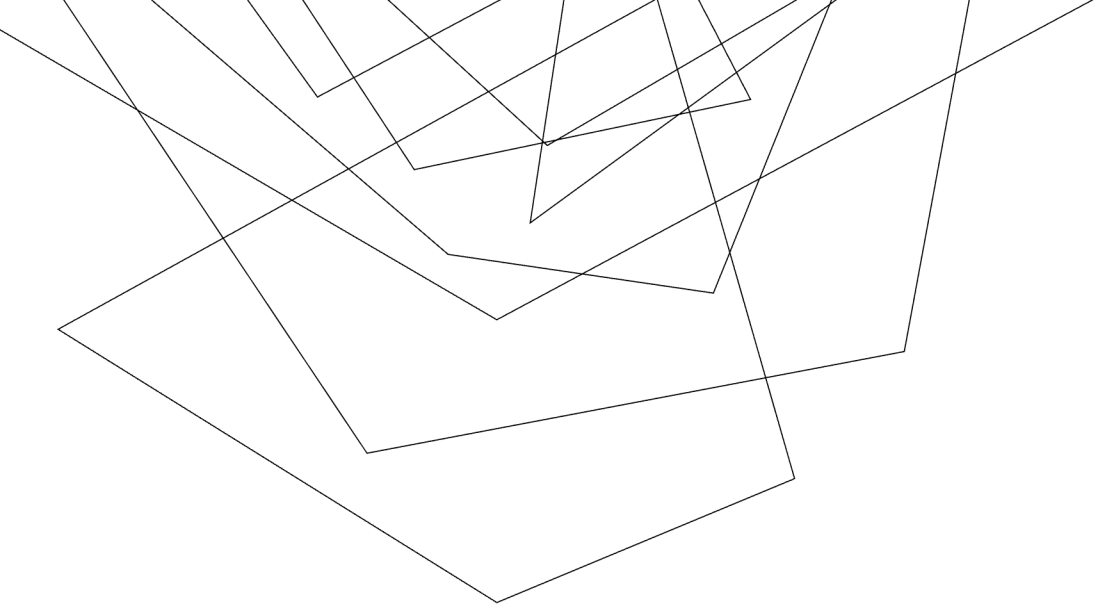


Optimizing Compiler Support for Processing-in-DRAM Hardware

NVRAMOS workshop

10/24/2025

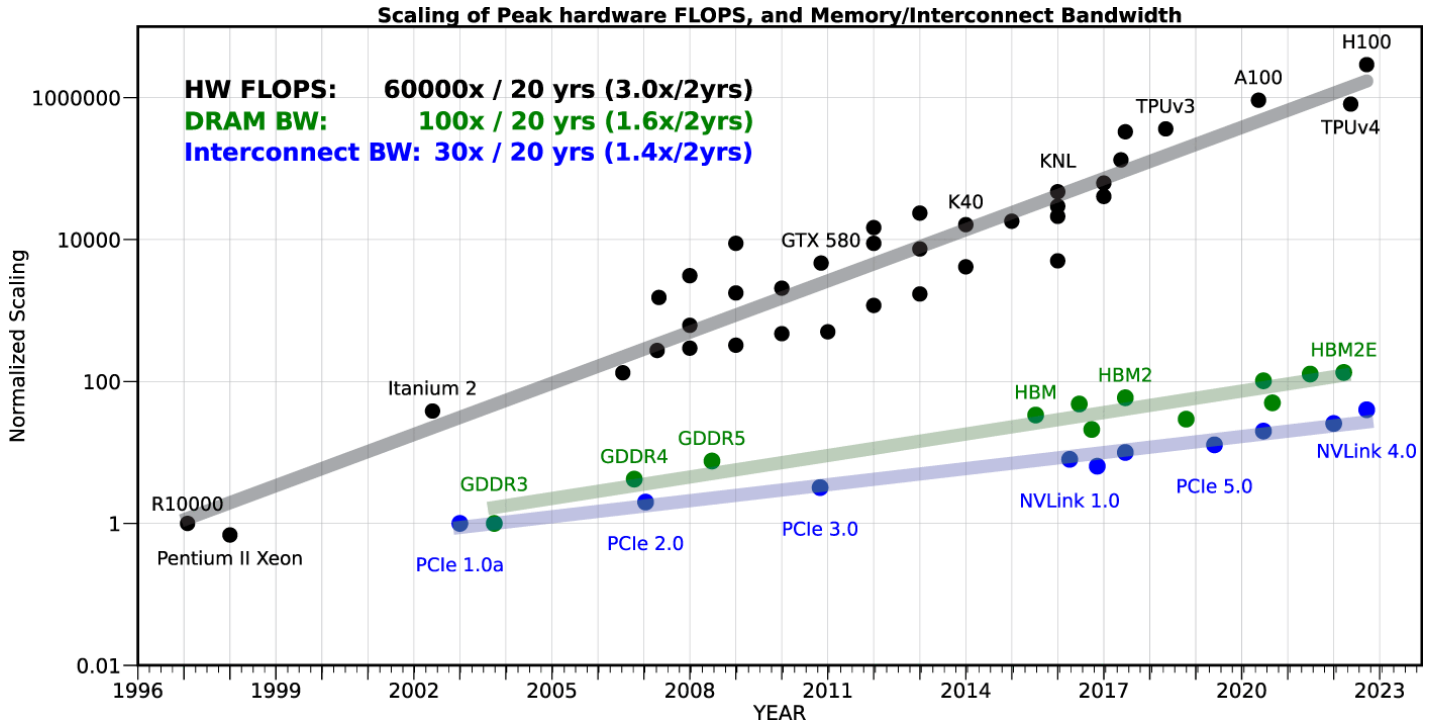
Hyojin Sung

AGENDA

- Software stack for Processing-in-Memory hardware
- PIMFlow: Compiler and Runtime Support for CNN Models on Processing-in-Memory DRAM (CGO '23)
- ATiM: Autotuning Tensor Programs for Processing-in-DRAM (ISCA '25)

MEMORY WALL IN MODERN SYSTEMS

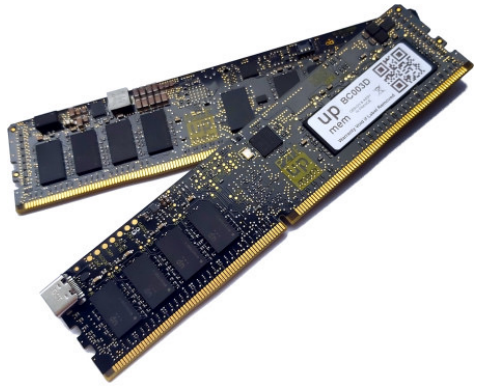
- **Memory performance is scaling more slowly** than compute speed
  - Creating a performance bottleneck
- Modern applications are increasingly **bandwidth-hungry**
  - Database systems, genomic analysis, ML/DL inferences (e.g., LLMs)



https://www.computer.org/csdl/magazine/mi/2024/03/10477550/1VBAlC0uYiA

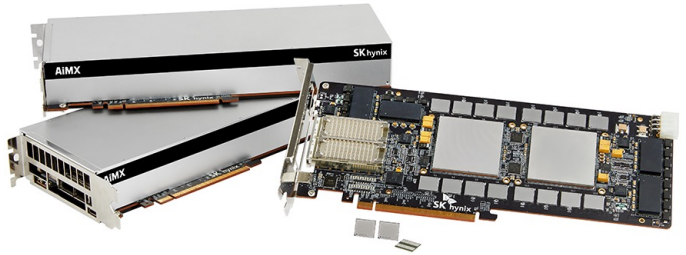
PROCESSING-IN-DRAM PRODUCTS/PROTOTYPES

- **Processing-in-DRAM (DRAM-PIM)** integrates compute units that **directly process data stored in memory**, significantly reducing data movement
- Several hardware/memory vendors have released DRAM-PIM products and prototypes
  - Include general-purpose cores vs specialized acceleration logic (e.g., MAC)



UPMEM[1]

SAMSUNG HBM2-PIM Aquabolt-XL[2]

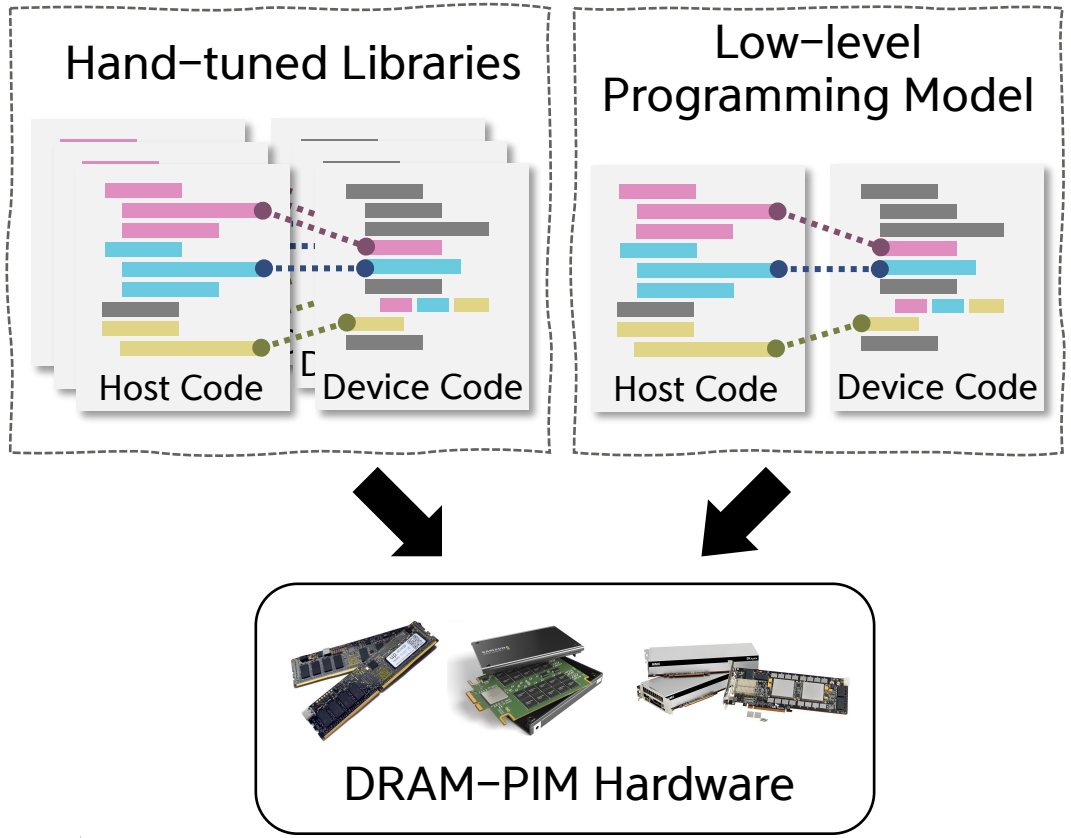


SK Hynix GDDR6-AiM[3]

[1] https://www.newswire.com/news/upmem-raises-7m-to-revolutionize-ai-and-analytics-processing-22126102
[2] https://news.samsung.com/global/samsung-electronics-introduces-industrys-first-512gb-cxl-memory-module
[3] https://news.skhynix.com/sk-hynix-debuts-first-gddr6-aim-accelerator-card-aimx-for-generative-ai/

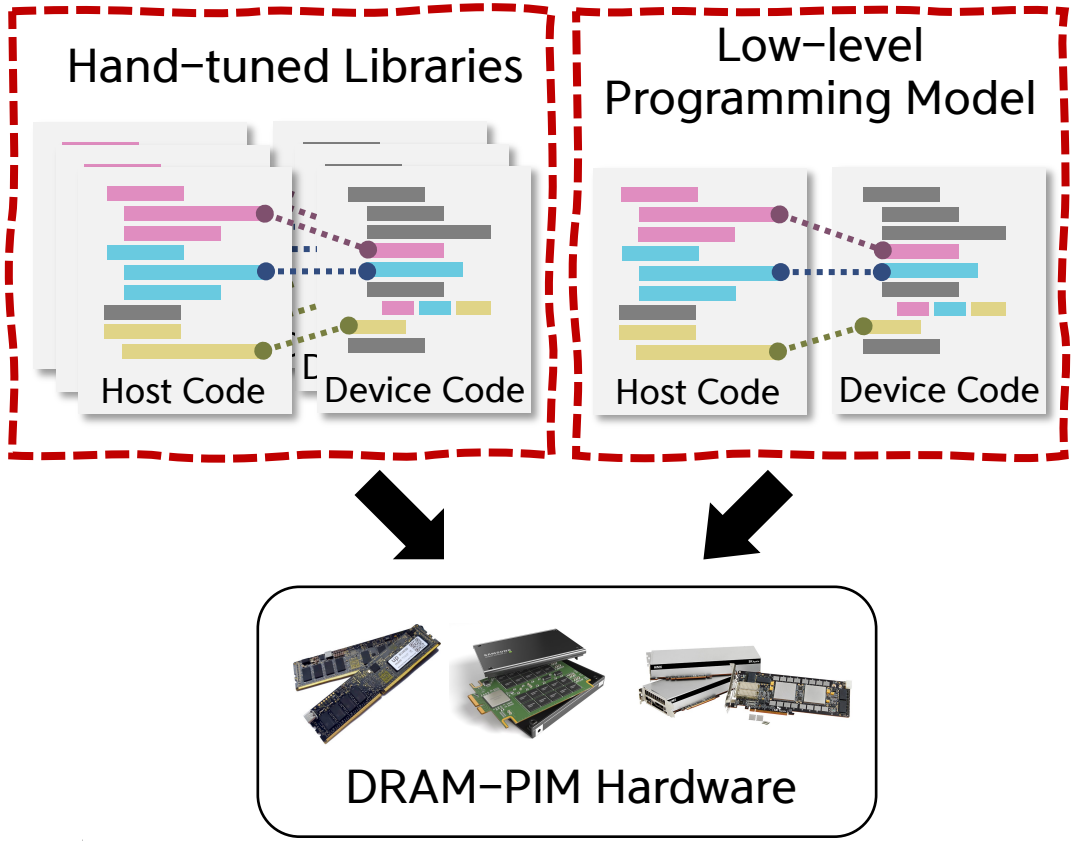
CURRENT SOFTWARE STACKS FOR DRAM-PIM

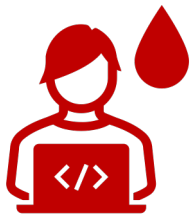
- Current software stacks support a narrow set of workloads with **hand-tuned libraries** or **low-level programming models**



CURRENT SOFTWARE STACKS FOR DRAM-PIM

- Current software stacks support a narrow set of workloads with **hand-tuned libraries** or **low-level programming models**

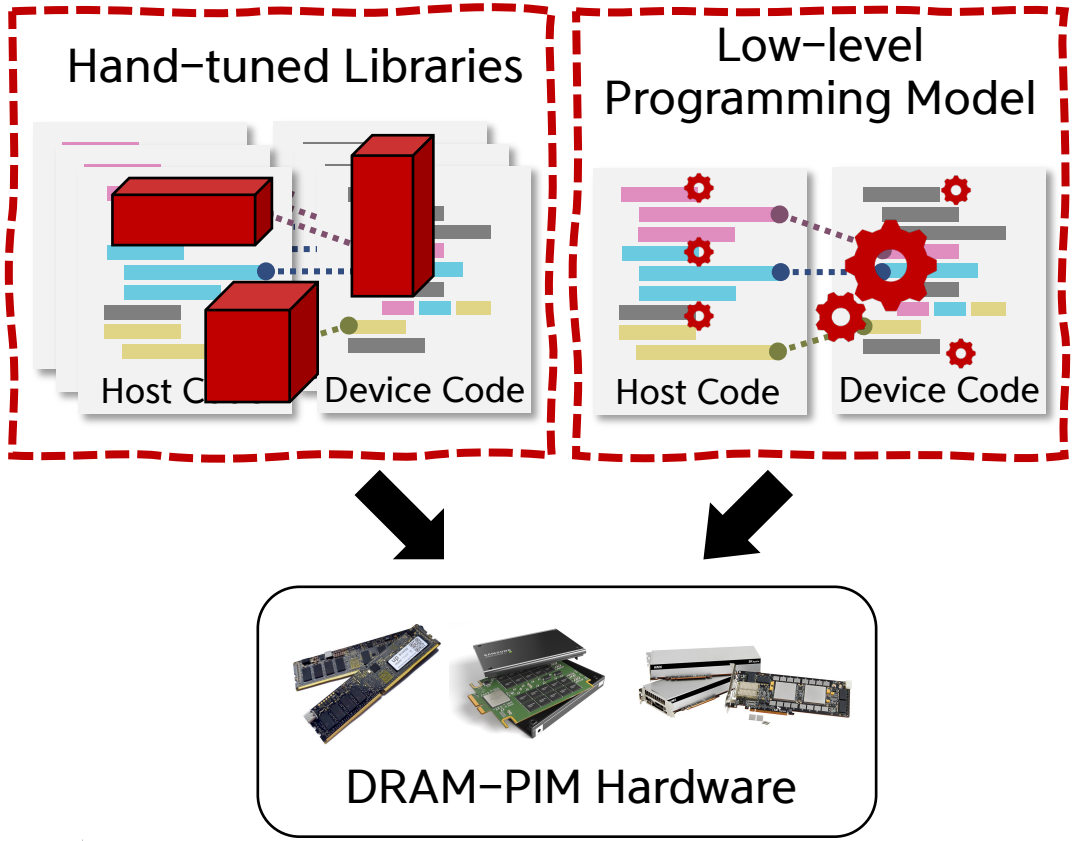


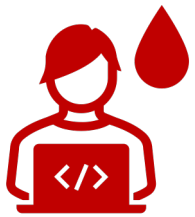


Substantial development efforts

CURRENT SOFTWARE STACKS FOR DRAM-PIM

- Current software stacks support a narrow set of workloads with **hand-tuned libraries** or **low-level programming models**



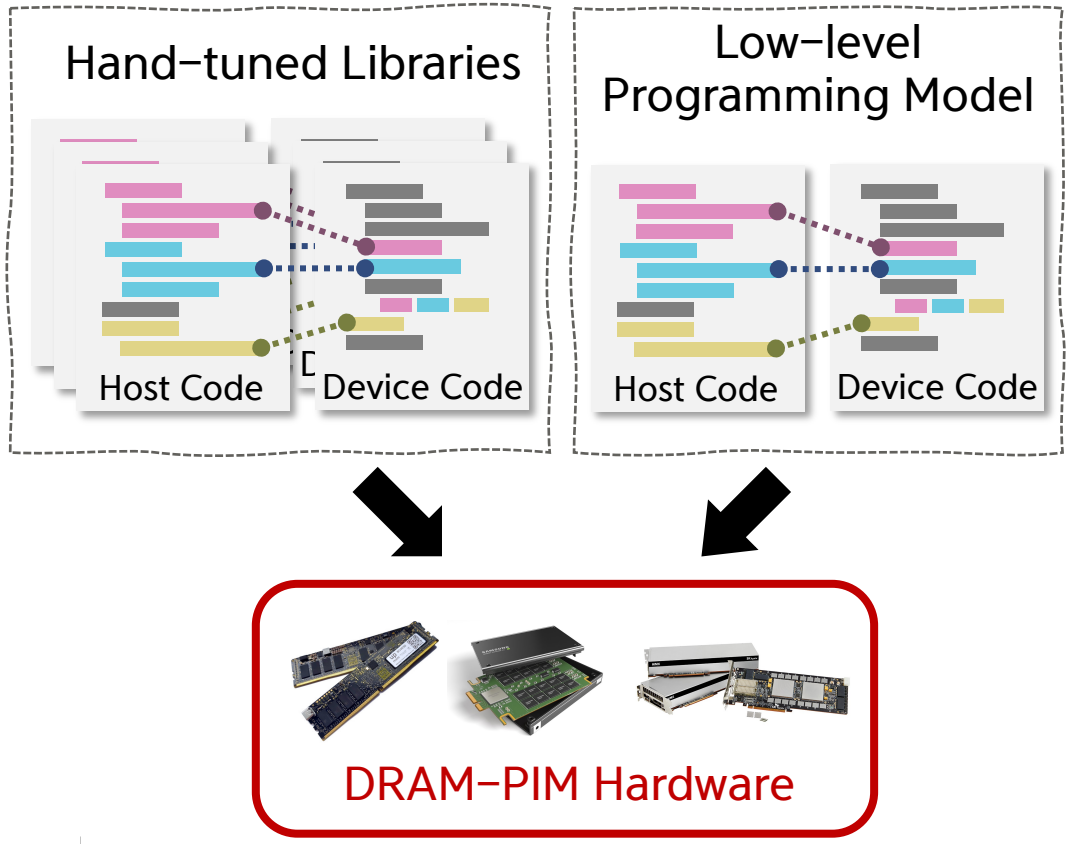


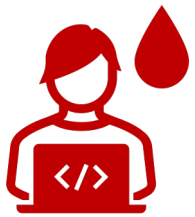
Substantial development efforts

Vast optimization search space

CURRENT SOFTWARE STACKS FOR DRAM-PIM

- Current software stacks support a narrow set of workloads with **hand-tuned libraries** or **low-level programming models**

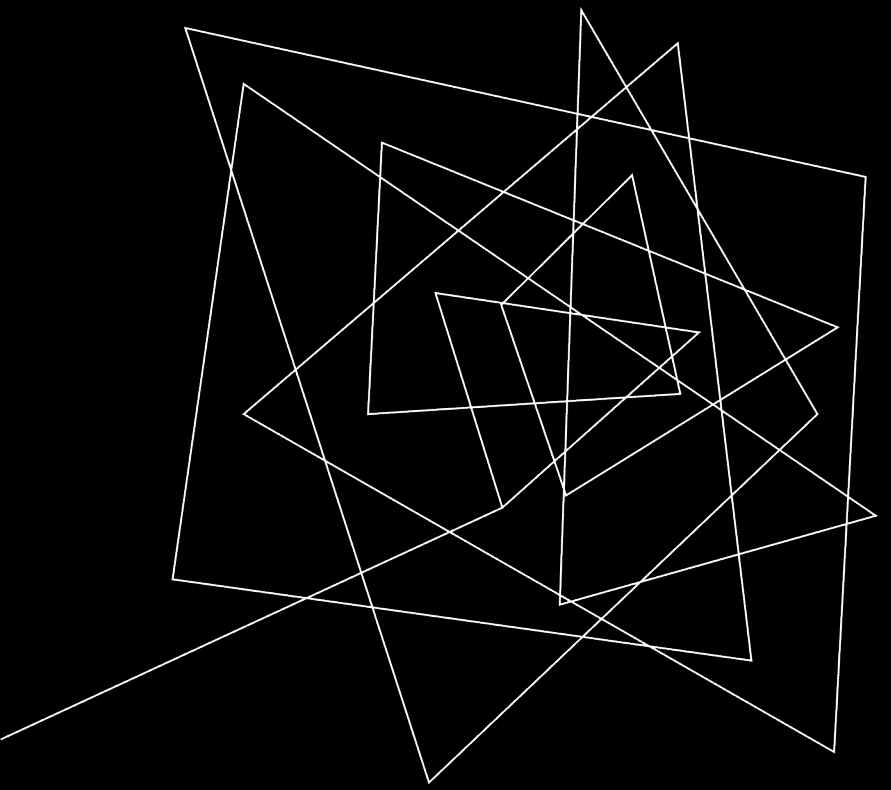




Substantial development efforts

Vast optimization search space

Missed optimization and acceleration opportunities



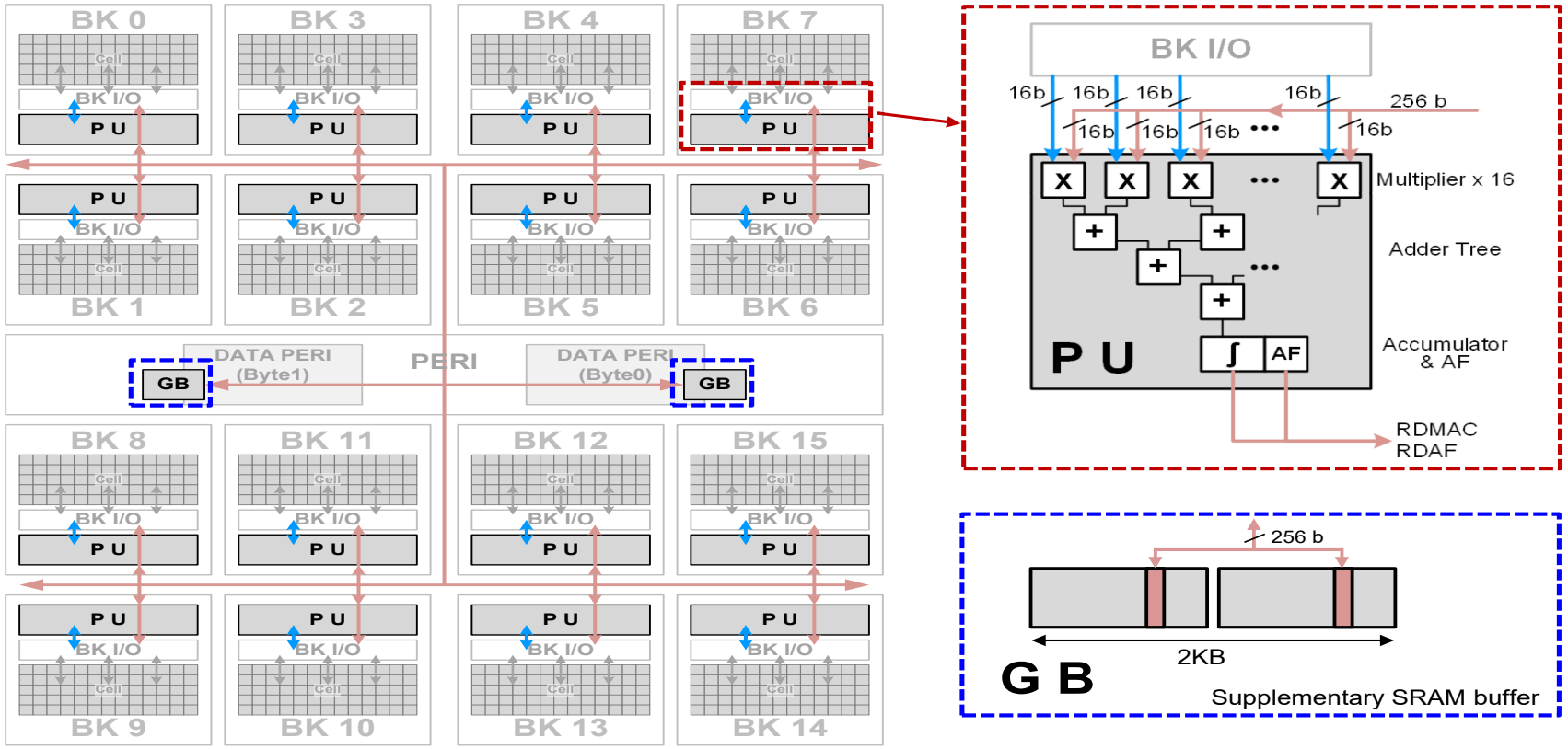
PIMFLOW: COMPILER AND RUNTIME SUPPORT FOR CNN MODELS ON PROCESSING-IN-MEMORY DRAM

