PIMFlow: Compiler and Runtime Support for CNN Models on Processing-in-Memory DRAM

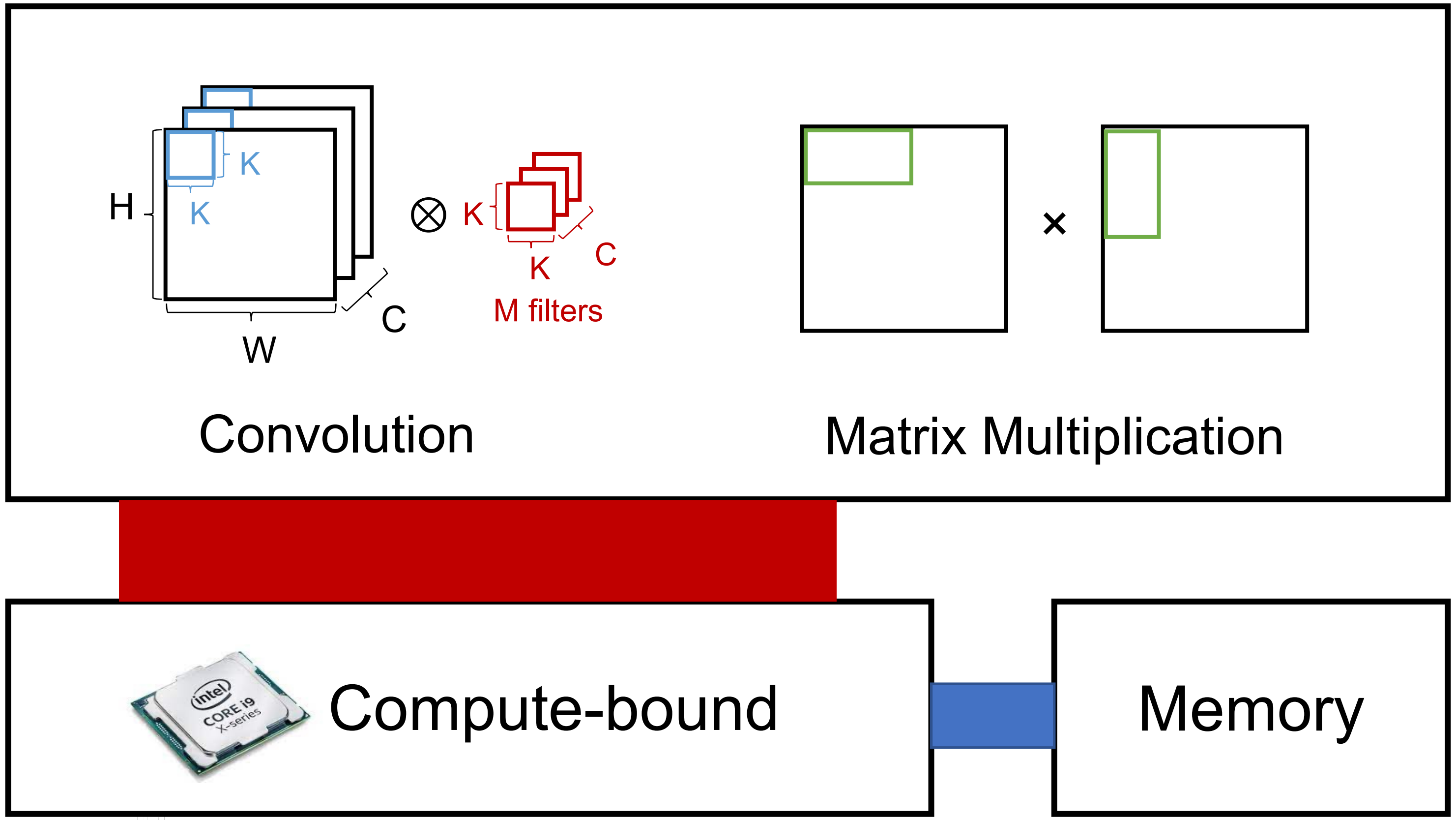
CGO 2023

Yongwon Shin[*,1], Juseong Park[*,2], Sungjun Cho[2], Hyojin Sung[1,2]

[1]Graduate School of AI
[2]Dept. of Computer Science and Engineering
Pohang University of Science and Technology (POSTECH), South Korea
[*]Equal contribution

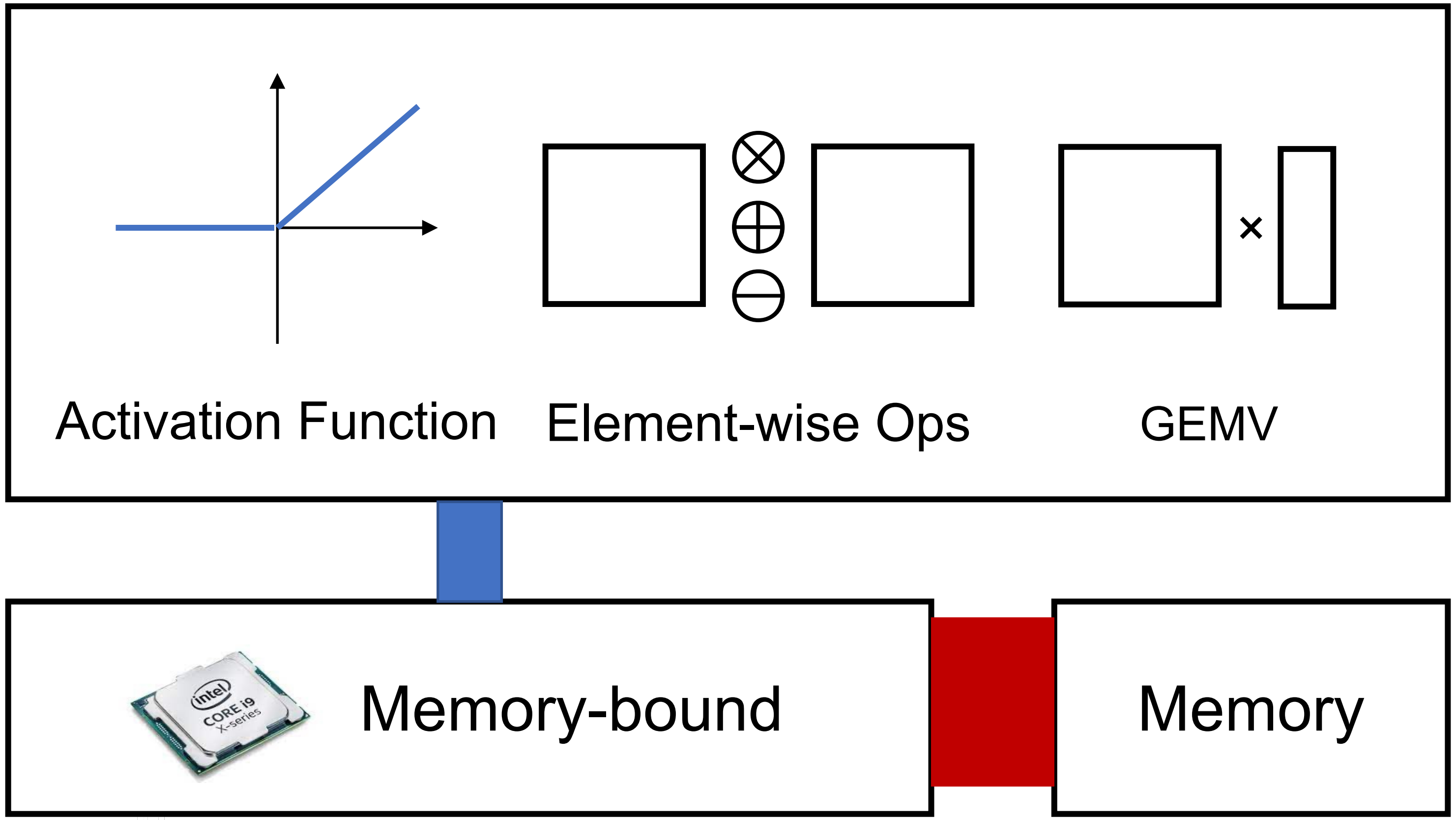
POSTECH

Increasing Demand for Computing Power



- Deep learning models drive FLOPS scaling due to their high computational loads
  - e.g., Convolution in CNN models, Matrix multiplication in Transformer models

Memory-bound Operations in Deep Learning



- Deep learning models also include many memory-intensive layers
  - e.g., activation functions, element-wise operations, FC (fully-connected) layers

Processing-in-Memory (PIM) for Rescue

- **Processing-in-Memory (PIM)** places computing units in or near memory
- Recent proposals from DRAM manufacturers showed the promising potential for commercialization
  - E.g., SK Hynix AiM, Samsung HBM-PIM
  - Focus on accelerating **matrix-vector** and elementwise computations
  - Target models: GPT, speech recognition, recommendation model, etc.

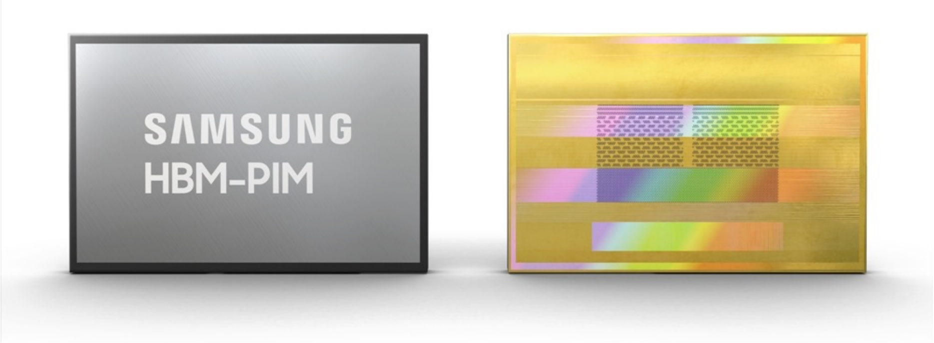
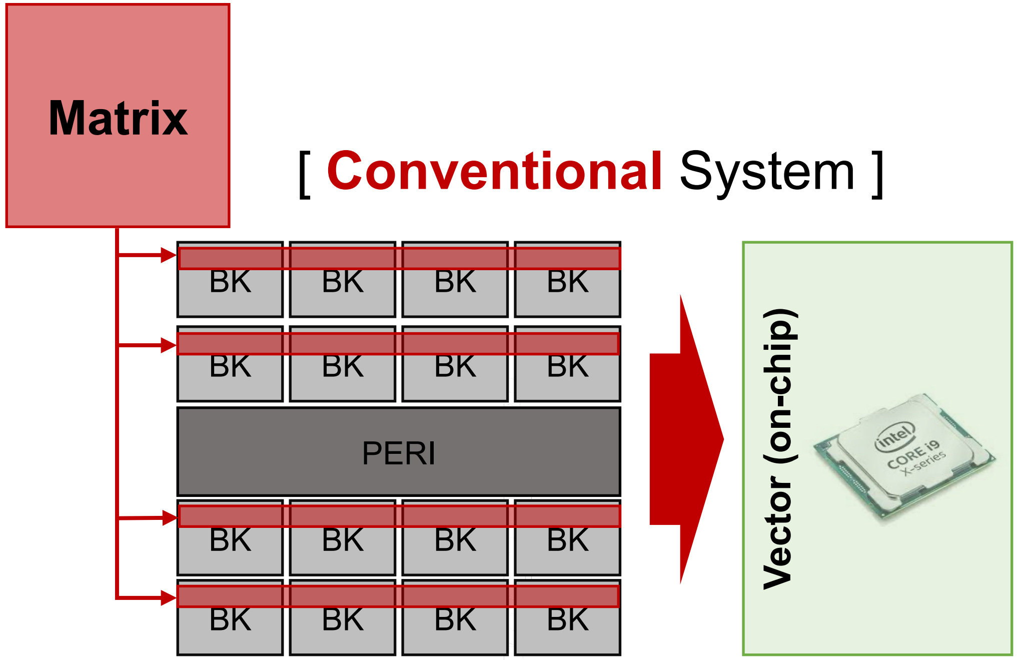


Image sources: https://news.skhynix.com/sk-hynix-develops-pim-next-generation-ai-accelerator/
https://news.samsung.com/global/samsung-brings-in-memory-processing-power-to-wider-range-of-applications

Matrix-Vector Multiplication (MVM)

- **Multiply-Accumulate (MAC)** is a key operation for many DL models
- **Large matrix** is used **only once** after fetched → **Little data reuse**

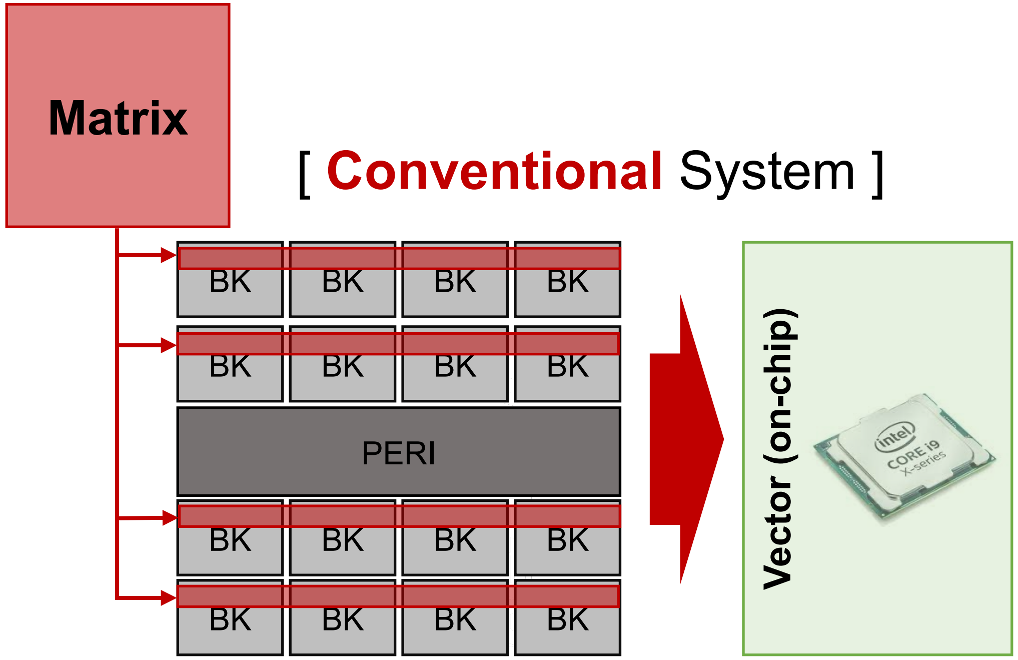


- **Large** data movement (**Matrix**)
- **High** latency & energy consumption

Matrix-Vector Multiplication (MVM)

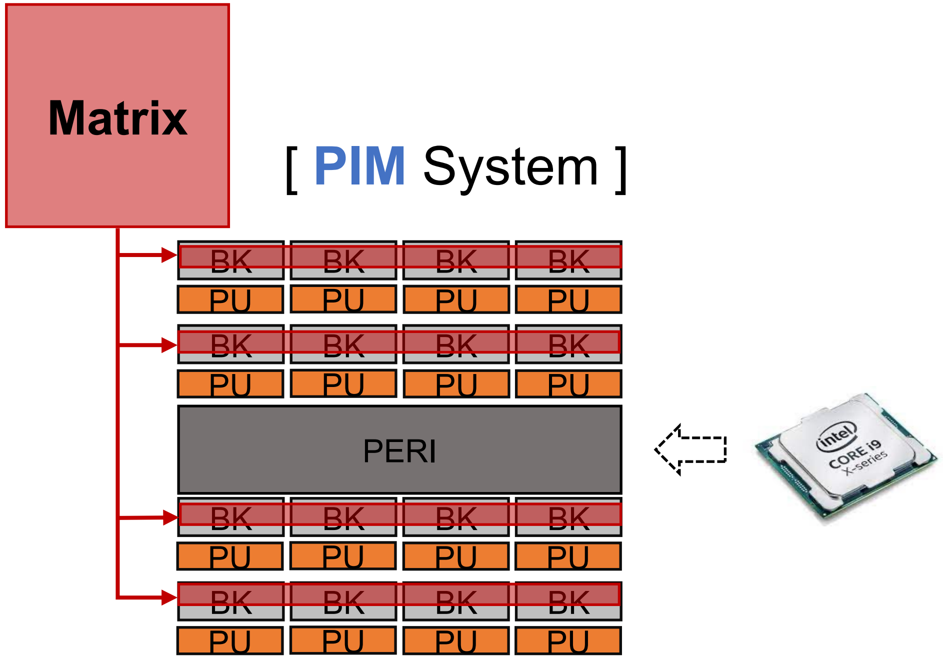
- PIM performs ***in-memory computation*** without moving the matrix
  - Bank-level Processing Units (PUs) compute MVM with data in memory and accumulates the results

[**Conventional** System]



- **Large** data movement (**Matrix**)
- **High** latency & energy consumption

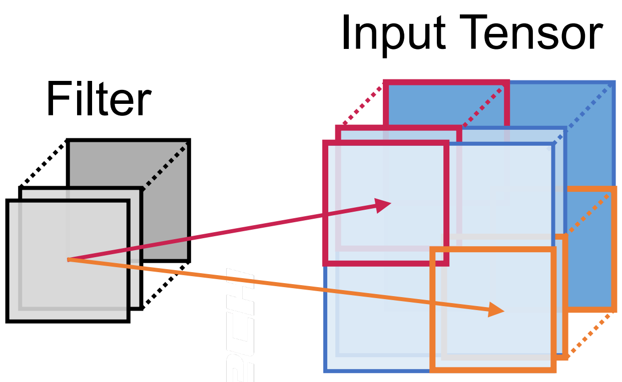
[**PIM** System]



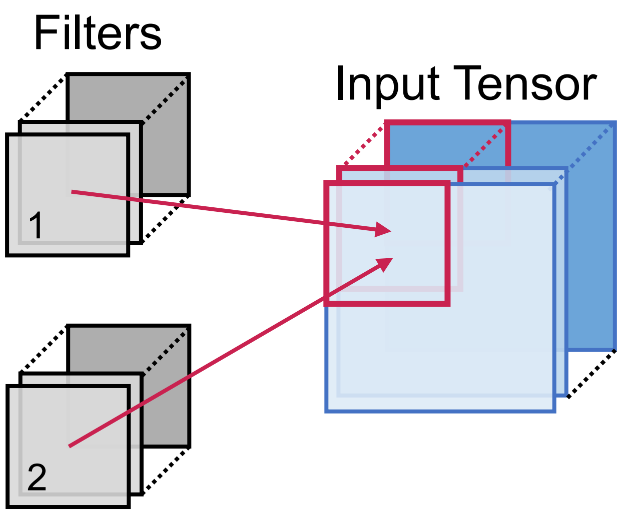
- **No** data movement
- **Low** latency & energy consumption

Unexplored Opportunity: CNN Models

- **Convolution** has multiple data reuse opportunities
  - Tiled data reused in **on-chip memory (scratchpad or cache)**
  - More compute-bound than memory-bound



Activation Reuse
Filter Reuse

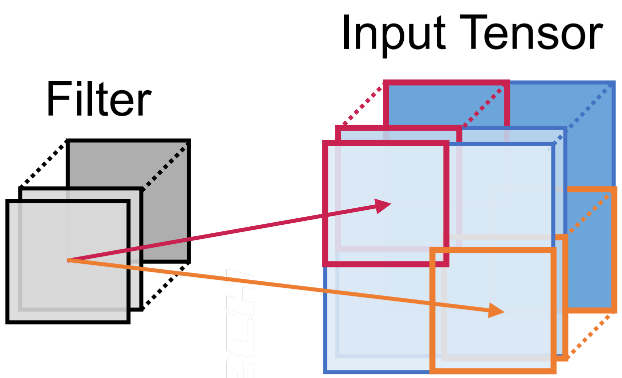


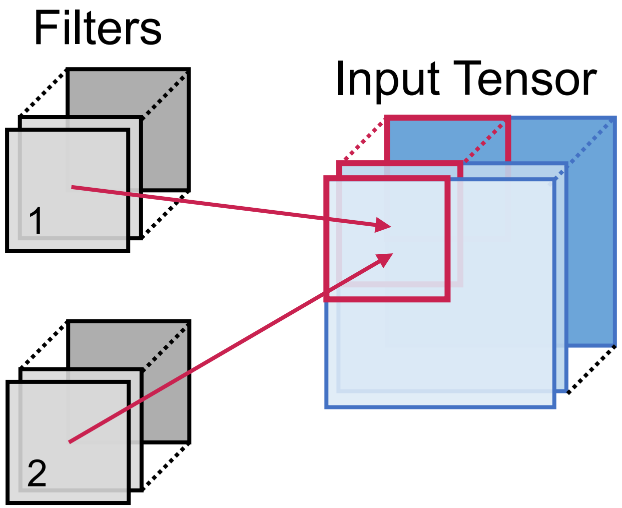
Activation Reuse

* Image adapted from V. Sze, et al., "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," in Proceedings of the IEEE, Dec. 2017

Unexplored Opportunity: CNN Models

- **Convolution** has multiple data reuse opportunities
  - Tiled data reused in **on-chip memory (scratchpad or cache)**
  - More compute-bound than memory-bound



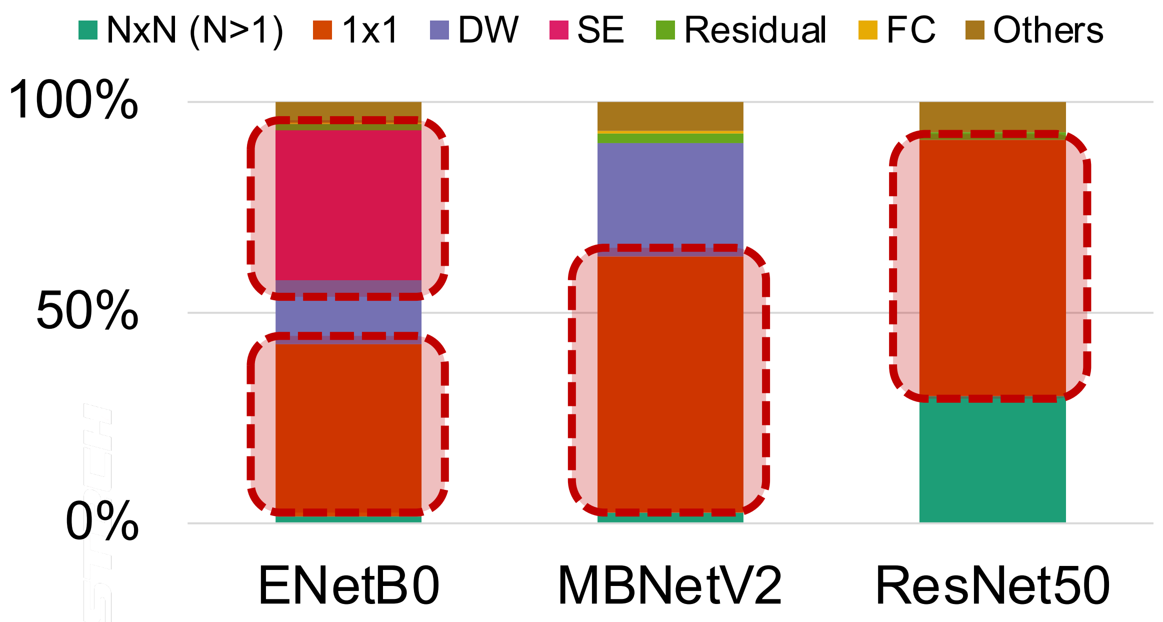


➔ ***have not been considered*** **a primary acceleration target for PIM**

* Image adapted from V. Sze, et al., "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," in Proceedings of the IEEE, Dec. 2017

Unexplored Opportunity: CNN Models

- Recently, CNNs increasingly adopt more **memory-intensive** layers
  - e.g., point-wise convolution (**1x1 CONV**), Squeeze-and-Excitation layer (**SE**)



1x1 CONV and **FC** take *60-80% of runtime* in modern CNNs

Unexplored Opportunity: CNN Models

- Recently, CNNs increasingly adopt more **memory-intensive** layers
  - e.g., point-wise convolution (**1x1 CONV**), Squeeze-and-Excitation layer (**SE**)

Can compiler and runtime support enable CNNs on DRAM-PIM without introducing hardware complexity?

1x1 CONV and **FC** take <u>*60-80% of runtime*</u> in modern CNNs

➔ CNNs can be a **potential PIM acceleration target**

POSTECH

Challenges

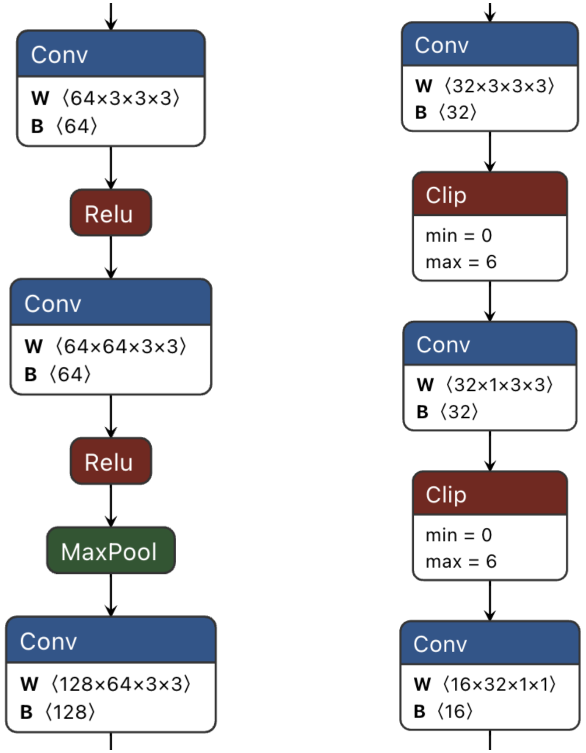
Many convolution layers perform comparably on PIM and GPU

- Exclusive GPU or PIM execution does not provide performance gain

Challenges

Many convolution layers perform comparably on PIM and GPU

CNN inference model graphs do not have much inherent parallelism



Challenges

Many convolution layers perform comparably on PIM and GPU

CNN inference model graphs do not have much inherent parallelism

End-to-end software interfaces to offload CNNs to PIM is crucial

Need DL framework front-end, device scheduling, PIM command generation, ...

