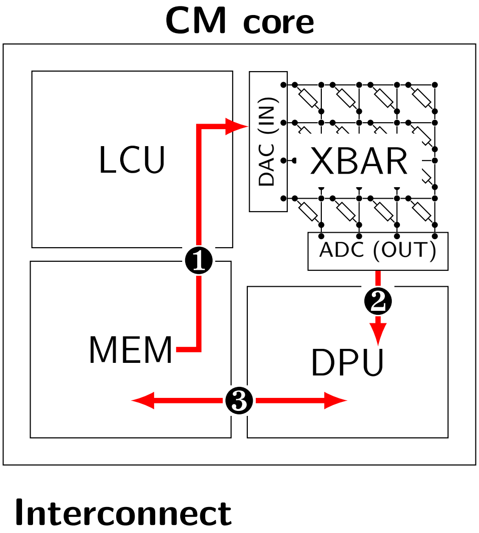
❶ LCU transfers data from MEM to XBAR, and initiates crossbar operation.

❷ XBAR output is made available to DPU, which executes its instructions.



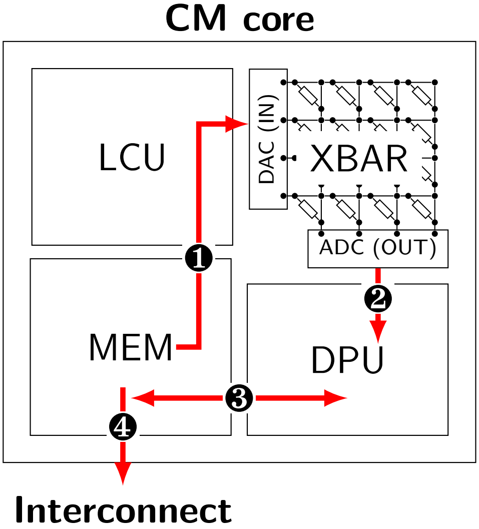
CM core

- ❶ LCU transfers data from MEM to XBAR, and initiates crossbar operation.
- ❷ XBAR output is made available to DPU, which executes its instructions.
- ❸ DPU may load and store data to local memory



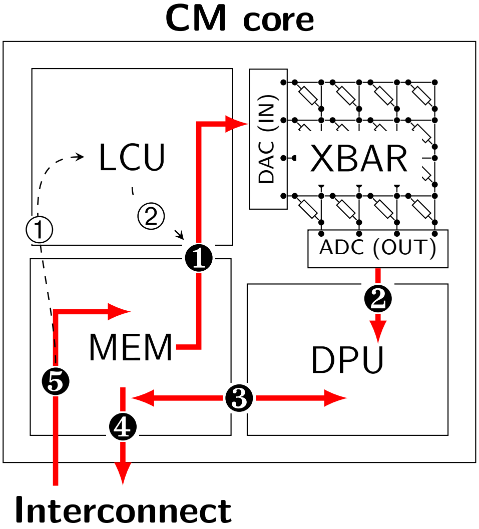
CM core

- ❶ LCU transfers data from MEM to XBAR, and initiates crossbar operation.
- ❷ XBAR output is made available to DPU, which executes its instructions.
- ❸ DPU may load and store data to local memory
- ❹ Data from local memory may be transferred to other cores via the interconnect.



CM core

- ❶ LCU transfers data from MEM to XBAR, and initiates crossbar operation.
- ❷ XBAR output is made available to DPU, which executes its instructions.
- ❸ DPU may load and store data to local memory
- ❹ Data from local memory may be transferred to other cores via the interconnect.
- ❺ 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).