EXECUTIVE SUMMARY

- Targets SK Hynix AiM (Newton)
  - GDDR6 based, bank-parallel MAC operation support
- Compiler and runtime support to accelerate 1D convolution layers both on GPU and PIM
  - Support GPU-PIM data-parallel and pipeline execution
    - Parition a single layer or pipeline layers across GPU and PIM
    - Search for optimal partition ratio and parallelism patterns (dynamic programming)
  - Optimize PIM command scheduling and data layout
  - Integrated with Apache TVM compiler
- Extended AiM architecture
  - GPU memory containing both regular and PIM banks
  - Add more global buffers for channel-level parallelism

AIM: SYSTEM ORGANIZATION

- GDDR6-based AiM architecture



https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9731711

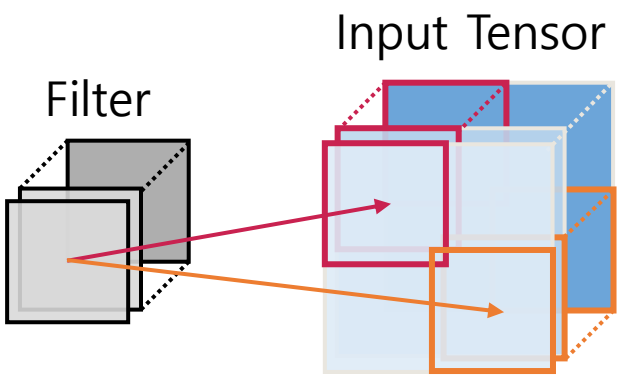
AIML COMMAND SET

Type	CMD	Description
Bank Activation	ACT4, ACT16	Activate four/sixteen banks in parallel
	ACTAF4, ACTAF16	Activate rows storing activation function LUTs in four/sixteen banks in parallel
Compute	MACSB, MAC4B, MACAB	Perform MAC in one/four/sixteen banks in parallel
	AF	Compute activation function in all banks
	EWMUL	Perform element-wise multiplication
Data	WRBK	Write to all activated banks in parallel
	WRGB	Write to Global Buffer
	RDMAC	Read from MAC result register
	RDAF	Read from activation function result register
	WRBIAS	Write bias to MAC result register
	RDCP	Copy data from a bank to Global Buffer
	WRCP	Copy data from Global Buffer to a bank

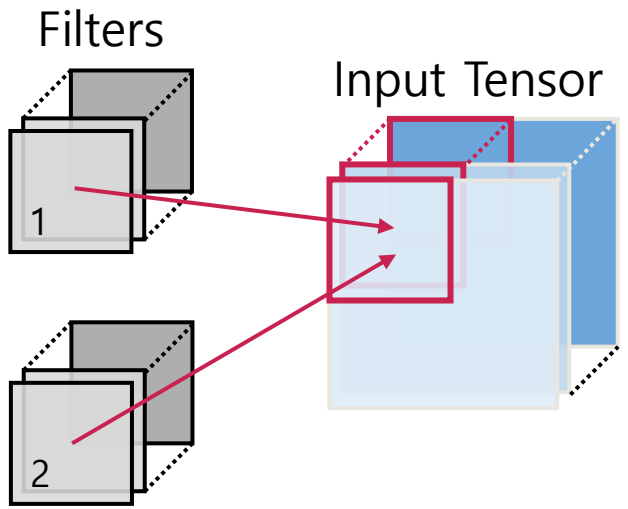
https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9731711

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

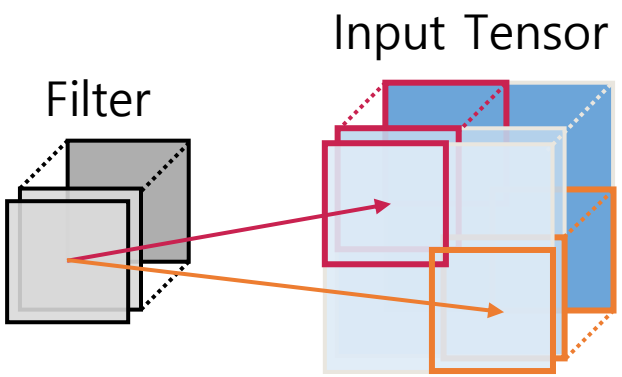


Activation Reuse

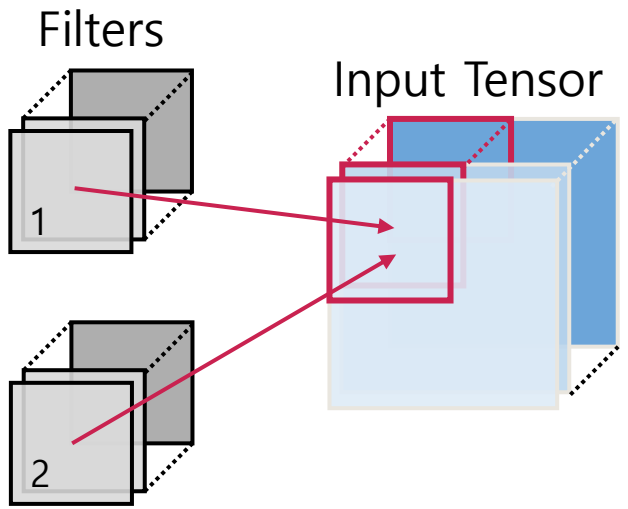
* Image adapted from V. Sze, et al., "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," in Proceedings of the IEEE, Dec. 2017

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



Activation Reuse
Filter Reuse



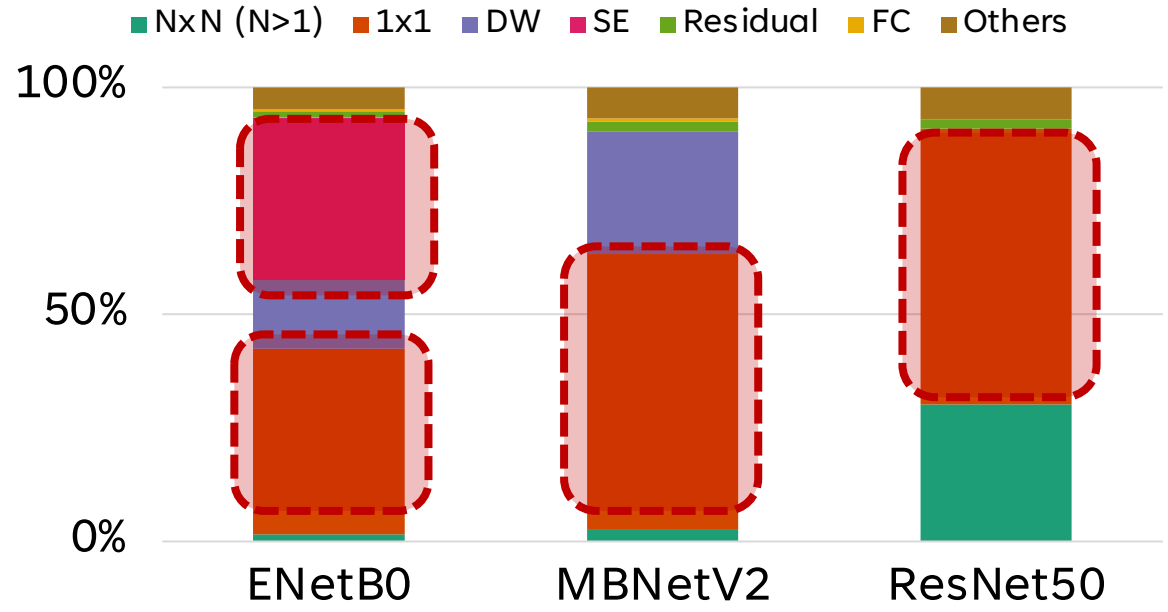
Activation Reuse

➔ 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**)

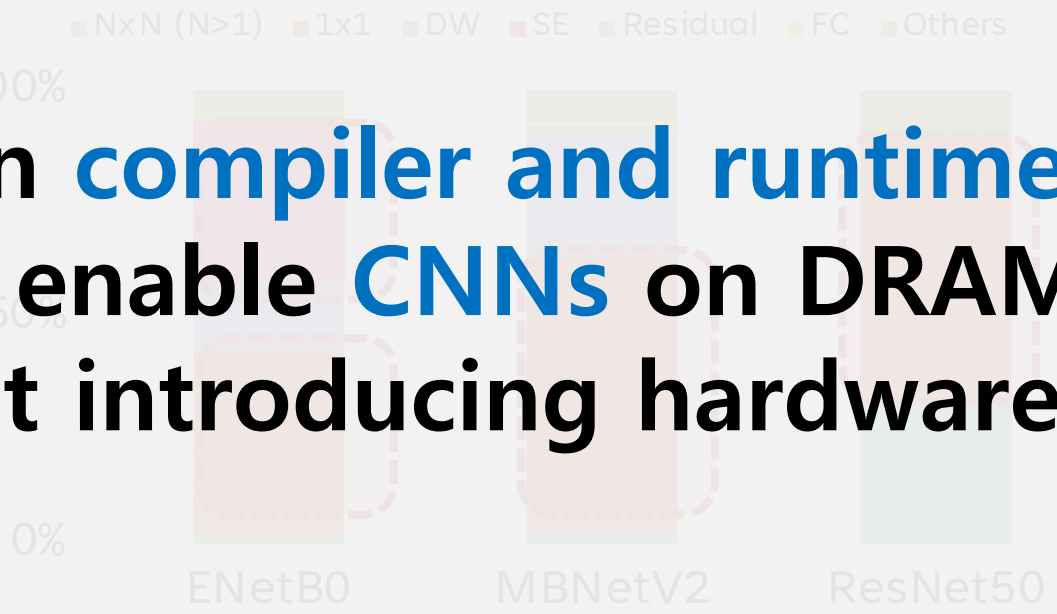


1x1 CONV and FC take 60-80% of runtime in modern CNNs

➔ CNNs can be a potential PIM acceleration target

UNEXPLORED OPPORTUNITY: CNN MODELS

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

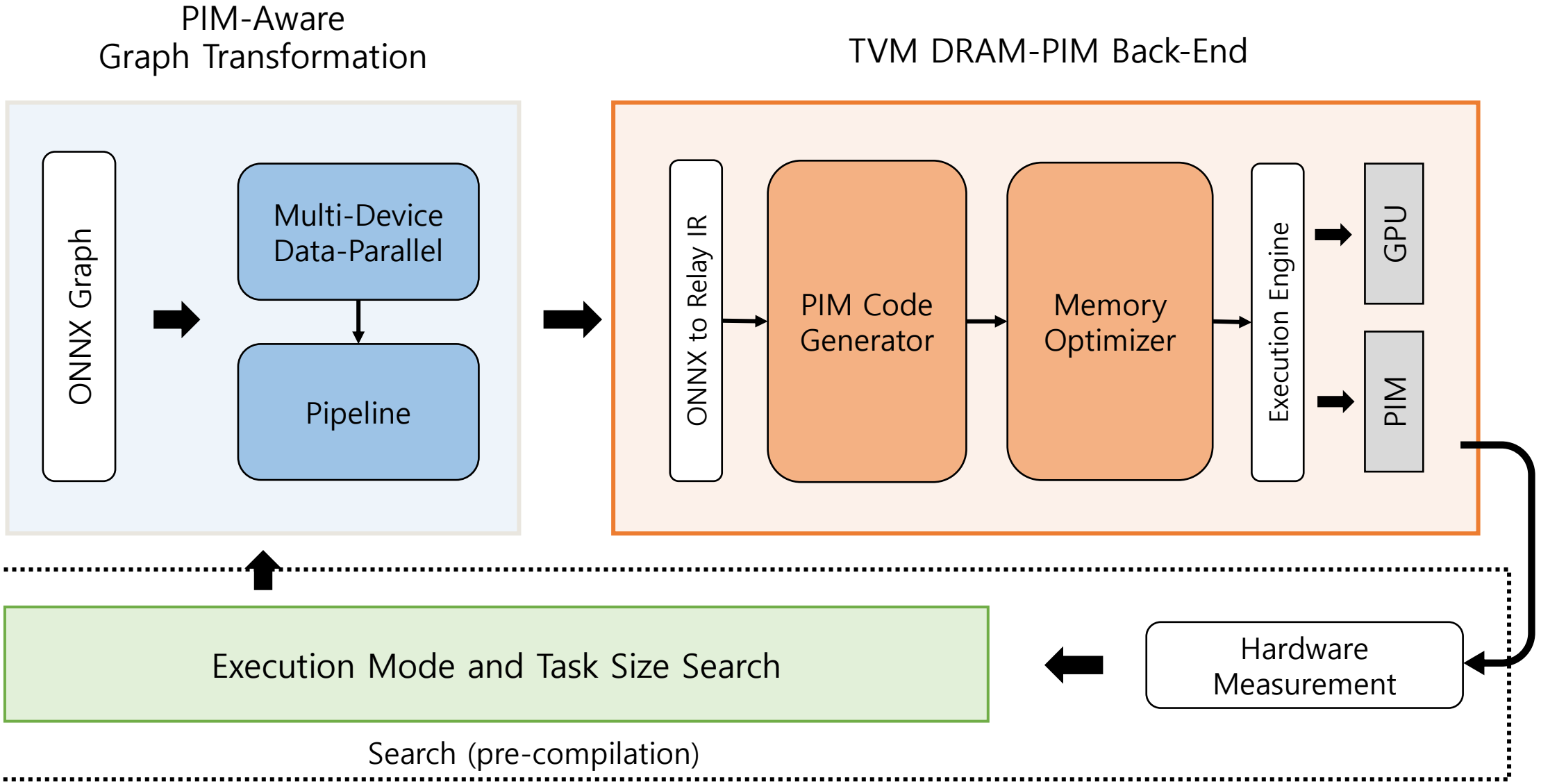


Can compiler and runtime support enable CNNs on DRAM-PIM without introducing hardware complexity?

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

➔ CNNs can be a potential PIM acceleration target

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)

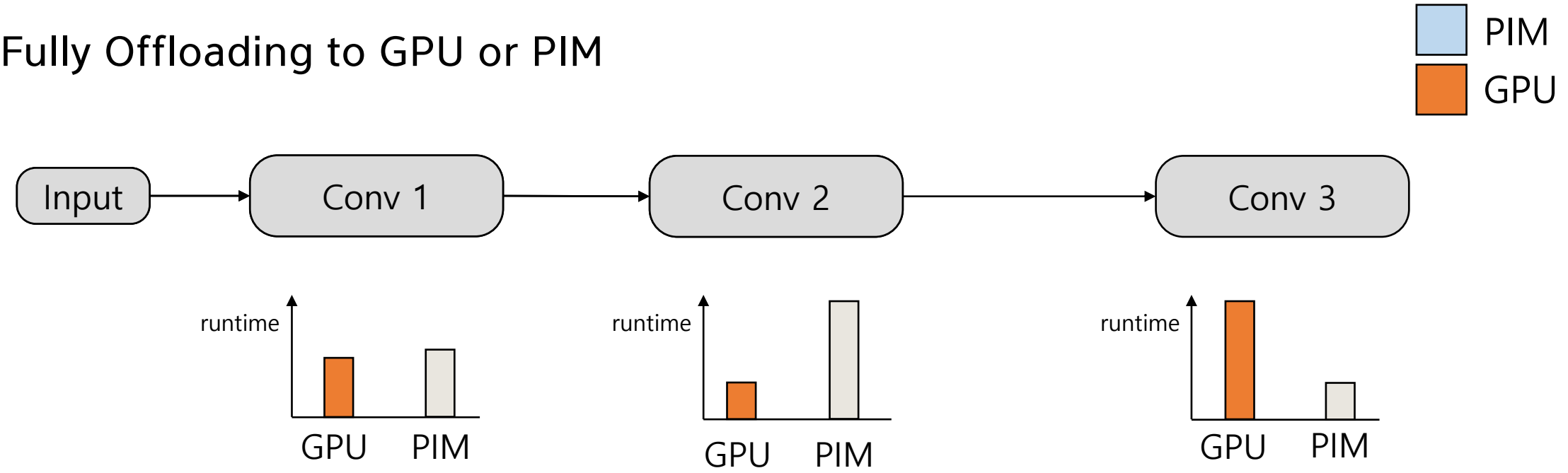
BASELINE EXECUTION

- Fully Offloading to GPU or PIM

Given a computation graph with convolution layers,

BASELINE EXECUTION

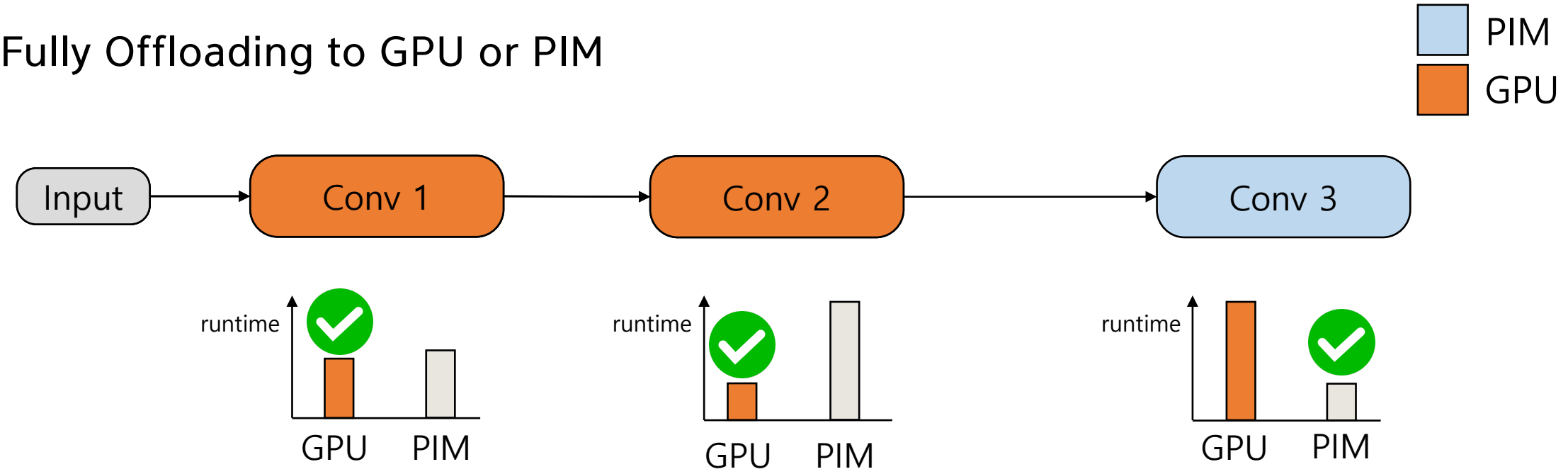
- Fully Offloading to GPU or PIM



Given a computation graph with convolution layers,
Measure performance of each node on GPU and PIM