PIMFlow

Many convolution layers perform comparably on PIM and GPU

→ **Support parallel execution** across PIM and GPU to reduce runtime

CNN inference model graphs do not have much inherent parallelism

End-to-end software interfaces to offload CNNs to PIM is crucial

PIMFlow

Many convolution layers perform comparably on PIM and GPU

→ **Support parallel execution** across PIM and GPU to reduce runtime

CNN inference model graphs do not have much inherent parallelism

→ Increase parallelism by **PIM-aware graph transformations**

End-to-end software interfaces to offload CNNs to PIM is crucial

PIMFlow

Many convolution layers perform comparably on PIM and GPU

→ **Support parallel execution** across PIM and GPU to reduce runtime

CNN inference model graphs do not have much inherent parallelism

→ Increase parallelism by **PIM-aware graph transformations**

End-to-end software interfaces to offload CNNs to PIM is crucial

→ Integrate **DRAM-PIM back-end** in deep learning compiler (TVM)

PIMFlow

Many convolution layers perform comparably on PIM and GPU

→ **Support parallel execution** across PIM and GPU to reduce runtime

30% (up to 82%) speedup

26% (up to 39%) energy savings

for modern CNNs

→ Increase parallelism by PIM-aware graph transformations

→ Integrate DRAM-PIM back-end in deep learning compiler (TVM)

Outline

Motivation

Overview

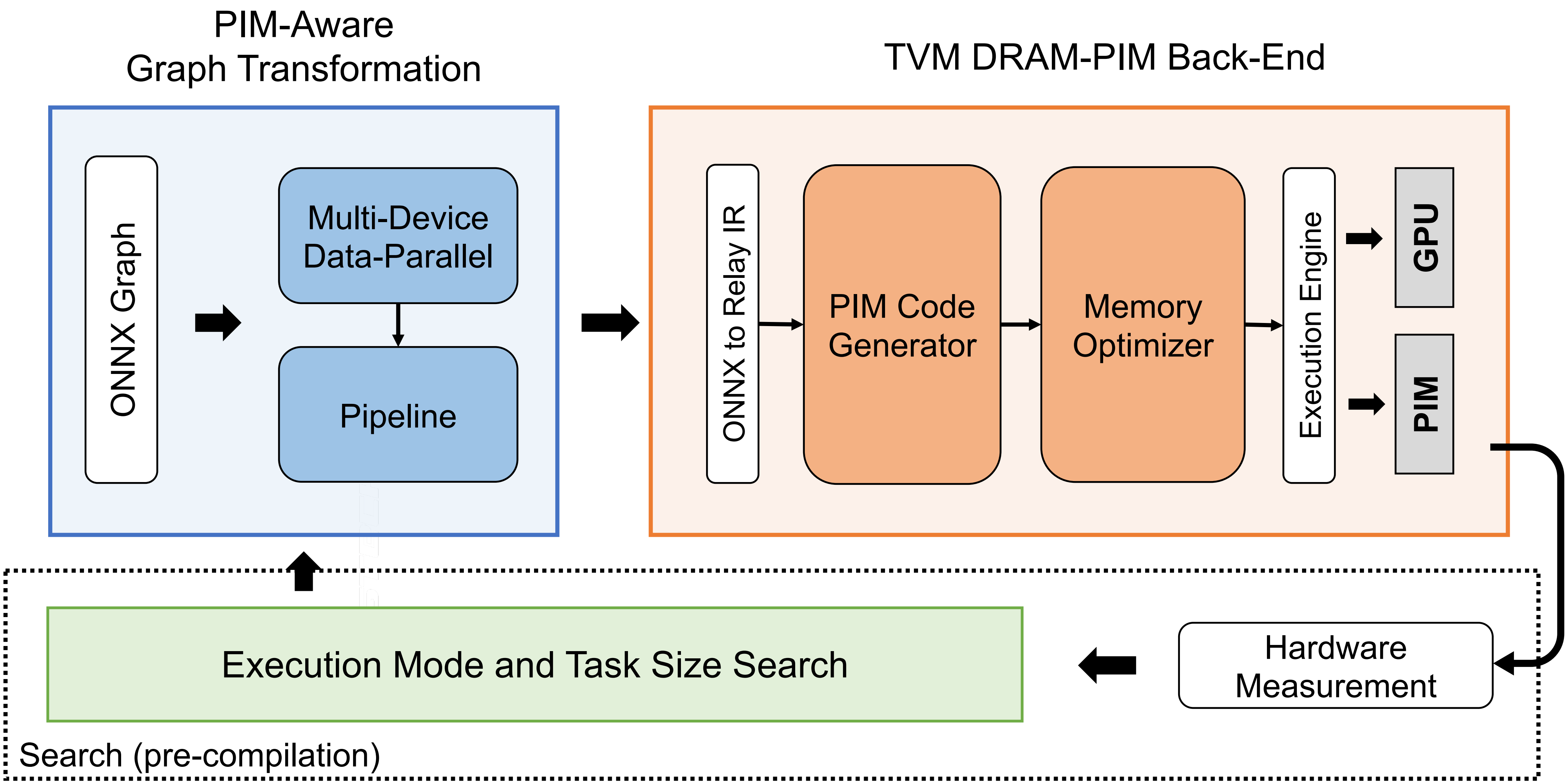
PIM-enabled GPU Memory Architecture

PIMFlow Design and Implementation

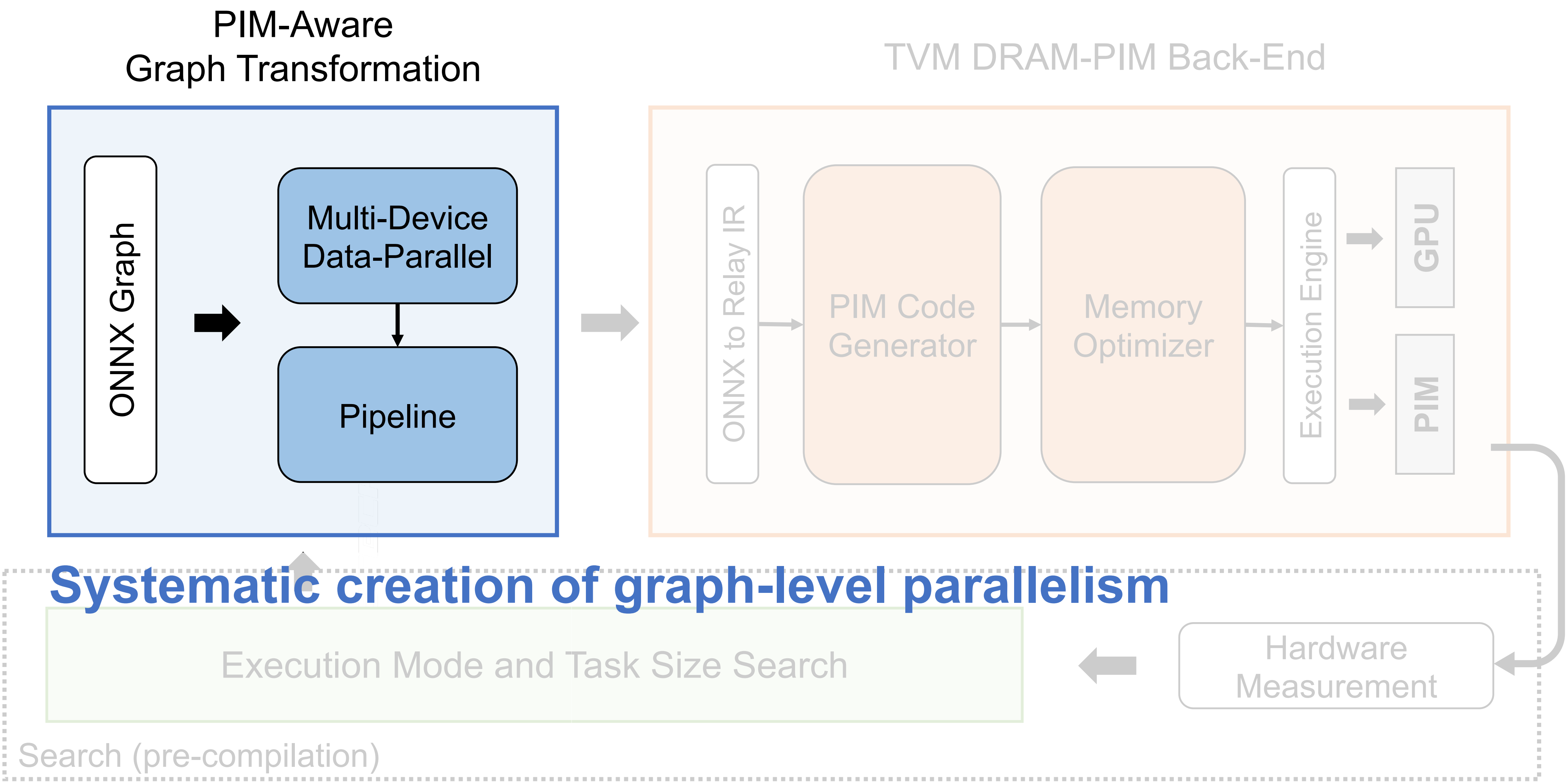
- PIM-Aware Graph Transformation
- Execution Mode and Task Size Search
- TVM Back-End for DRAM-PIM

Evaluation result

Overview

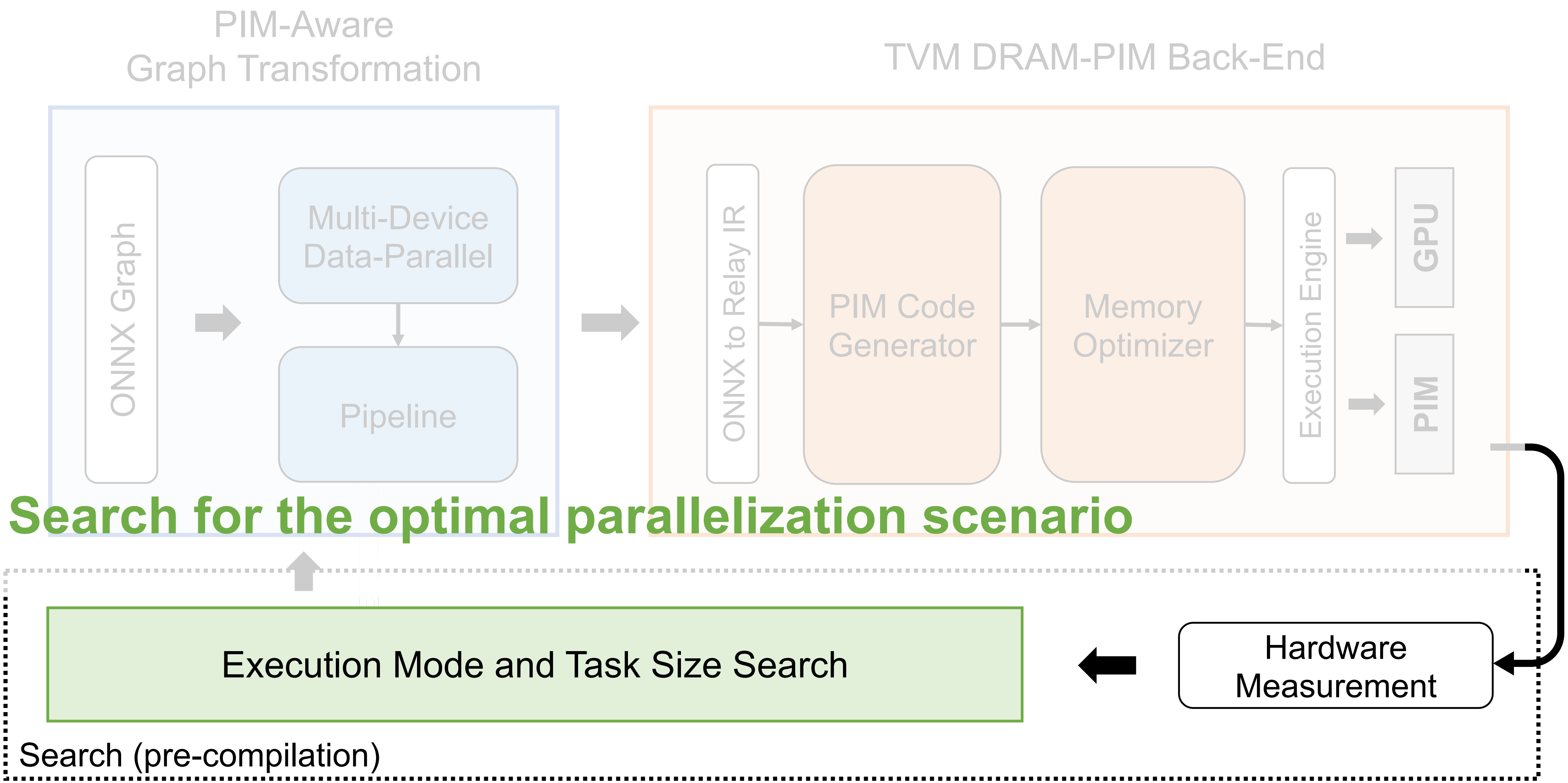

PIM-Aware
Graph Transformation
TVM DRAM-PIM Back-End
ONNX Graph
Multi-Device
Data-Parallel
Pipeline
ONNX to Relay IR
PIM Code
Generator
Memory
Optimizer
Execution Engine
GPU
PIM
Execution Mode and Task Size Search
Hardware
Measurement
Search (pre-compilation)

Overview



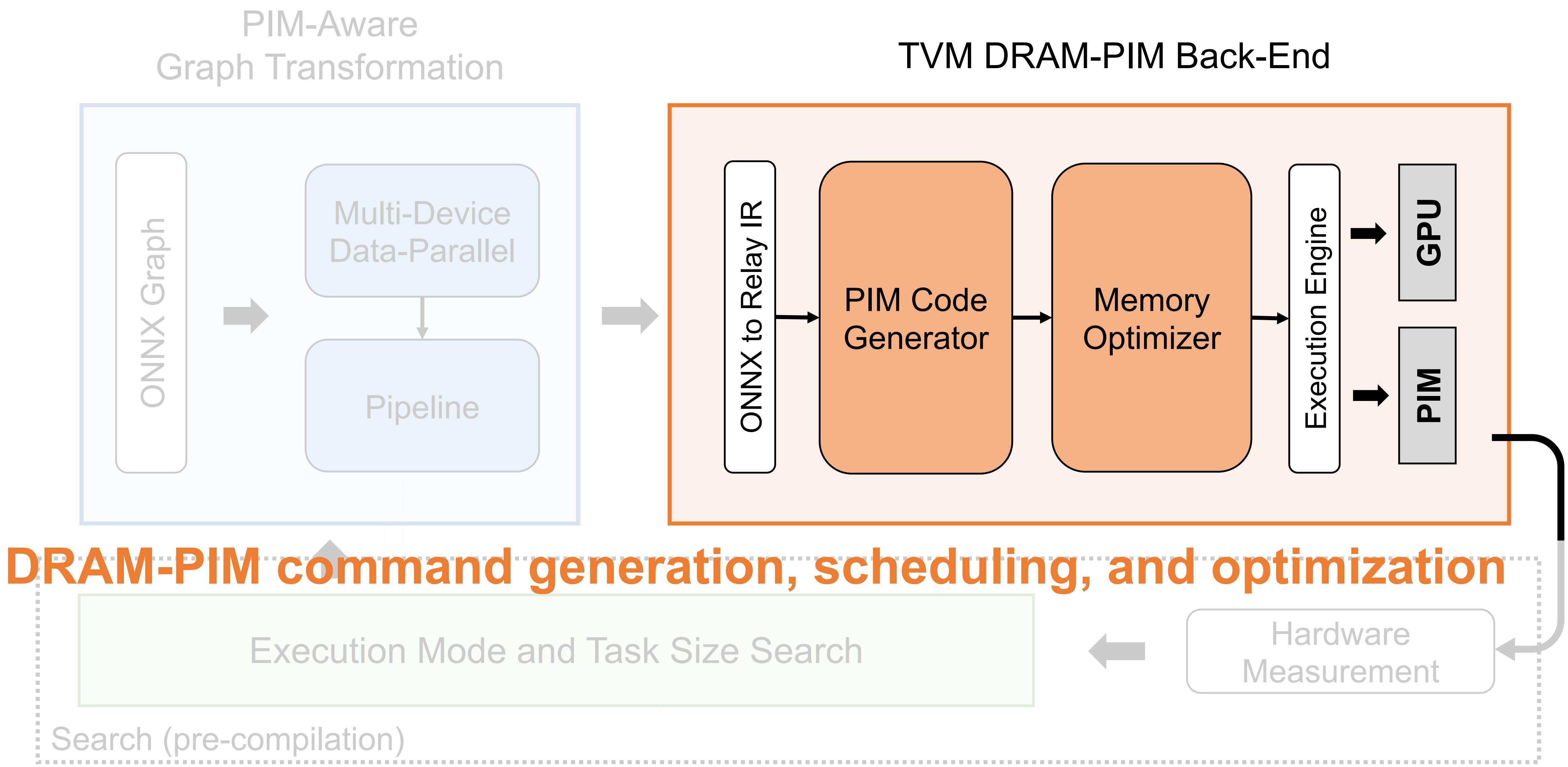
Systematic creation of graph-level parallelism

Overview



PIM-Aware
Graph Transformation
TVM DRAM-PIM Back-End
ONNX Graph
Multi-Device
Data-Parallel
Pipeline
ONNX to Relay IR
PIM Code
Generator
Memory
Optimizer
Execution Engine
GPU
PIM
Search for the optimal parallelization scenario
Execution Mode and Task Size Search
Hardware
Measurement
Search (pre-compilation)

Overview


PIM-Aware
Graph Transformation
ONNX Graph
Multi-Device
Data-Parallel
Pipeline
TVM DRAM-PIM Back-End
ONNX to Relay IR
PIM Code
Generator
Memory
Optimizer
Execution Engine
GPU
PIM
DRAM-PIM command generation, scheduling, and optimization
Execution Mode and Task Size Search
Hardware
Measurement
Search (pre-compilation)

Outline

Motivation

Overview

PIM-enabled GPU Memory Architecture

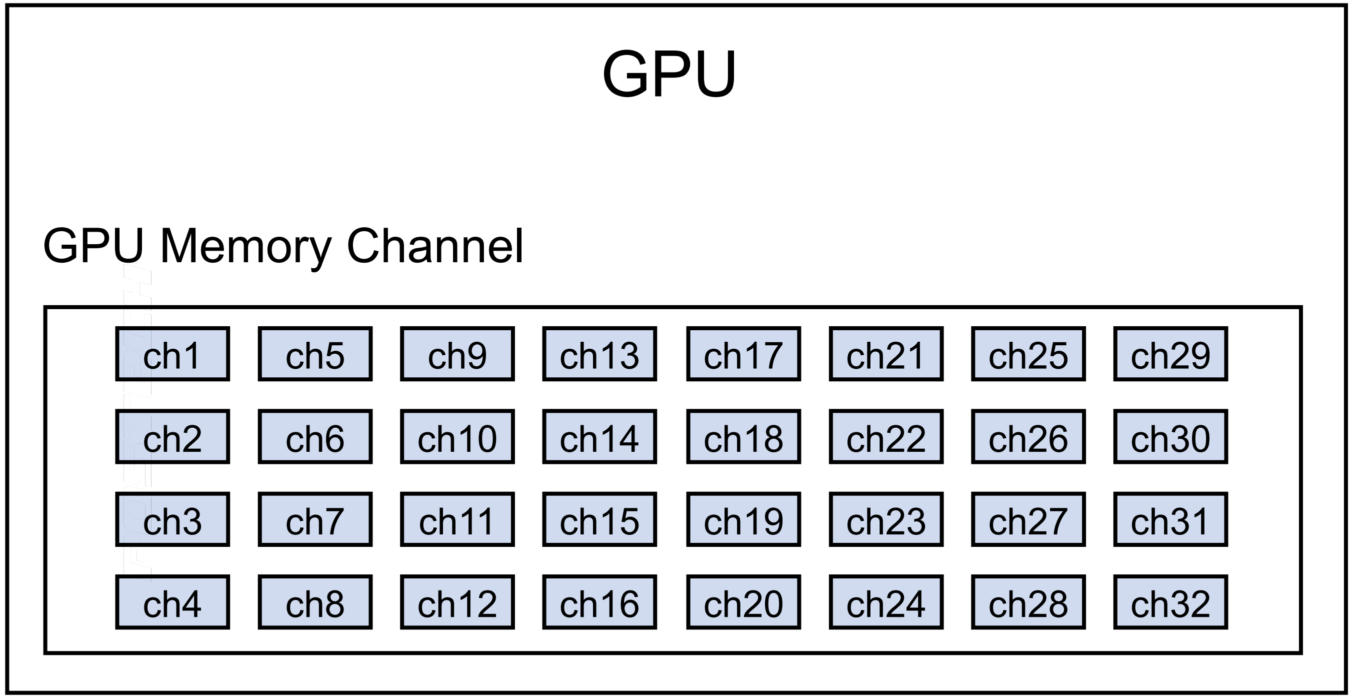
PIMFlow

- PIM-Aware Graph Transformation
- Execution Mode and Task Size Search
- TVM Back-End for DRAM-PIM

Evaluation result

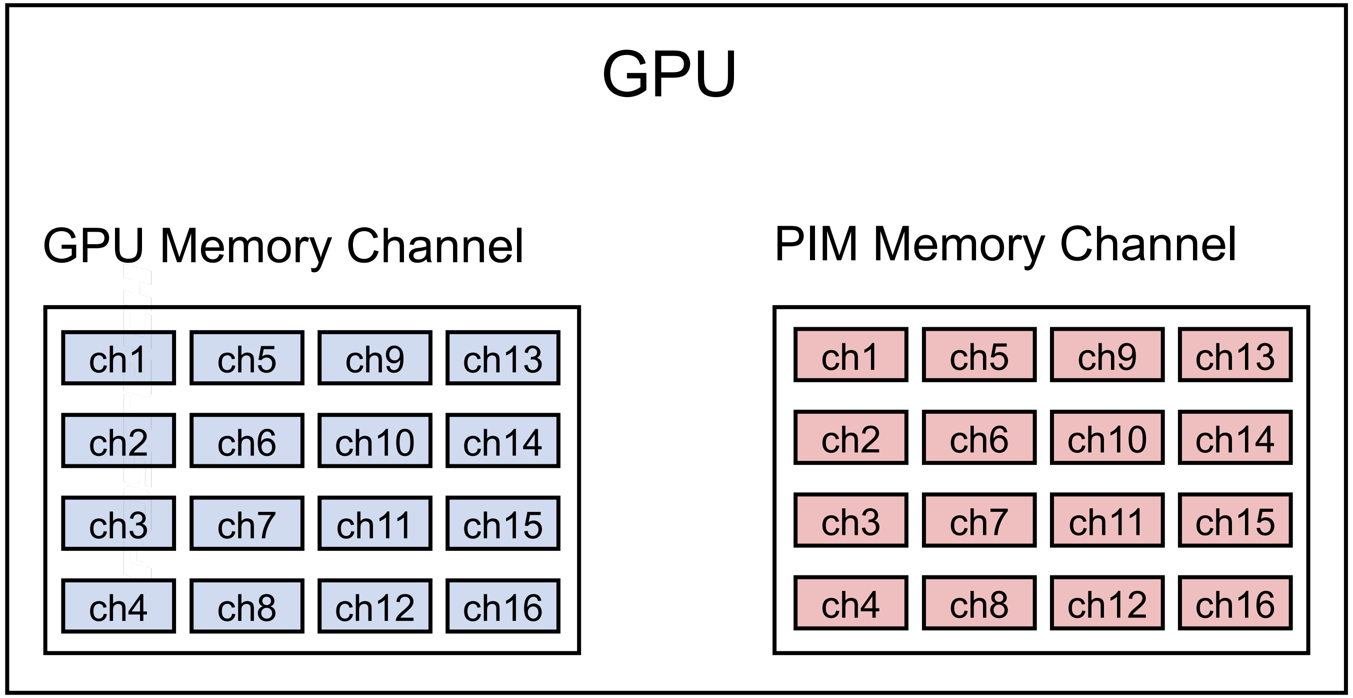
PIM-enabled GPU Memory Architecture

- In a conventional GPU, **all memory channels** are used for GPU



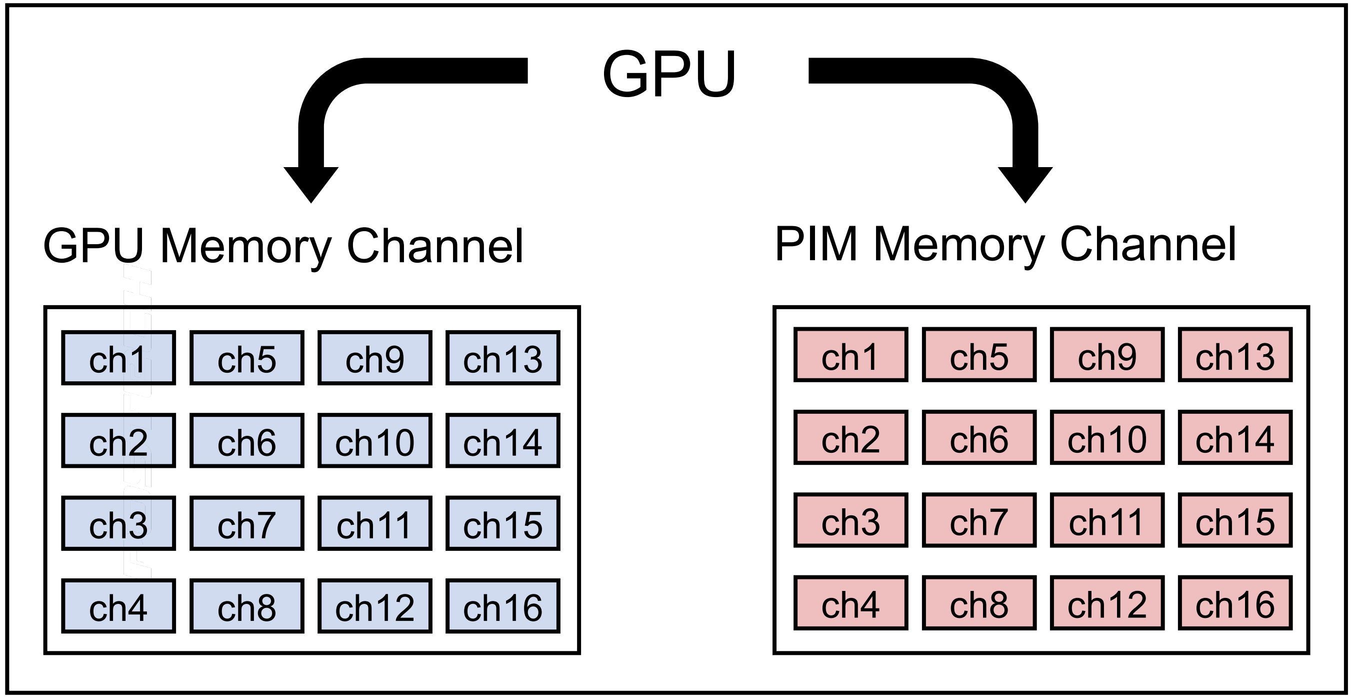
PIM-enabled GPU Memory Architecture

- We assume a PIM-enabled GPU with regular memory channels and **a set of memory channels** equipped with PIM compute units



PIM-enabled GPU Memory Architecture

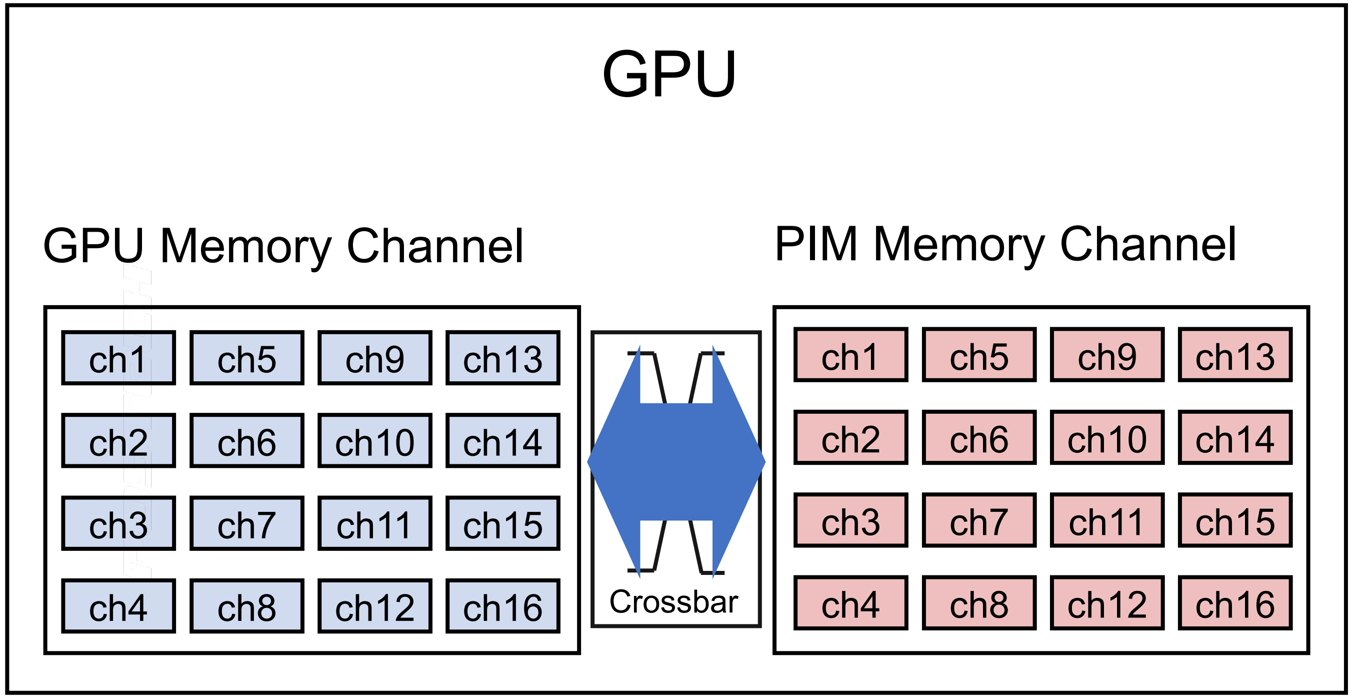
- We assume a PIM-enabled GPU with regular memory channels and **a set of memory channels** equipped with PIM compute units
- PIM memory channels can be used as regular channels too
  - GPU-only mode and GPU-PIM mode



PIM-enabled GPU Memory Architecture

- GPU and PIM memory channels are **connected to each other** by a crossbar

➔ Move data directly between the channels **without going through GPU caches**



Outline

Motivation

Overview

PIM-enabled GPU Memory Architecture

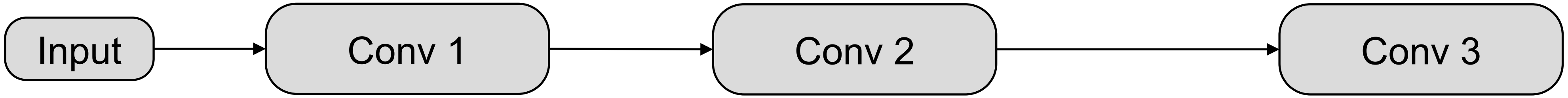
PIMFlow

- **PIM-Aware Graph Transformation**
- Execution Mode and Task Size Search
- TVM Back-End for DRAM-PIM