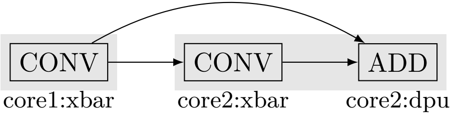


Executing CNNs on the CM accelerator

- 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

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 (e.g., ONNX)
  - 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

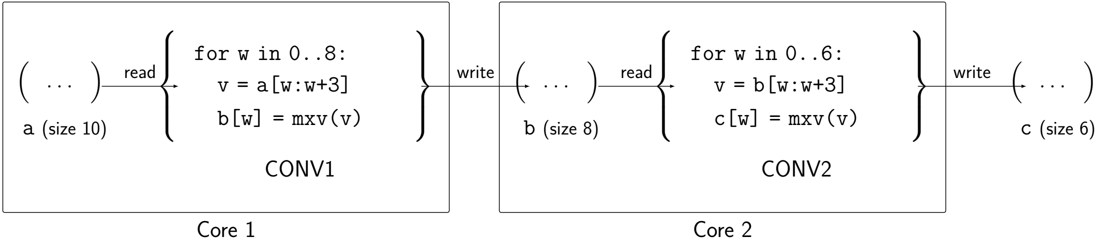
- Partitioning and Mapping
  partition the NN dataflow graph and map each partition to a CM core, respecting interconnect constrains.

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

Data dependencies between cores



(. . .) a (size 10) read

```
for w in 0..8:
    v = a[w:w+3]
    b[w] = mxv(v)
```

CONV1

Core 1

write (. . .) b (size 8) read

```
for w in 0..6:
    v = b[w:w+3]
    c[w] = mxv(v)
```

CONV2

Core 2

write (. . .) c (size 6)

- Core 2 can only start executing after `b[0]`, `b[1]`, `b[3]` are written from Core 1.

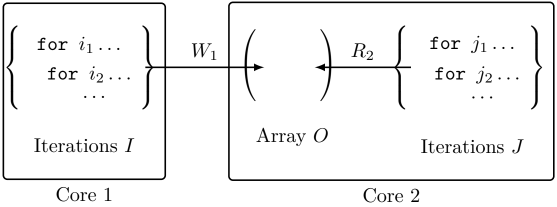
LCU state machine

- 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 computation

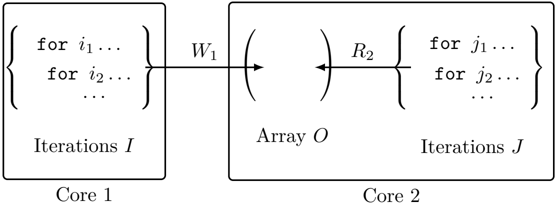


Modeling dependencies

- We need to model the dependencies of the computation

Polyhedral model:

- allows reasoning about nested loops computations that access multi-dimensional arrays
- works well with NN operations



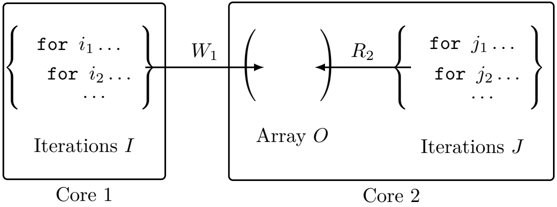
Modeling dependencies

- We need to model the dependencies of the 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

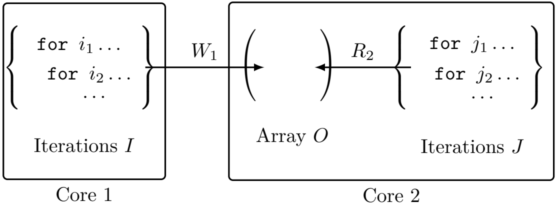


Modeling dependencies

- We need to model the dependencies of the 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

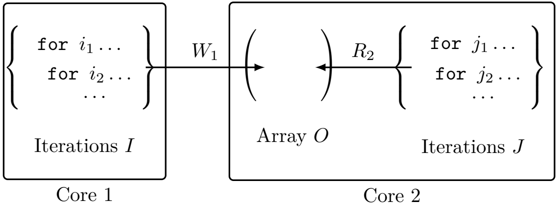


ISL Example: read access relation

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

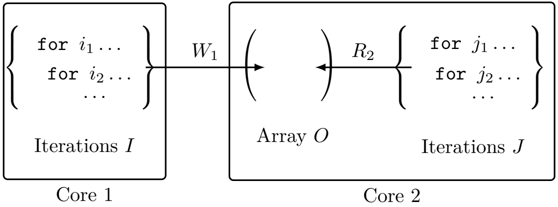
LCU state machine with polyhedral model

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



LCU state machine with polyhedral model

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



(for more details please check our 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`.

Thank you!

Questions?