BASELINE EXECUTION

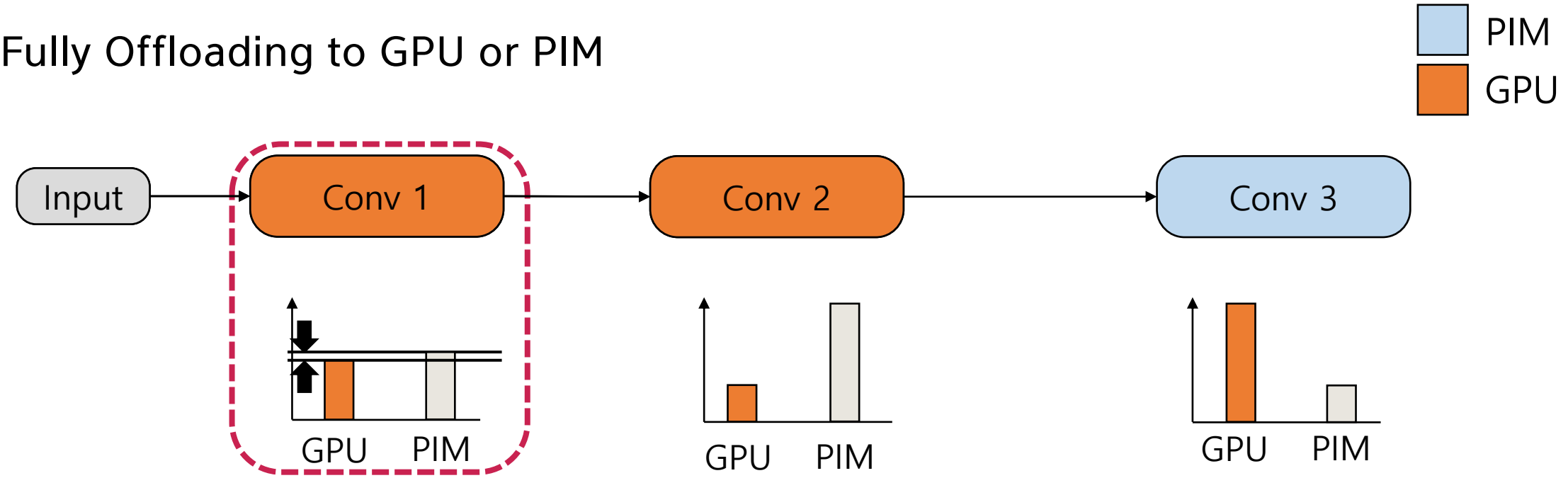
- Fully Offloading to GPU or PIM



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

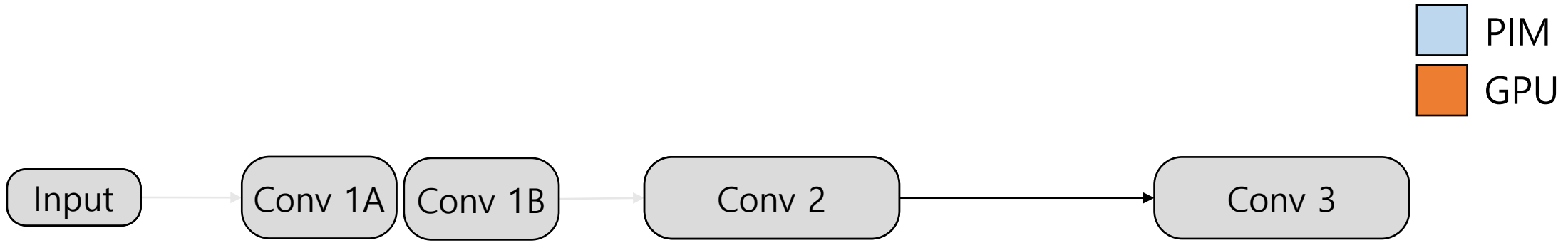
BASELINE EXECUTION

- Fully Offloading to GPU or PIM



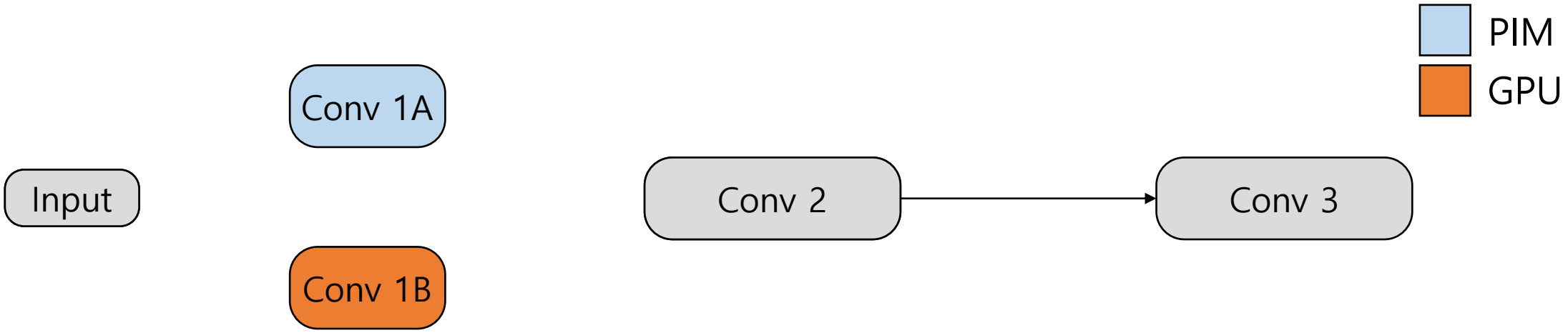
Parallel speedup when executed on both GPU AND PIM
➔ Multi-Device Data-Parallel Execution (MD-DP)

MULTI-DEVICE DATA-PARALLEL EXECUTION



Split a convolution node into two nodes

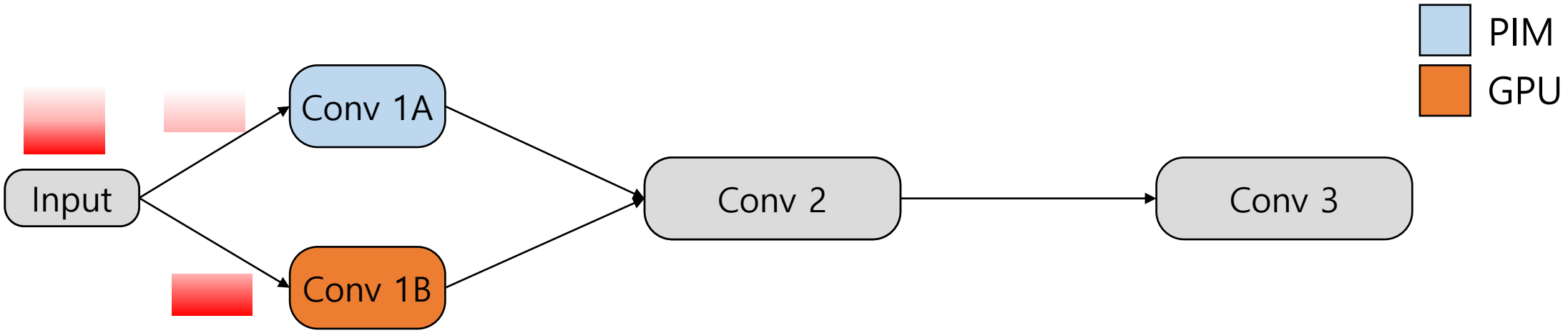
MULTI-DEVICE DATA-PARALLEL EXECUTION



Split a convolution node into two nodes

Assign each node to GPU and PIM

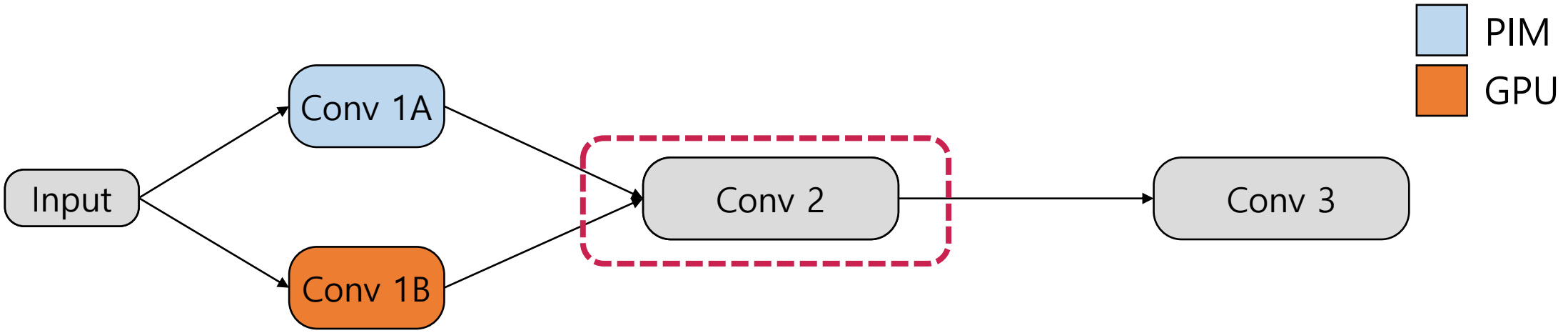
MULTI-DEVICE DATA-PARALLEL EXECUTION



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

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

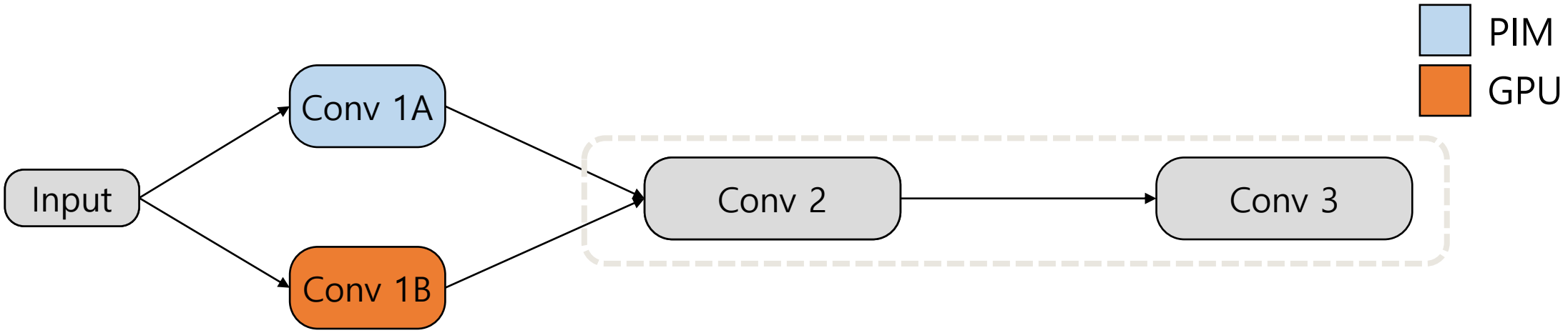
PIPELINED EXECUTION



Certain types of convolution nodes always run on GPU

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

PIPELINED EXECUTION

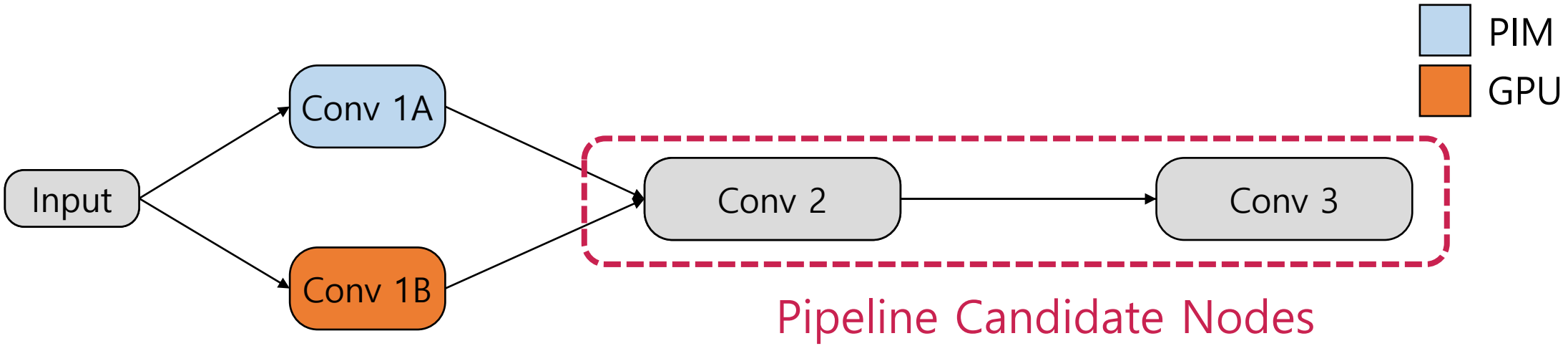


Certain types of convolution nodes always run on GPU

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

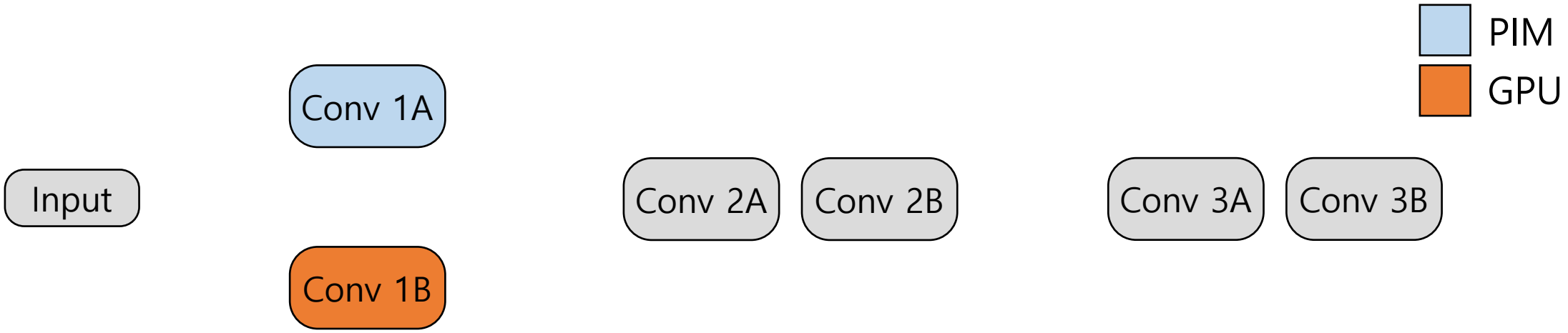
→ Pipeline the GPU node with the following PIM node

PIPELINED EXECUTION



Select a group of nodes as pipeline candidates

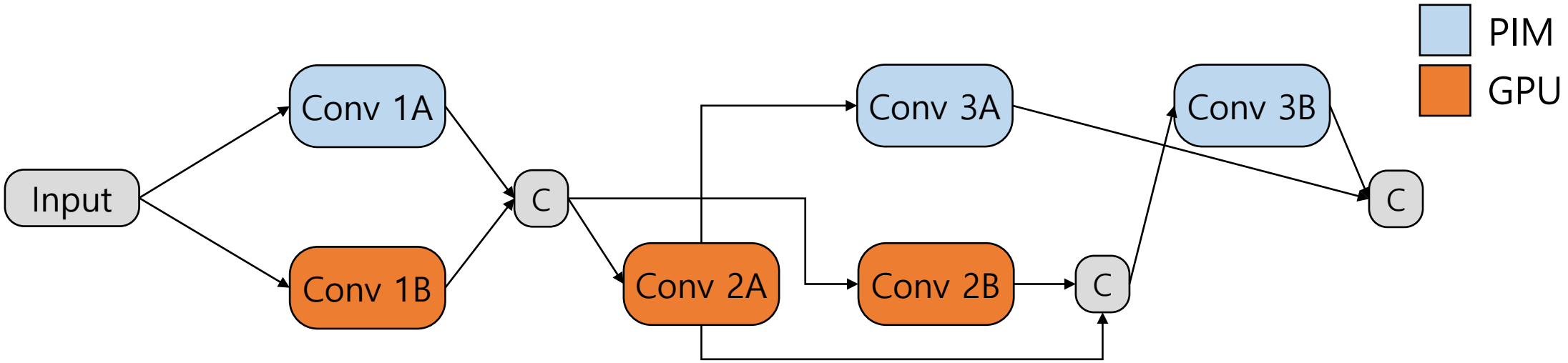
PIPELINED EXECUTION



Select a group of nodes as pipeline candidates

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

PIPELINED EXECUTION

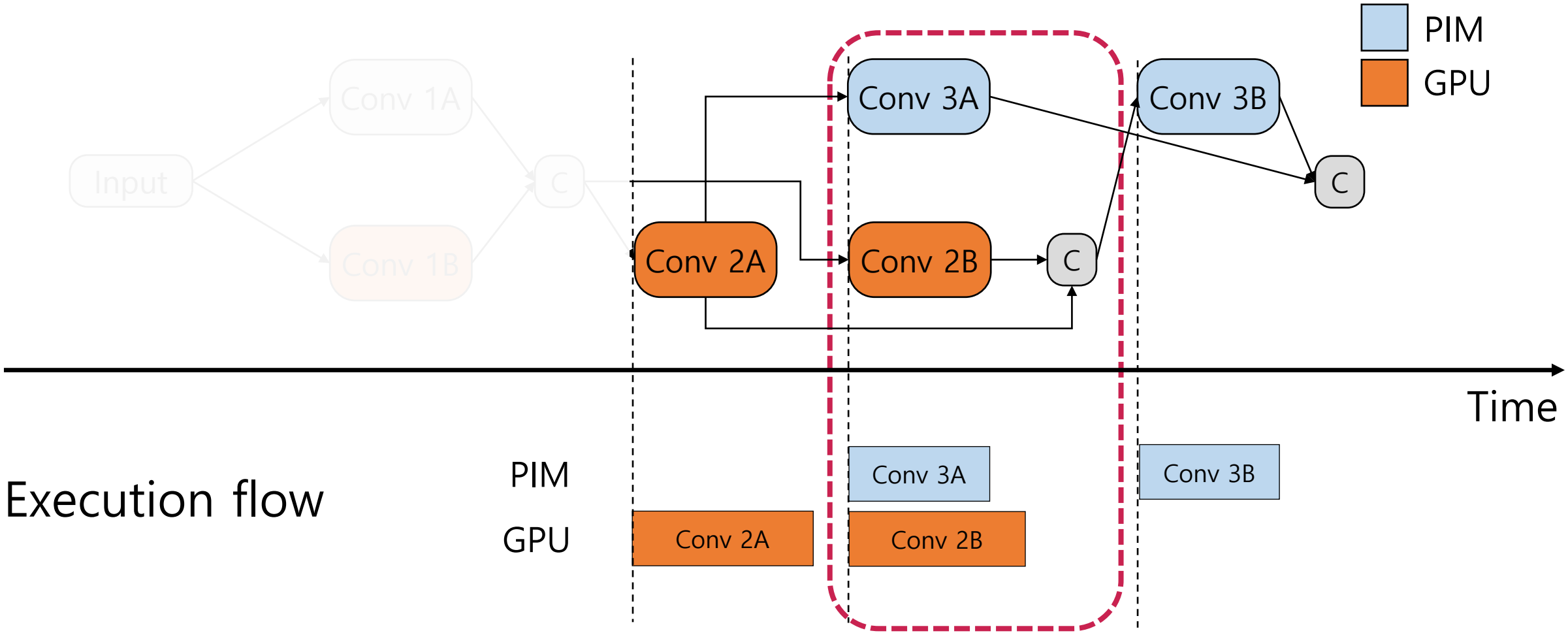


Select a group of nodes as pipeline candidates

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

Add data-flow edges for pipelined execution

PIPELINED EXECUTION

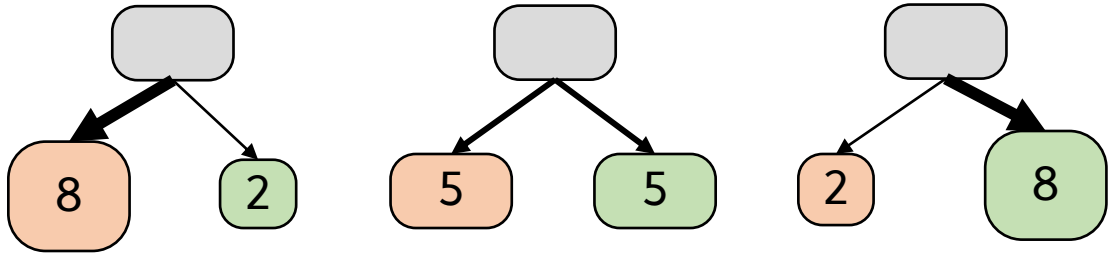


Conv 2B and Conv 3A can be executed *in parallel* on GPU and PIM

EXECUTION MODE AND TASK SIZE SEARCH

Profiling Phase

① Determine the optimal split ratio for MD-DP

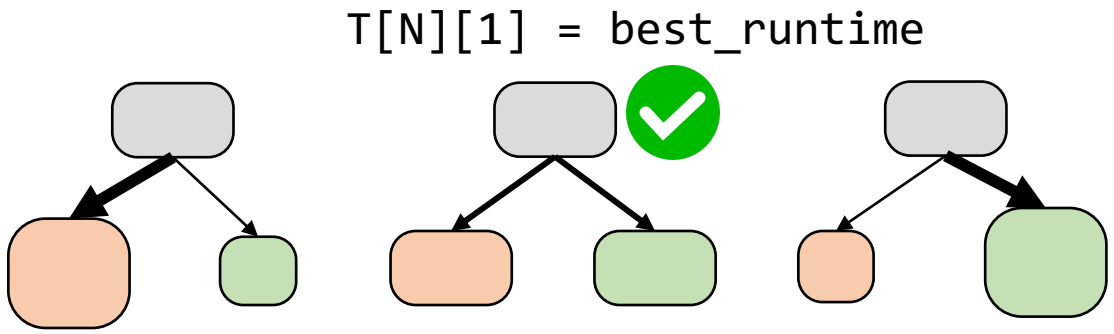


Split ratio is profiled at every 10%

EXECUTION MODE AND TASK SIZE SEARCH

Profiling Phase

① Determine the optimal split ratio for MD-DP



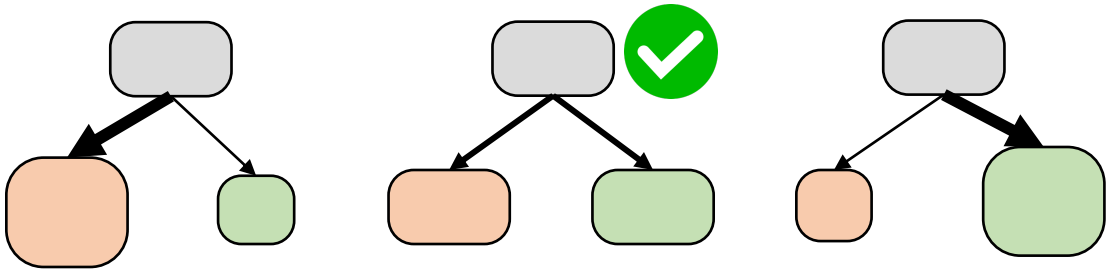
Best split ratio is recorded for the node

EXECUTION MODE AND TASK SIZE SEARCH

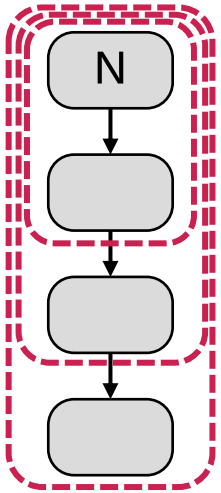
Profiling Phase

① Determine the optimal split ratio for MD-DP

```
T[N][1] = best_runtime
```



② Record every possible pipelining result



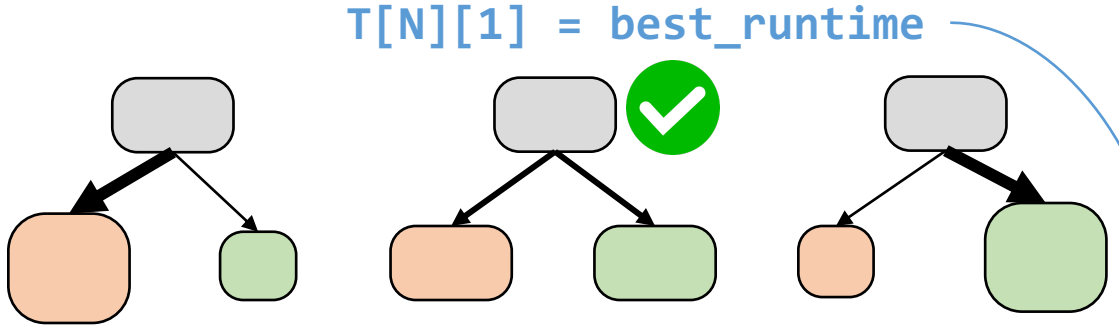

```
T[N][2] = pipeline_runtime
T[N][3] = pipeline_runtime
T[N][4] = pipeline_runtime
```