Evaluation result

Baseline Execution

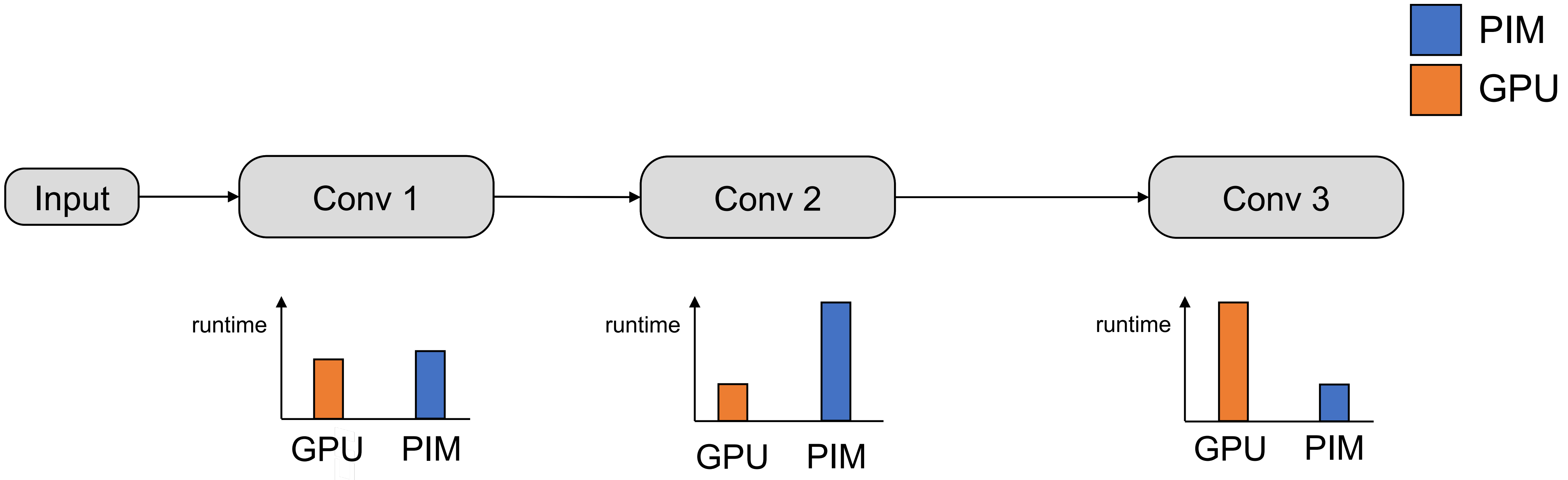
Fully Offloading to GPU or PIM



Given a **computation graph** with convolution layers,

Baseline Execution

Fully Offloading to GPU or PIM

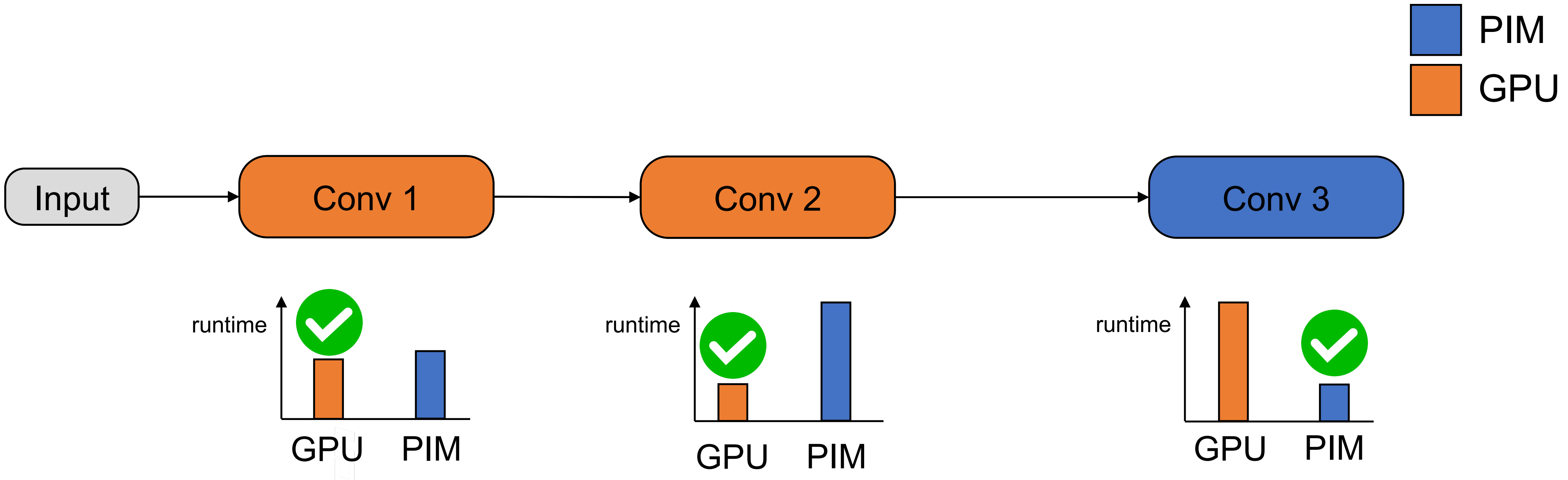


Given a **computation graph** with convolution layers,

Measure performance of each node on GPU and PIM

Baseline Execution

Fully Offloading to GPU or PIM



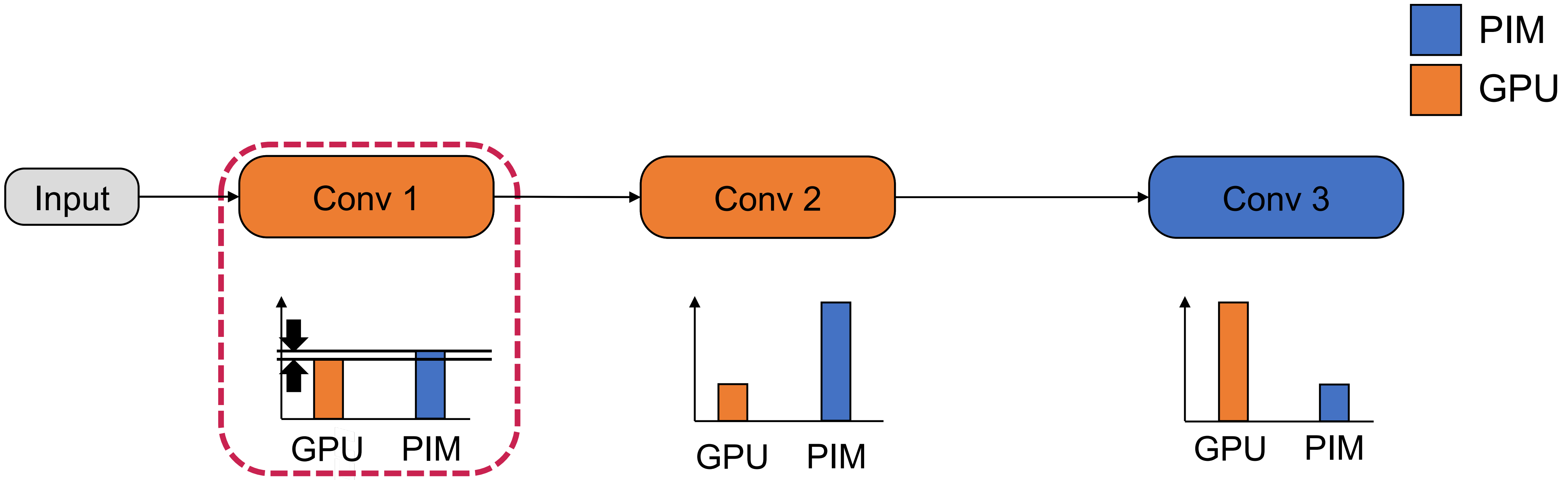
Given a **computation graph** with convolution layers,

Measure performance of each node on GPU and PIM

Offload the node to the device where it runs faster

Baseline Execution

Fully Offloading to GPU or PIM



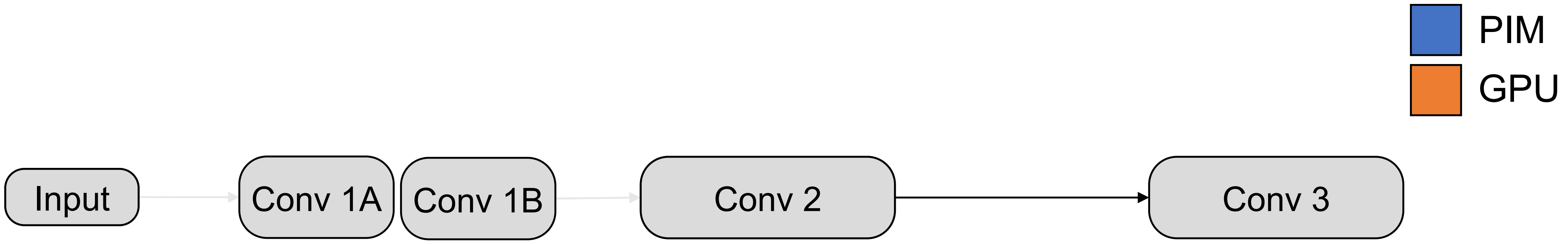
Parallel speedup when executed on both GPU AND PIM

➔ Multi-Device Data-Parallel Execution (**MD-DP**)

POSTECH

Multi-Device Data-Parallel (MD-DP) Execution

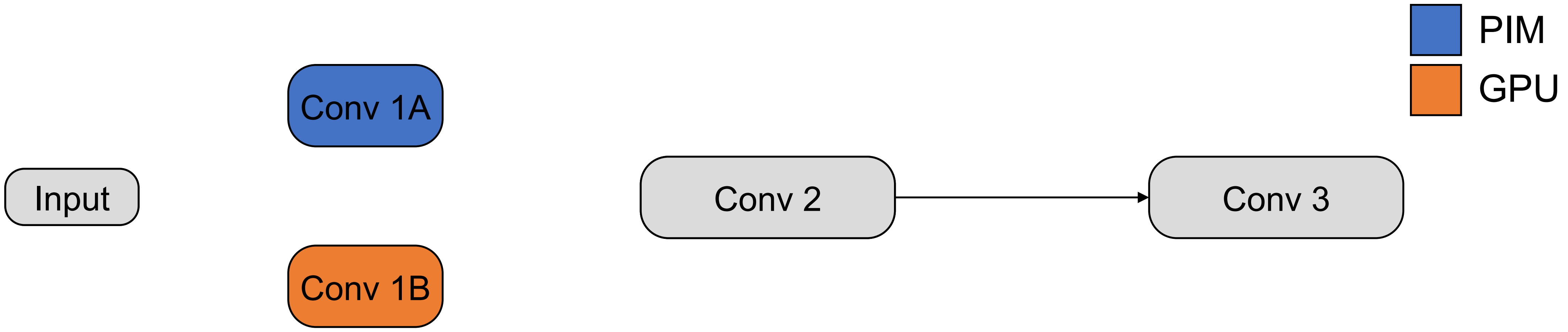
PIM-aware Graph Optimization #1



Split a convolution node into two nodes

Multi-Device Data-Parallel (MD-DP) Execution

PIM-aware Graph Optimization #1

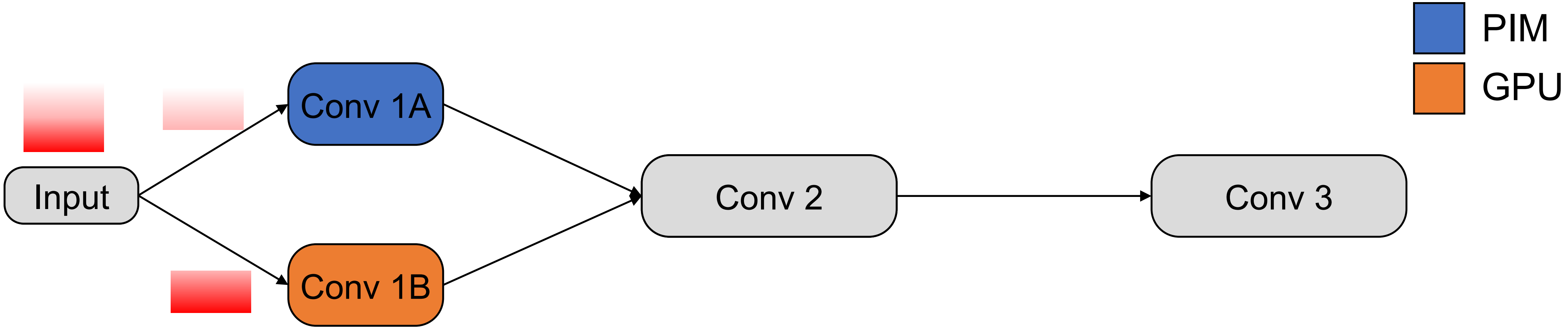


Split a convolution node into two nodes

Assign each node to GPU and PIM

Multi-Device Data-Parallel (MD-DP) Execution

PIM-aware Graph Optimization #1

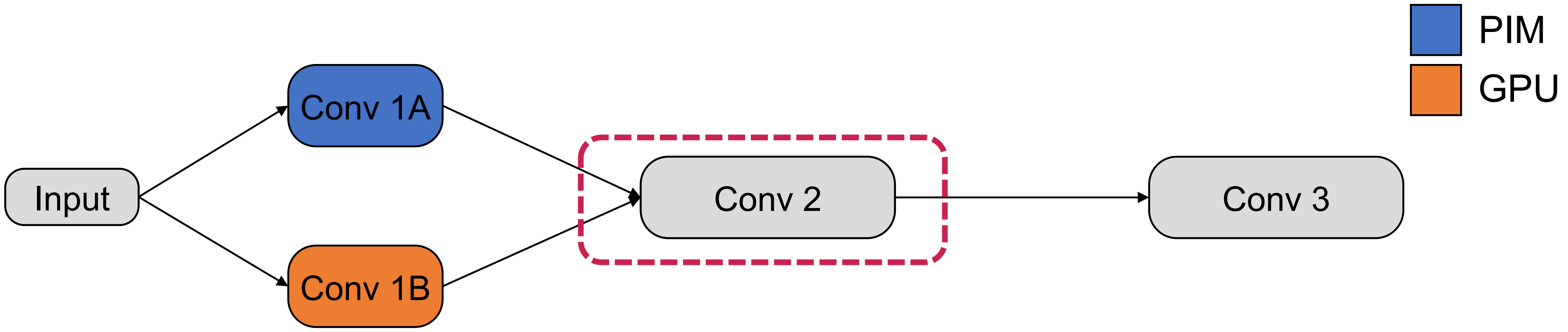


Split the input tensor for Conv 1A and Conv 1B

➔ Data-parallel execution multiple devices (**MD-DP**)

Pipelined Execution

PIM-aware Graph Optimization #2

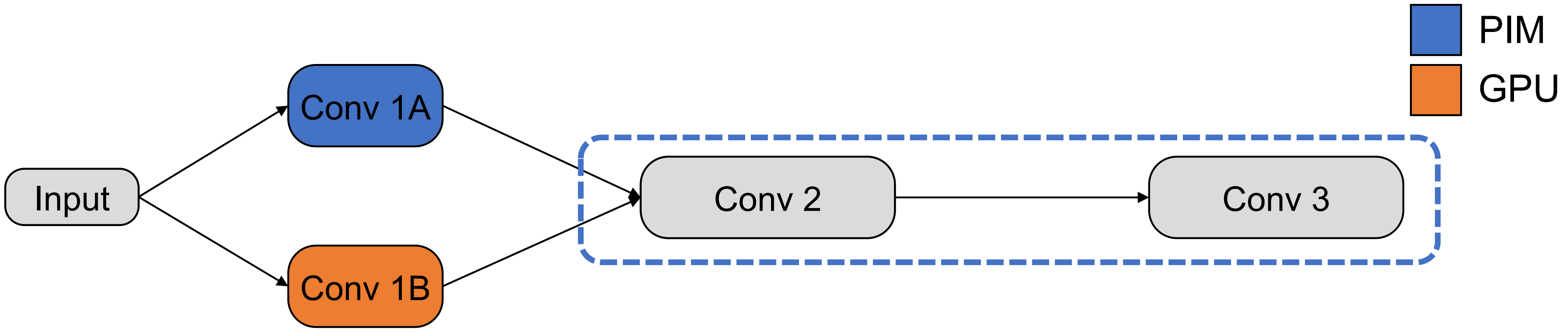


Certain types of convolution nodes always run on GPU

- Not supported on PIM, or
- GPU runtime is much faster than PIM runtime

Pipelined Execution

PIM-aware Graph Optimization #2



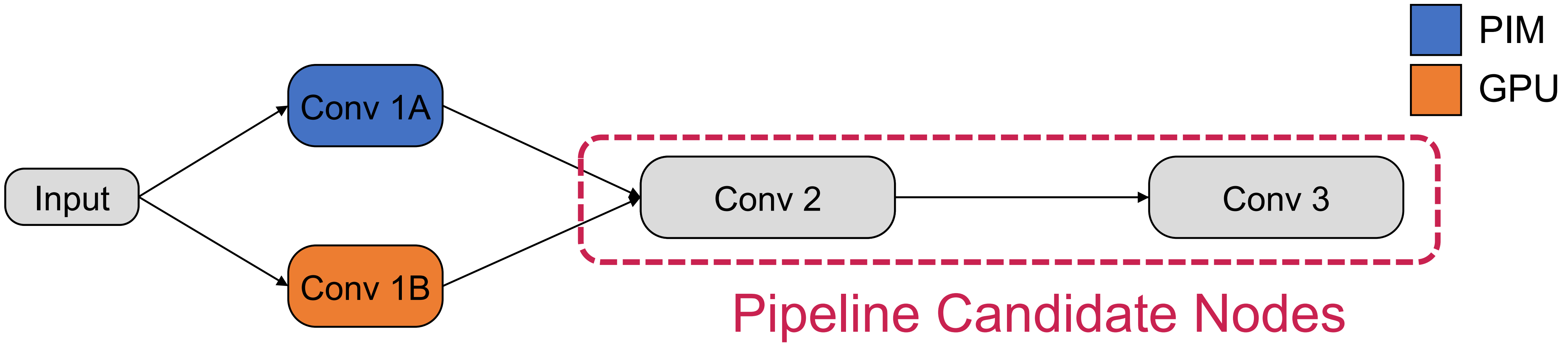
Certain types of convolution nodes always run on GPU

- Not supported on PIM, or
- GPU runtime is much faster than PIM runtime

→ **Pipeline the GPU node with the following PIM node**

Pipelined Execution

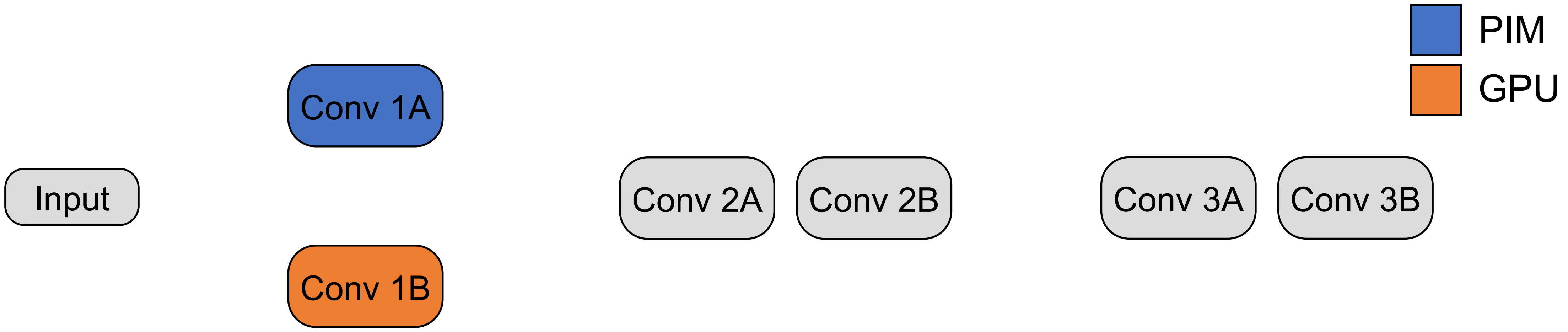
PIM-aware Graph Optimization #2



Select a group of nodes as **pipeline candidates**

Pipelined Execution

PIM-aware Graph Optimization #2

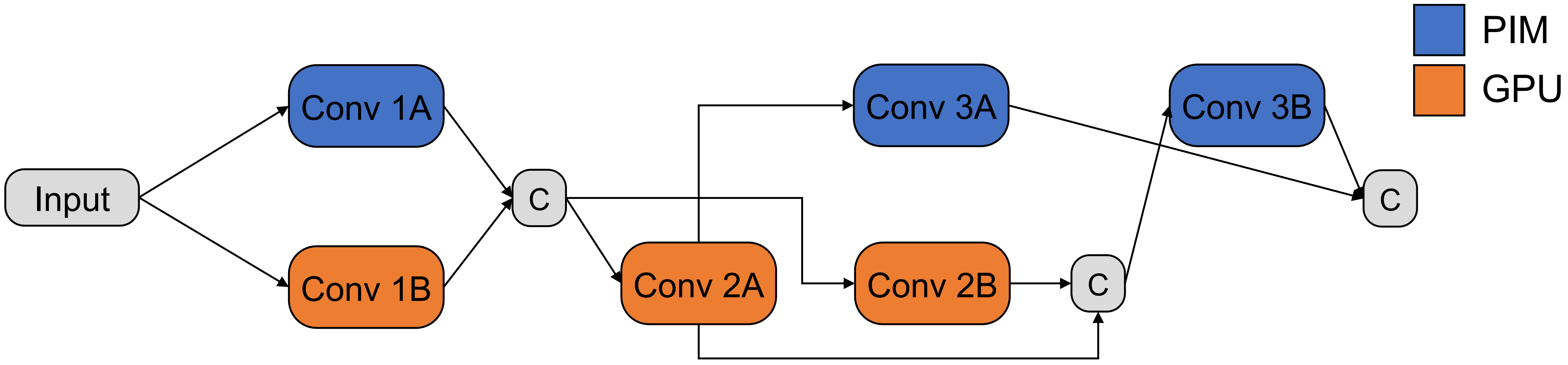


Select a group of nodes as **pipeline candidates**

Split Conv 2 and Conv 3 into **pipeline stages**

Pipelined Execution

PIM-aware Graph Optimization #2



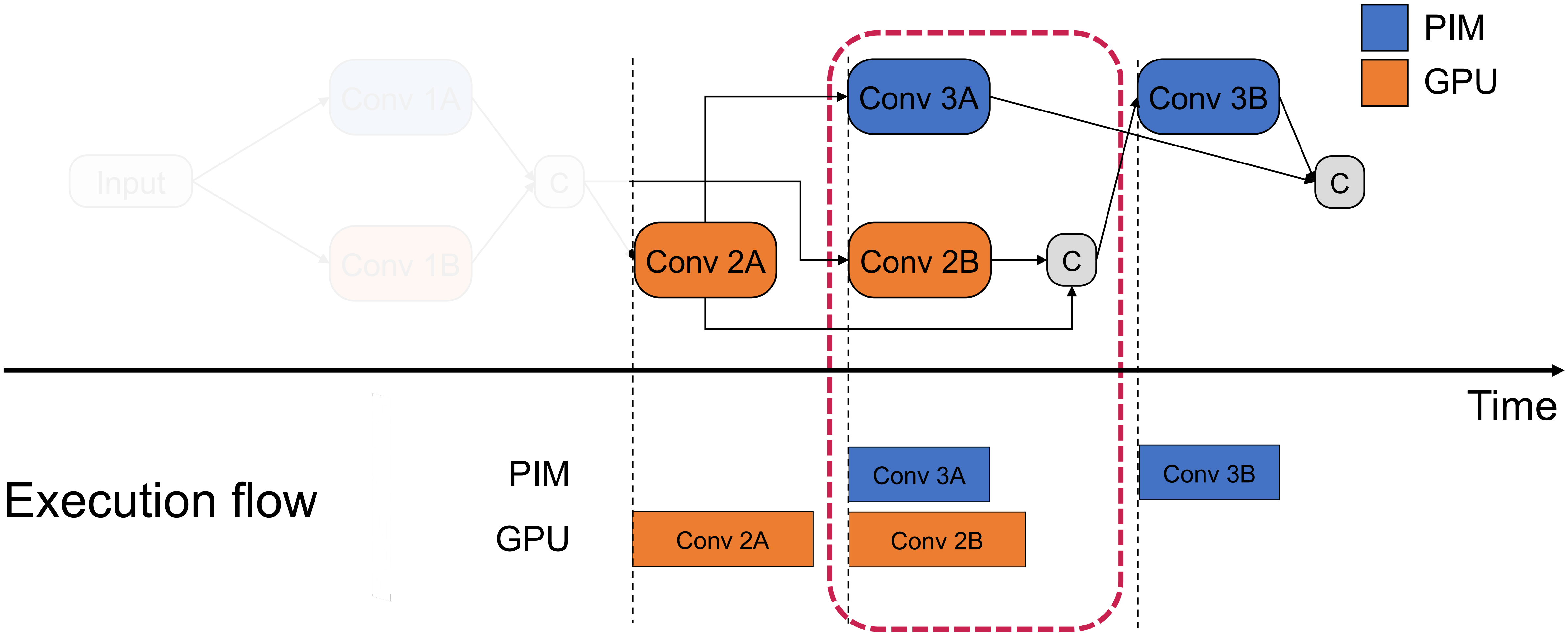
Select a group of nodes as **pipeline candidates**

Split Conv 2 and Conv 3 into pipeline stages

Add **data-flow edges** for pipelined execution

Pipelined Execution

PIM-aware Graph Optimization #2



Conv 2B and Conv 3A can be executed ***in parallel*** on GPU and PIM

POSTECH

Outline

Motivation

Overview of PIMFlow

PIM-enabled GPU Memory Architecture

PIMFlow

- PIM-Aware Graph Transformation
- **Execution Mode and Task Size Search**
- TVM Back-End for DRAM-PIM

Evaluation result

Execution Mode and Task Size Search

Profiling Phase

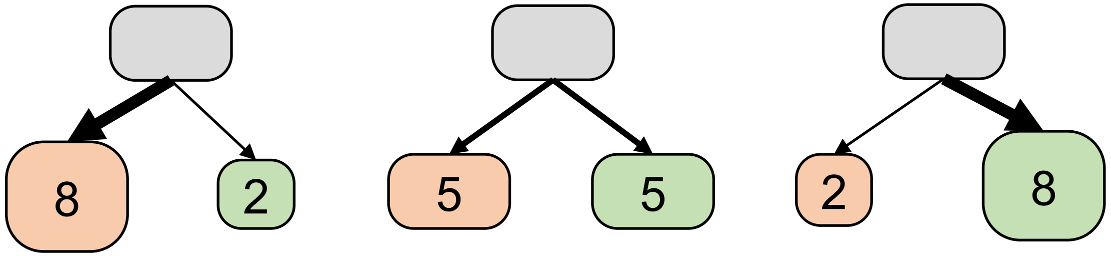
Solving Phase

Execution Mode and Task Size Search

Profiling Phase

Solving Phase

① Determine the ***optimal*** split ratio for **MD-DP**



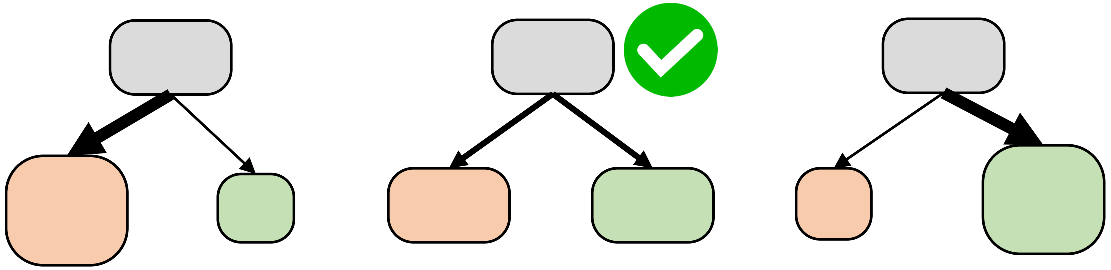
Split ratio is profiled at every 10%

Execution Mode and Task Size Search

Profiling Phase

① Determine the ***optimal*** split ratio for **MD-DP**

`T[N][1] = best_runtime`



Best split ratio is recorded for the node

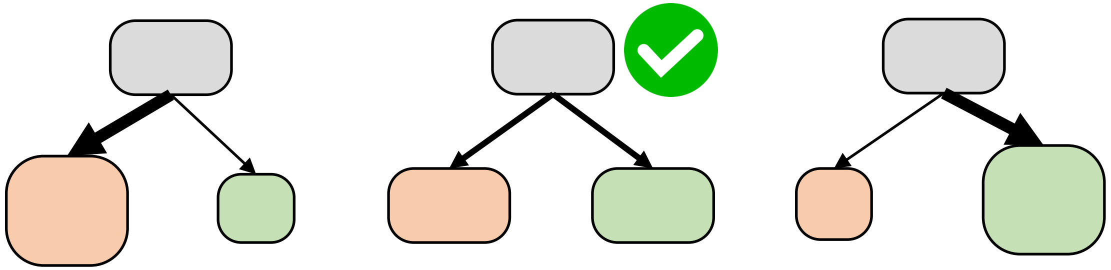
Solving Phase

Execution Mode and Task Size Search

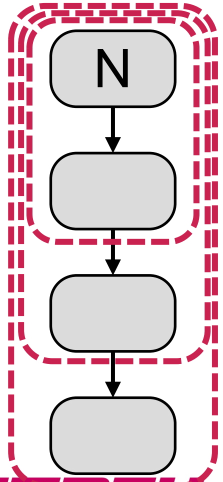
Profiling Phase

① Determine the ***optimal*** split ratio for **MD-DP**

`T[N][1] = best_runtime`



② Record ***every*** possible **pipelining** result




```
T[N][2] = pipeline_runtime
T[N][3] = pipeline_runtime
T[N][4] = pipeline_runtime
```

pipeline length form the node

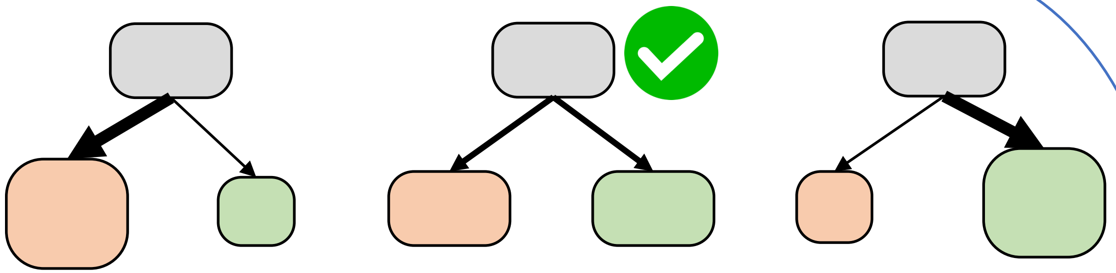
Solving Phase

Execution Mode and Task Size Search

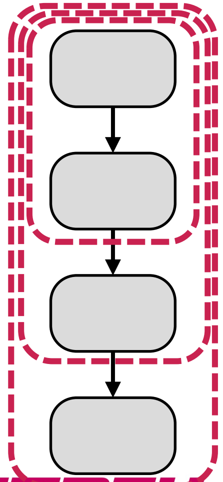
Profiling Phase

① Determine the ***optimal*** split ratio for **MD-DP**

`T[N][1] = best_runtime`



② Record ***every*** possible **pipelining** result



`T[N][2] = pipeline_runtime`
`T[N][3] = pipeline_runtime`
`T[N][4] = pipeline_runtime`

Solving Phase

③ Obtain the ***optimal policy*** by solving DP with the runtime table (T) information

```
for l ← 1 to N do       // Solve by dynamic programming
  for i ← 1 to N do
    for k ← 1 to l - 1 do
      if i + k > N then
        continue
      T[i][l] ← min(T[i][l], T[i][k] + T[i+k][l-k])
return T[i][N]
```

Outline

Motivation

Overview of PIMFlow

PIM-enabled GPU Memory Architecture

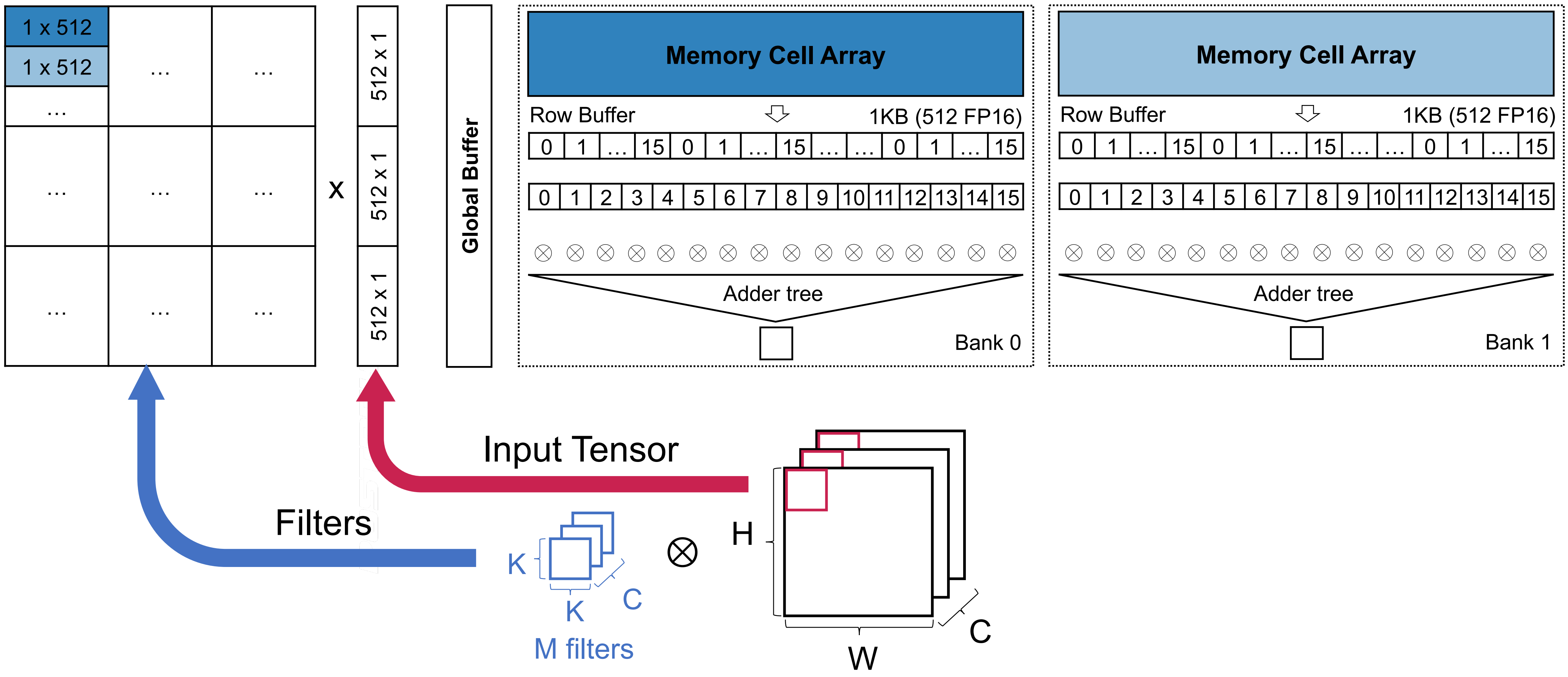
PIMFlow

- PIM-Aware Graph Transformation
- Execution Mode and Task Size Search
- **TVM Back-End for DRAM-PIM**

Evaluation result

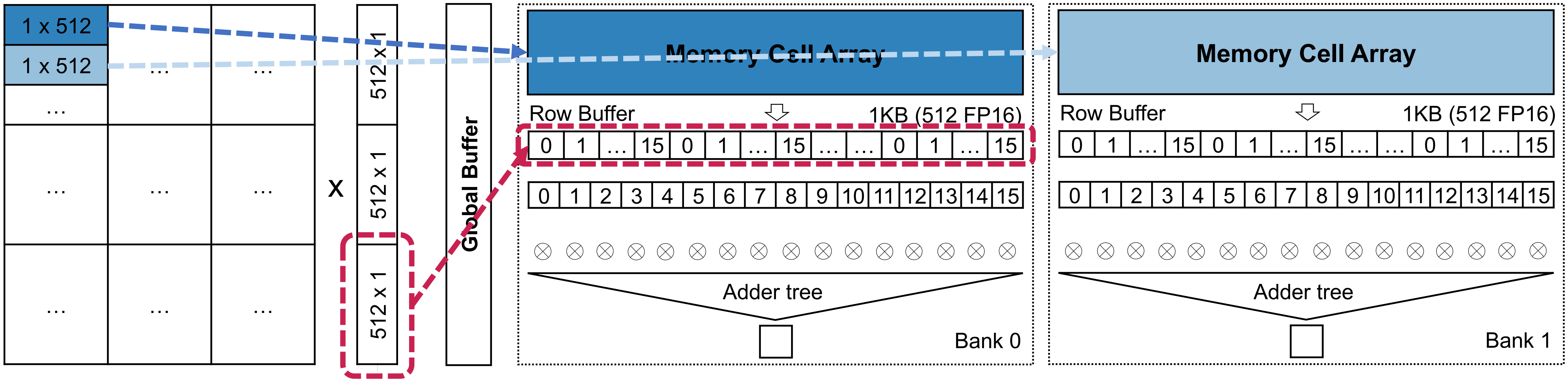
TVM Back-End for DRAM-PIM

- Generate PIM commands to map matrix-vector multiplications to DRAM-PIM
  - Input data to CONV layers → vector, filters → matrix tiles



TVM Back-End for DRAM-PIM

- Generate PIM commands to map matrix-vector multiplications to DRAM-PIM
  - Input data to CONV layers → vector, filters → matrix tiles

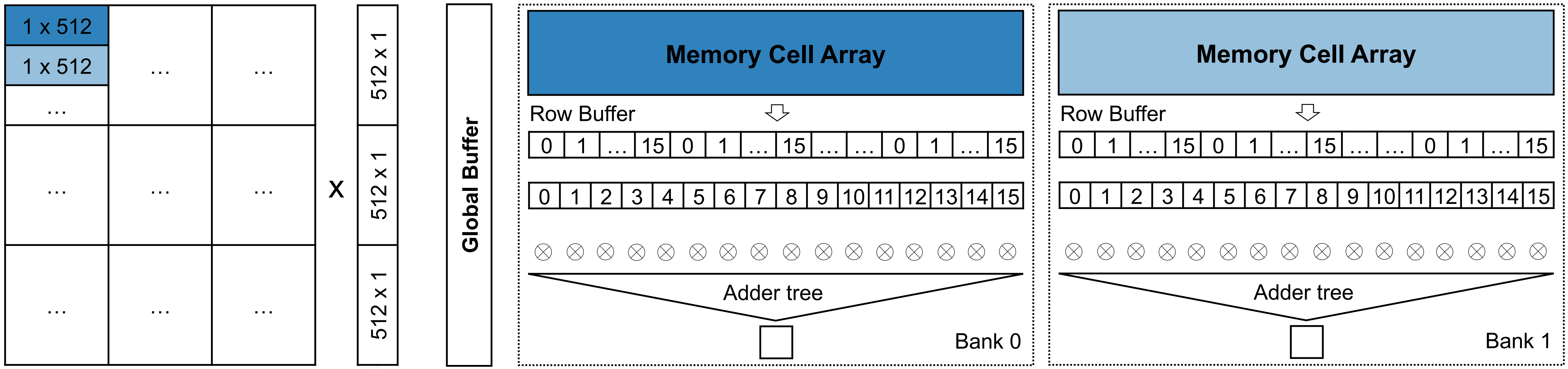


- Matrix is partitioned to 2 x 512 tiles
  - **2** is the number of the banks in a channel (bank-level parallelism)
  - **512** is the number of matrix elements in a memory
- Tiles are stored in the memory cell array

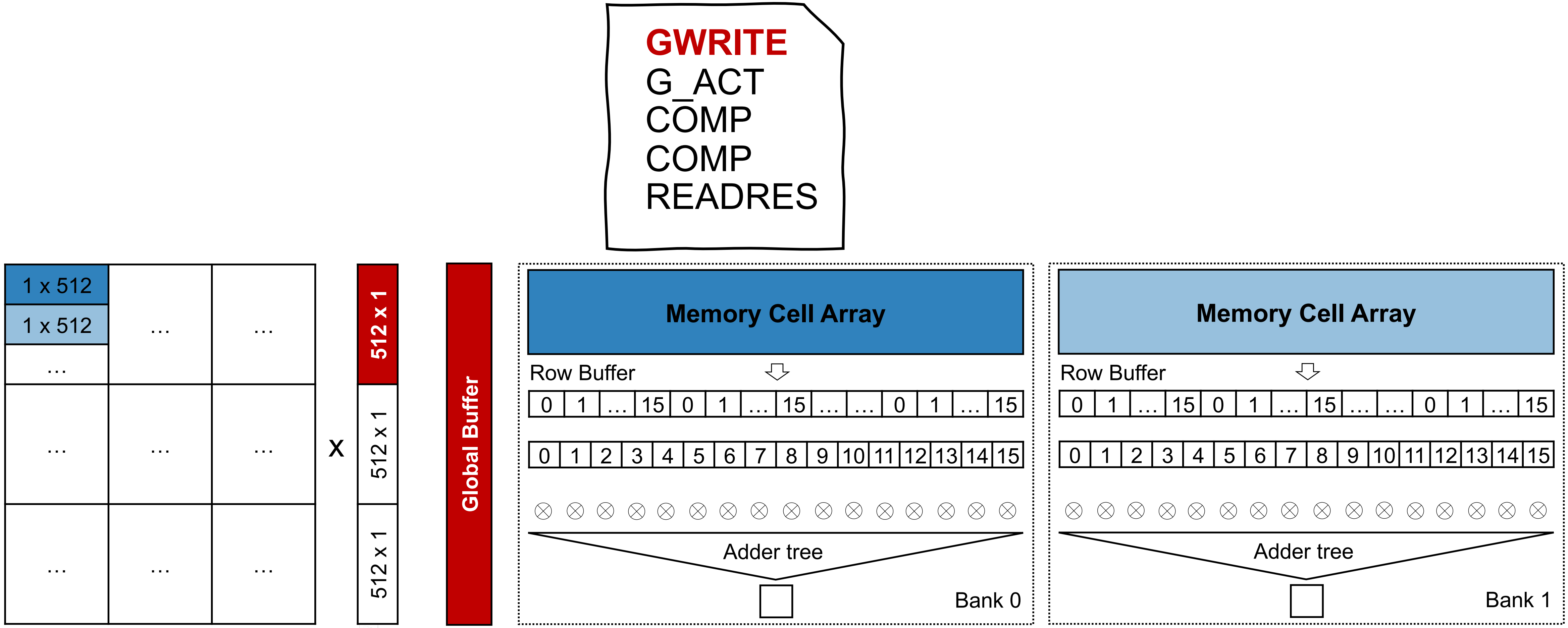
PIM Command Execution Flow

Example command sequence

GWRITE
G_ACT
COMP
COMP
READRES

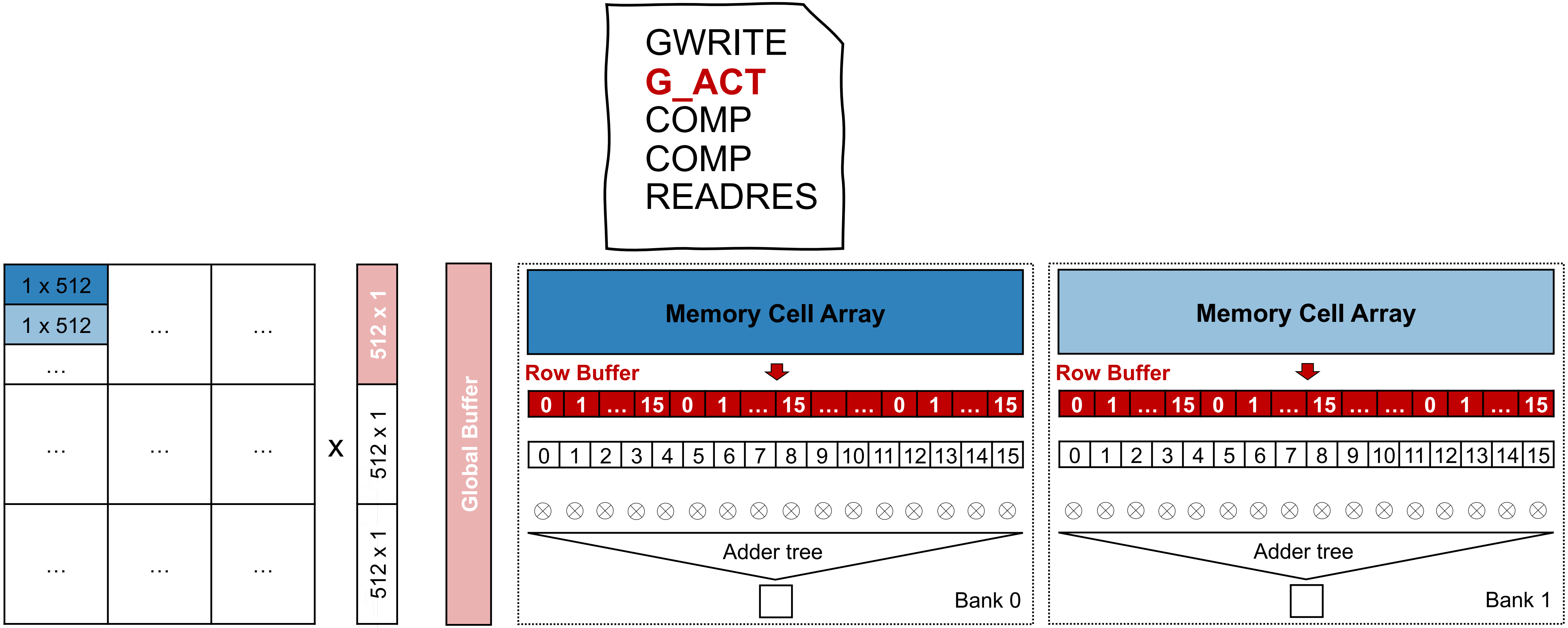


PIM Command Execution Flow



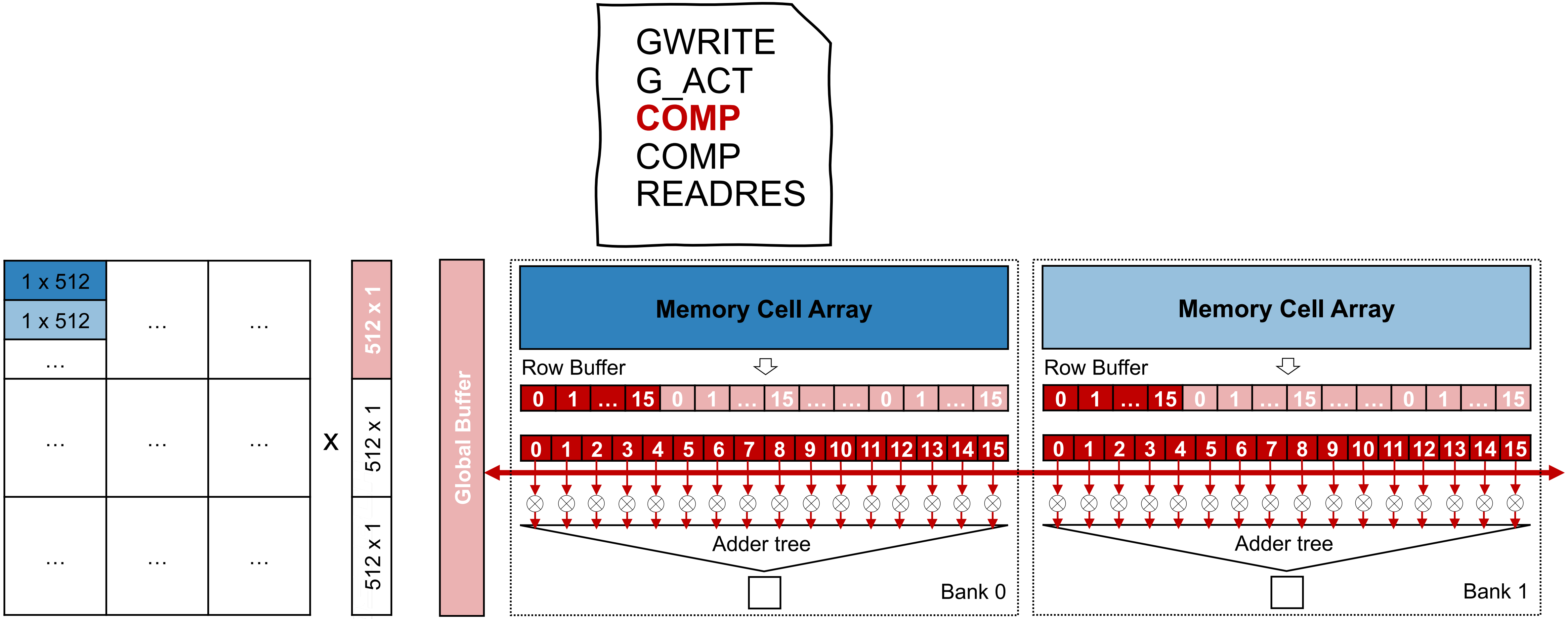
- **GWRITE**: Fill the global buffer from the input tensor (vector)

PIM Command Execution Flow



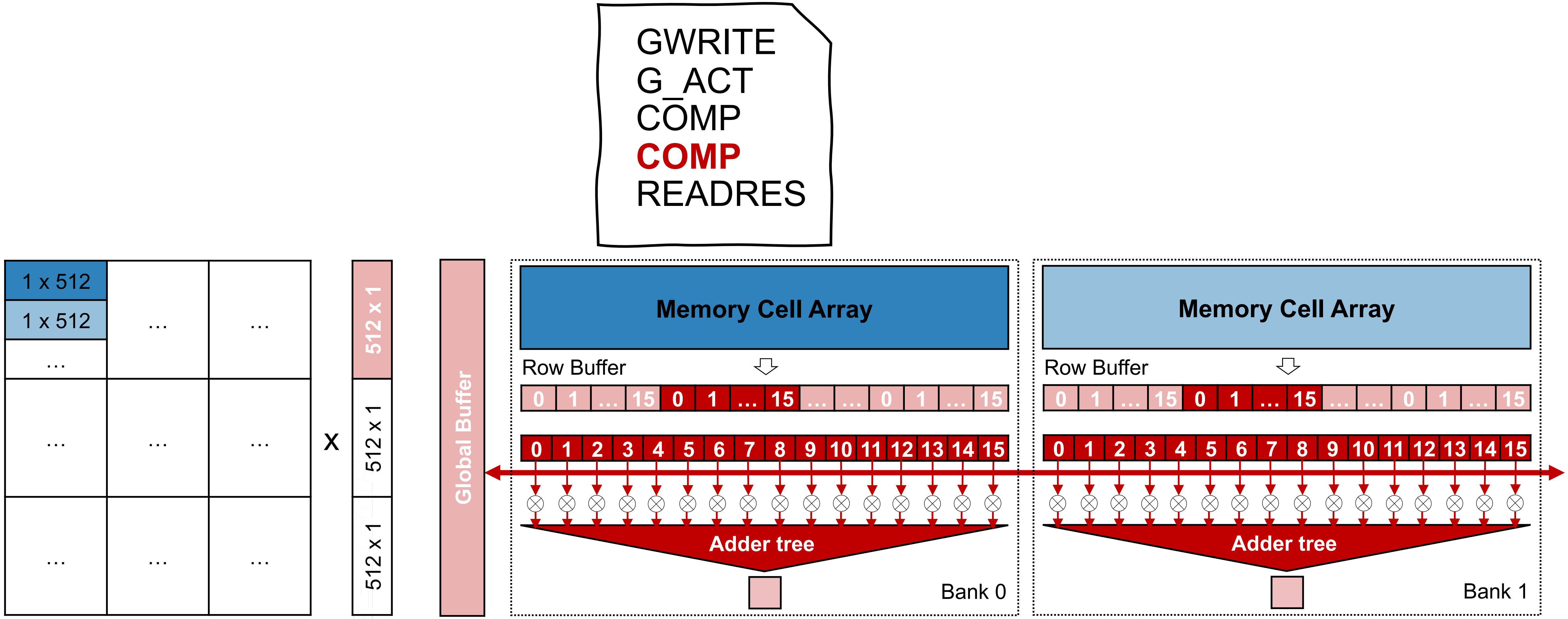
- **G_ACT**: Activates rows of multiple memory banks (*ganged* activation)
  - Fetch matrix (convolution) weights

PIM Command Execution Flow



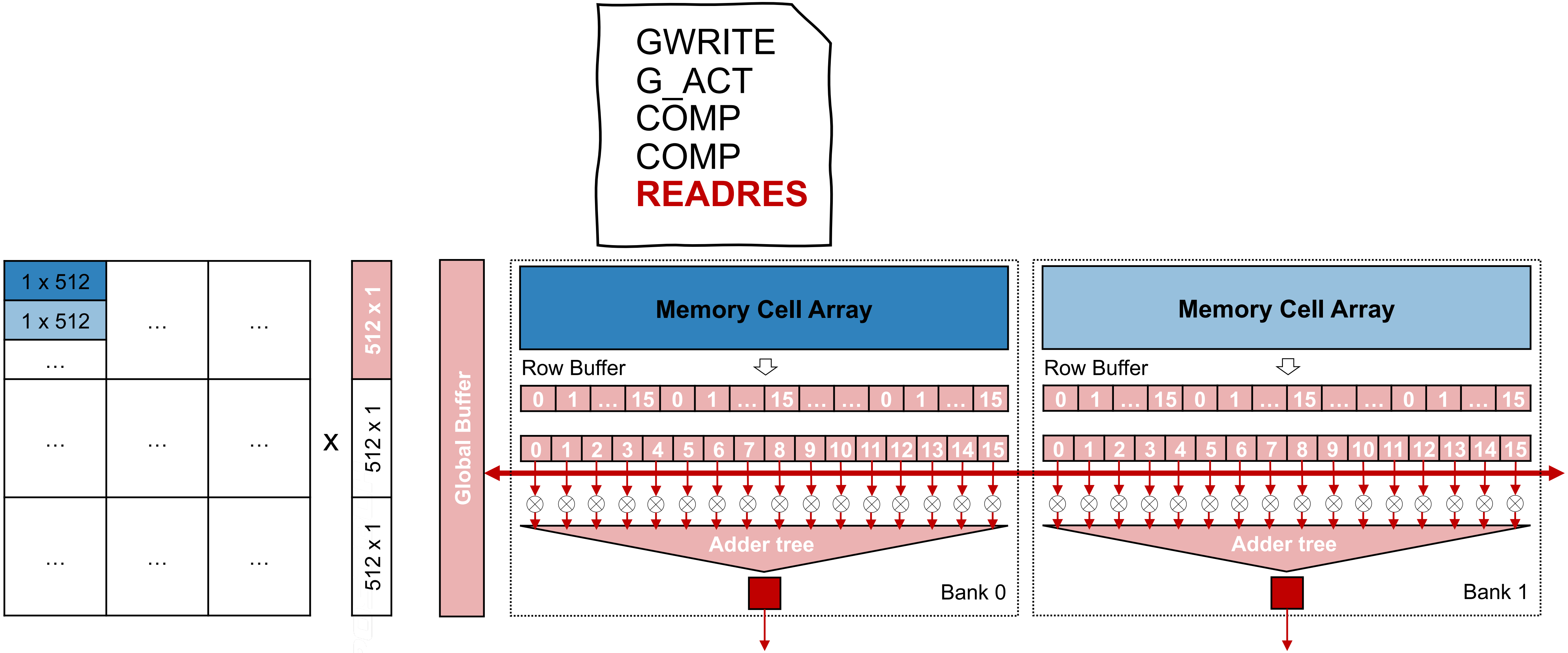
- **COMP**: computes multiply-accumulate (MAC) operation

PIM Command Execution Flow



- Consecutive **COMPs** can be computed in a **pipelined** manner
- Partial results are **accumulated** in the result latch