pipeline length form the node

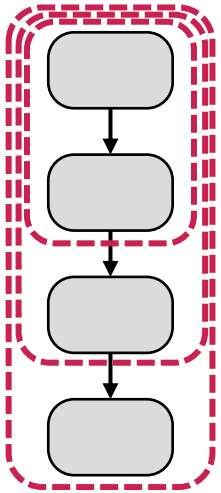
EXECUTION MODE AND TASK SIZE SEARCH

Profiling Phase

① Determine the optimal split ratio for MD-DP



② Record every possible pipelining result



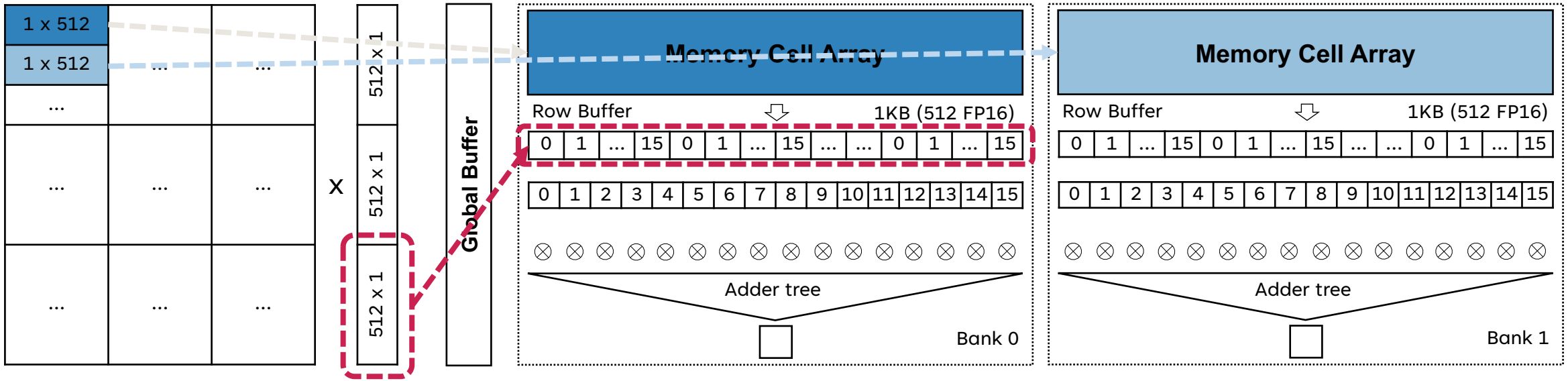
Solving Phase

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]
```

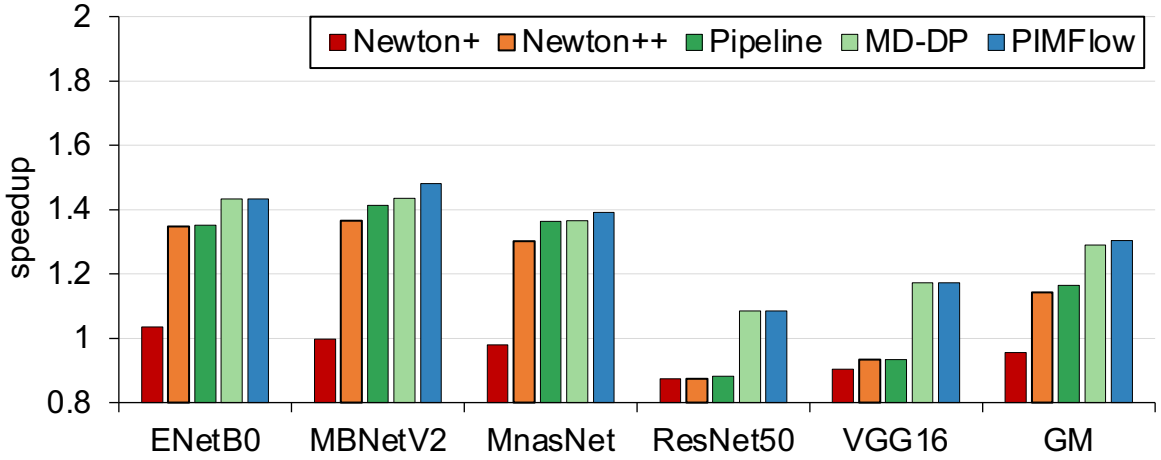
TVM BACK-END FOR DRAM-PIM

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

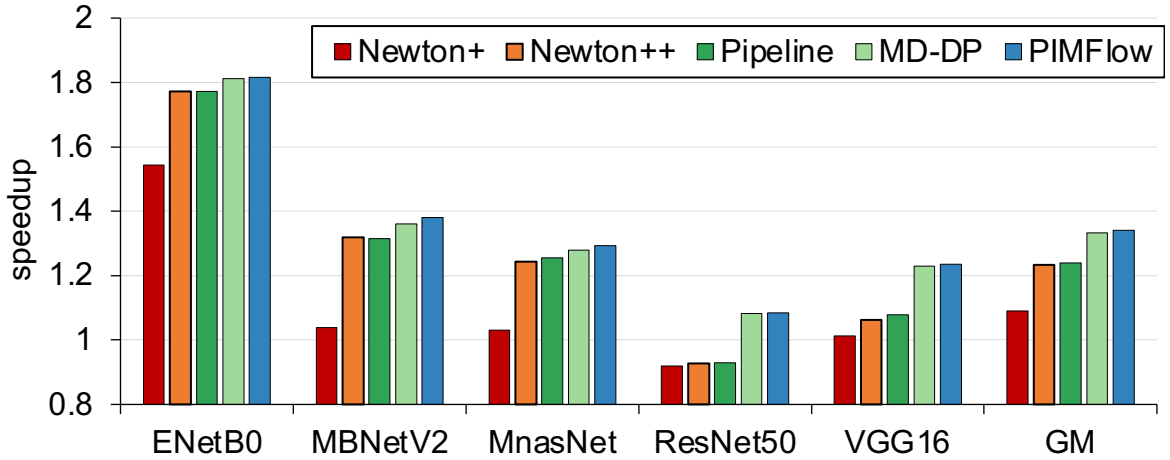


- Matrix is partitioned to 2 x 512 tiles
  - **2** is the number of the banks in a channel (bank-level parallelism)
  - **512** is the number of matrix elements in a memory
- Tiles are stored in the memory cell array

EXECUTION TIME (SPEEDUP)



[All CONV layers]



[End-to-end model inference]

- Evaluated on five configurations

Newton+ —(Multiple GB / GW Latency Hiding)→ Newton++

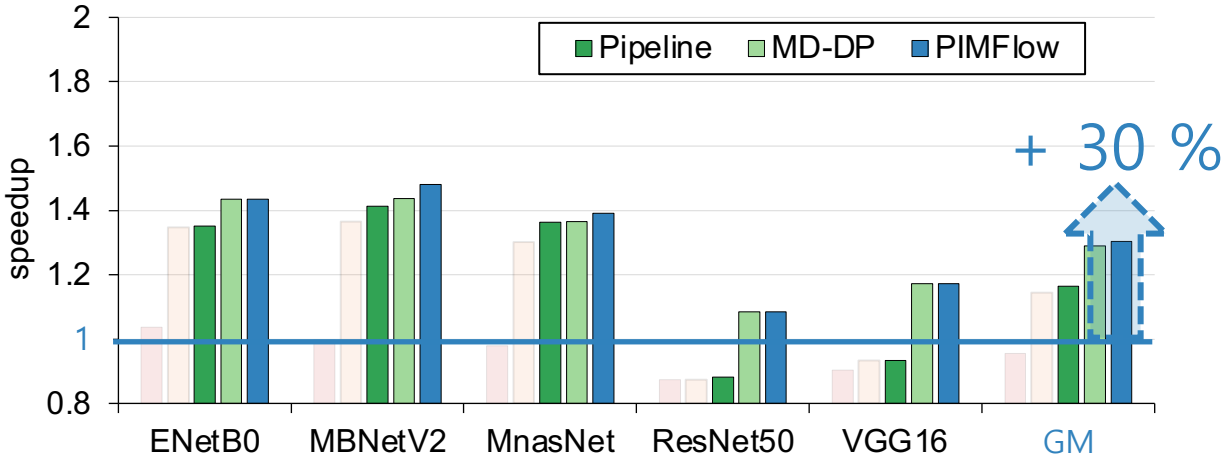
[PIM command optimizations]

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

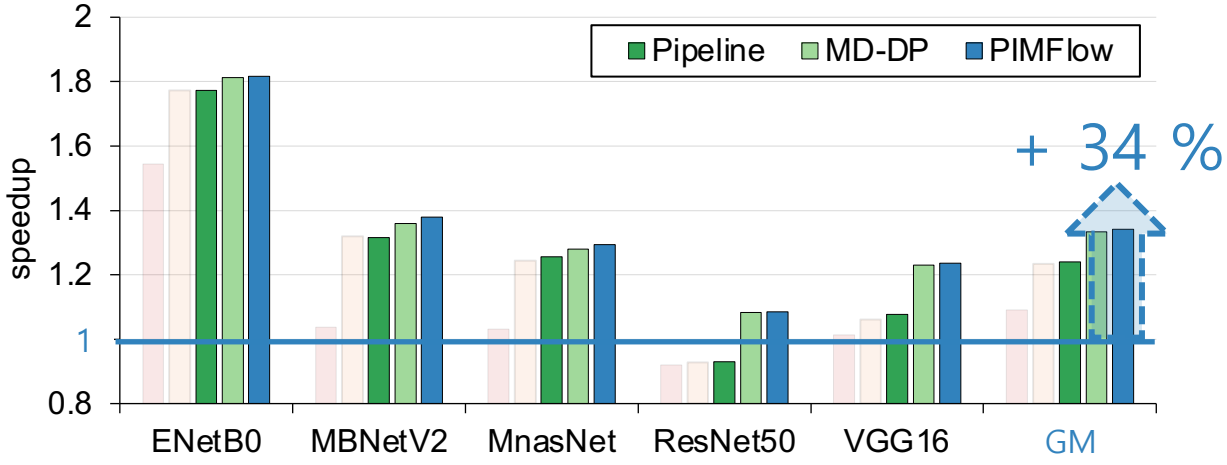
- Inference time normalized to the GPU baseline

EXECUTION TIME (SPEEDUP)

PIMFlow: Pipeline + MD-DP



[All CONV layers]

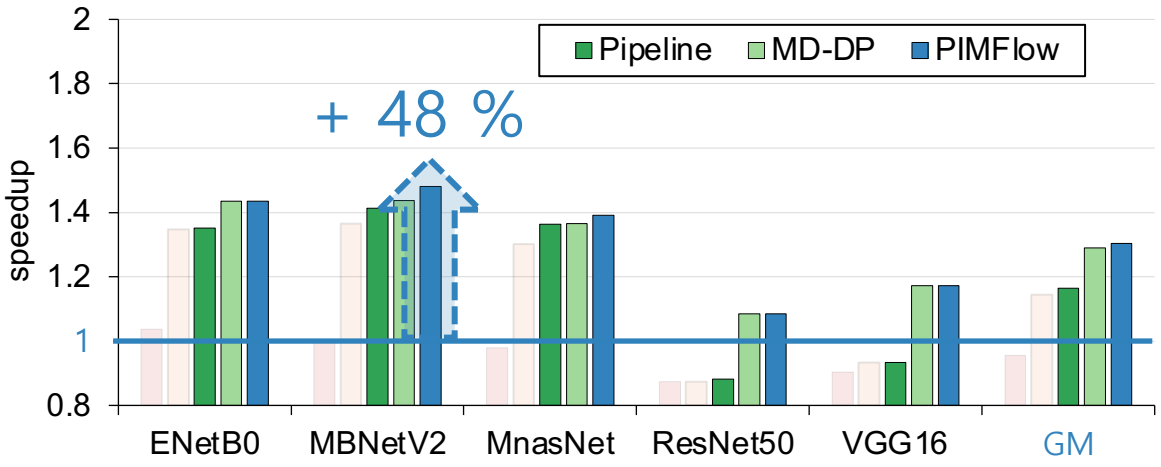


[End-to-end model inference]

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

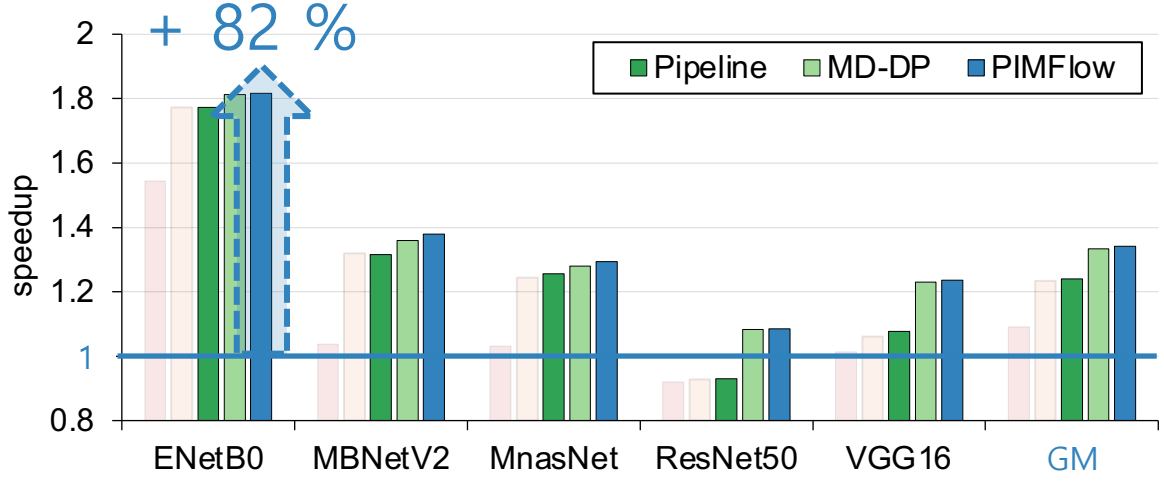
➔ 30% (34%) speedup for **all CONV (end-to-end)** performance on average

EXECUTION TIME (SPEEDUP)



[All CONV layers]

PIMFlow: Pipeline + MD-DP



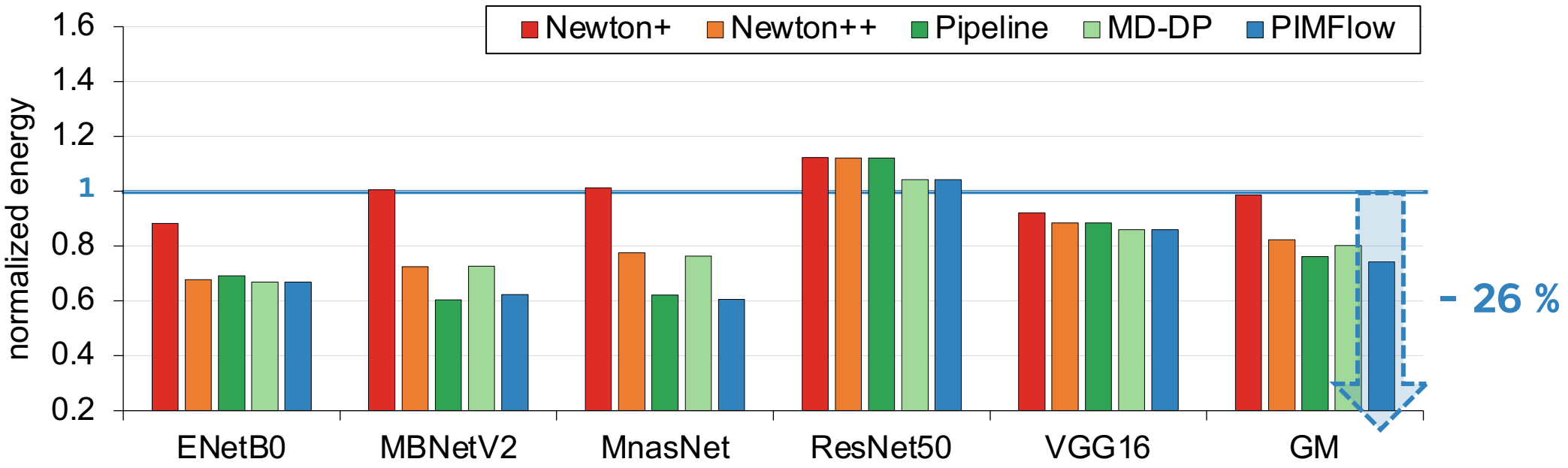
[End-to-end model inference]

- 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

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

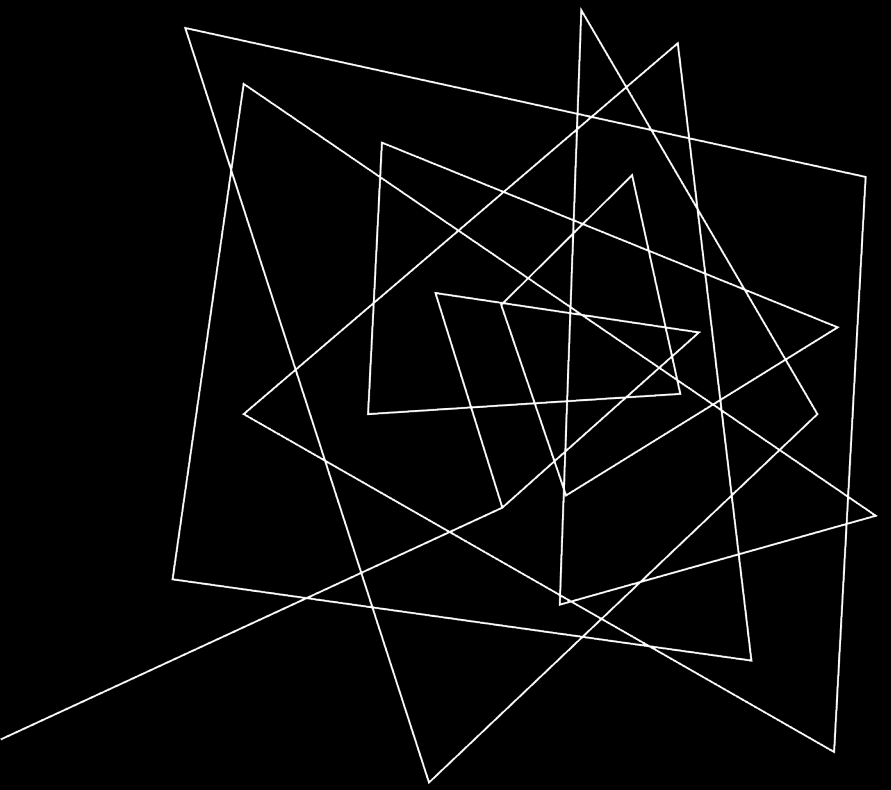
Execution Mode and Task Size Search

Optimal graph transformation search

TVM DRAM-PIM Back-End

DRAM-PIM command generation, scheduling and opt.

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



ATiM: Autotuning Tensor Programs for Processing-in-DRAM

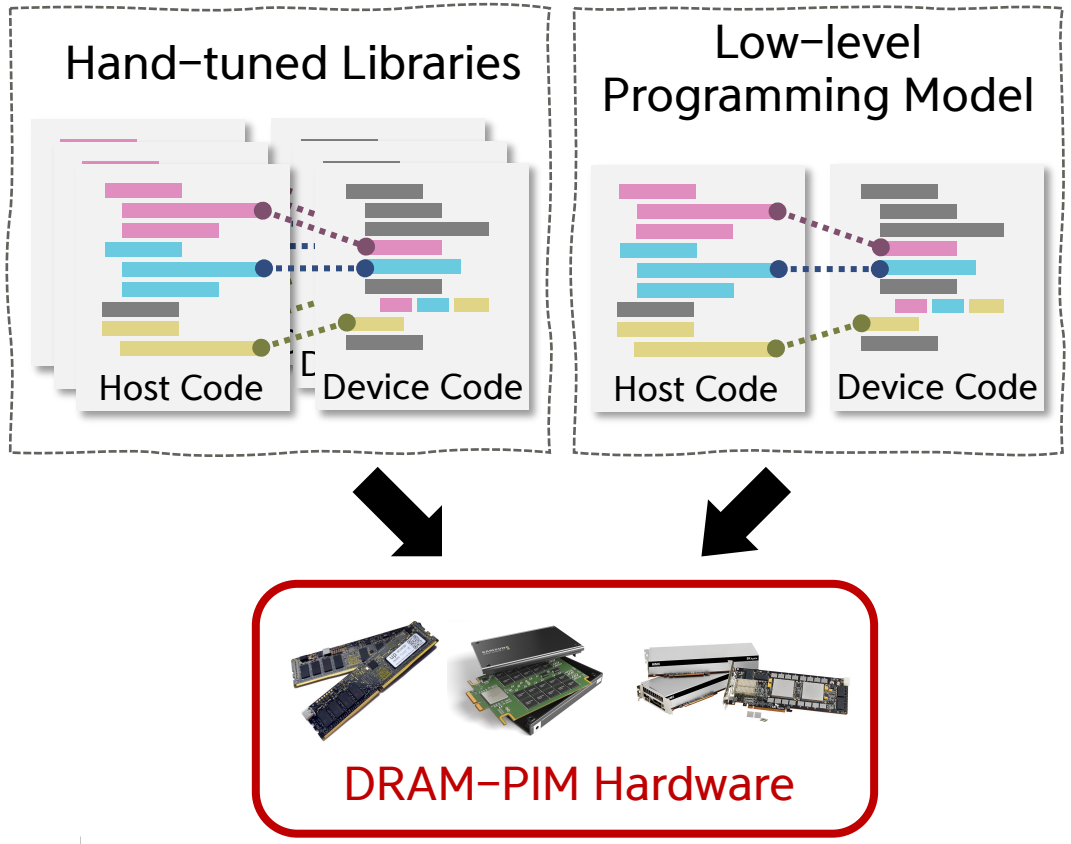
EXECUTIVE SUMMARY

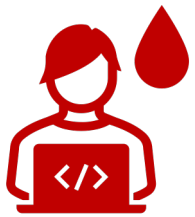
- Targets UPMEM-PIM
  - DDR4 based, near-bank RISC cores, distributed PIM-enabled DIMMs (DPUs)

- Tensor compiler with autotuning support
  - Search the optimization space of host-DPU distribution, kernel tiling and parallelization, reduction dimensions
  - Automatically generate host and kernel code
  - Provide PIM-aware kernel optimizations to reduce branch overheads
  - Adapt search mechanisms to a larger search space
  - Integrated with Apache TVM compiler

CURRENT SOFTWARE STACKS FOR DRAM-PIM