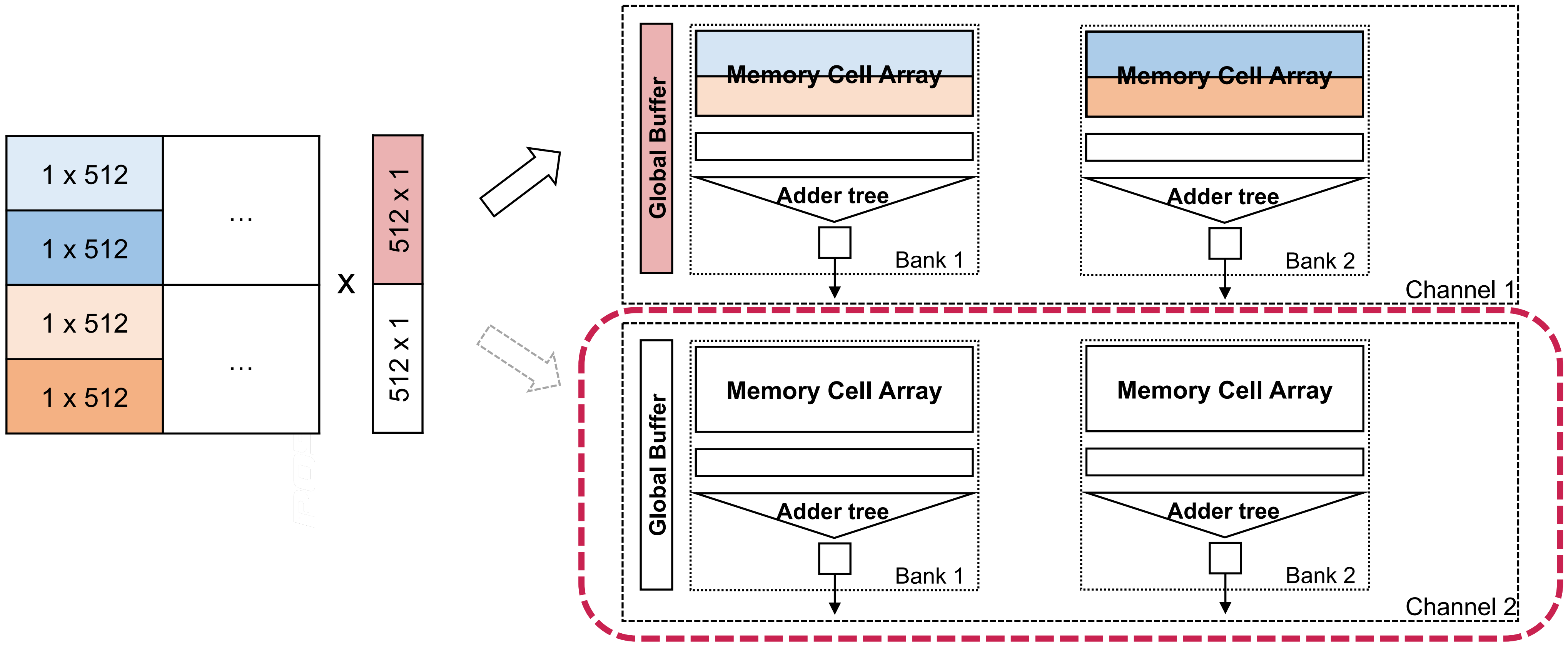
PIM Command Execution Flow



- **READRES**: read MAC results

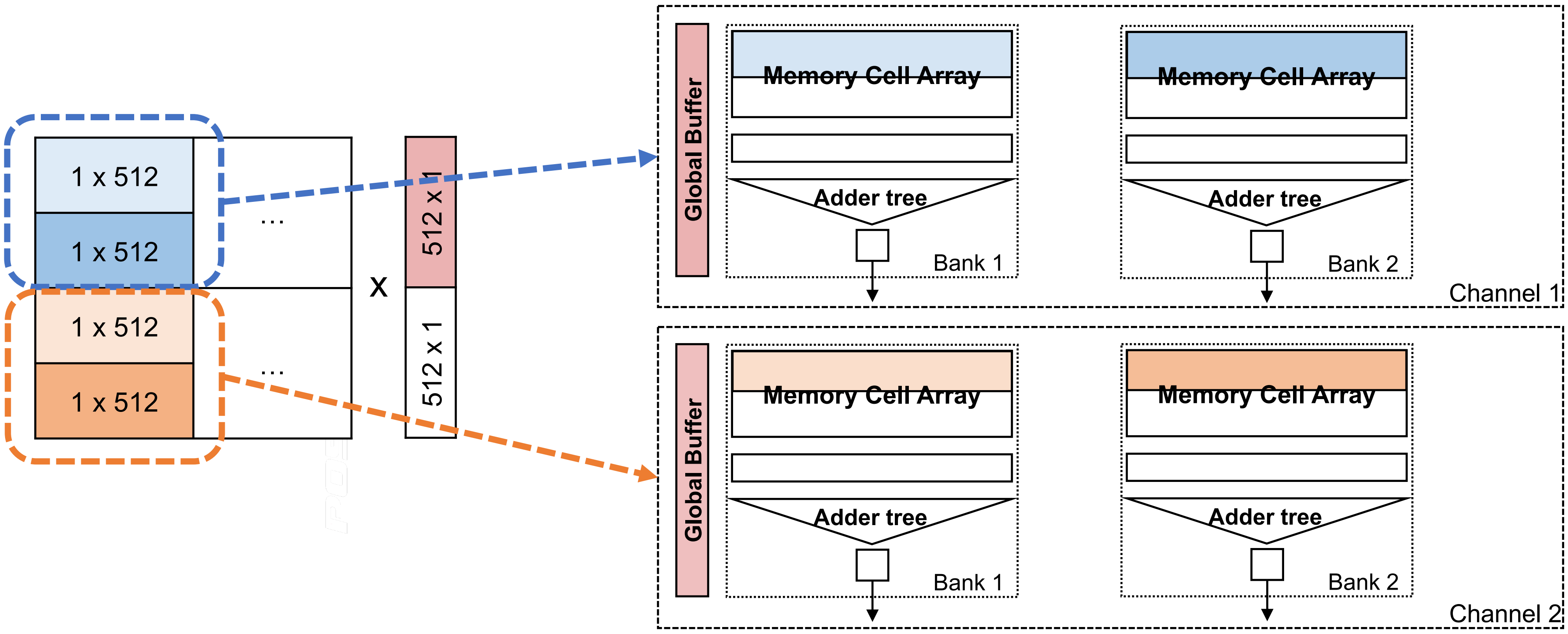
Command Scheduling

- Naïve scheduling can suffer from **channel under-utilization**



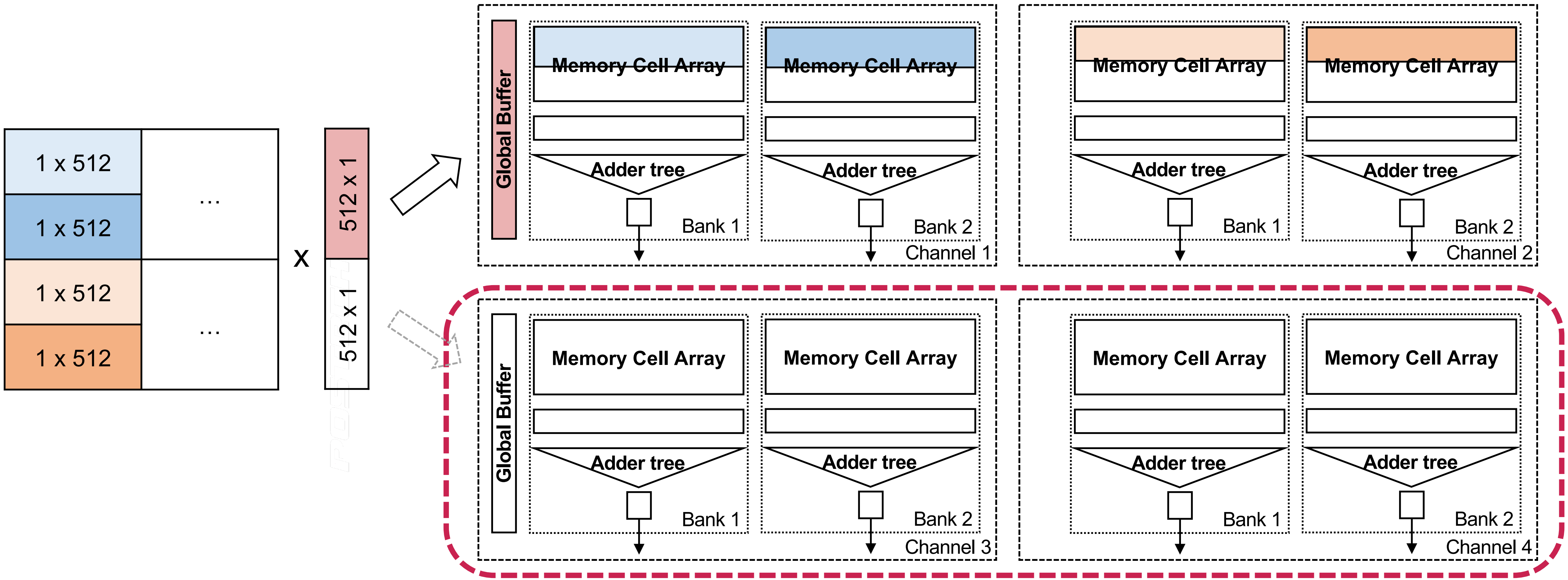
Command Scheduling

- Naïve scheduling can suffer from **channel under-utilization**
  → **Distribute tiles (G_ACT)** to increase channel-level parallelism



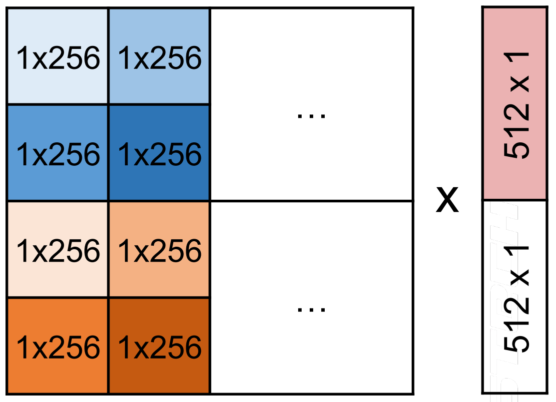
Command Scheduling

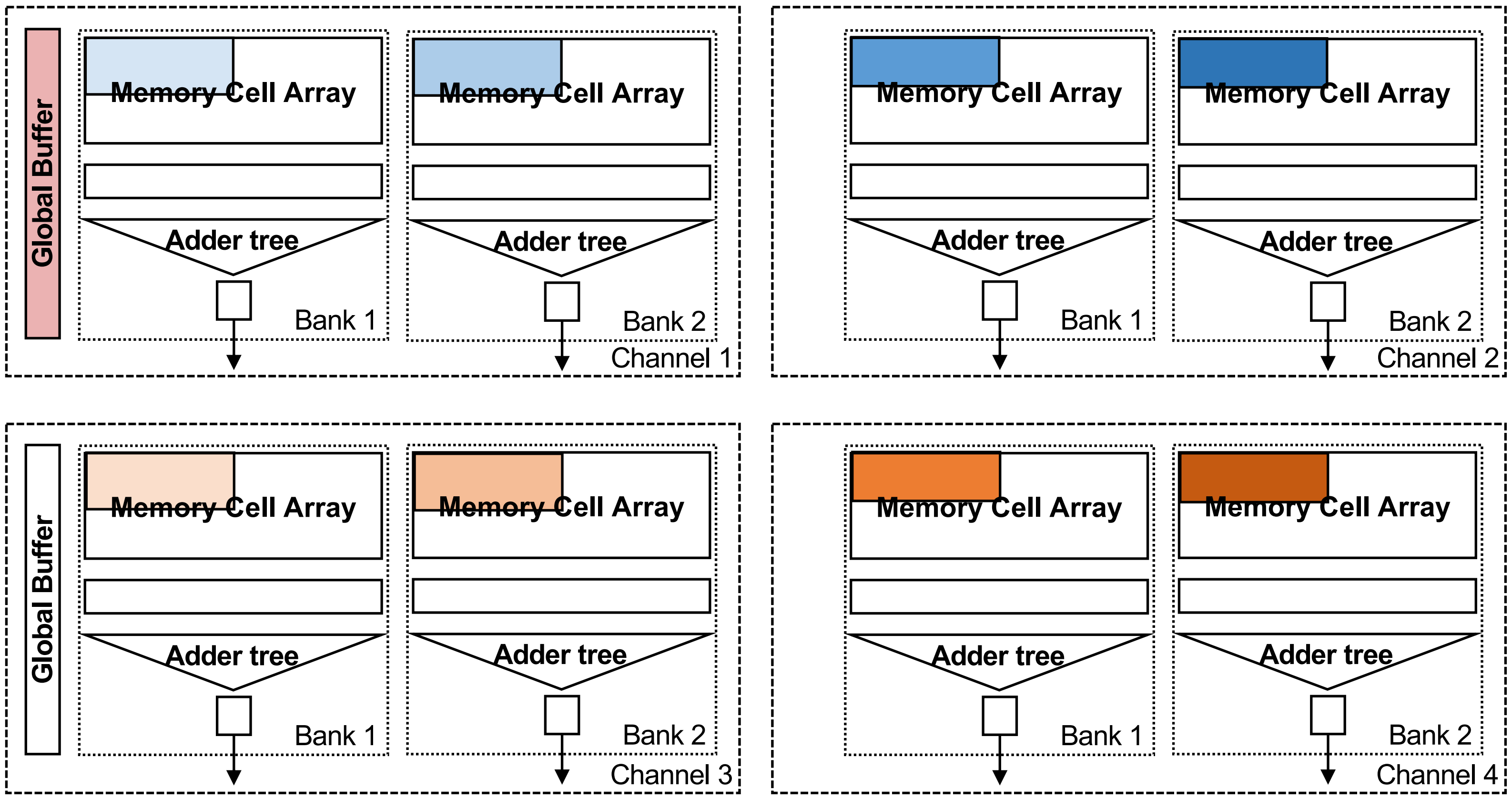
- Still can suffer from **under-utilization when (# channels) > (# tiles)**



Command Scheduling

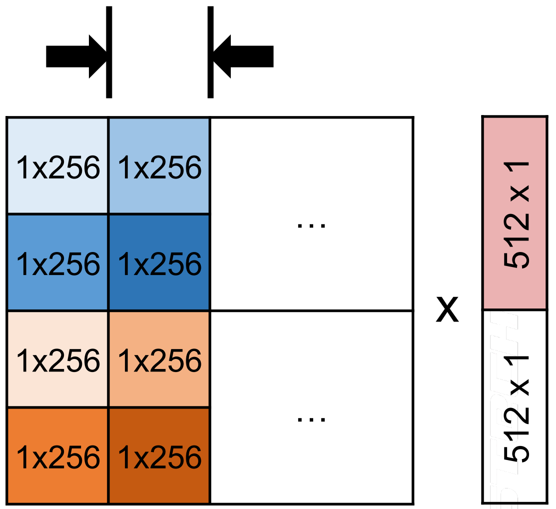
- Still can suffer from **under-utilization when (# channels) > (# tiles)**
  - **Reducing the tile size can increase channel utilization**

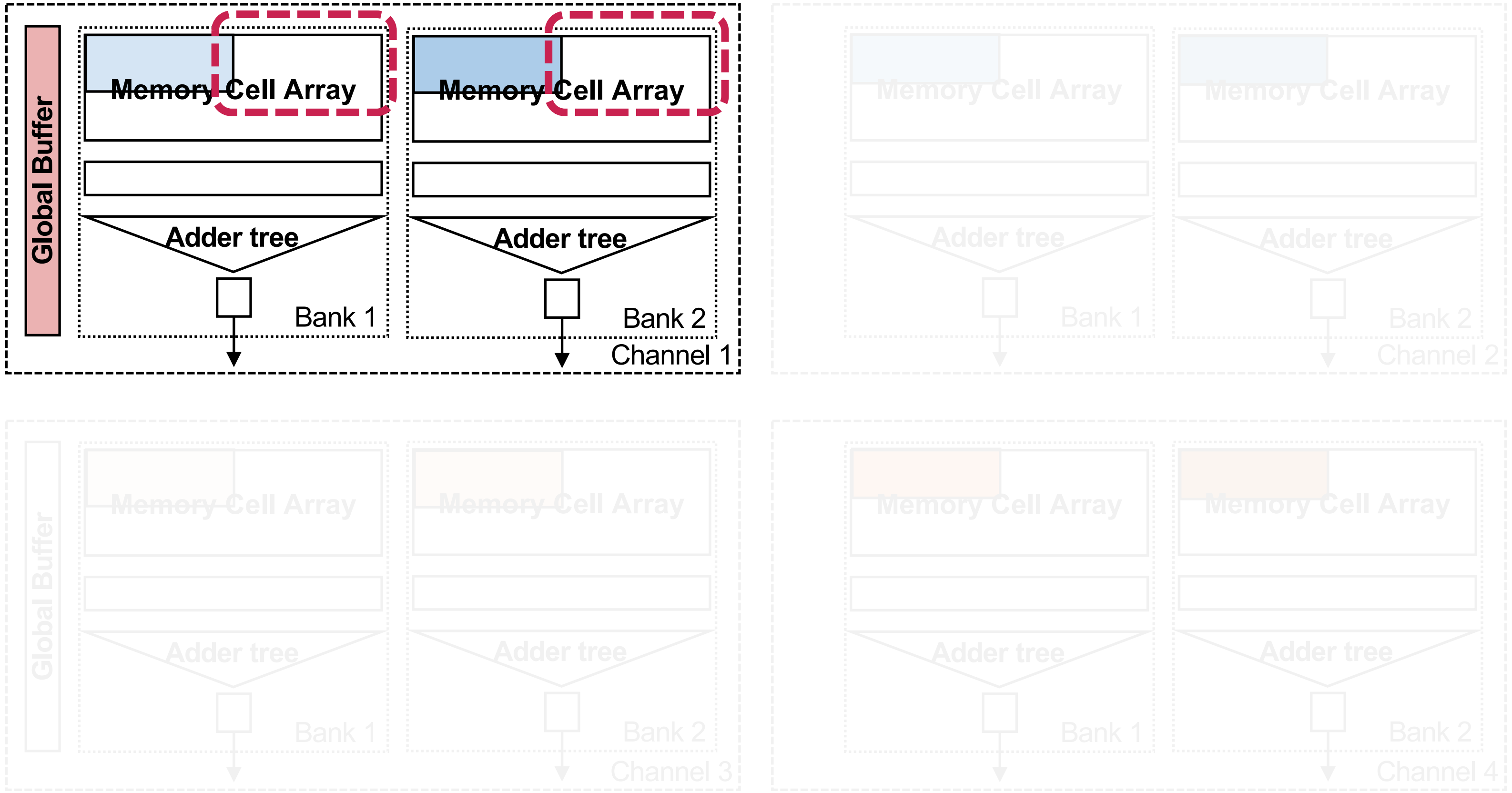




Command Scheduling

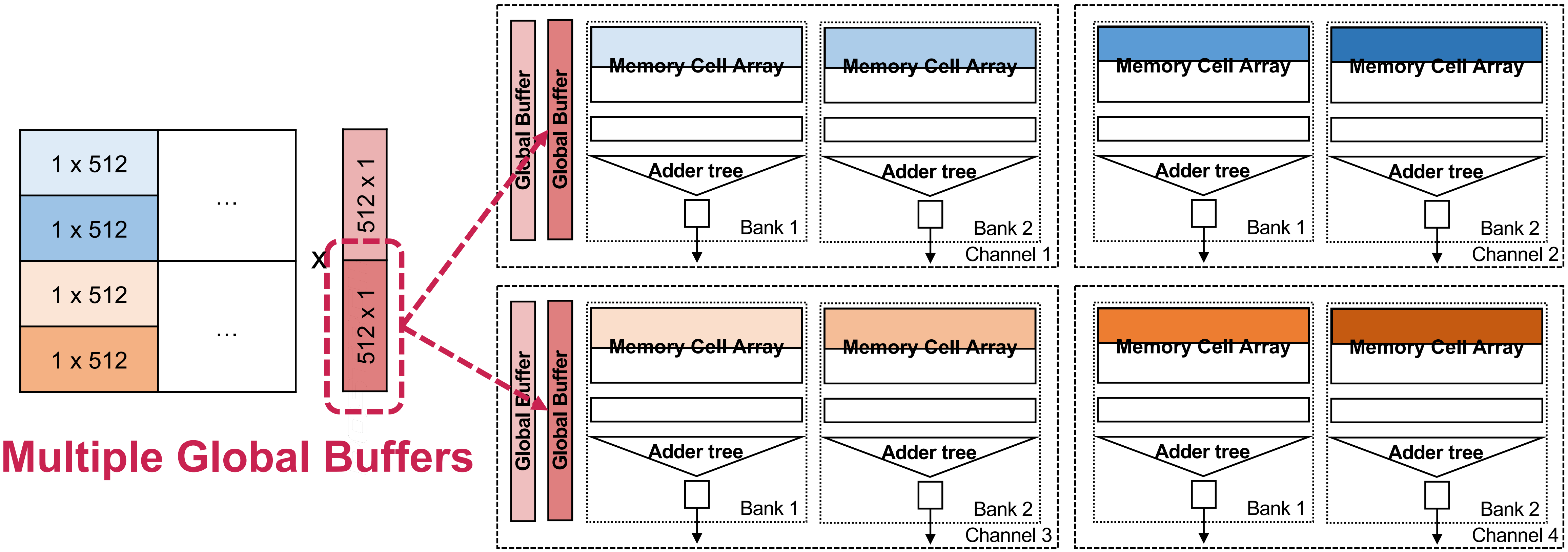
- Still can suffer from **under-utilization when (# channels) > (# tiles)**
  - **Reducing the tile size can increase channel utilization**
  - **Can introduce wasted row elements**





Command Scheduling

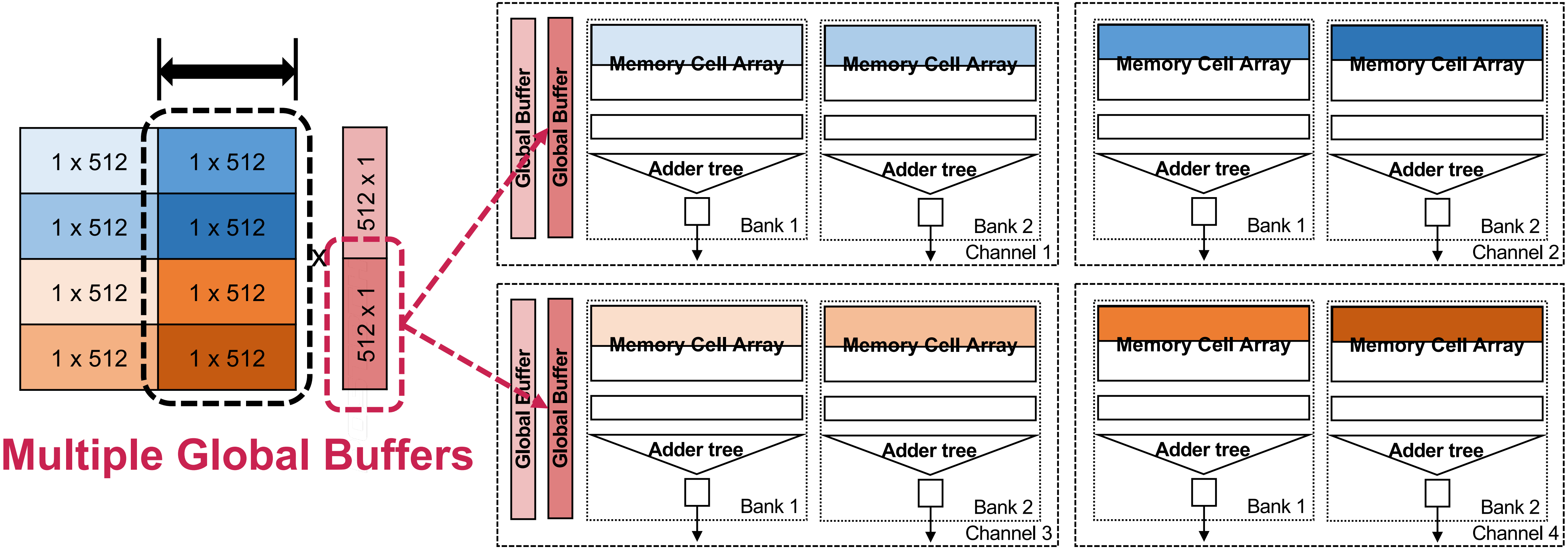
- Additional global buffers can address both row-level and channel-level utilization issues



Multiple Global Buffers

Command Scheduling

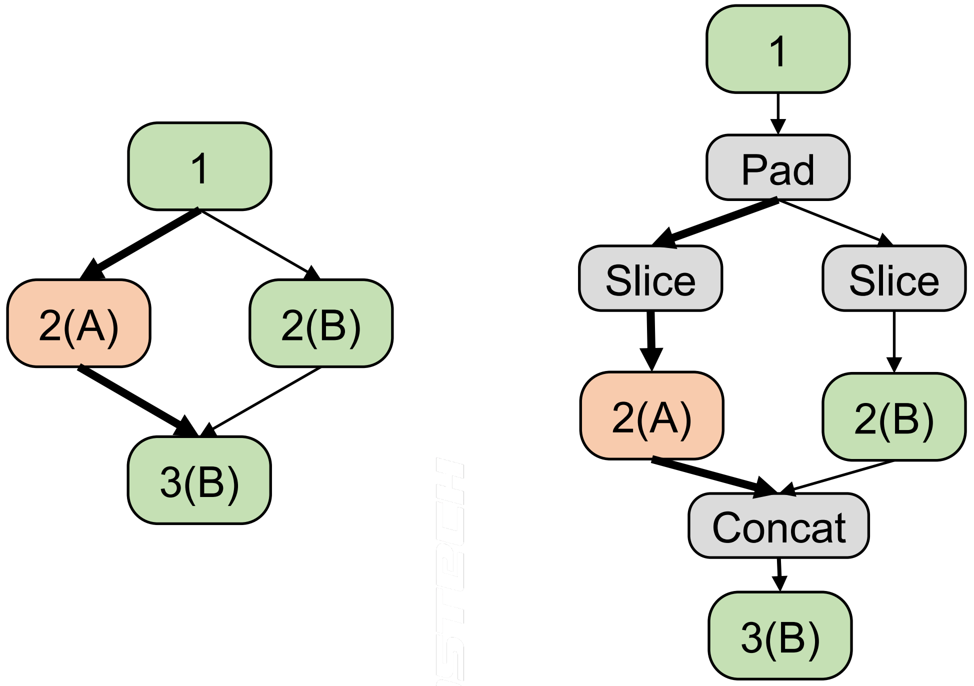
- Additional global buffers can address both row-level and channel-level utilization issues
  - Incurs a small overhead (0.7% GPU die area)



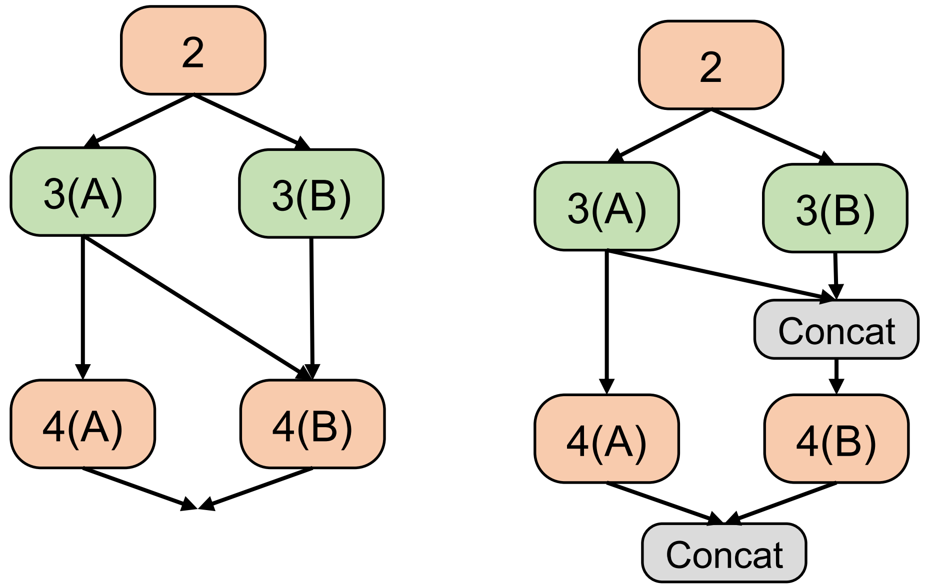
Multiple Global Buffers

Memory Optimizer

- PIM-aware graph transformations generate extra nodes (**Slice, Pad, Concat**) to adjust tensor shapes and placement for correctness



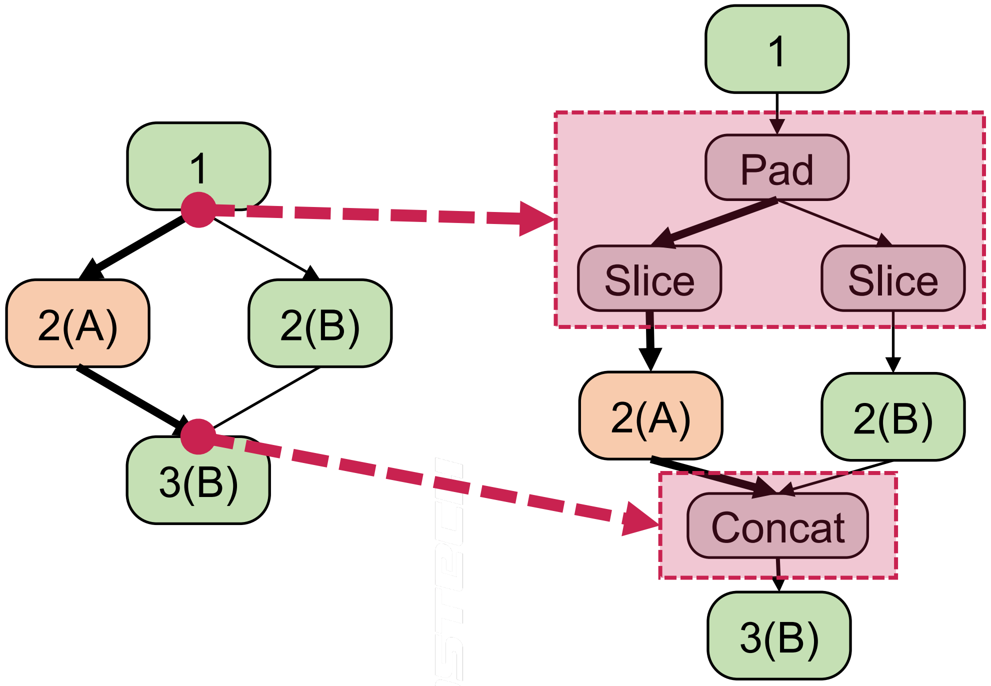
MD-DP Transformation



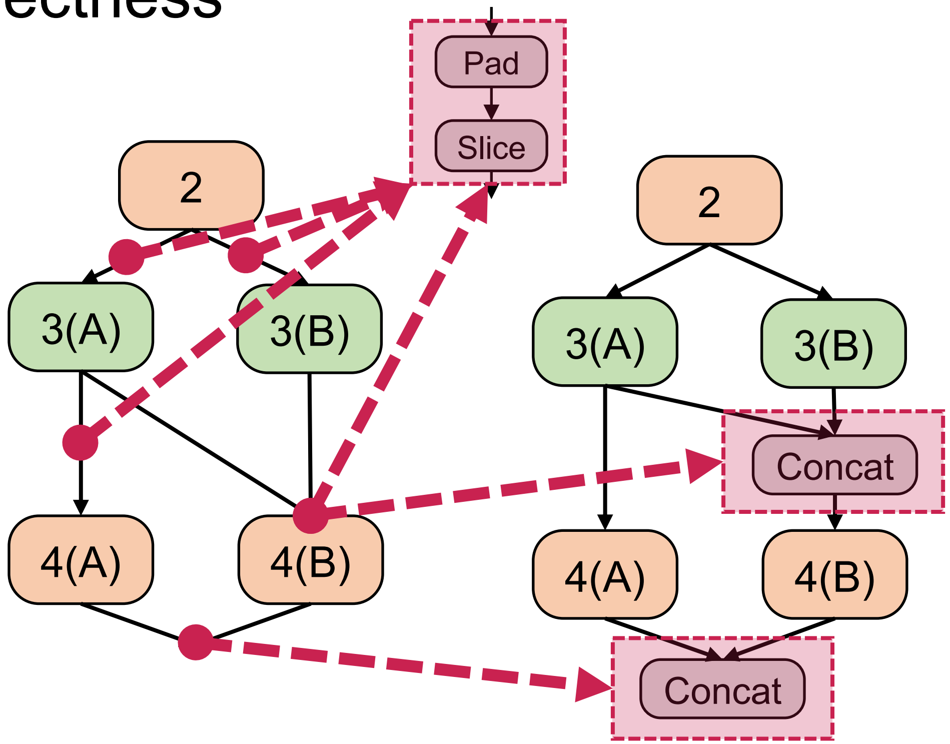
Pipeline Transformation

Memory Optimizer

- PIM-aware graph transformations generate extra nodes (**Slice, Pad, Concat**) to adjust tensor shapes and placement for correctness



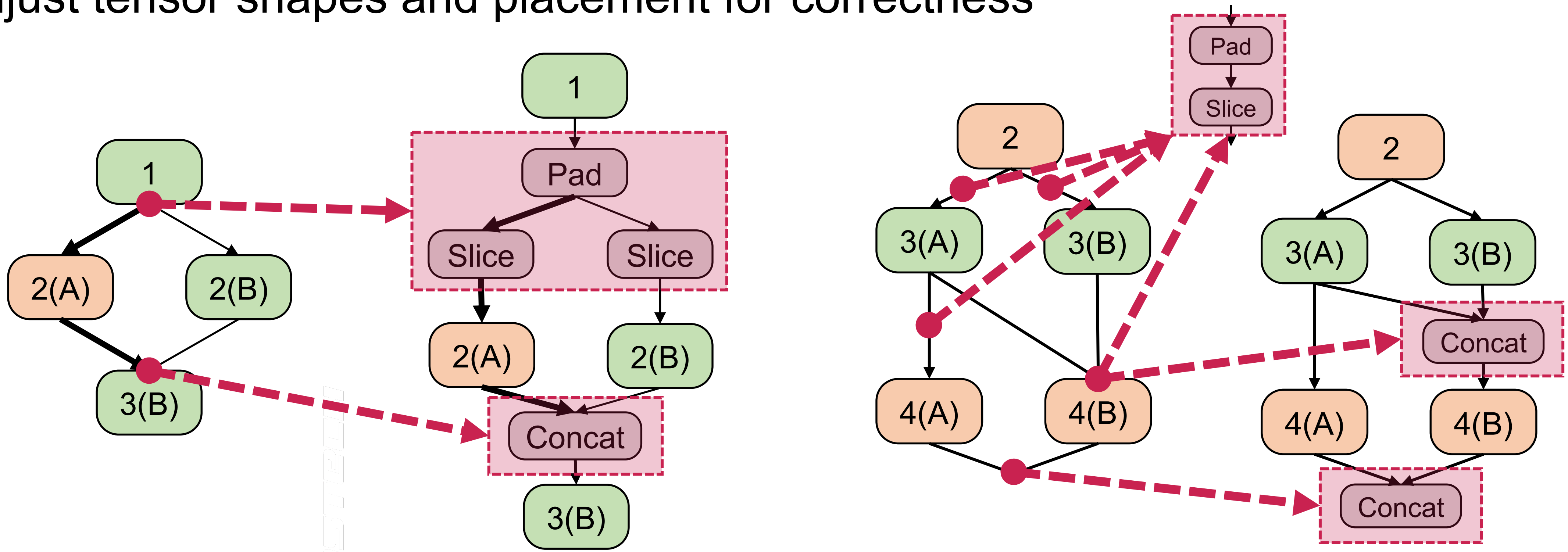
MD-DP Transformation



Pipeline Transformation

Memory Optimizer

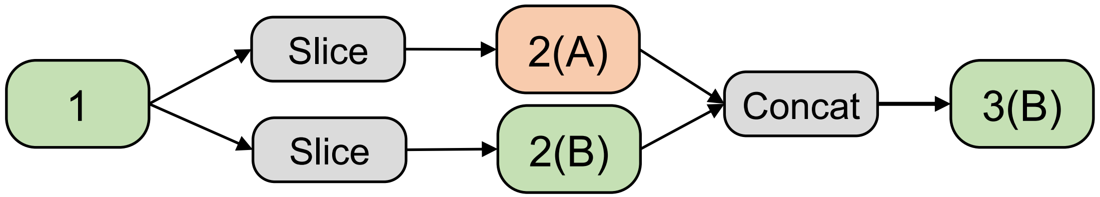
- PIM-aware graph transformations generate extra nodes (**Slice, Pad, Concat**) to adjust tensor shapes and placement for correctness



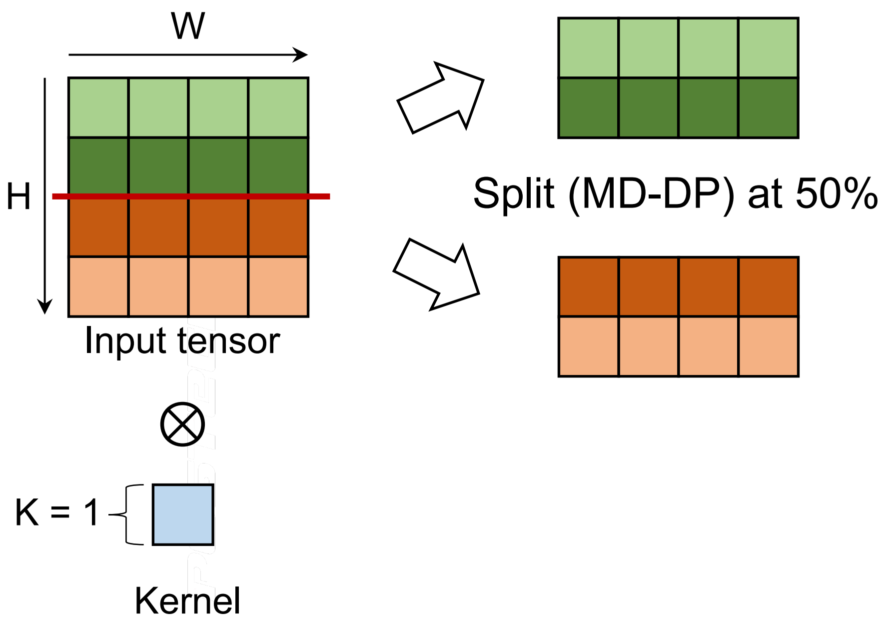
- **Can increase** the cost of graph transformations

➔ **Memory Optimizer** can ***eliminate*** them in runtime

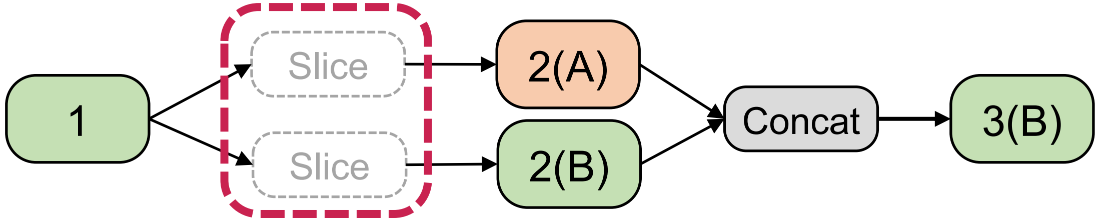
Memory Optimizer



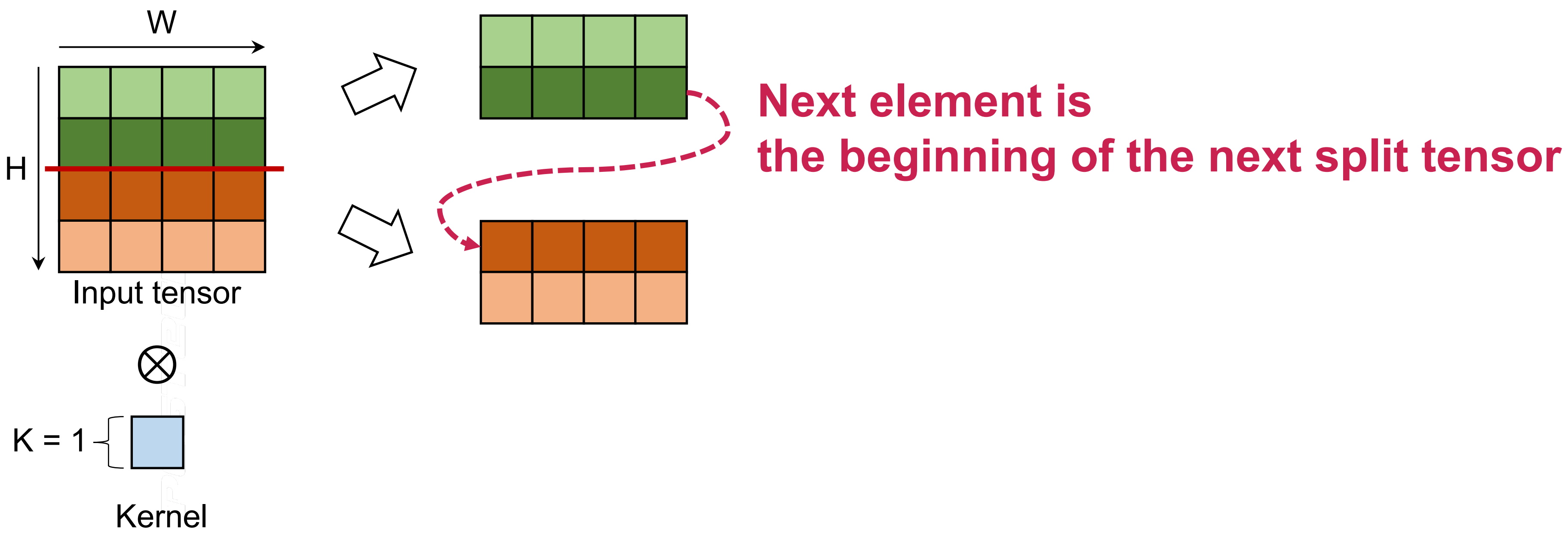
- Suppose the input tensor for an 1x1 convolution layer is split at 50% ratio



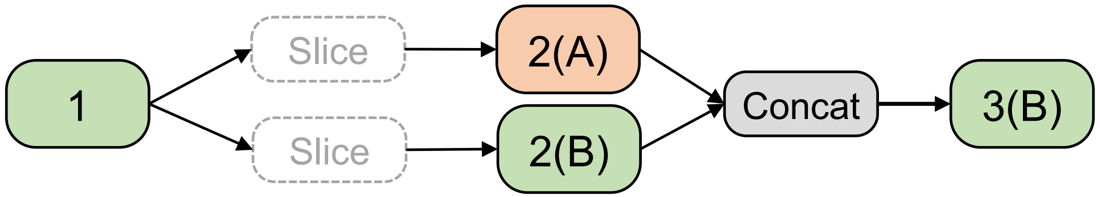
Memory Optimizer



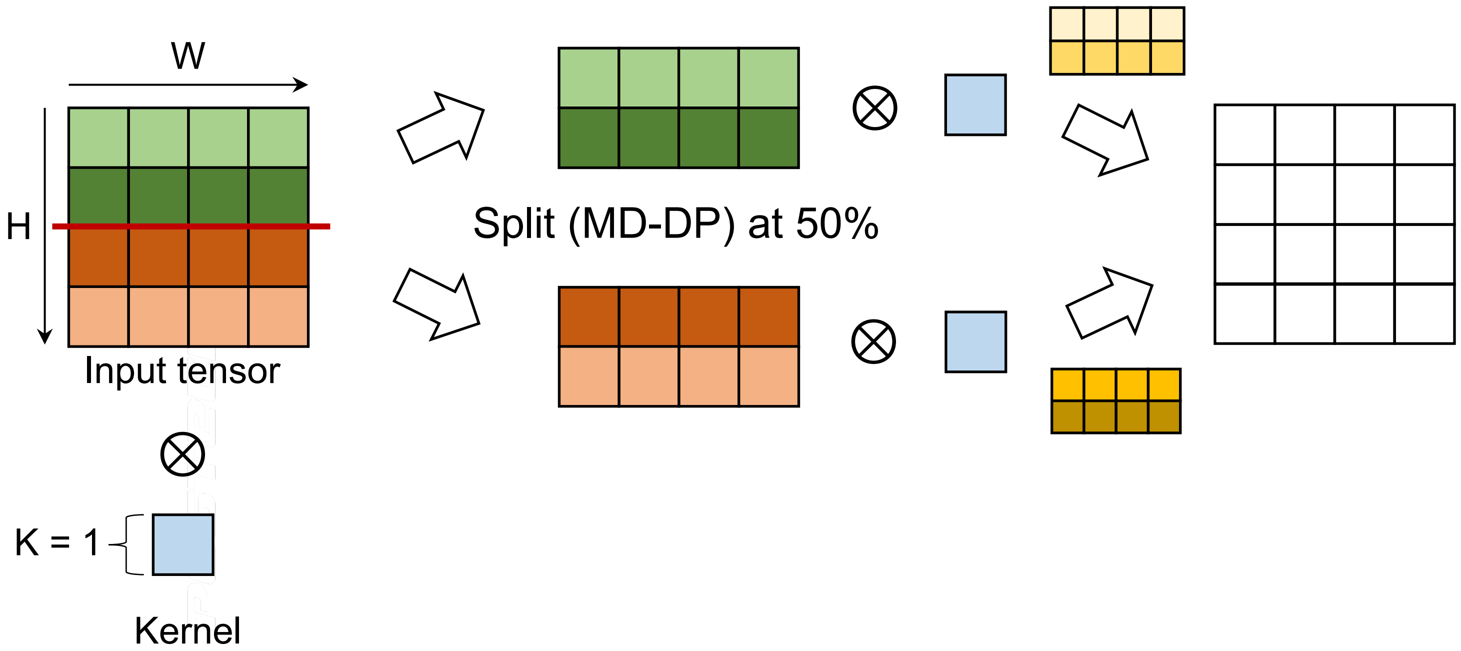
- Suppose the input tensor for an 1x1 convolution layer is split at 50% ratio
- Split tensors are already **contiguous** → No need to "**Slice**" tensors



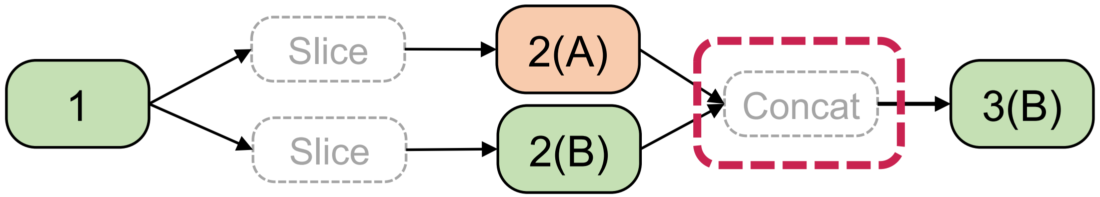
Memory Optimizer



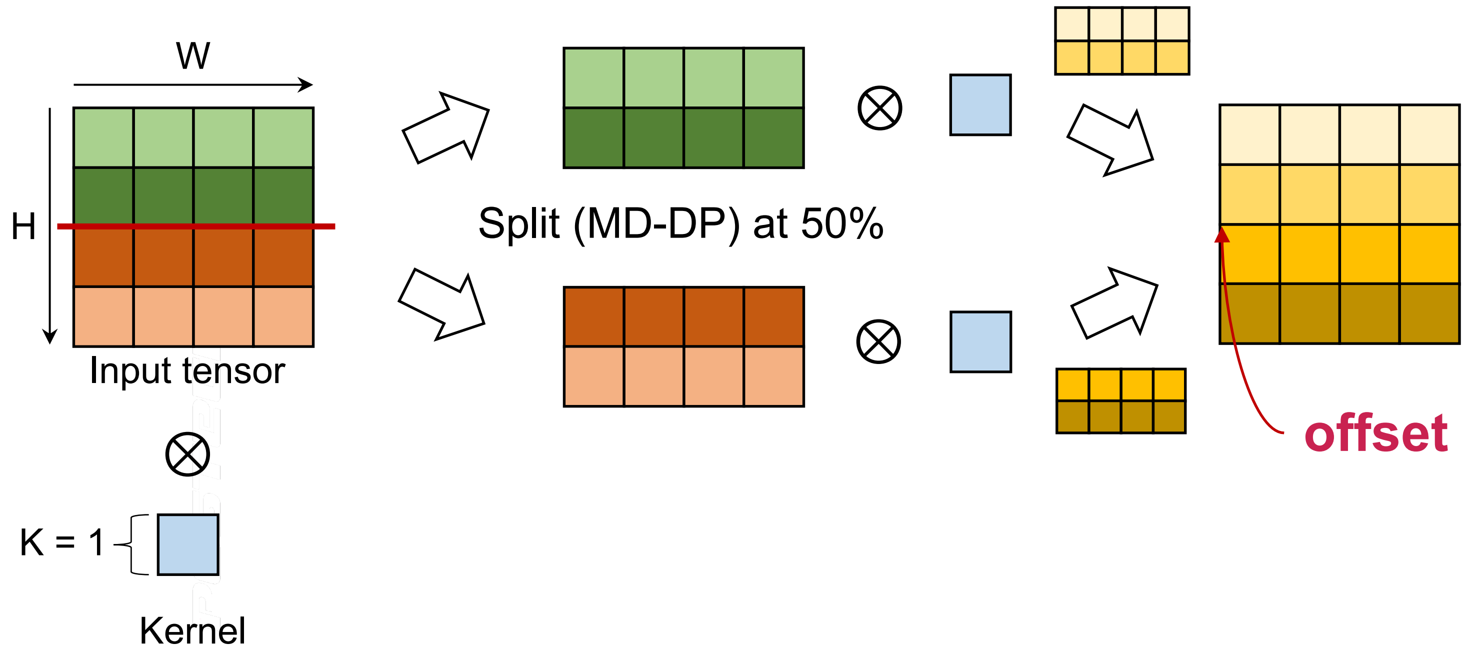
- Suppose the input tensor for an 1x1 convolution layer is split at 50% ratio
- Allocate a **contiguous memory region** for the output tensors



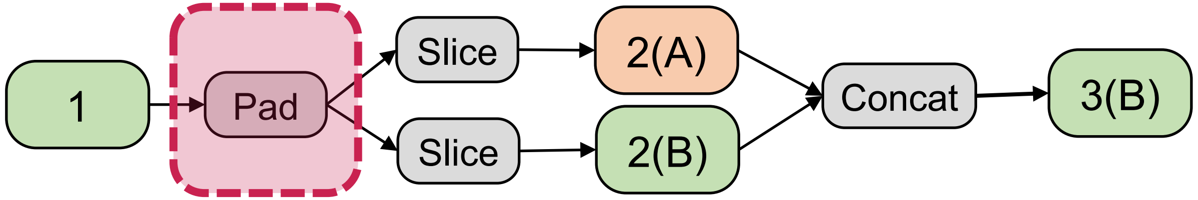
Memory Optimizer



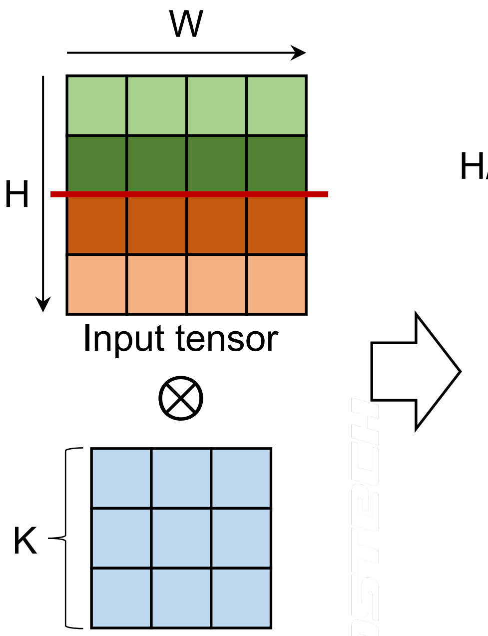
- Suppose the input tensor for an 1x1 convolution layer is split at 50% ratio
- Allocate a **contiguous memory region** for the output tensors
- Write the output tensors to the **specified offset** ➔ **Remove** "**Concat**" operator



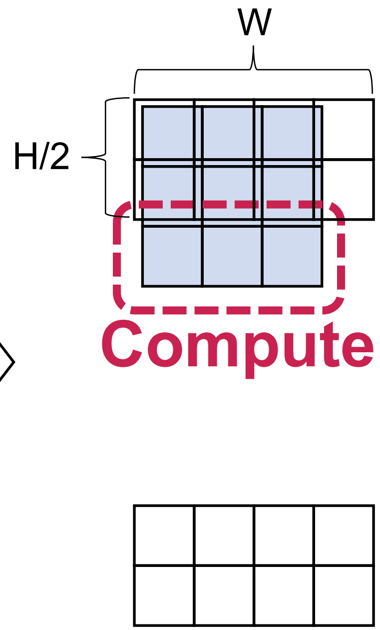
Memory Optimizer



- When the kernel size is not 1x1, **additional "Pad" operator is** needed

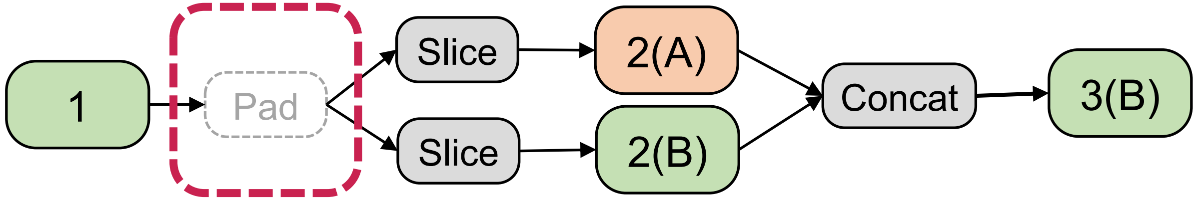


Split (MD-DP) at 50%

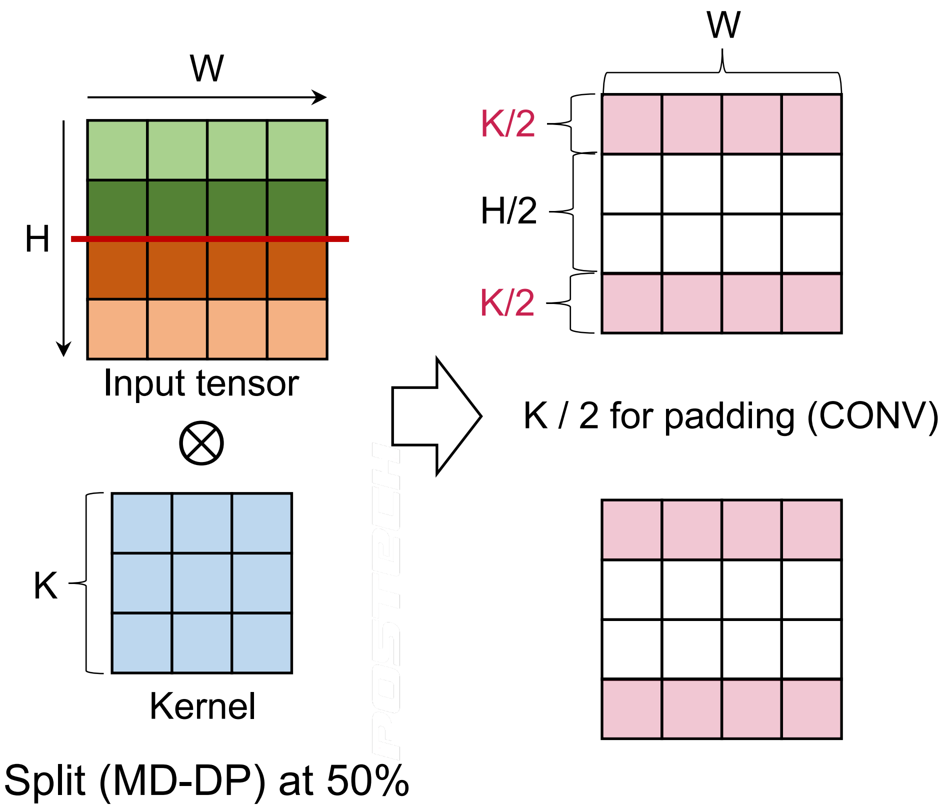


Compute wrong results at the boundary

Memory Optimizer



- When the kernel size is not 1x1, **additional "Pad" operator is** needed
- Reserve **more space** for padding ➔ Remove "**Pad**" operator



Outline

Motivation

Overview of PIMFlow

PIM-enabled GPU Memory Architecture

PIMFlow

- PIM-Aware Graph Transformation
- Execution Mode and Task Size Search
- TVM Back-End for DRAM-PIM

Evaluation result

Methodology

- **Evaluated Models**
  - EfficientNet-V1, MobileNet-V2, MnasNet, ResNet-50, VGG-16

Methodology

- **Evaluated Models**
  - EfficientNet-V1, MobileNet-V2, MnasNet, ResNet-50, VGG-16

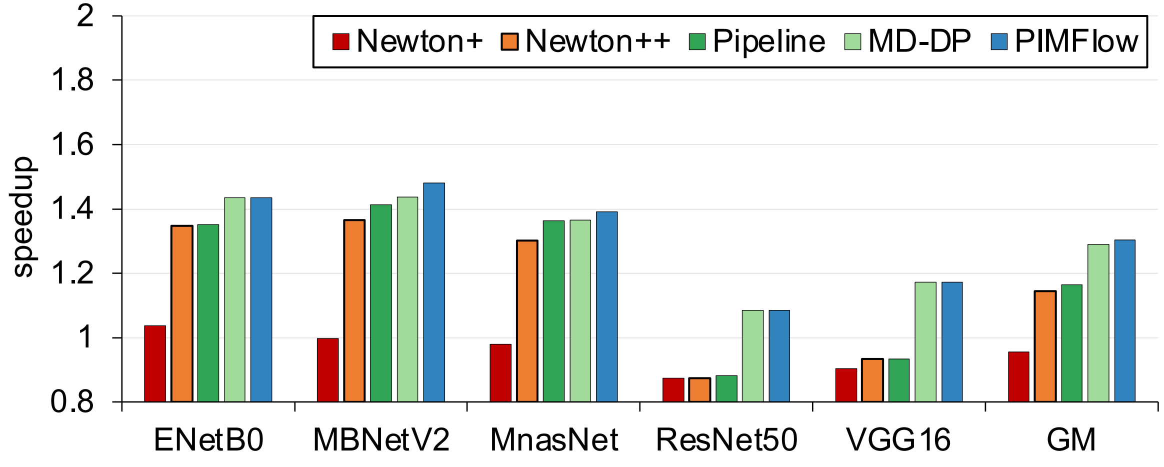
- **Simulators**
  - GPU: Accel-Sim
  - DRAM-PIM: Ramulator (DRAM-PIM command latency from Newton[1])

[DRAM configuration]

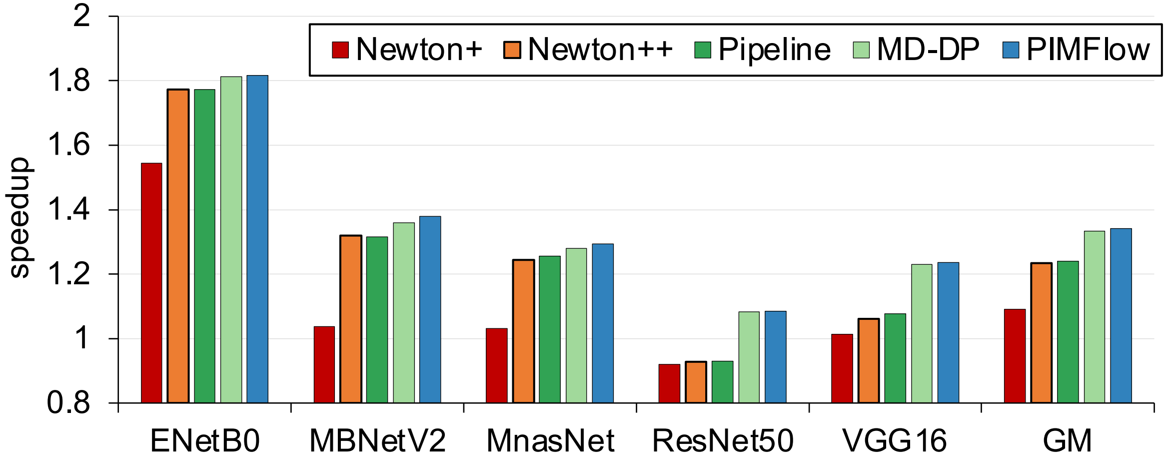
Num of Ranks	1	Num of Column I/Os per row	32
Num of Banks	16	Column I/O bit width	256 b
Global buffer size	4 KB	Num of Multipliers per bank	16
Timing Parameters (in clock cycles) t_{BL}: 2, t_{CL}: 11, t_{RP}: 11, t_{RCD}: 11, t_{CCD}: 2, t_{RAS}: 25			