- Current software stacks support a narrow set of workloads with **hand-tuned libraries** or **low-level programming models**





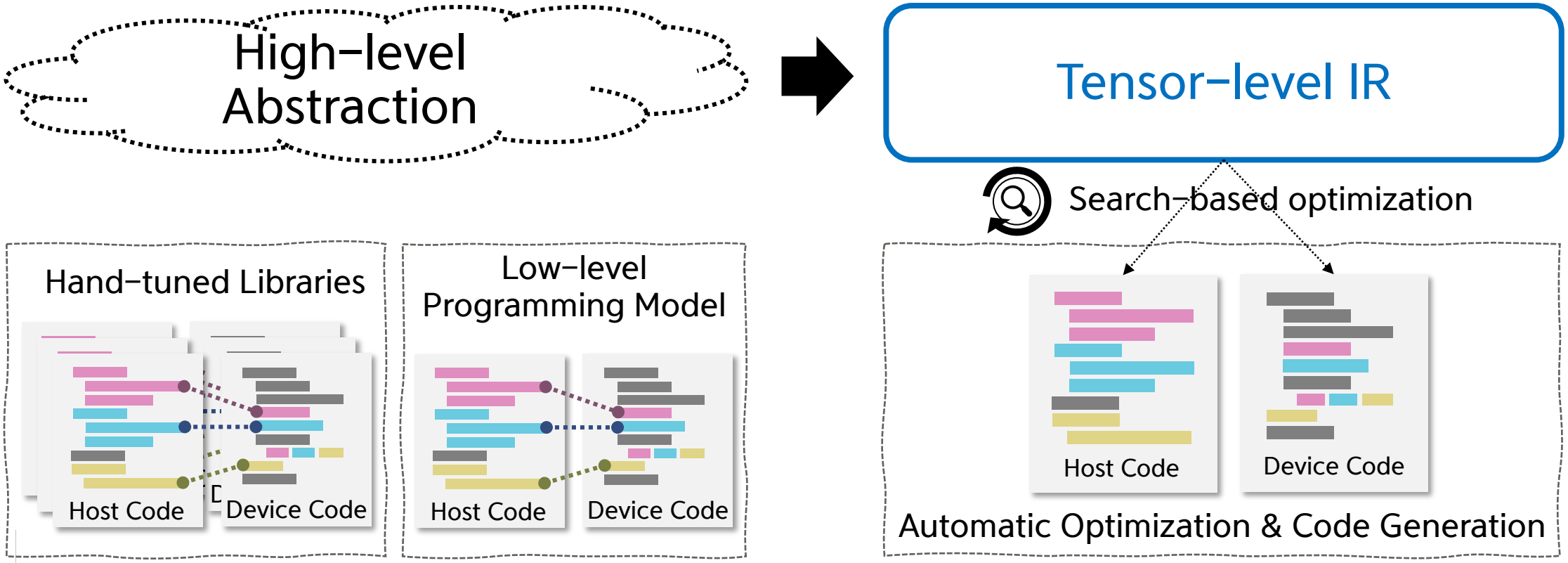
Substantial development efforts

Vast optimization search space

Missed optimization opportunities

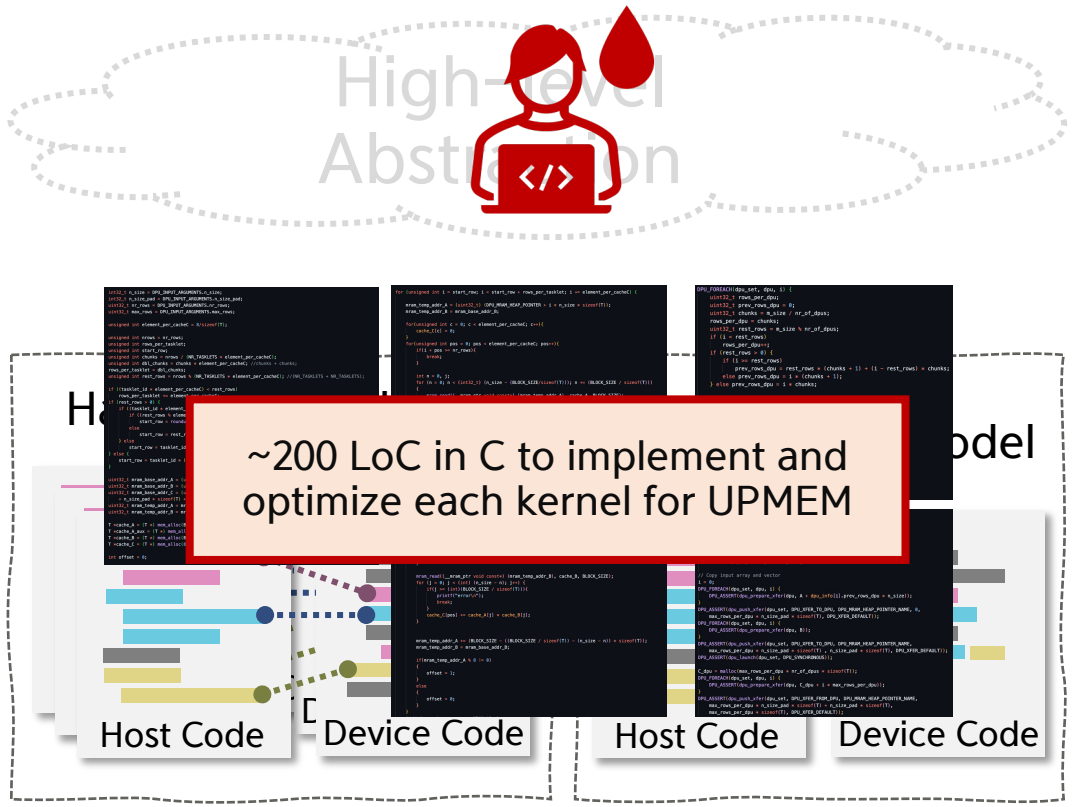
OUR PROPOSAL: ATIM

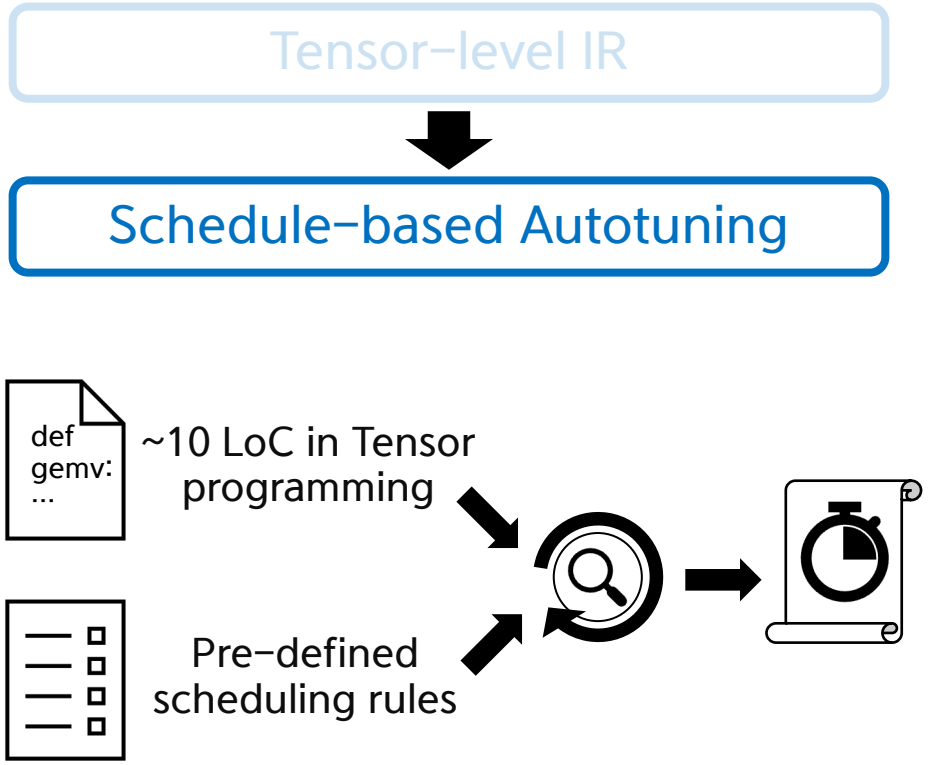
- **Key point:** Tensor-level IR
  - Enables search-based automatic optimization and code generation



OUR PROPOSAL: ATIM

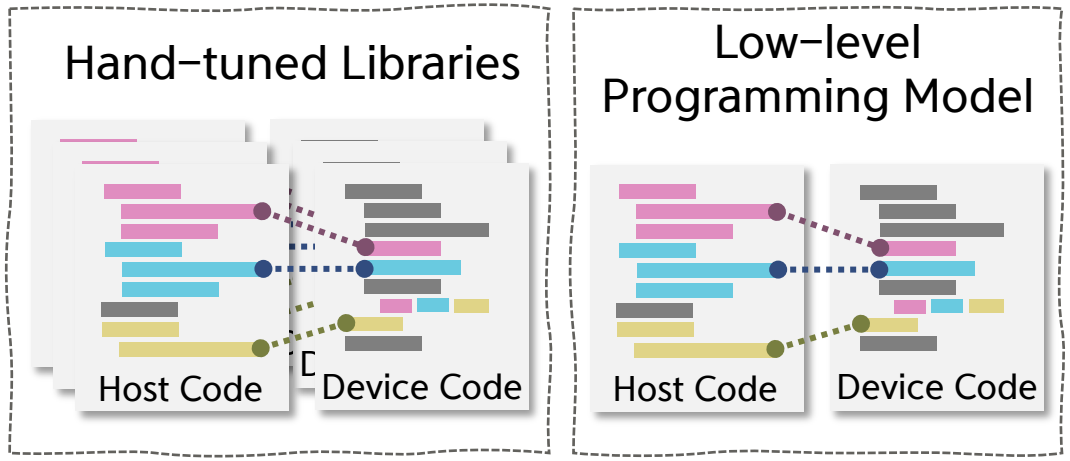
- **Key point:** Tensor-level IR
  - Enabling search-based automatic optimization and code generation

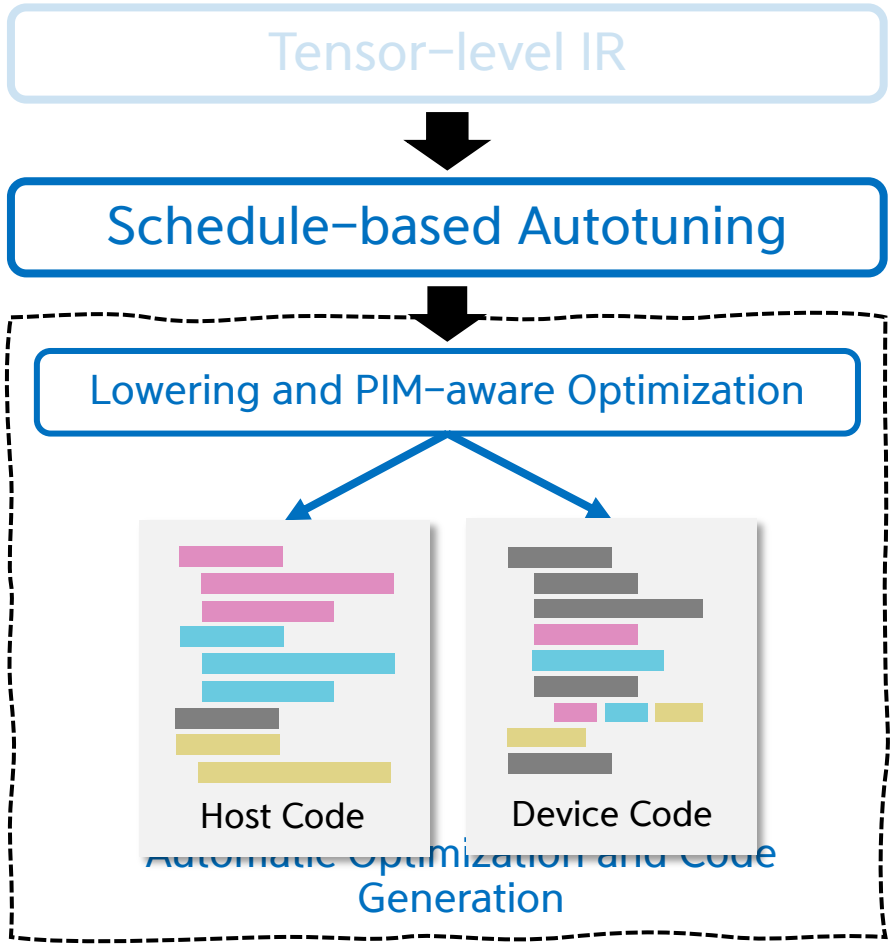




OUR PROPOSAL: ATIM

- **Key point:** Tensor-level IR
  - Enabling search-based automatic optimization and code generation





OUR PROPOSAL: ATIM

We propose ATiM, a search-based optimizing tensor compiler

- Fully automated host and kernel code generation
- PIM-aware optimizations

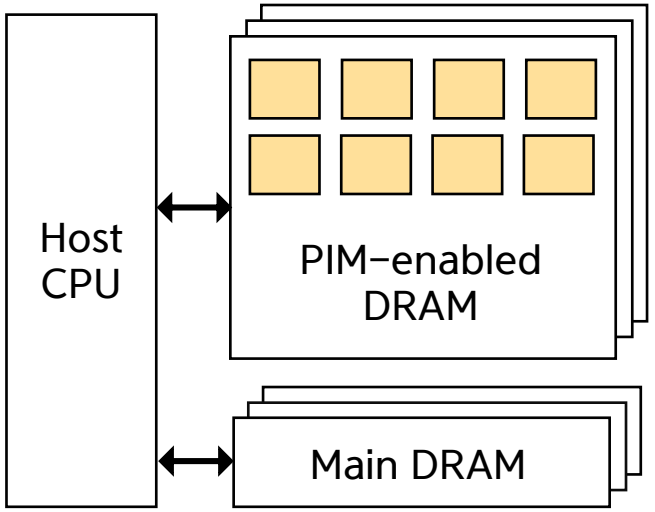
OUR PROPOSAL: ATIM

ATiM achieves up to 6.2× speedup for UPMEM benchmark kernels and 8.3× for core operations of GPT-J models, compared to hand-tuned libraries and existing frameworks

Automatic Optimization and Code Generation

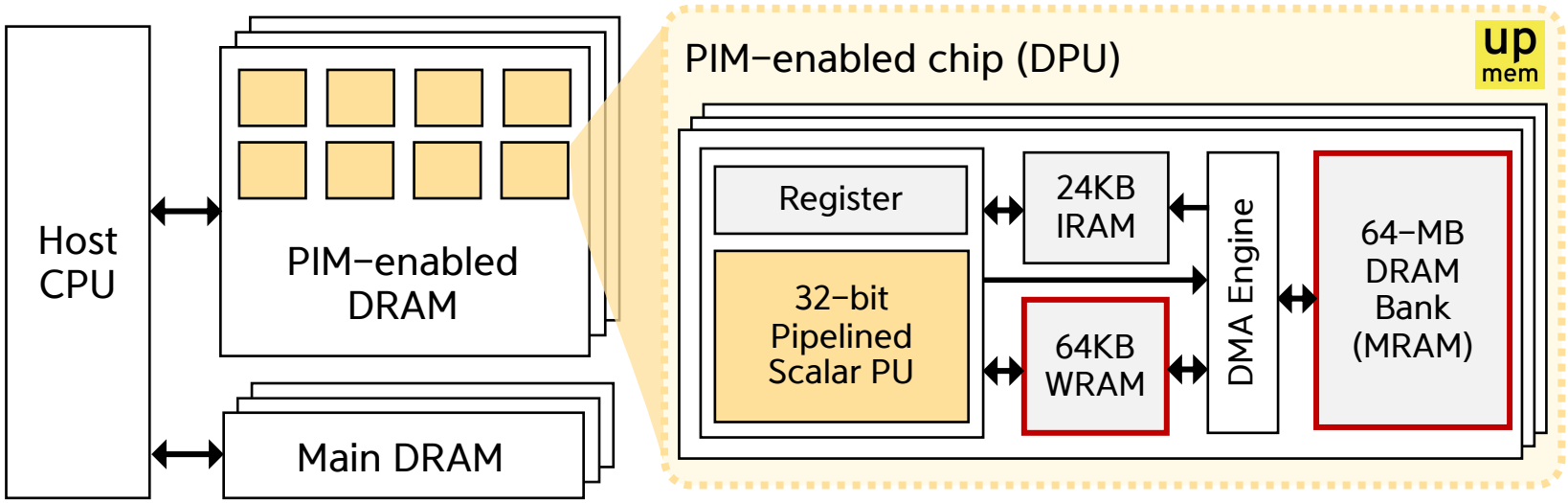
UPMEM ARCHITECTURE

- Consists of a **host CPU**, **standard DRAM**, and **PIM-enabled DRAM**



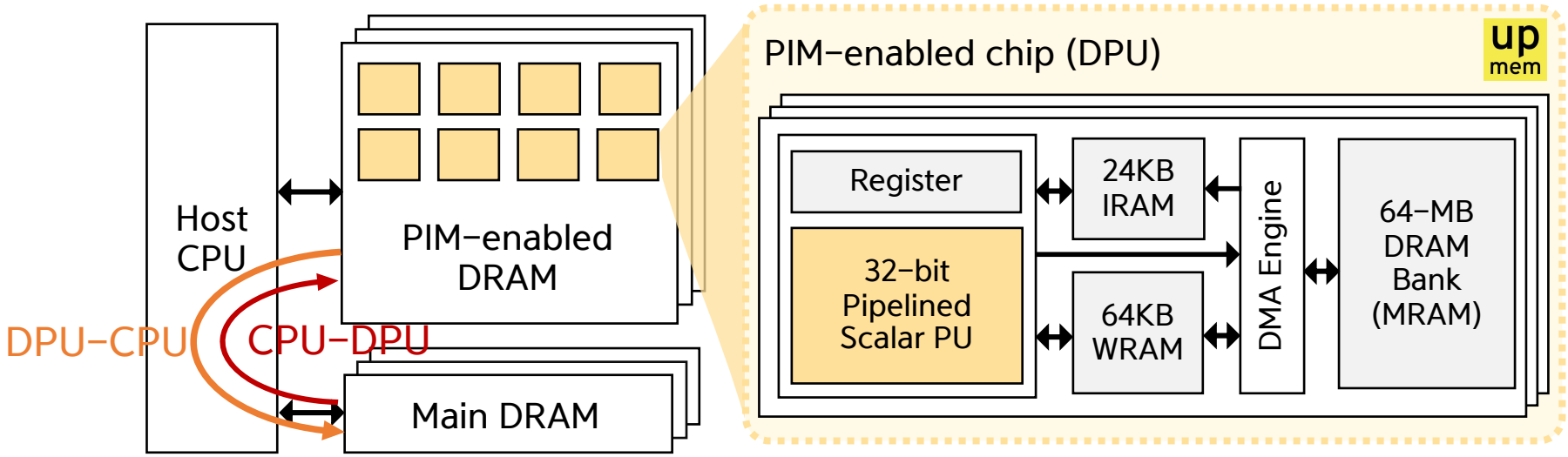
UPMEM ARCHITECTURE

- **Data Processing Unit (DPU)**
  - Performing **bank-parallel computation**
  - Multiple hardware threads (tasklets) enable **thread-level parallelism**
  - **WRAM** (scratchpad) is used for staging data from **MRAM**



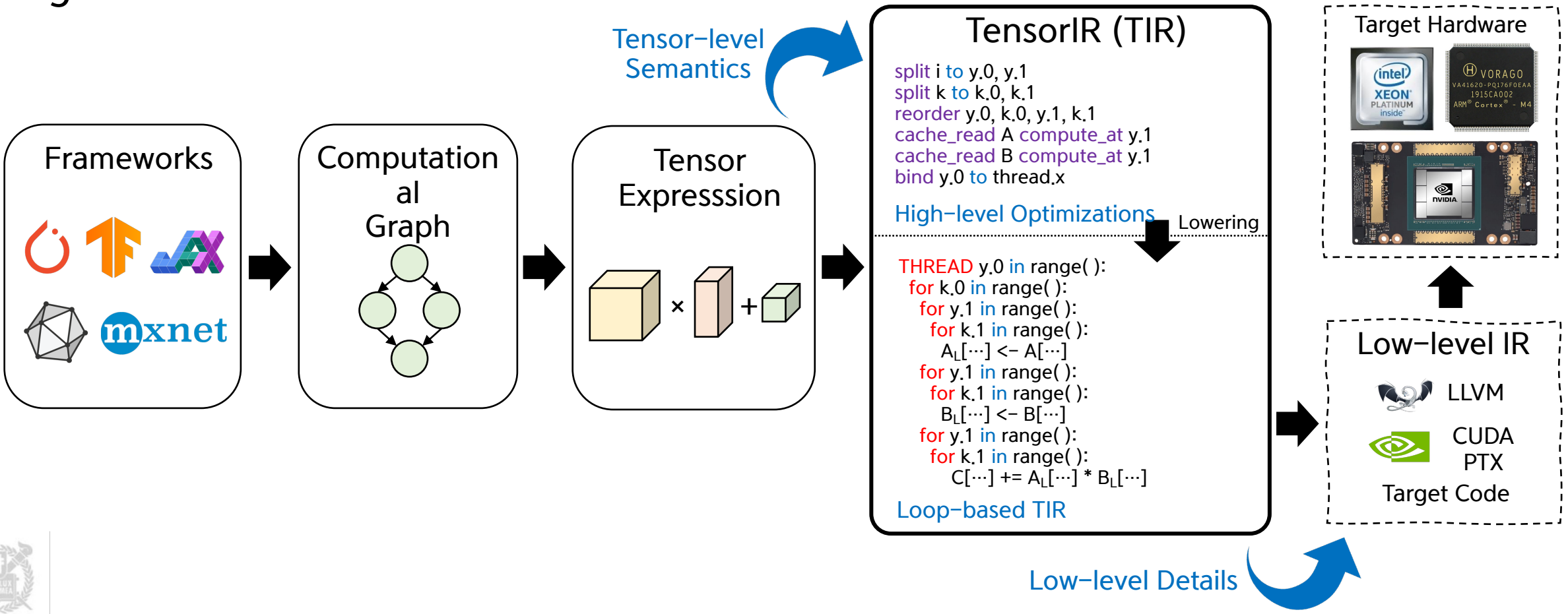
UPMEM ARCHITECTURE

- **Data movements must be routed via the host**
  - Direction: Host-to-DPU and DPU-to-host
  - Minimizing data movement is critical for performance



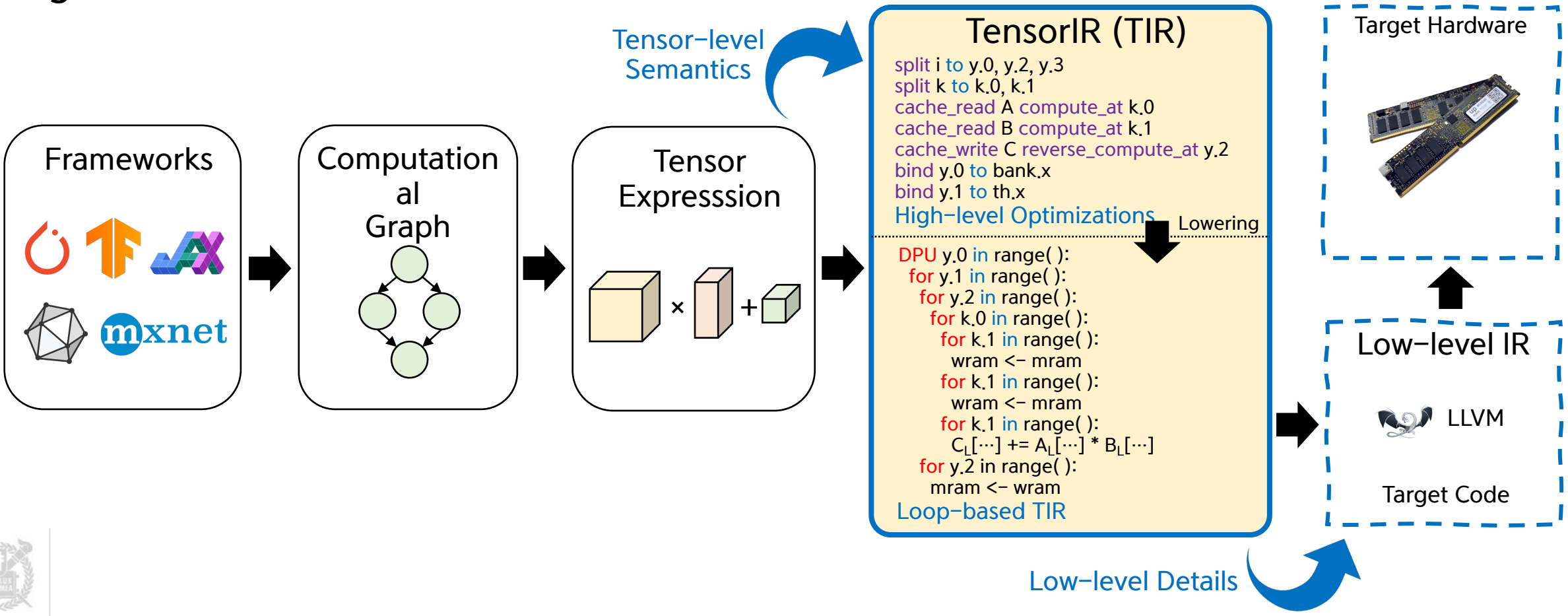
TENSORIR IN APACHE TVM

- **TensorIR** preserves **tensor-level semantics** for **high-level optimizations**, and lowers computations to **loop-based code** for hardware code generation

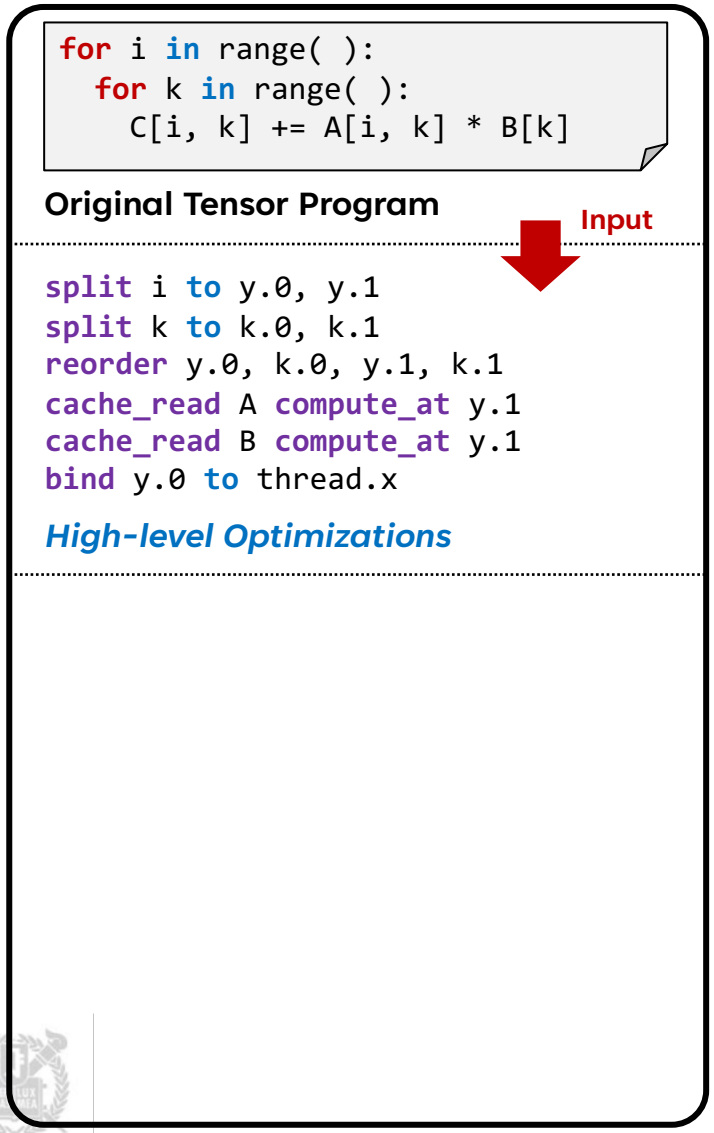


TENSORIR IN APACHE TVM

- **TensorIR** preserves **tensor-level semantics** for **high-level optimizations**, and lowers computations to **loop-based code** for hardware code generation



TENSORIR – SCHEDULE PRIMITIVES



Schedule primitives

- Abstracting the policy for **how code should be executed**
  - How to tile loops, parallelize operations, or manage data caching

➔ Concise and flexible description of optimizations

TENSORIR – SCHEDULE PRIMITIVES