[1] M. He et al., "Newton: A DRAM-maker's Accelerator-in-Memory (AiM) Architecture for Machine Learning," MICRO, 2020

Execution Time (Speedup)



[All CONV layers]



[End-to-end model inference]

- Evaluated on five configurations

Newton+ $\xrightarrow[\text{GW Latency Hiding}]{\text{Multiple GB}}$ **Newton++**

[PIM command optimizations]

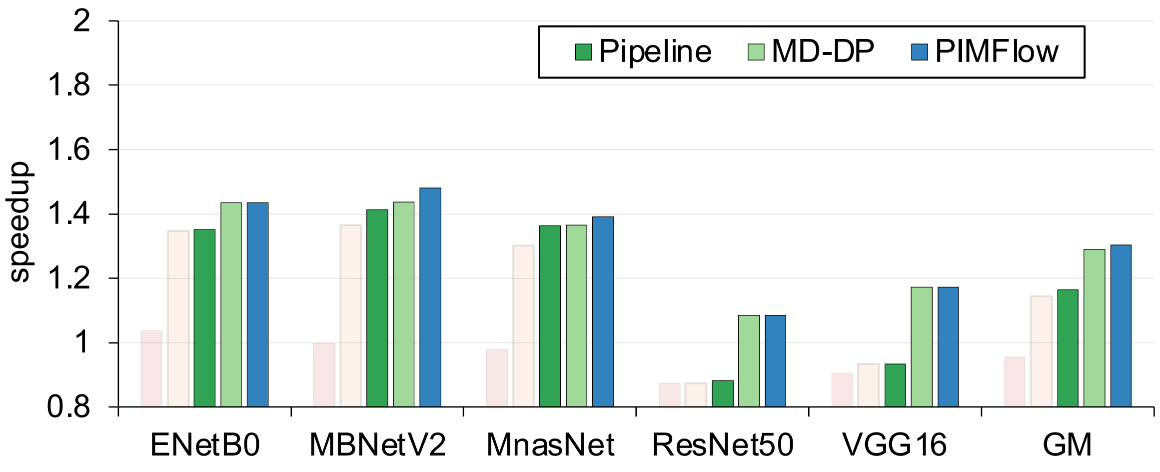
PIMFlow: **Pipeline** + **MD-DP**
(Based on Newton++)

- Inference time normalized to the GPU baseline

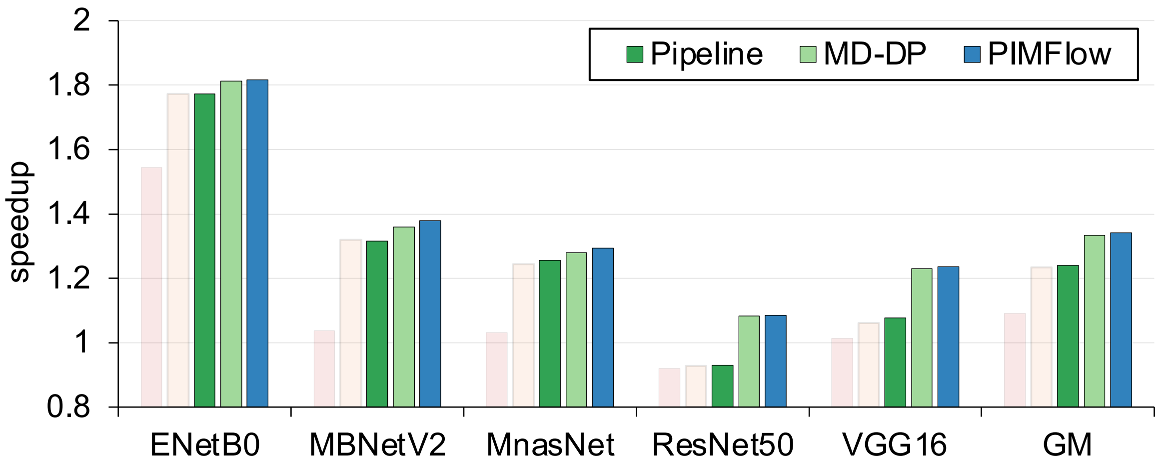
POSTECH

Execution Time (Speedup)

PIMFlow: **Pipeline** + **MD-DP**



[All CONV layers]

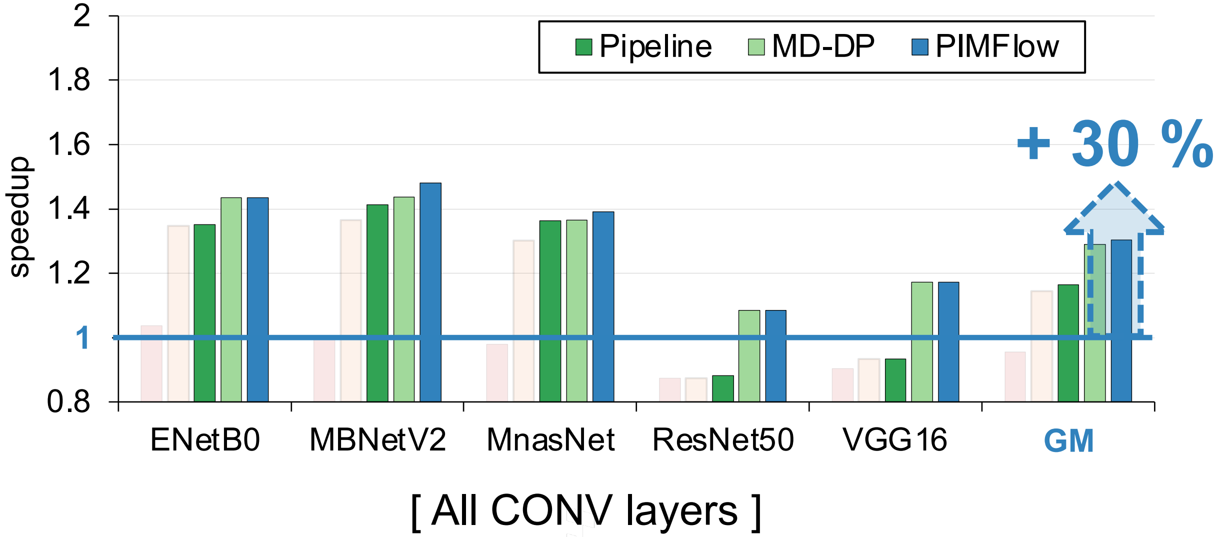


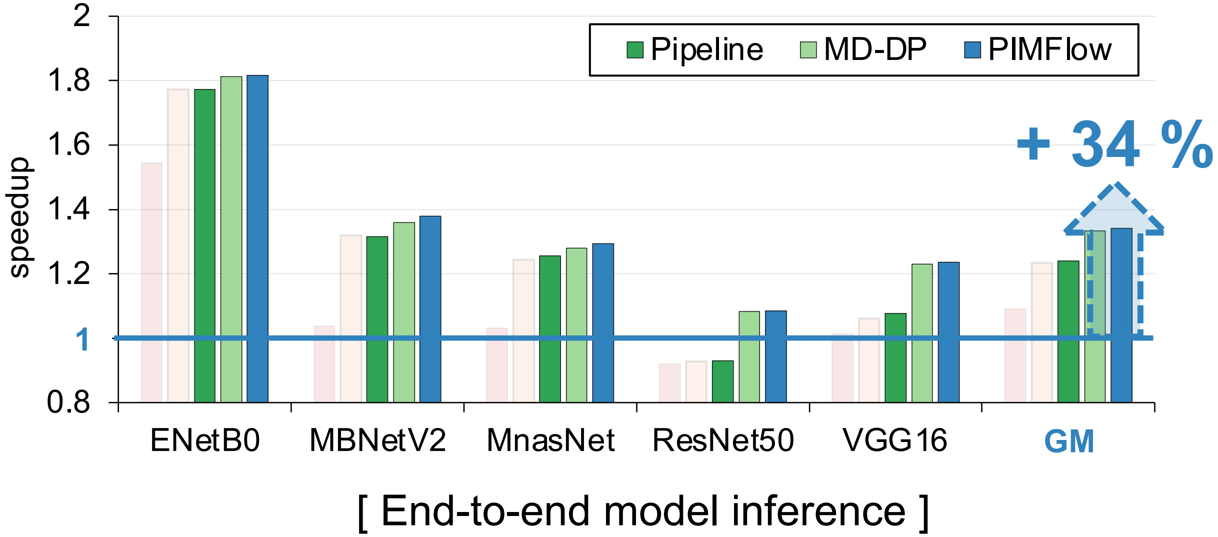
[End-to-end model inference]

- **PIMFlow** ***enables*** mixed-parallelism for all evaluated models (**MD-DP** and **Pipeline**)

Execution Time (Speedup)

PIMFlow: Pipeline + MD-DP



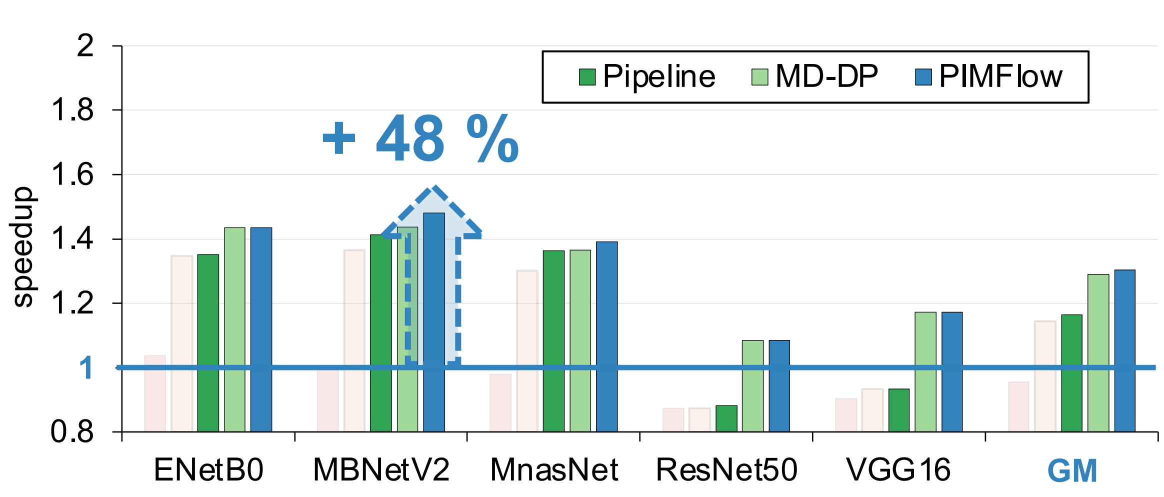


- **PIMFlow** ***enables*** mixed-parallelism for all evaluated models (**MD-DP** and **Pipeline**)

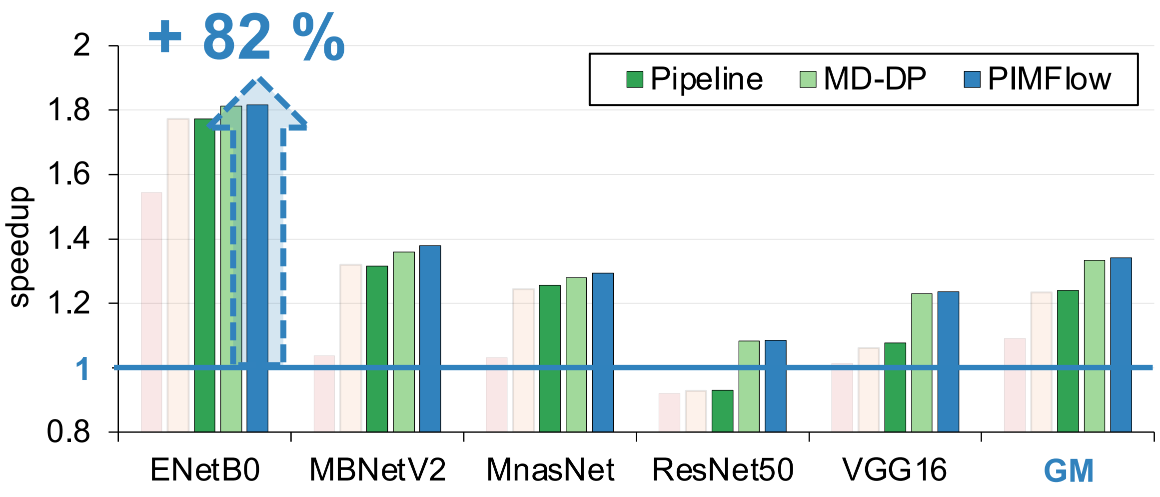
➔ **30% (34%)** speedup for **all CONV (end-to-end)** performance on average

Execution Time (Speedup)

PIMFlow: **Pipeline** + **MD-DP**



[All CONV layers]



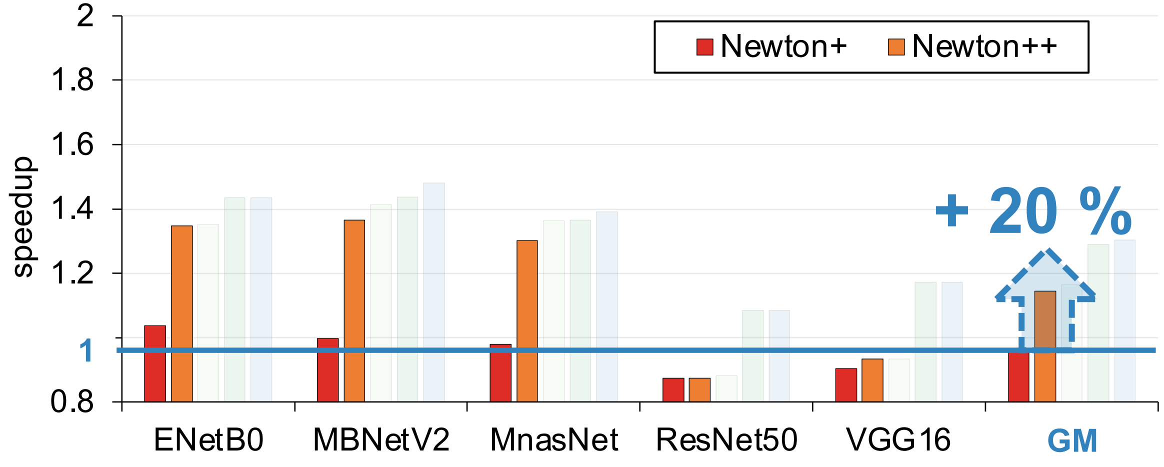
[End-to-end model inference]

- **PIMFlow** ***enables*** mixed-parallelism for all evaluated models (**MD-DP** and **Pipeline**)

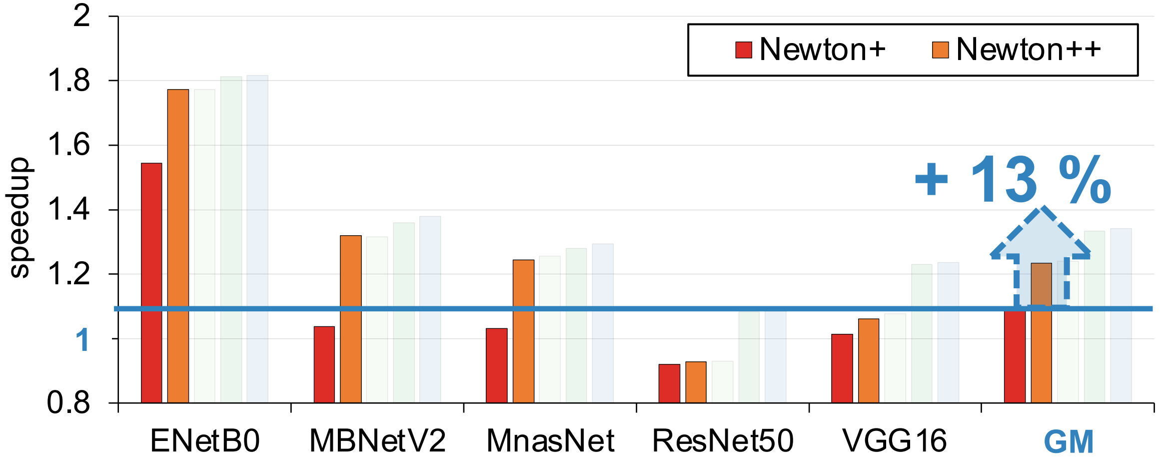
➔ **30% (34%)** speedup for **all CONV (end-to-end)** performance on average

➔ Up to **48% (82%)** speedup for **all CONV (end-to-end)** performance

Execution Time (Speedup)



[All CONV layers]

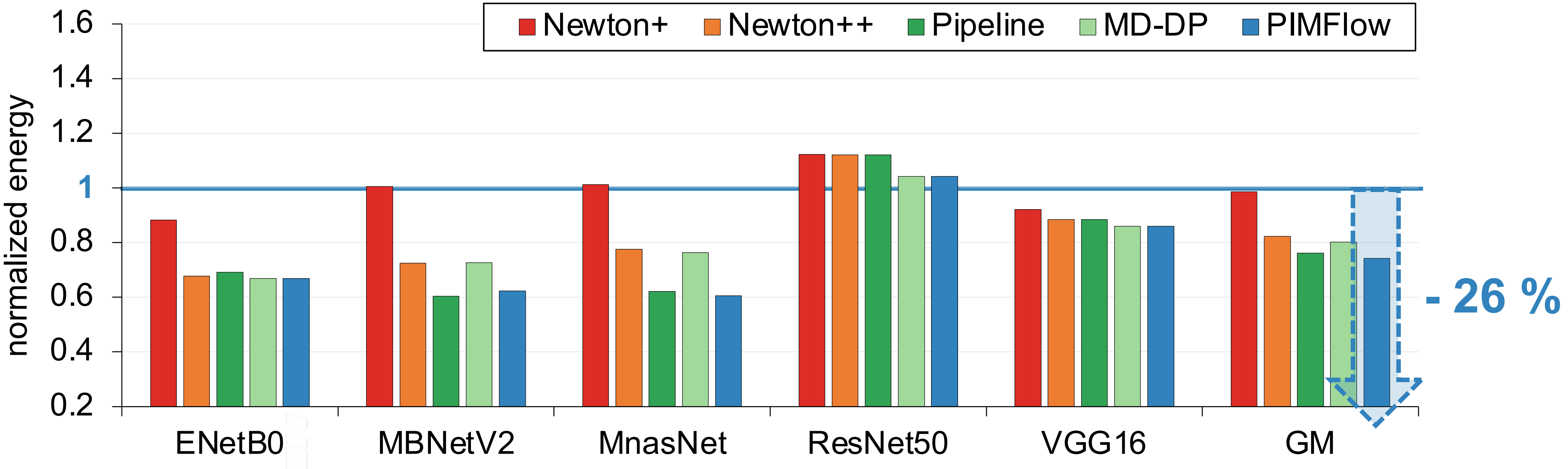


[End-to-end model inference]

- **Optimizing PIM commands** alone can boost PIM capabilities (Multiple Global Buffers & GWRITE Latency Hiding)

➔ Provide a **20% (13% end-to-end) speedup** on average

Energy Consumption



- **PIMFlow provides a significant energy saving by 26%**
  - Due to reduced runtime and energy-efficient fixed-function MAC logic in PIM

PIMFlow

Compiler and Runtime Support for CNNs on PIM-enabled DRAM

PIM-Aware Graph Transformation

Systematic creation of graph-level parallelism

PIMFlow

Compiler and Runtime Support for CNNs on PIM-enabled DRAM

PIM-Aware Graph Transformation
Systematic creation of graph-level parallelism

Execution Mode and Task Size Search
Optimal graph transformation search

PIMFlow

Compiler and Runtime Support for CNNs on PIM-enabled DRAM

PIM-Aware Graph Transformation

Systematic creation of graph-level parallelism

Execution Mode and Task Size Search

Optimal graph transformation search

TVM DRAM-PIM Back-End

DRAM-PIM command generation, scheduling and opt.

PIMFlow

Compiler and Runtime Support for CNNs on PIM-enabled DRAM

PIM-Aware Graph Transformation
Systematic creation of graph-level parallelism

Execution Mode and Task Size Search
Optimal graph transformation search

TVM DRAM-PIM Back-End
DRAM-PIM command generation, scheduling and opt.

Code available at https://github.com/yongwonshin/PIMFlow

PIMFlow: Compiler and Runtime Support for CNN Models on Processing-in-Memory DRAM

CGO 2023

Yongwon Shin[*,1], Juseong Park[*,2], Sungjun Cho[2], Hyojin Sung[1,2]

[1]Graduate School of AI
[2]Dept. of Computer Science and Engineering
Pohang University of Science and Technology (POSTECH), South Korea
[*]Equal contribution

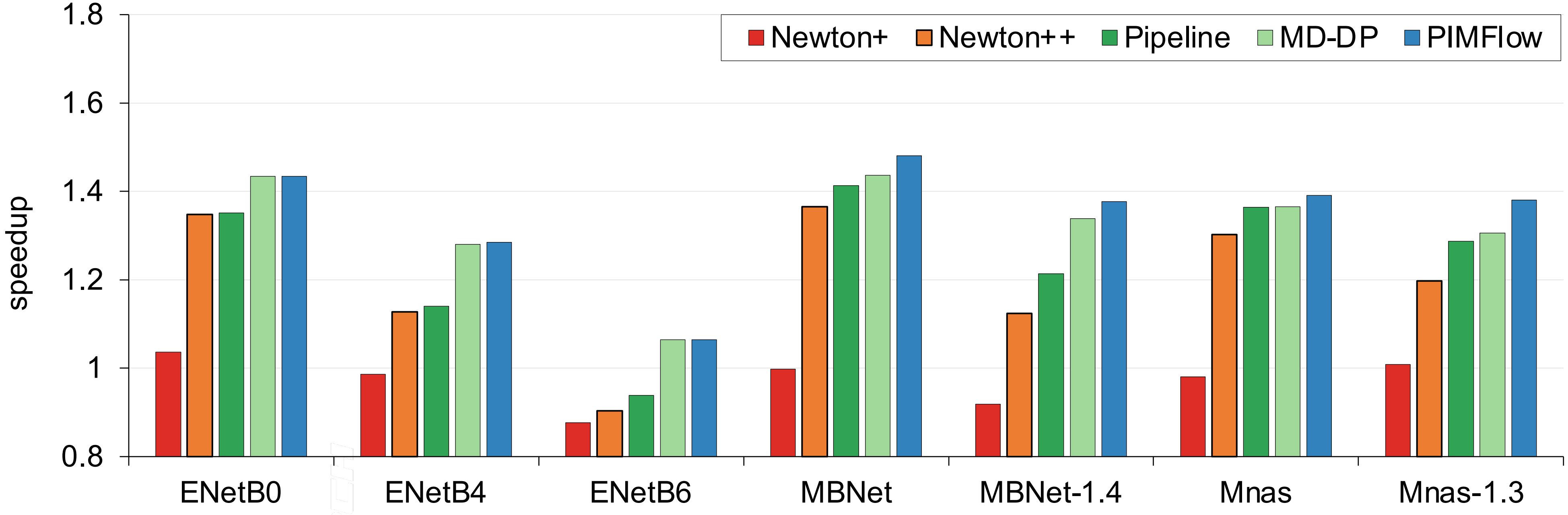
Code available at https://github.com/yongwonshin/PIMFlow

POSTECH

Backup Slides

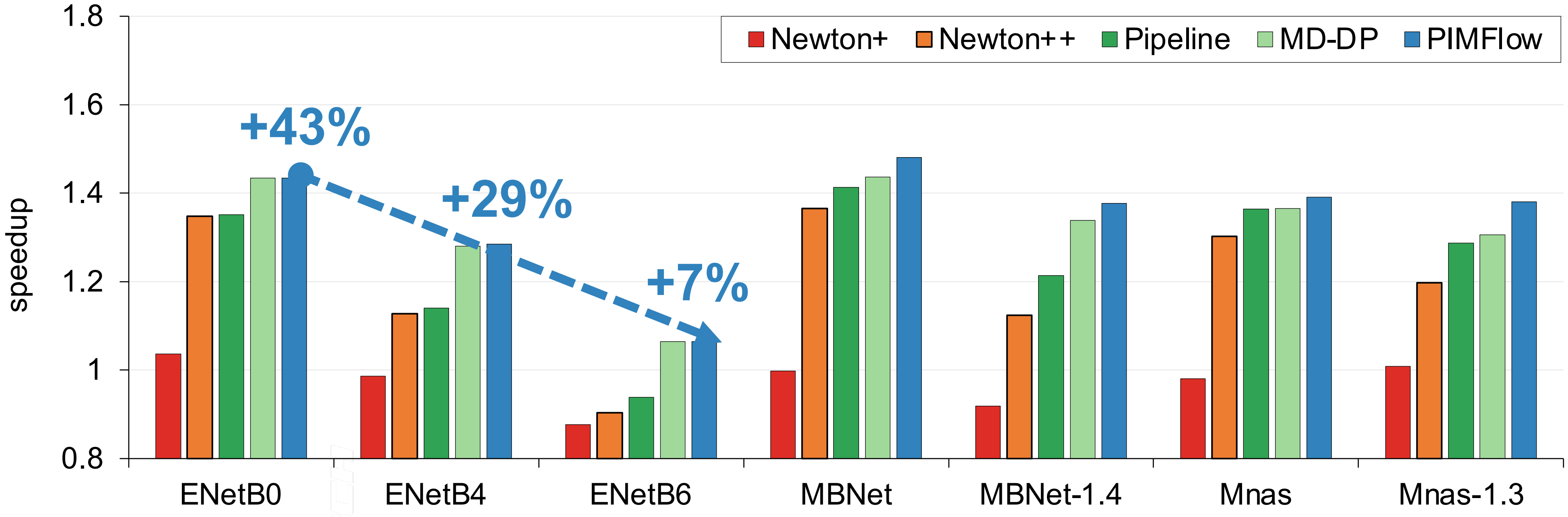
- Model Size Sensitivity Study
- Memory Optimizer

Model Size Sensitivity Study



- As model size increases, 1x1 CONV has higher arithmetic intensity and data reuse

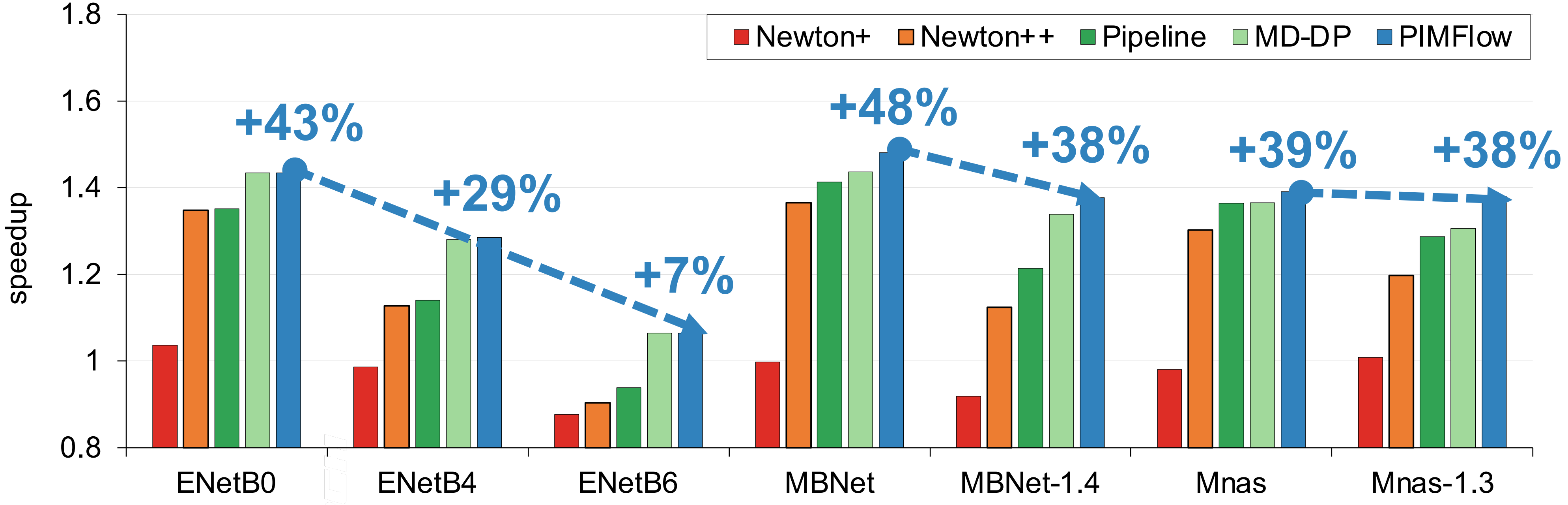
Model Size Sensitivity Study



- As model size increases, 1x1 CONV has higher arithmetic intensity and data reuse