```
for i in range( ):
  for k in range( ):
    C[i, k] += A[i, k] * B[k]
```

Original Tensor Program

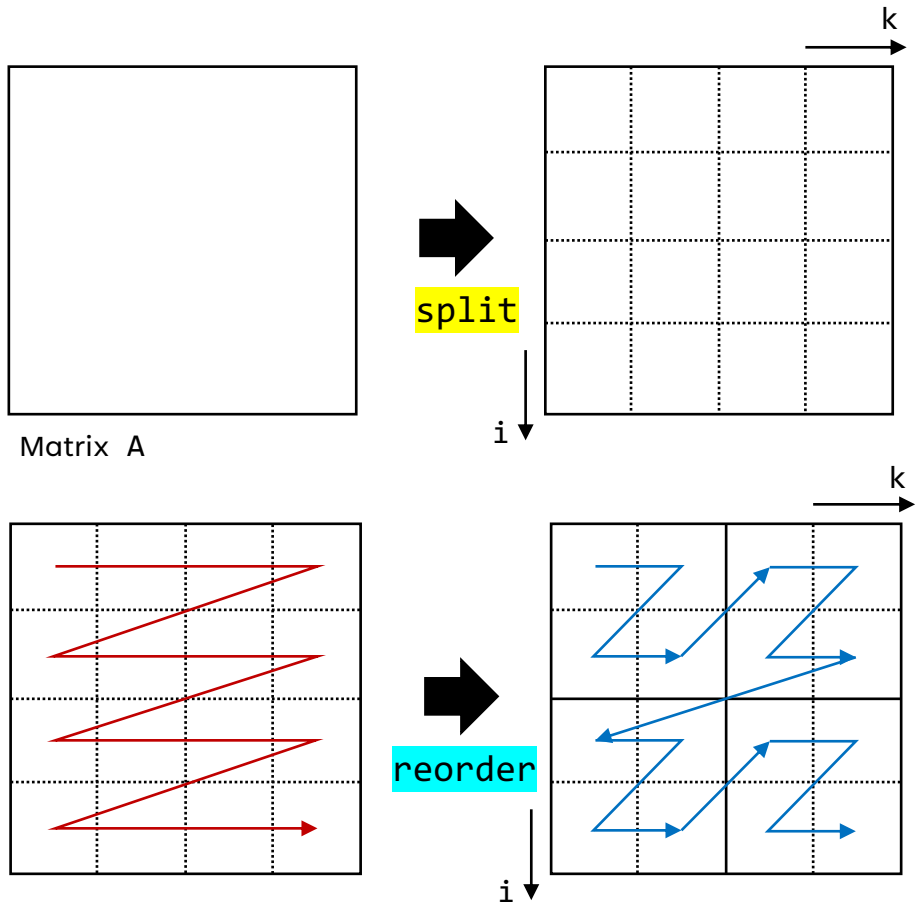
Input

```
split i to y.0, y.1
split k to k.0, k.1
reorder y.0, k.0, y.1, k.1
cache_read A compute_at y.1
cache_read B compute_at y.1
bind y.0 to thread.x
```

High-level Optimizations

Schedule primitives

- Tiling for locality by splitting tensors and setting ordering



TENSORIR – SCHEDULE PRIMITIVES

```
for i in range( ):
  for k in range( ):
    C[i, k] += A[i, k] * B[k]
```

Original Tensor Program

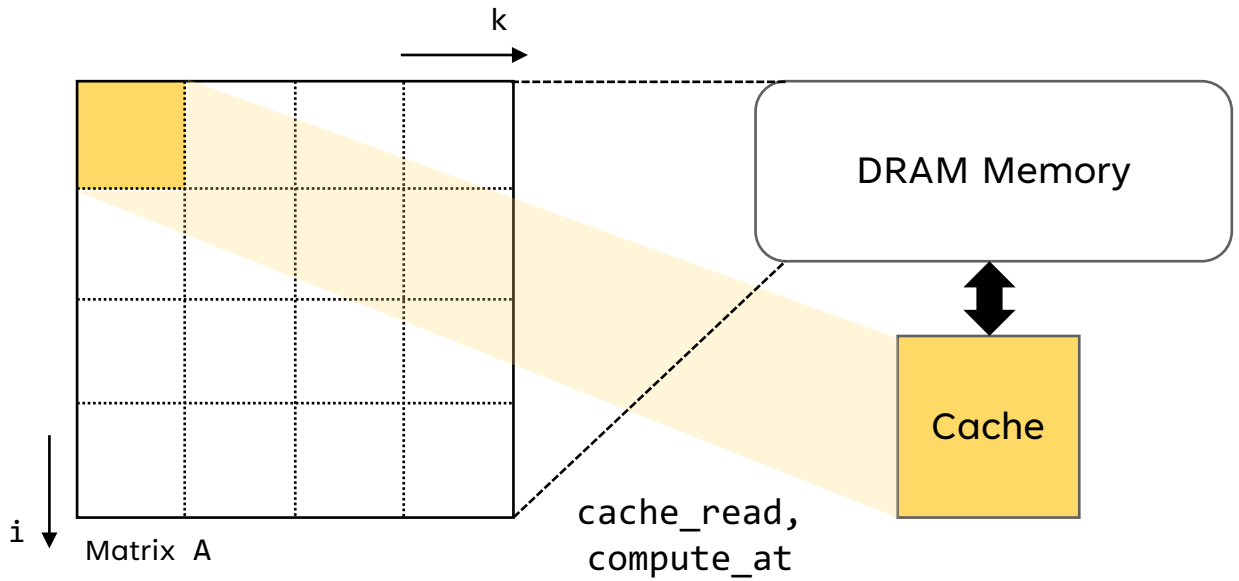
Input

```
split i to y.0, y.1
split k to k.0, k.1
reorder y.0, k.0, y.1, k.1
cache_read A compute_at y.1
cache_read B compute_at y.1
bind y.0 to thread.x
```

High-level Optimizations

Schedule primitives

- Specify explicit data caching



TENSORIR – SCHEDULE PRIMITIVES

```
for i in range( ):
  for k in range( ):
    C[i, k] += A[i, k] * B[k]
```

Original Tensor Program

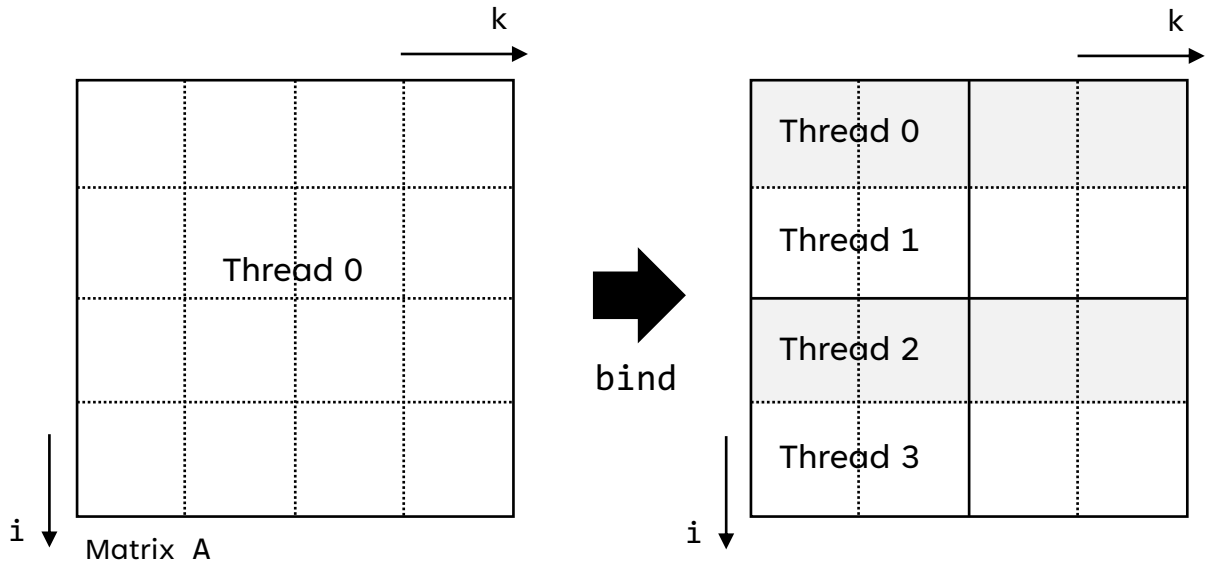
Input

```
split i to y.0, y.1
split k to k.0, k.1
reorder y.0, k.0, y.1, k.1
cache_read A compute_at y.1
cache_read B compute_at y.1
bind y.0 to thread.x
```

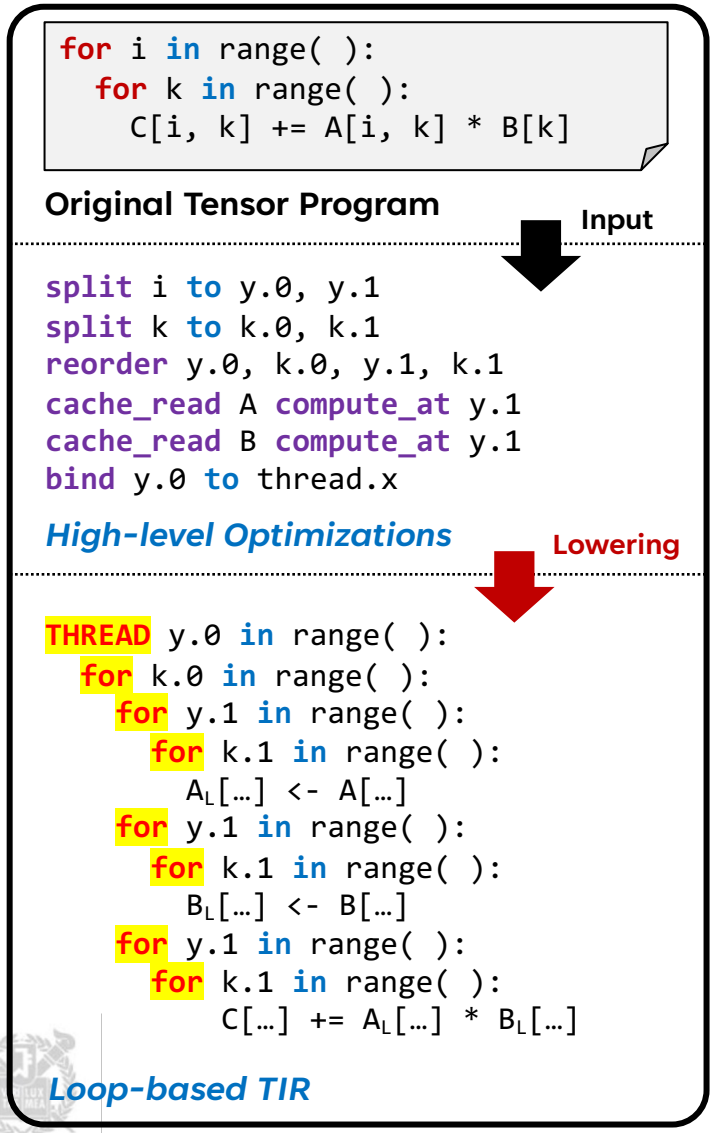
High-level Optimizations

Schedule primitives

- Specify thread-level parallelism



TENSORIR – LOWERING PROCESS



TIR lowering process

- Schedule primitives are lowered to TIR with explicit loops
- Enabling fine-grained hardware-aware optimization
- Seamless code generation for hardware
  - For both host and kernel code

TENSORIR – AUTOTUNING SUPPORT

```
for i in range( ):
  for k in range( ):
    C[i, k] += A[i, k] * B[k]
```

Original Tensor Program

Input

```
split i to y.0, y.1
split k to k.0, k.1
reorder y.0, k.0, y.1, k.1
cache_read A compute_at y.1
cache_read B compute_at y.1
bind y.0 to thread.x
```

High-level Optimizations

Lowering

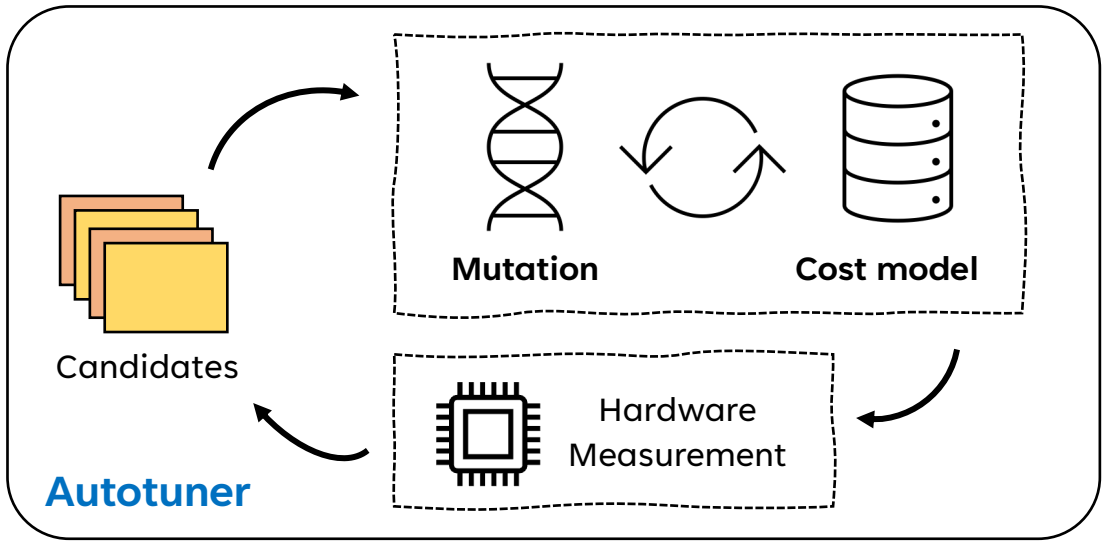
```
THREAD y.0 in range(⚙):
  for k.0 in range(⚙):
    for y.1 in range(⚙):
      for k.1 in range(⚙):
        A_L[…] <- A[…]
    for y.1 in range(⚙):
      for k.1 in range(⚙):
        B_L[…] <- B[…]
    for y.1 in range(⚙):
      for k.1 in range(⚙):
        C[…] += A_L[…] * B_L[…]
```

Loop-based TIR

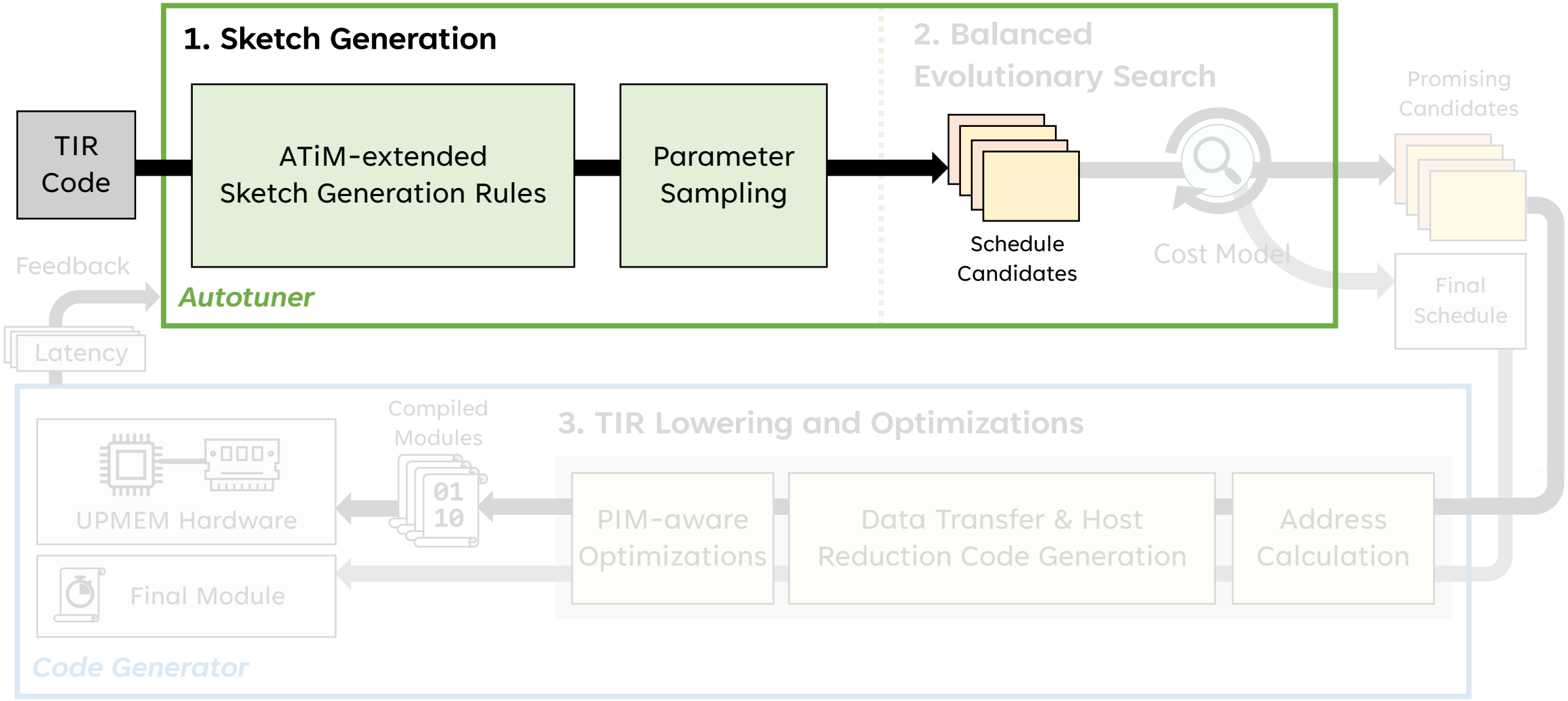
Adjust tunable parameters

Autotuning support

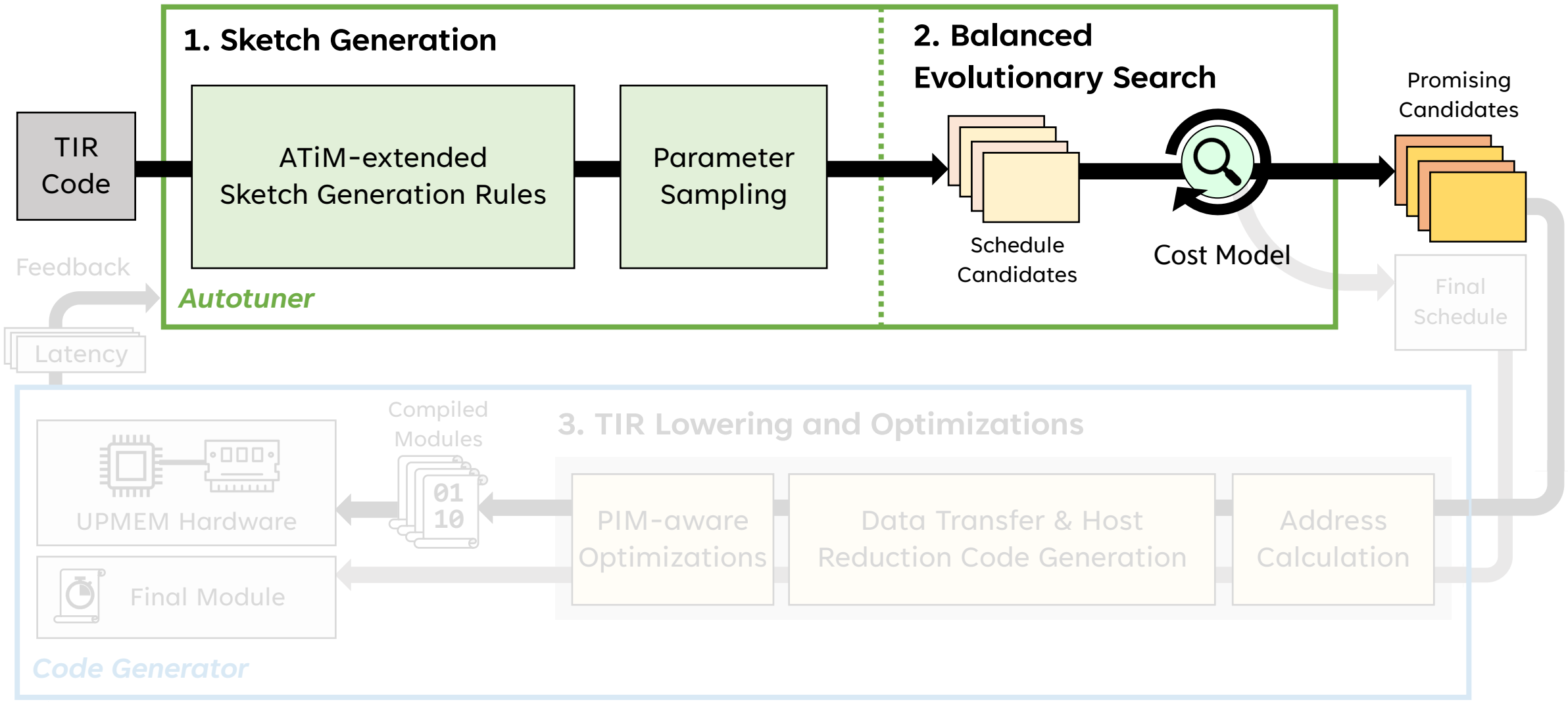
- The autotuner iteratively explores the code space for optimal performance using cost model–guided search and profiled runs



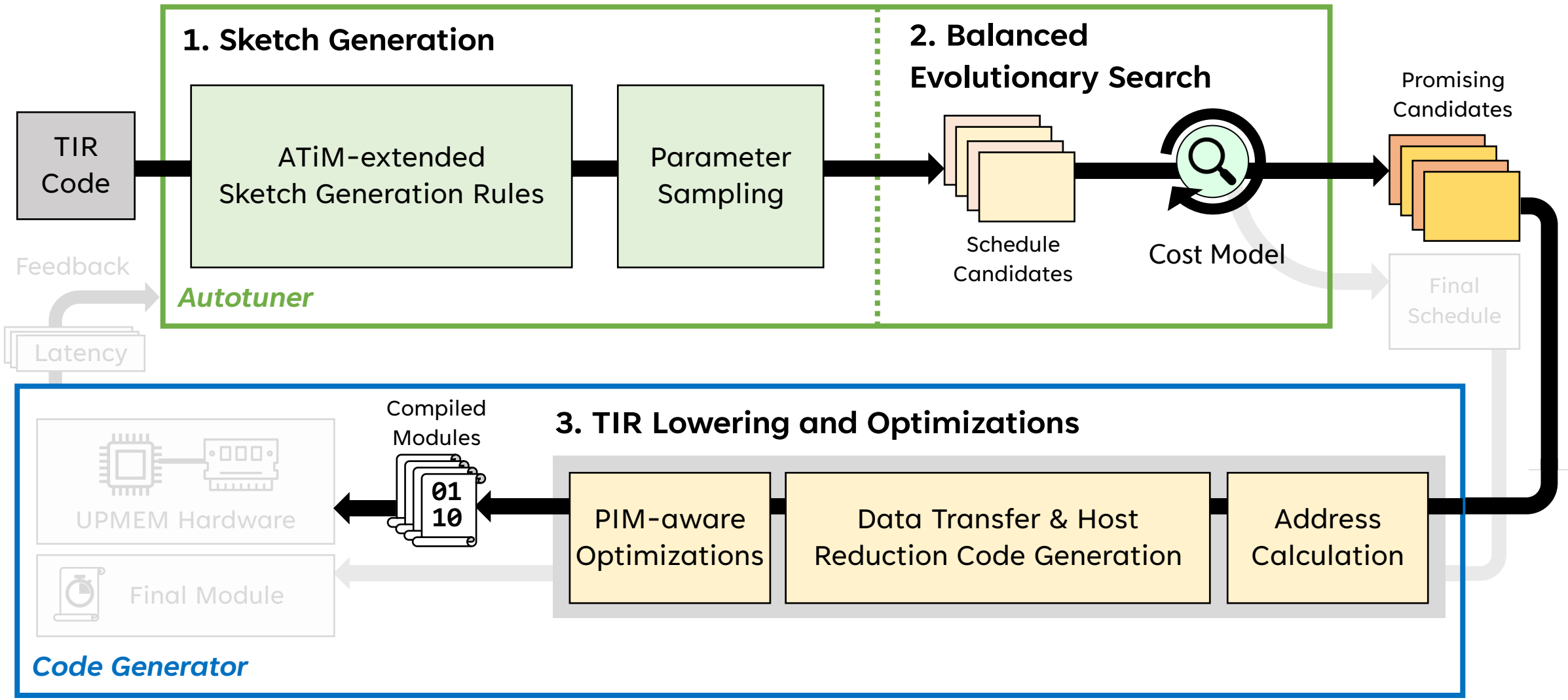
ATIM OVERVIEW



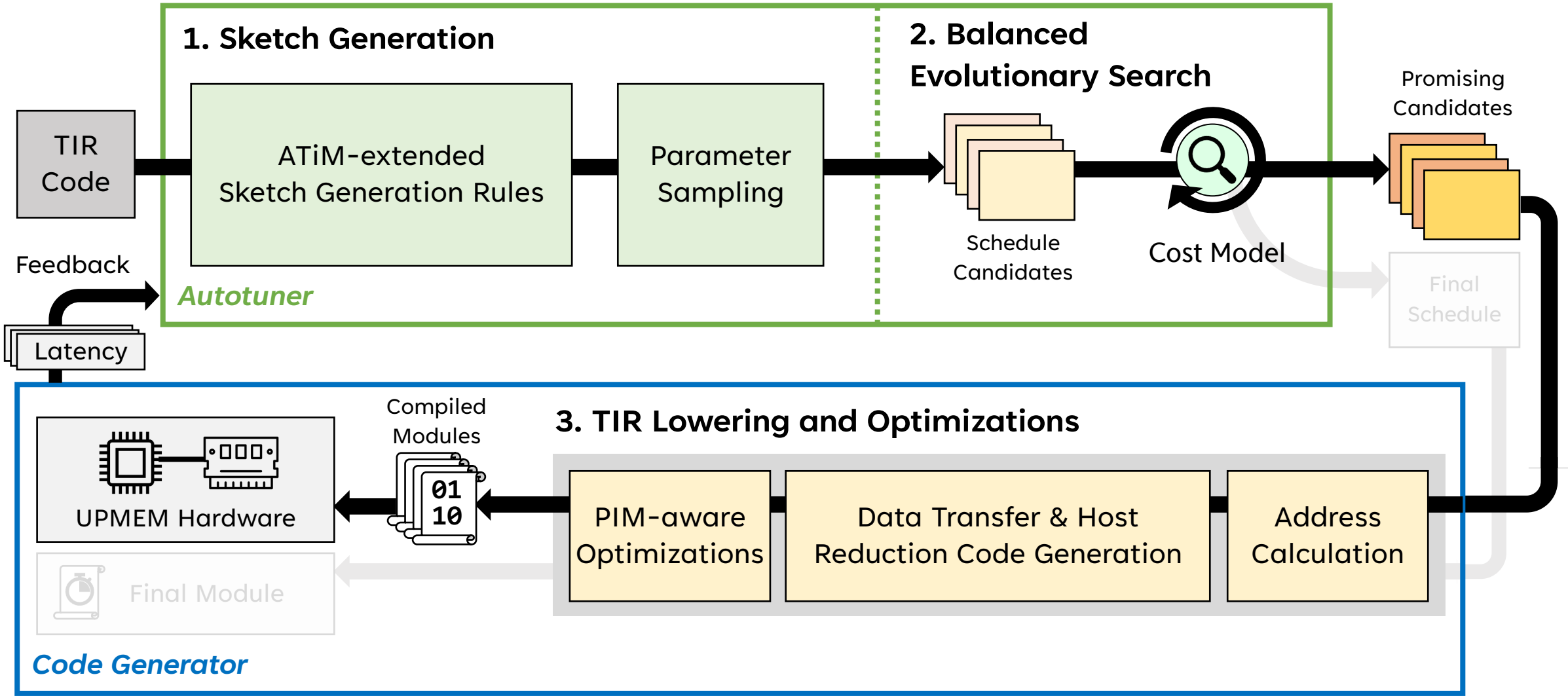
ATIM OVERVIEW



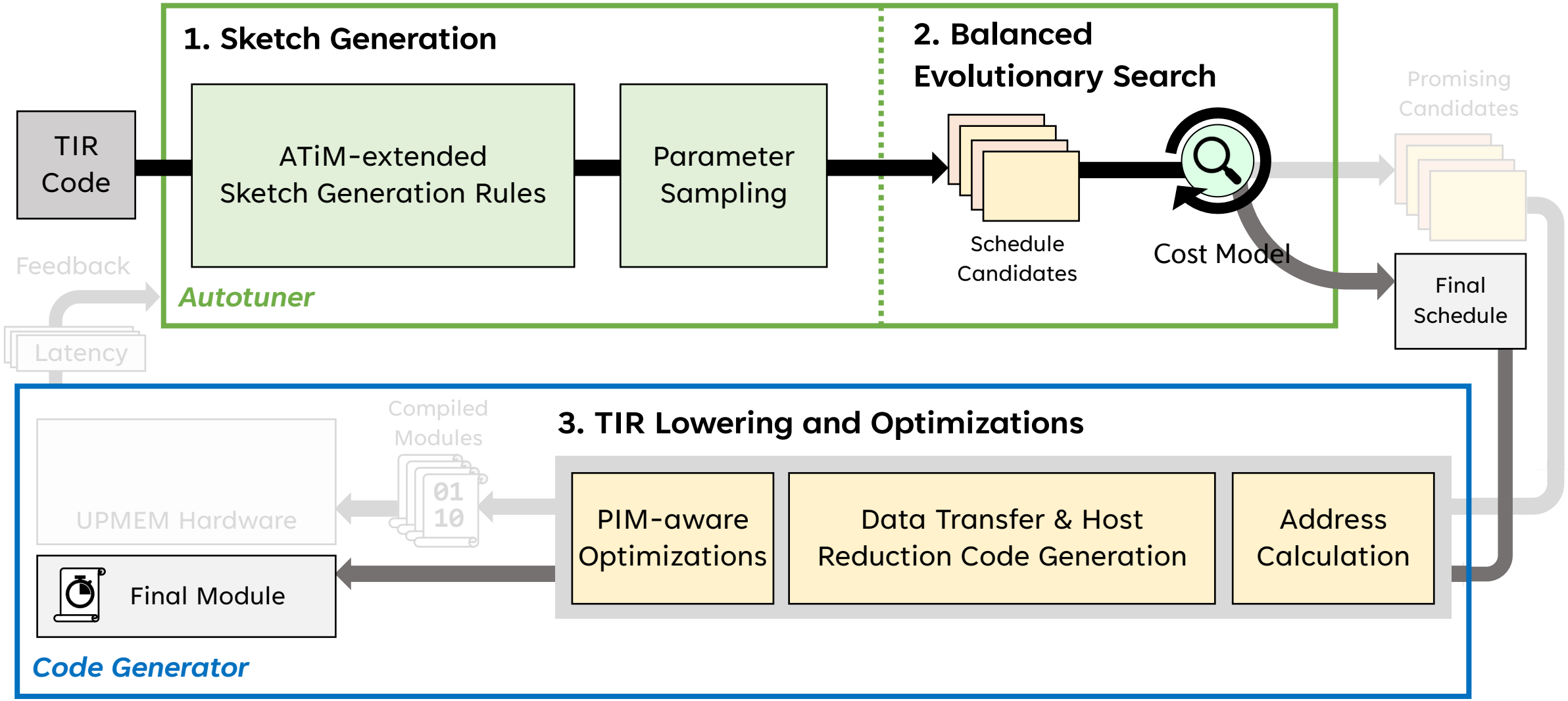
ATIM OVERVIEW



ATIM OVERVIEW

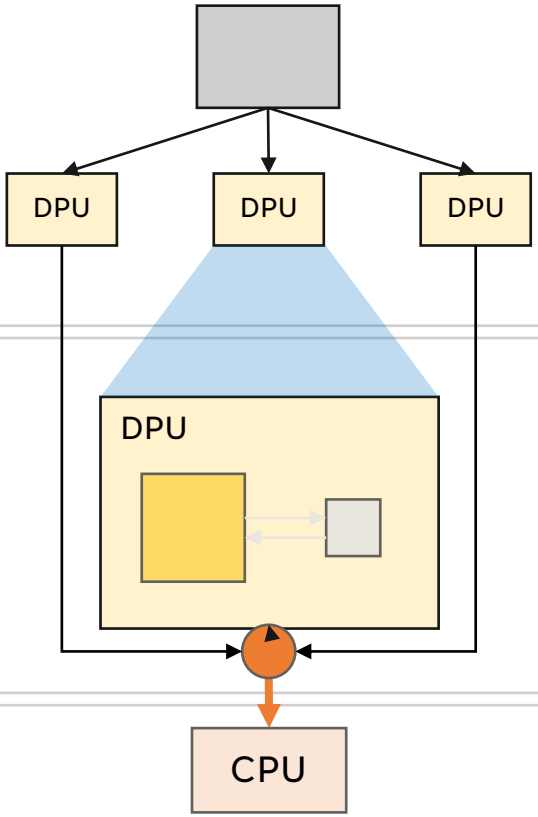


ATIM OVERVIEW

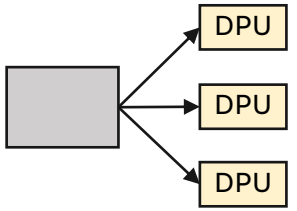


SCHEDULE CANDIDATE GENERATION

High-level Optimizations	Optimization Categories and Schedule Primitives
Host-to-DPU data distribution	Inter-DPU optimizations • `split, reorder, bind` • `rfactor`
Reduction Strategy	
Multi-level Tiling	Intra-DPU optimizations • `split, reorder, bind` • `cache_read, cache_write` • `compute_at, reverse_compute_at`
Intra-DPU Caching	
Post-processing	Post-processing at the host • `split, parallel`



INTER-DPU OPTIMIZATIONS

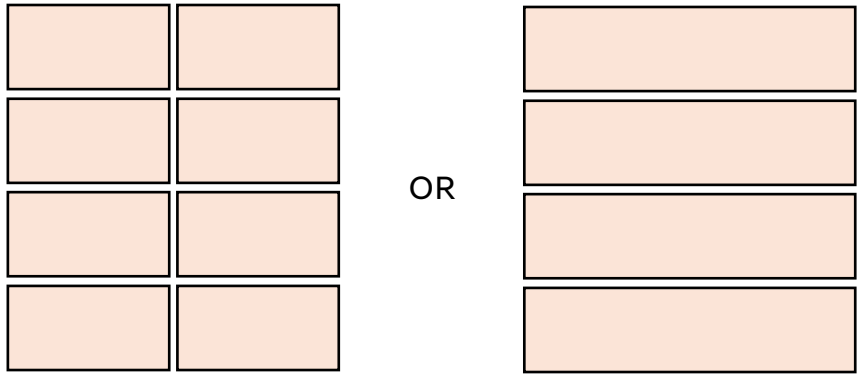


High-level Optimizations	Schedule Primitives	Tunable Parameters

Host-to-DPU Data Distribution	`split`
	`reorder`
	`bind`
		• Data distribution **policy** and **parallelism**

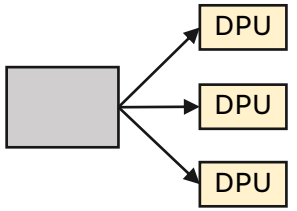
⚙: tunable parameters

```
x_dpu, x_in_dpu = sch.split(x, factors=[⚙])
y_dpu, y_in_dpu = sch.split(y, factors=[⚙])
sch.reorder(x_dpu, y_dpu, x_in_dpu, y_in_dpu)
sch.bind(x_dpu, “blockIdx.x”)
sch.bind(y_dpu, “blockIdx.y”)
block_dpu = sch.rfactor(x_dpu, factor_axis=0)
```



Control **tiling dimension** (2D vs 1D) and **parallelism** (8 vs 4)

INTER-DPU OPTIMIZATIONS

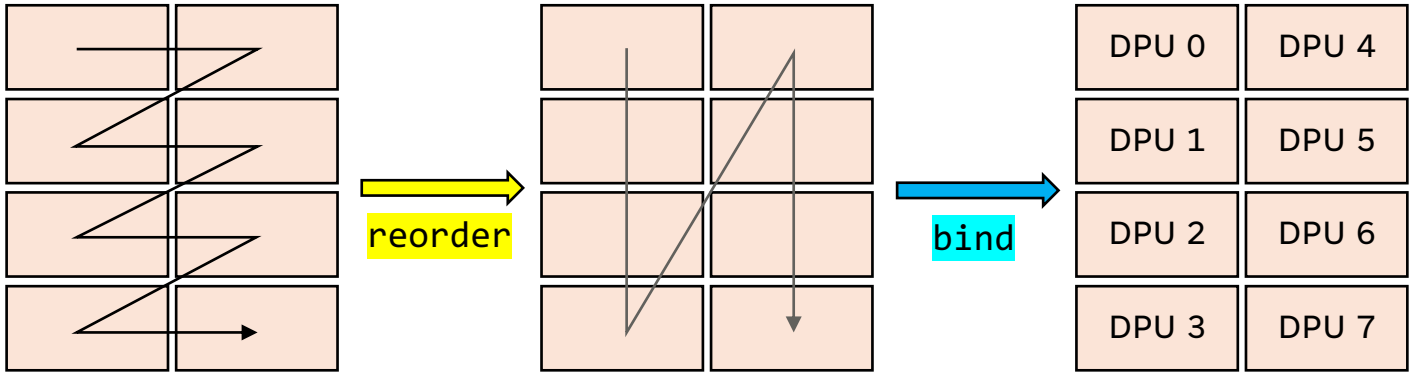


High-level Optimizations	Schedule Primitives	Tunable Parameters

Host-to-DPU Data Distribution	`split`
	`reorder`
	`bind`
		• Data distribution **policy** and **parallelism**

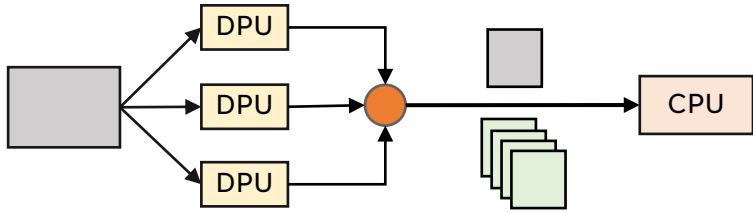
⚙: tunable parameters

```
x_dpu, x_in_dpu = sch.split(x, factors=[⚙])
y_dpu, y_in_dpu = sch.split(y, factors=[⚙])
sch.reorder(x_dpu, y_dpu, x_in_dpu, y_in_dpu)
sch.bind(x_dpu, “blockIdx.x”)
sch.bind(y_dpu, “blockIdx.y”)
block_dpu = sch.rfactor(x_dpu, factor_axis=0)
```



Reorder **tiles** and set **tile-to-DPU mapping**
to finalize distribution policy

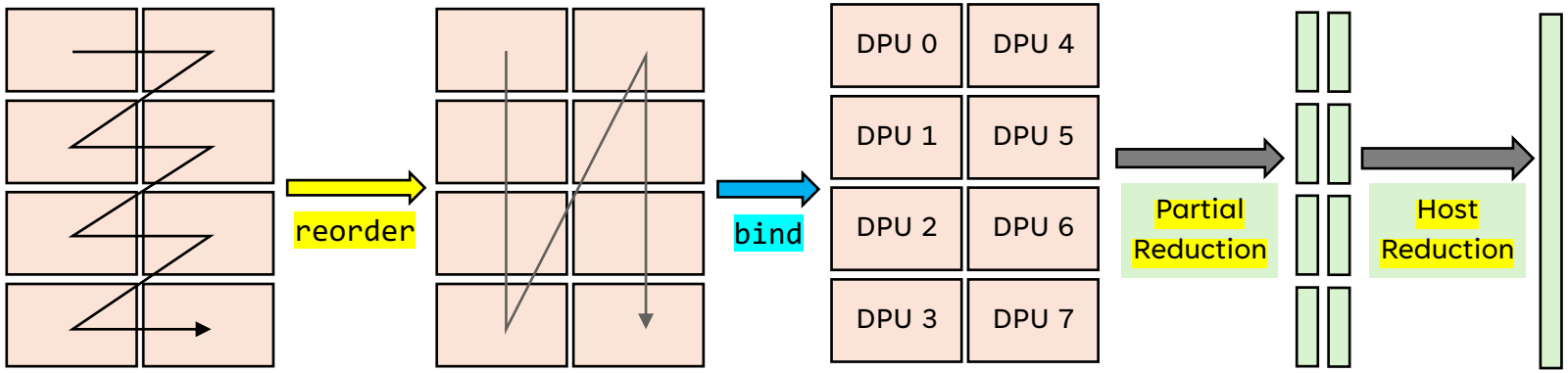
INTER-DPU OPTIMIZATIONS



High-level Optimizations	Schedule Primitives	Tunable Parameters
Reduction Strategy	`rfactor`	• Whether to partially compute results on DPUs and aggregate them on the host

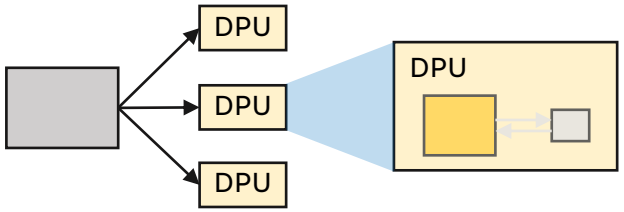
⚙: tunable parameters

```
x_dpu, x_in_dpu = sch.split(x, factors=[⚙])
y_dpu, y_in_dpu = sch.split(y, factors=[⚙])
sch.reorder(x_dpu, y_dpu, x_in_dpu, y_in_dpu)
sch.bind(x_dpu, “blockIdx.x”)
sch.bind(y_dpu, “blockIdx.y”)
block_dpu = sch.rfactor(x_dpu, factor_axis=0)
```



Optionally apply **hierarchical reduction** to reduce input data movement

INTRA-DPU OPTIMIZATIONS

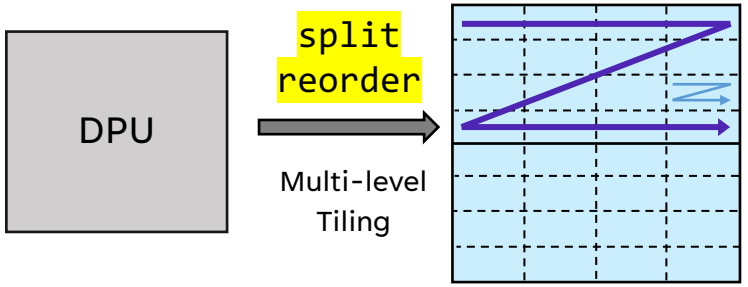


High-level Optimizations	Schedule Primitives	Tunable Parameters

Multi-level Tiling	`split`
	`reorder`
	`bind`
		• Caching policy in WRAM & register blocking
		• Tasklet-level parallelism

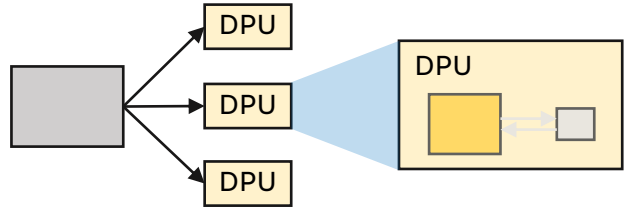
```
thread_dpu, y_cache, y_in = sch.split(y_in_dpu, factors=[⚙])
x_cache, x_in = sch.split(x_in_dpu, factors=[⚙])
sch.reorder(thread_dpu, y_cache, x_cache, y_in, x_in)
sch.bind(thread_dpu, "threadIdx.x")
```

⚙: tunable parameters



Perform multi-level tiling of the intra-DPU tensor for data locality (**WRAM caching** and **register blocking**)

INTRA-DPU OPTIMIZATIONS

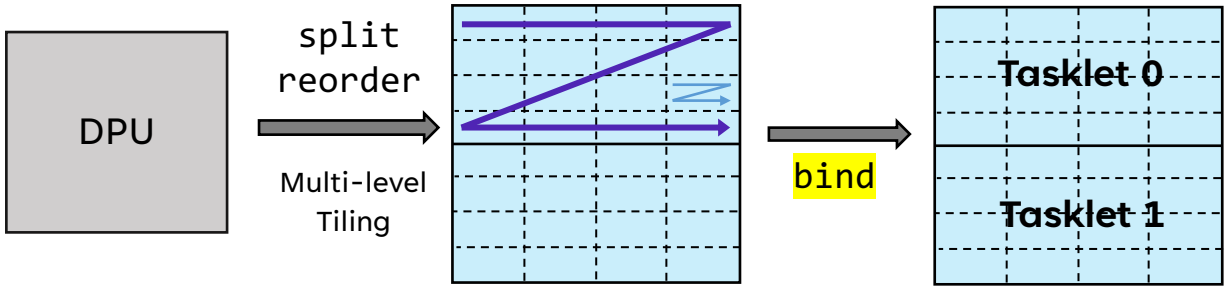


High-level Optimizations	Schedule Primitives	Tunable Parameters

Multi-level Tiling	`split`
	`reorder`
	`bind`
		• Caching policy in WRAM & register blocking
		• Tasklet-level parallelism

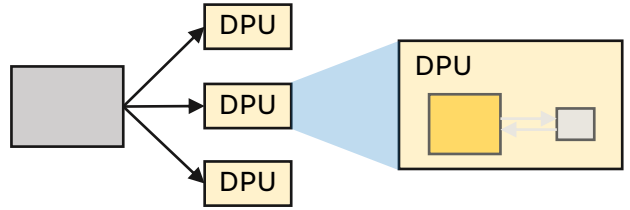
```
thread_dpu, y_cache, y_in = sch.split(y_in_dpu, factors=[⚙])
x_cache, x_in = sch.split(x_in_dpu, factors=[⚙])
sch.reorder(thread_dpu, y_cache, x_cache, y_in, x_in)
sch.bind(thread_dpu, "threadIdx.x")
```

⚙: tunable parameters



Control **tasklet-level parallelism**

INTRA-DPU OPTIMIZATIONS

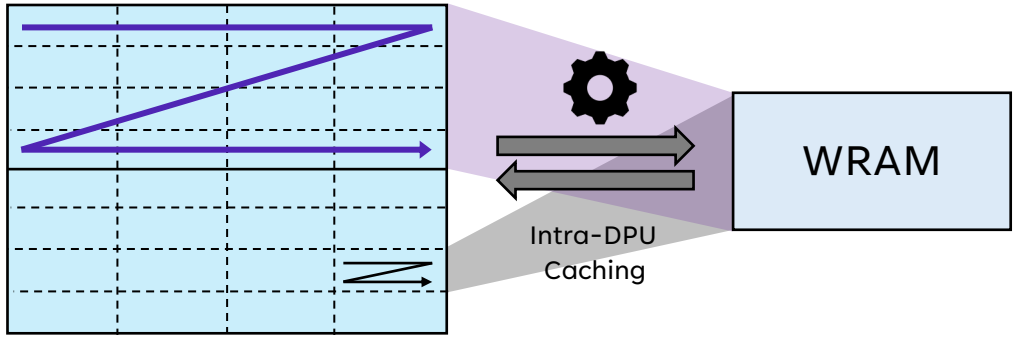


High-level Optimizations	Schedule Primitives	Tunable Parameters

Intra-DPU Caching	`cache_read`
	`cache_write`
	`compute_at`
	`reverse_compute_at`
		• Caching locations and sizes
		• Data transfer granularities (MRAM ↔ WRAM)