➔ **ENetB6** shows only **7%** speedup compared to **43%** in **ENetB0**

Model Size Sensitivity Study

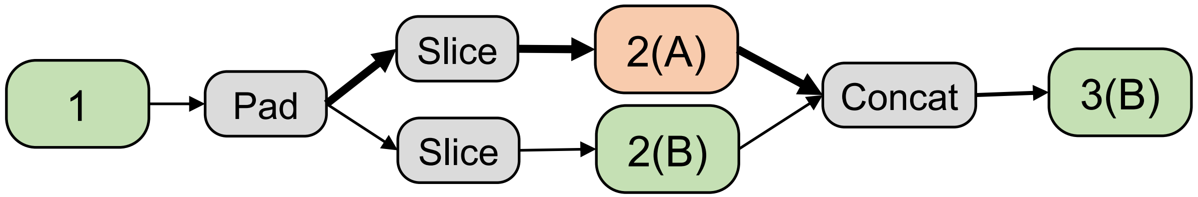


- As model size increases, 1x1 CONV has higher arithmetic intensity and data reuse

➔ **ENetB6** shows only **7%** speedup compared to **43%** in **ENetB0**

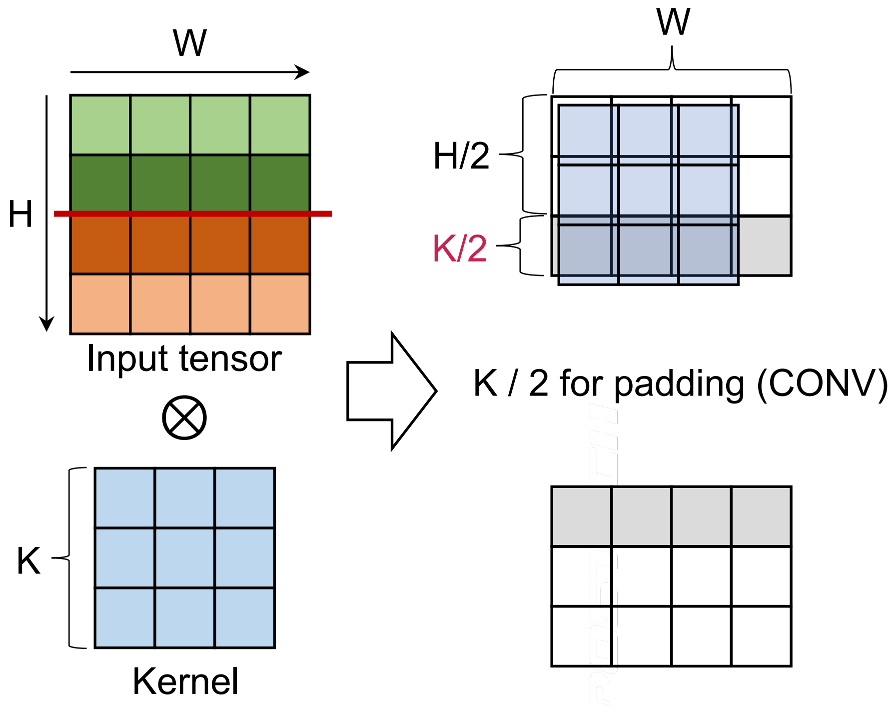
➔ **MBNet** and **Mnas** also undergo slightly dropped performance

Memory Optimizer



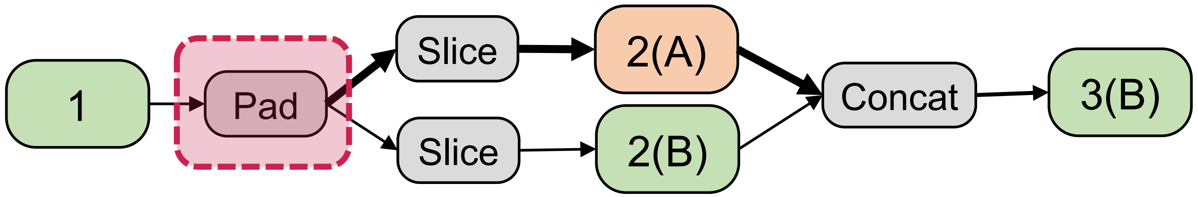
- Naïve split will generate wrong result since non-zero padding is needed

➔ Allocate space for non-zero padding in advance

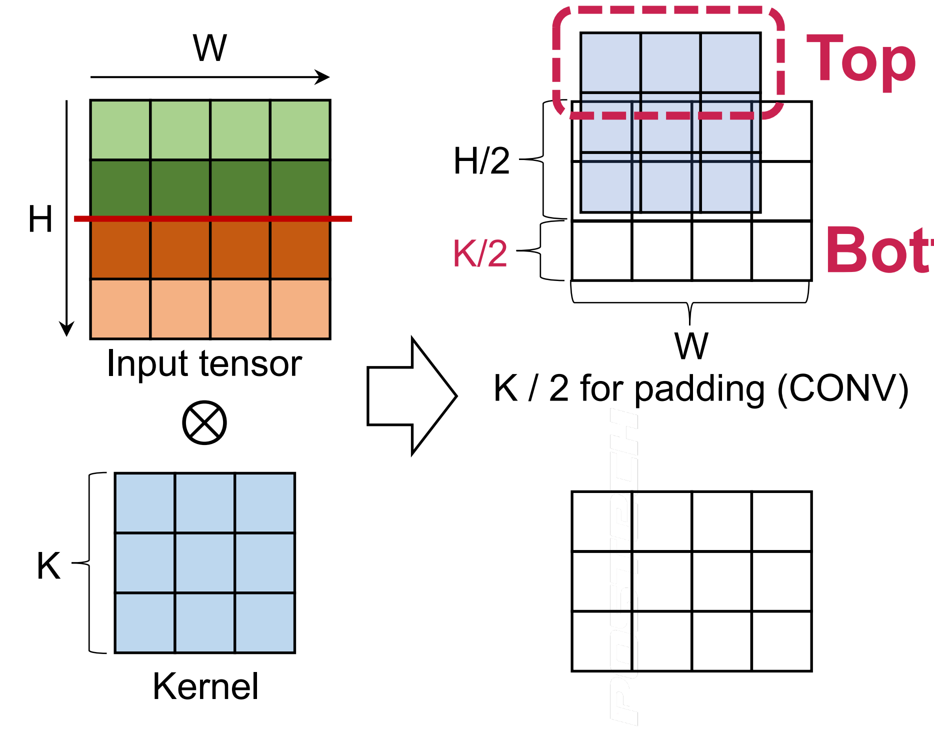


Split (MD-DP) at 50%

Memory Optimizer



- cuDNN does not allow asymmetric padding

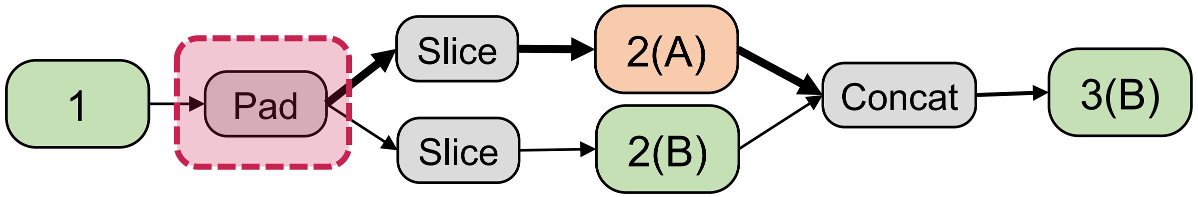


Top elements requires <u>*one*</u> padding

Bottom elements requires <u>*zero*</u> padding

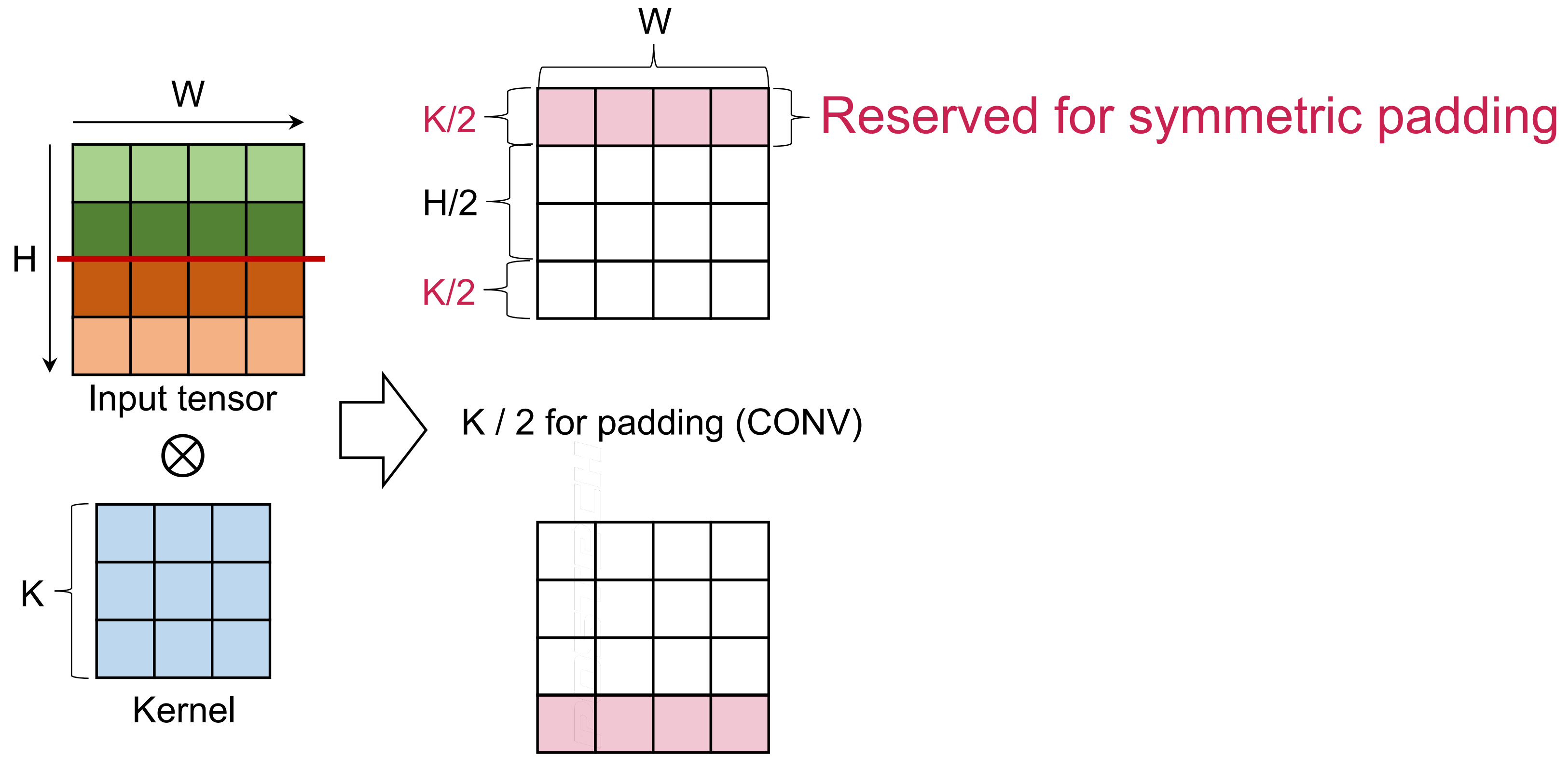
Split (MD-DP) at 50%

Memory Optimizer



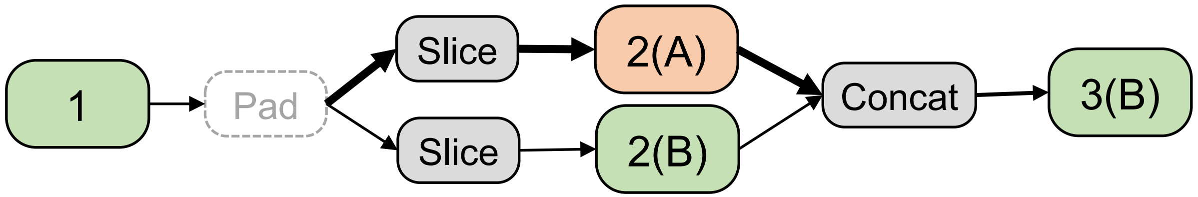
- cuDNN does not allow asymmetric padding

➔ Reserve more space for symmetric padding



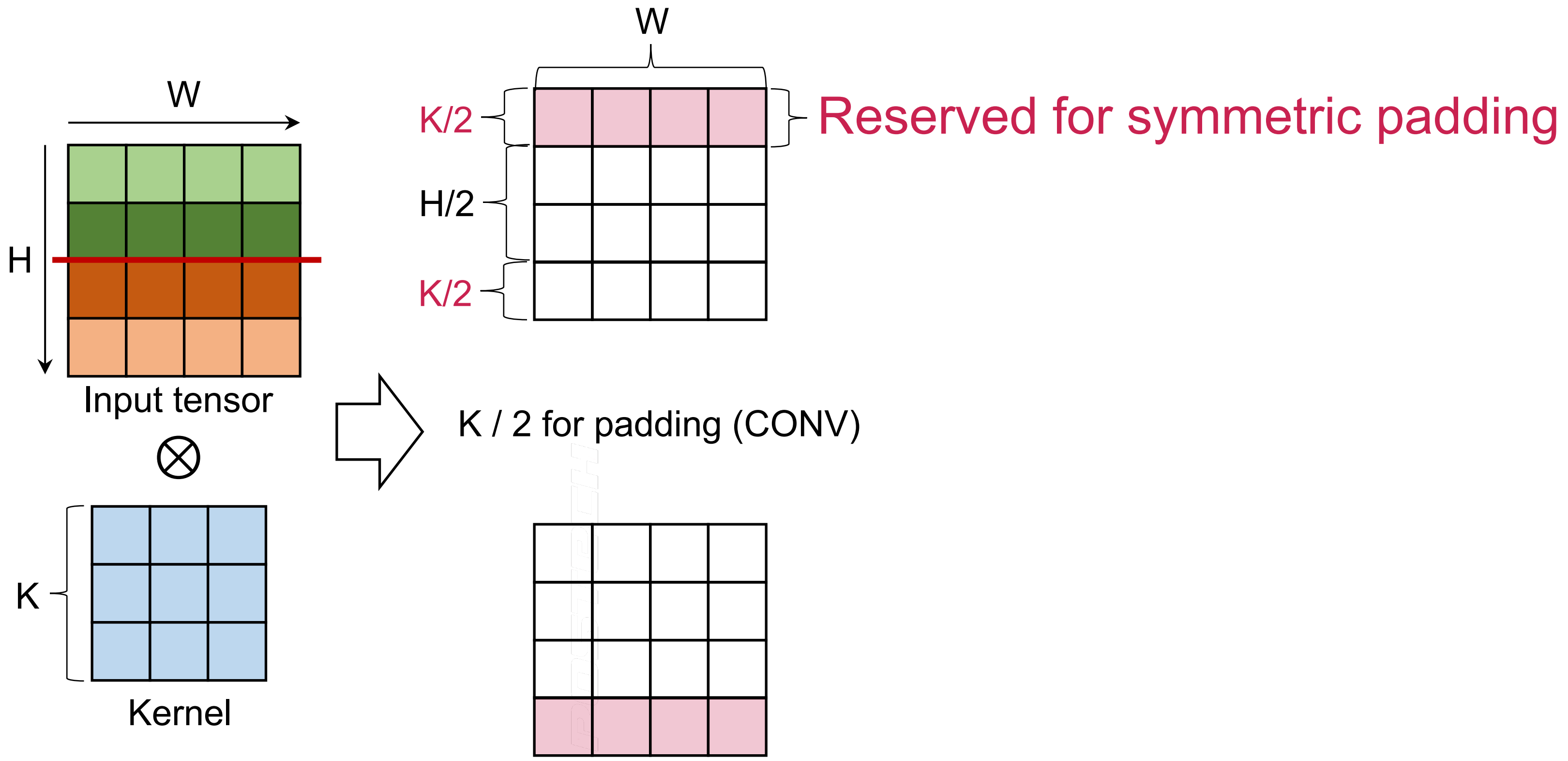
Split (MD-DP) at 50%

Memory Optimizer



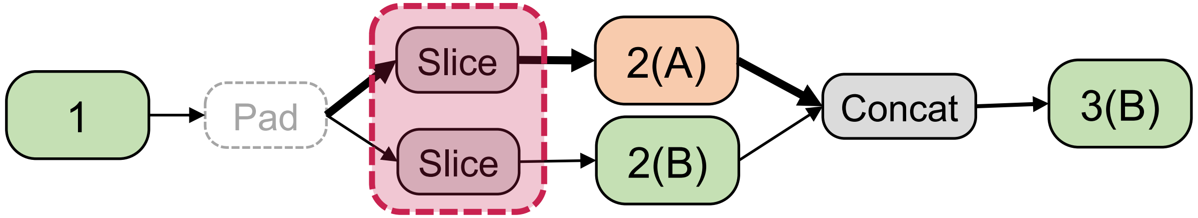
- cuDNN does not allow asymmetric padding

➔ Reserve more space for symmetric padding ➔ Remove "**Pad**" operator

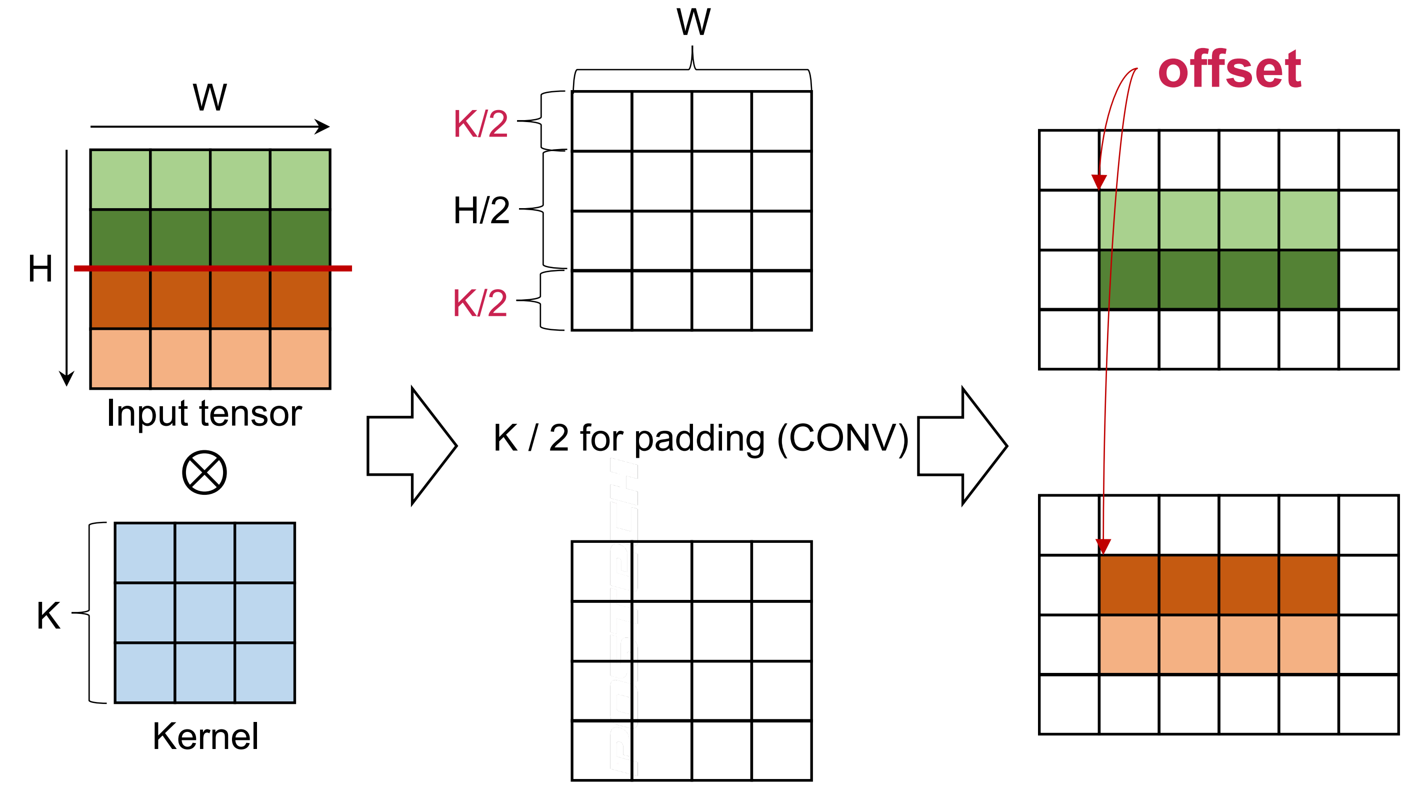


Split (MD-DP) at 50%

Memory Optimizer



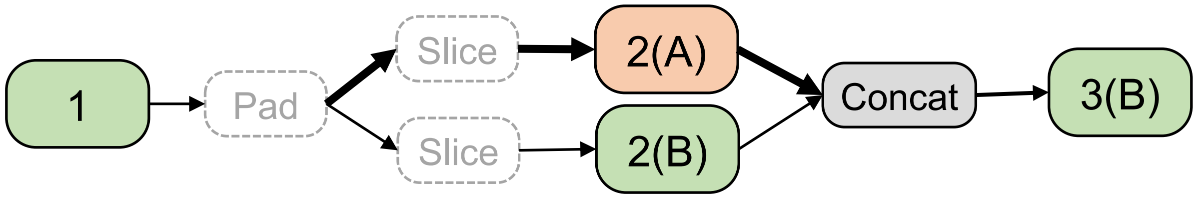
- Locate split tensor to the specified offset



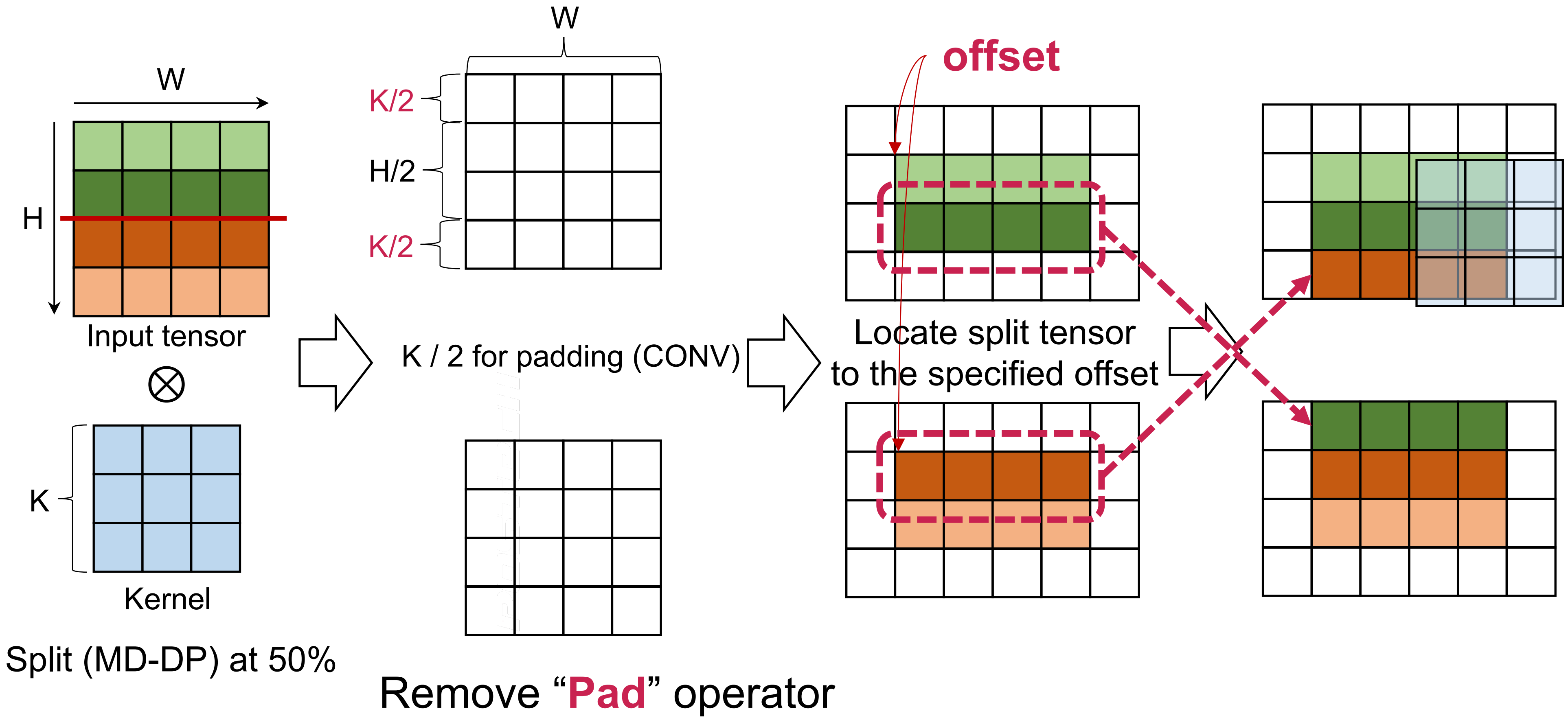
Split (MD-DP) at 50%

Remove "**Pad**" operator

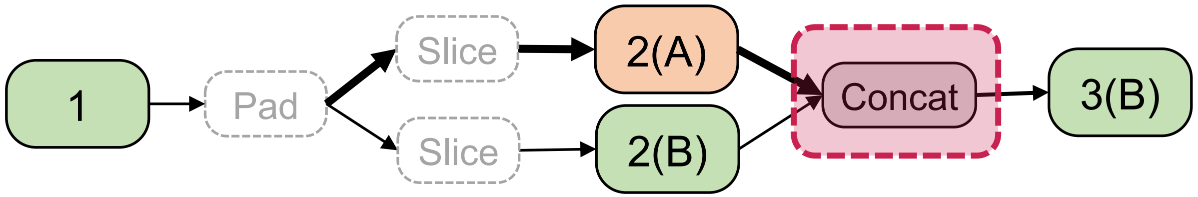
Memory Optimizer



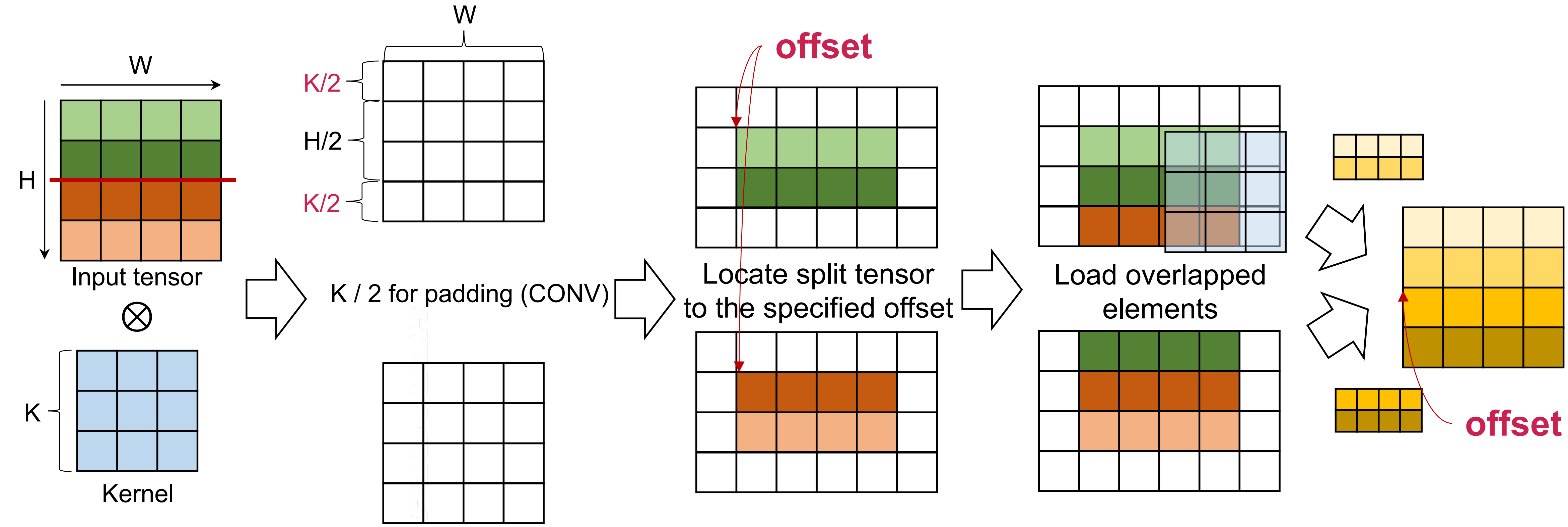
- Load overlapped elements ➔ Remove "**Slice**" operator



Memory Optimizer



- Write to contiguous region to the specified offset

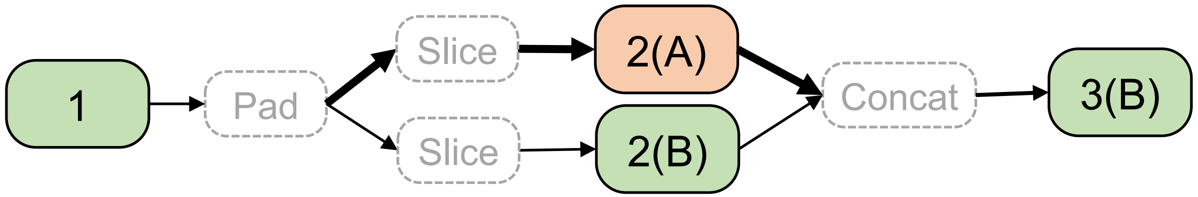


Split (MD-DP) at 50%

Remove “**Pad**” operator

Remove “**Slice**” operator

Memory Optimizer



- Write to contiguous region to the specified offset ➔ Remove "**Concat**" operator