```
cache_a = sch.cache_read(block_dpu, 0, "local")
cache_b = sch.cache_read(block_dpu, 1, "local")
cache_c = sch.cache_write(block_dpu, 0, "local")
sch.compute_at(cache_a ⚙, x_cache)
sch.compute_at(cache_b ⚙, x_cache)
sch.reverse_compute_at(cache_c ⚙, y_cache)
```

⚙: tunable parameters



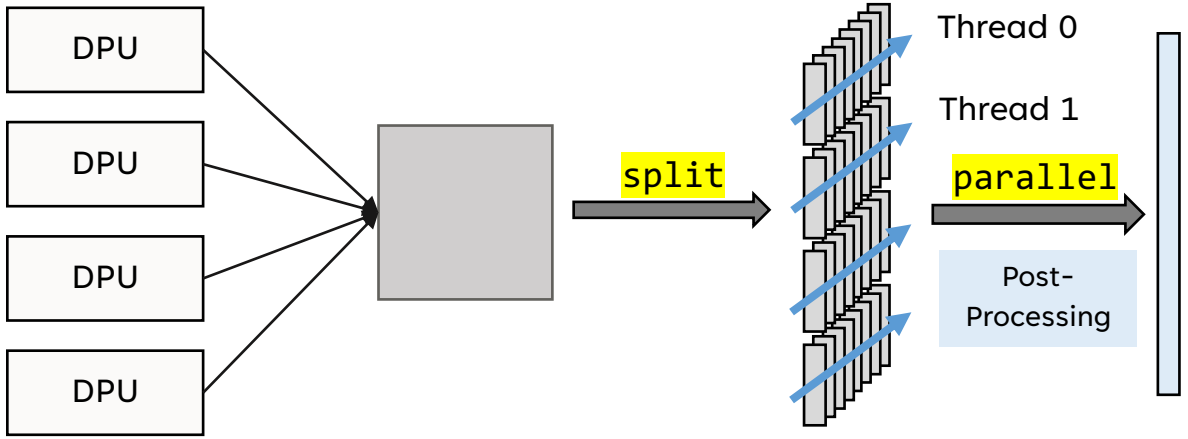
Set **caching locations** and **data transfer granularities**

POST-PROCESSING AT THE HOST

High-level Optimizations	Schedule Primitives	Optimization Target
Post-Processing	`split` `parallel`	• Thread-level parallelism for parallel reduction on the host

```
thread, _ = sch.split(y, factors=[⚙])
sch.parallel(thread)
```

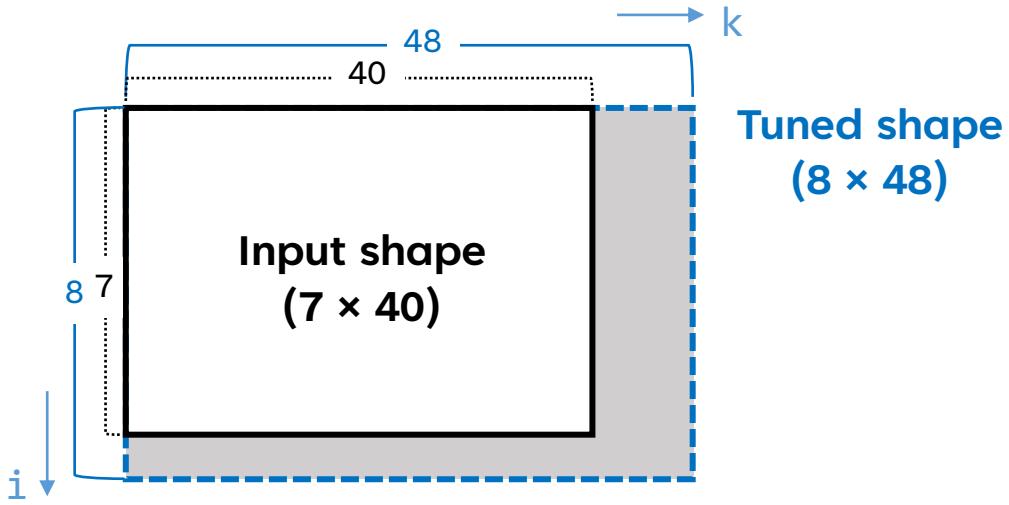
⚙: tunable parameters



Control **Thread-level parallelism** at the host CPU

PIM-AWARE OPTIMIZATIONS

- **Boundary check conditions** are generated when input tensor shape does not match the tuned module's expected shape




```
for j in [0, 3):
  for k in [0, 16):
    A_L[k] = A[base(i) + k]
  for k in [0, 16):
    B_L[K] = B[base + k]
  for k in [0, 16):
    C_L[i] += A_L[k] + B_L[k]
```

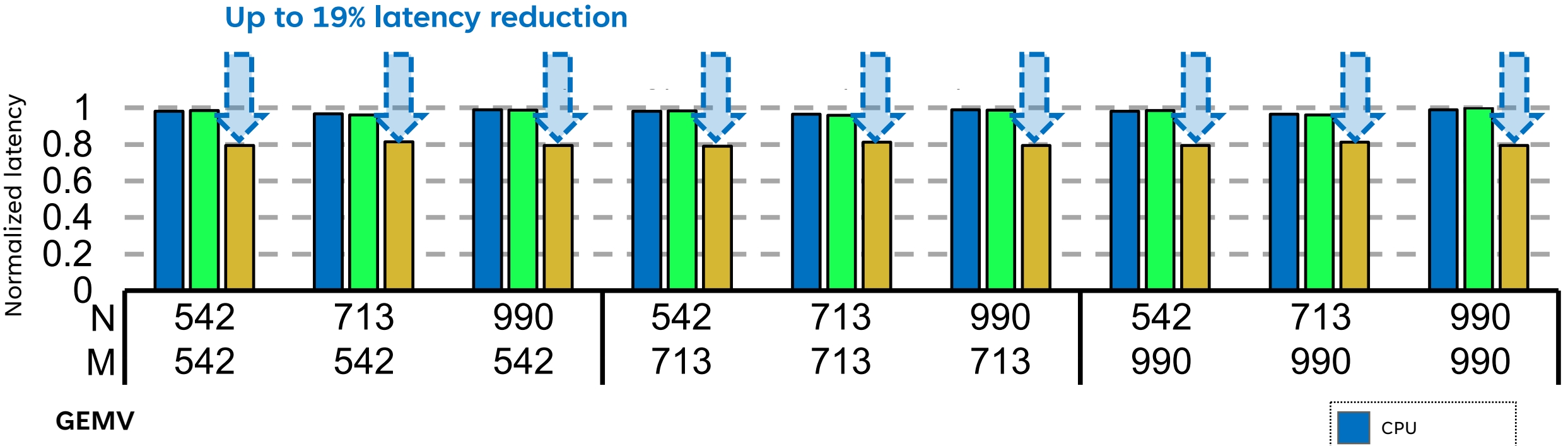
```
for j in [0, 3):
  for k in [0, 16):
    if boundary(k) ∧ boundary(i):
      A_L[k] = A[base(i) + k]
  for k in [0, 16):
    if boundary(k):
      B_L[K] = B[base + k]
  for k in [0, 16):
    if boundary(k) ∧ boundary(i):
      C_L[i] += A_L[k] + B_L[k]
```

Boundary check code is generated

PIM–AWARE OPTIMIZATIONS

UPMEM lacks advanced latency–hiding hardware logic

➔ **Reducing boundary checks** can significantly lower latency



PIM-AWARE OPTIMIZATIONS

- UPMEM lacks advanced latency-hiding hardware logic

➔**Reducing boundary checks** can significantly lower latency

- Thanks to TIR's high-level semantics, ATiM can rewrite code aggressively to reduce boundary checks

- Low-level compiler (e.g., LLVM) cannot apply the above rewriting as it is considered unsafe without high-level semantics

PIM-AWARE OPTIMIZATIONS

1. DMA-aware boundary check elimination

Original TIR Code

```
for j in [0, 3):
  for k in [0, 16):
    if boundary(k) ∧ boundary(i):
      A_L[k] = A[base(i) + k]
  for k in [0, 16):
    if boundary(k):
      B_L[K] = B[base + k]
  for k in [0, 16):
    if boundary(k) ∧ boundary(i):
      C_L[i] += A_L[k] + B_L[k]
```

```
for j in [0, 3):
  mram_read(A, 16)
  mram_read(B, 16)
  for k in [0, 16):
    if boundary(k) ∧ boundary(i):
      C_L[i] += A_L[k] + B_L[k]
```

Remove boundary check and replace the loop with a DMA function call

Why is this safe?

(1) Memory regions are locally padded to the tile size

(2) The same boundary checks are repeated later (in the computation statement)

PIM-AWARE OPTIMIZATIONS

2. Loop bound tightening

TIR Code

```
for j in [0, 3):
  mram_read(A, 16)
  mram_read(B, 16)
  for k in [0, 16):
    if boundary(k) ∧ boundary(i):
      C_L[i] += A_L[k] + B_L[k]
```

```
for j in [0, 3]:
  mram_read(A, 16)
  mram_read(B, 16)
  for k in [0, min(16, K')):
    if boundary(i):
      C_L[i] += A_L[k] + B_L[k]
```

Fold boundary checks into loop bound by solving linear inequalities

Preconditions

(1) The boundary check condition can intersect with the loop bound condition

(2) No other instructions exist in the loop body (perfectly nested)

PIM-AWARE OPTIMIZATIONS

3. Invariant branch hoisting

TIR Code

```
for j in [0, 3]:
  mram_read(A, 16)
  mram_read(B, 16)
  for k in [0, min(16, K')):
    if boundary(i):
      C_L[i] += A_L[k] + B_L[k]
```

```
if boundary(i):
  for j in [0, 3):
    mram_read(A, 16)
    mram_read(B, 16)
    for k in [0, min(16, k')):
      C_L[i] += A_L[k] + B_L[k]
```

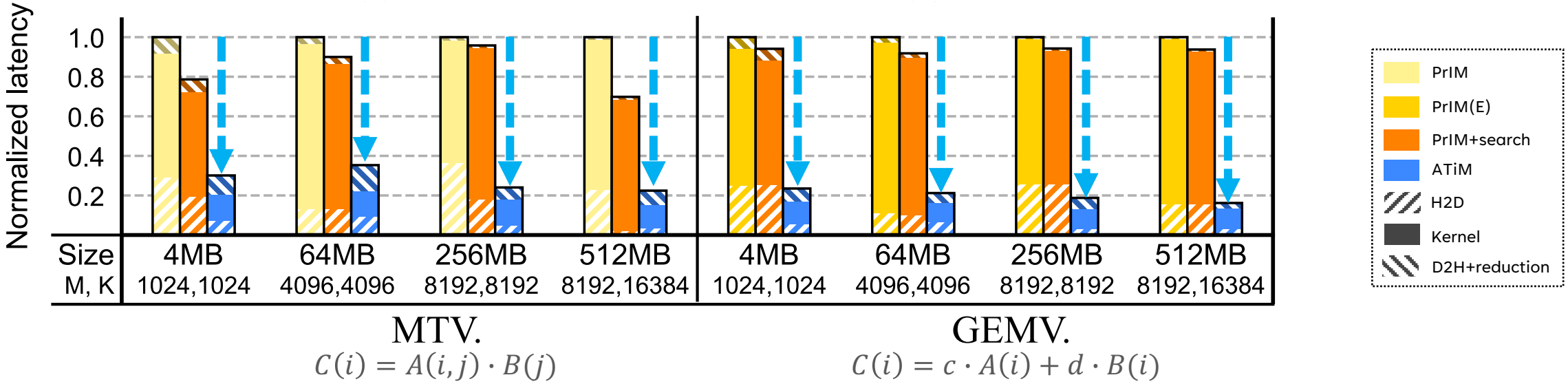
Aggressively hoist invariant boundary checks to outer loops

How it works?

- Hoisting boundary check condition above data transfer intrinsics (mram_read)

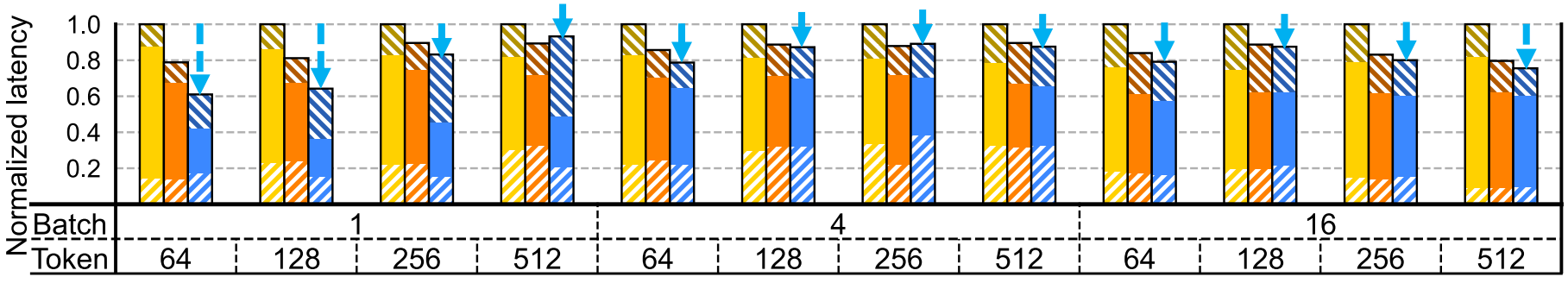
➔ Cannot guarantee safety without high-level semantics (e.g., LLVM)

AUTOTUNED PERFORMANCE OF MTV AND GEMV



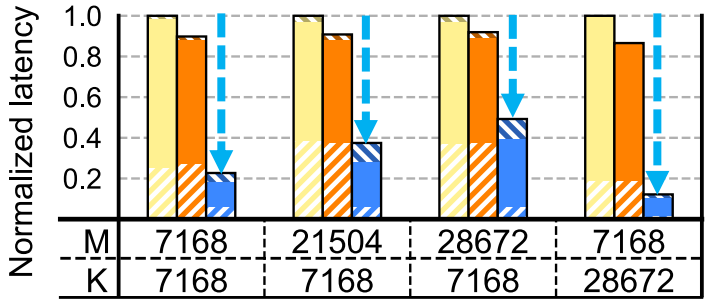
- ATiM achieves speedups of up to 6.2× and 5.8× over PrIM/(E) and PrIM+search, respectively
- ATiM finds optimal parameters for
  - 2D tiling with hierarchical reduction → reducing the amount of input vector data movement
  - Autotuned caching locations and sizes → hard to discover by hand-tuned code

AUTOTUNED PERFORMANCE OF GPT-J LAYERS



MMTV in GPT-J 30B.

$$C(i,j) = \sum_k A(i,j,k) \cdot B(i,k)$$

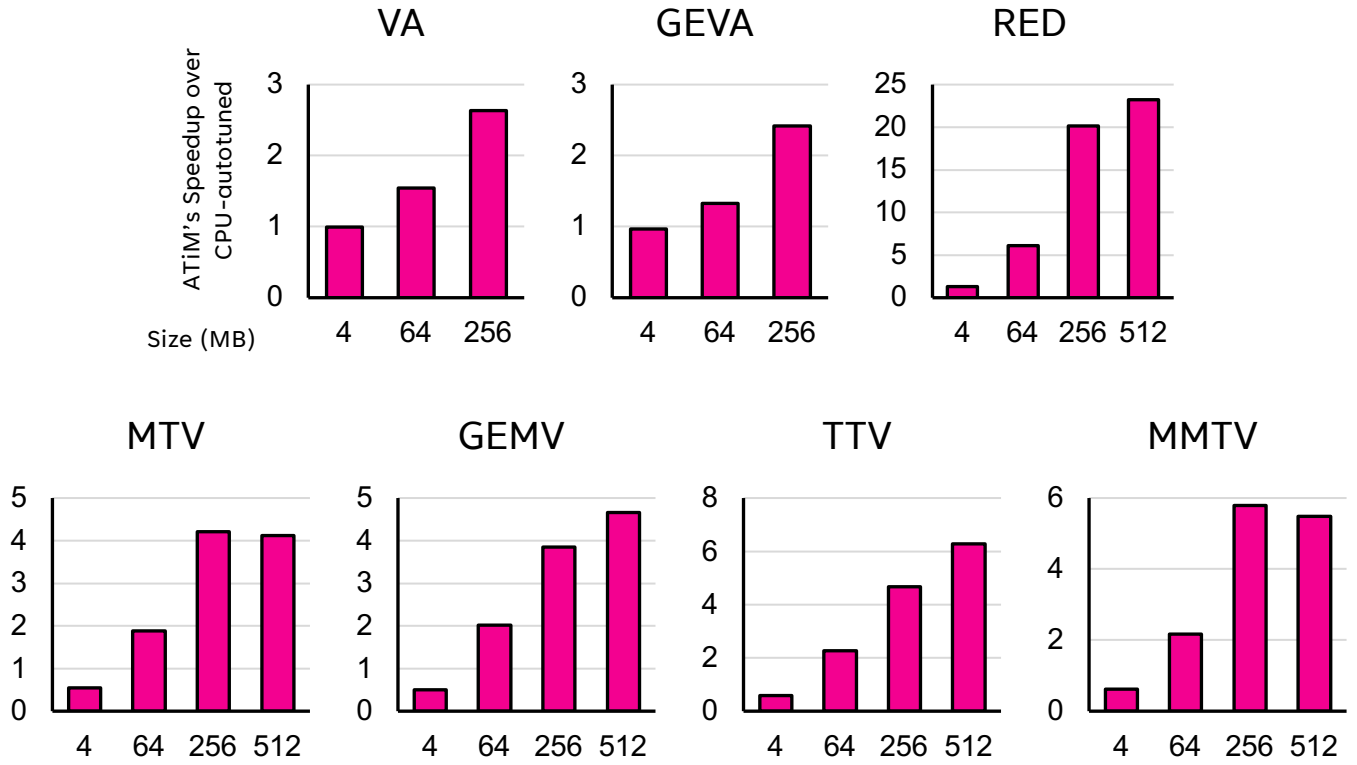


MTV in GPT-J 30B.

$$C(i) = A(i,j) \cdot B(j)$$

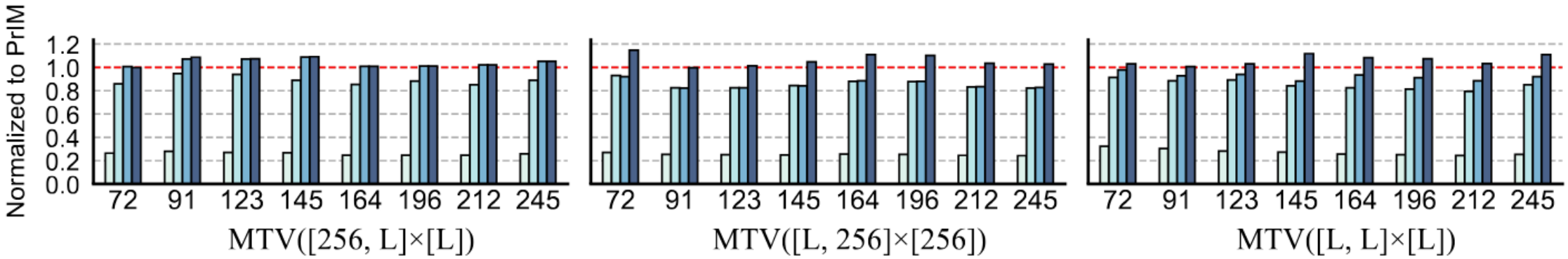
- For MMTV, ATiM outperforms PrIM by an average of 5.7%, reaching 11.7% on average for single batch scenarios
  - Efficient caching data sizes
  - More DPU-level parallelism for small tensors due to its 2D DPU tiling
- For MTV, ATiM shows significant performance improvement over PrIM and PrIM+search up to 6× across all four MTV kernels in GPT-J 30B

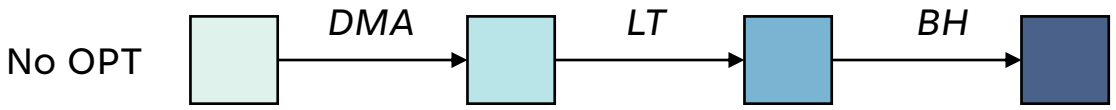
AUTOTUNED PERFORMANCE OF CPU VS. UPMEM



- ATiM consistently outperforms CPU up to 23× for tensors ≥ 64MB
  - CPU suffers from data movement overhead for large tensors
  - ATiM benefits from in-place parallel computation across DPUs

PERFORMANCE OF PIM-AWARE OPTIMIZATIONS





DMA: DMA-aware boundary check elimination
LT: loop-bound tightening
BH: invariant branch hoisting

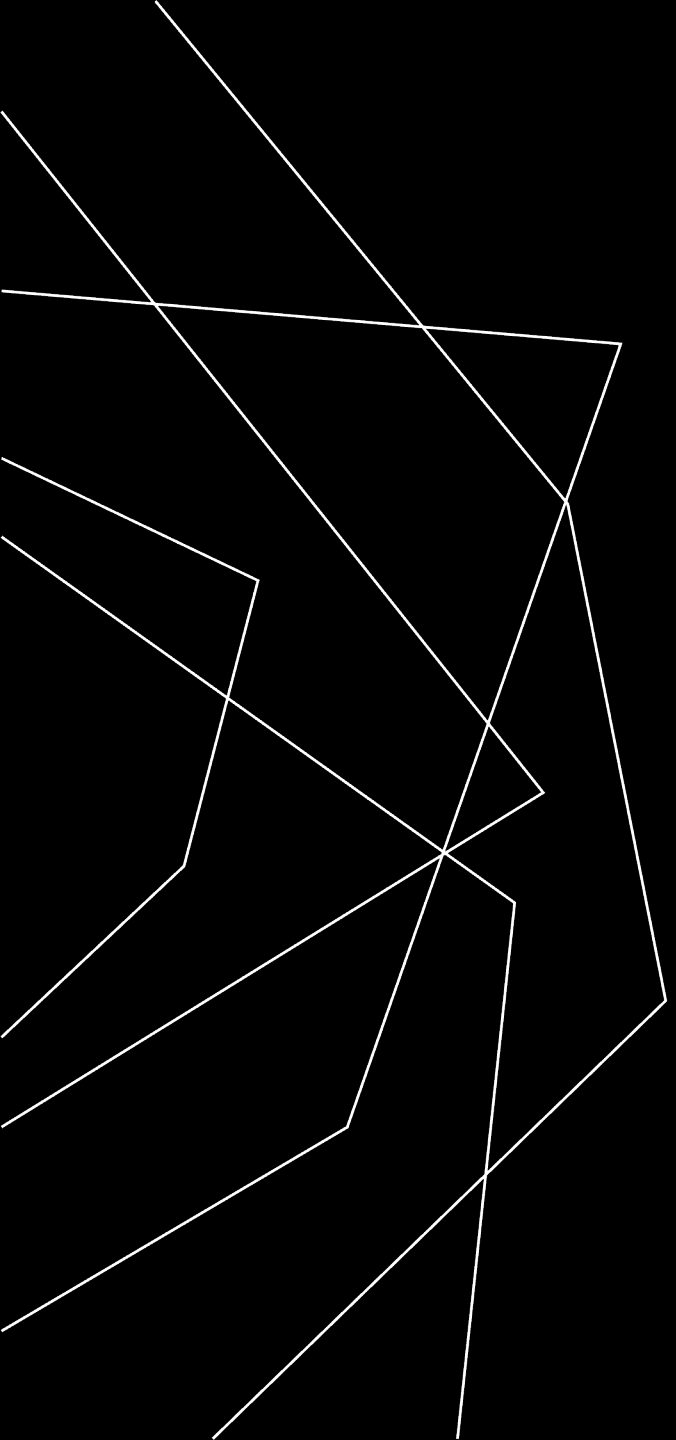
- ATiM achieves up to 14.7% speedup for various input shapes of MTV
  - DMA provides largest performance gain
  - LT and BH improve performance for workloads with misaligned columns and rows, respectively

SUMMARY

ATiM

- Enables search-based code generation for PIM systems
- Proposes PIM-aware optimizations
- Improves autotuning through a balanced evolutionary search

Practical and performant tensor compiler for current and future PIM hardware



QUESTIONS?