# DCC

# Data-Centric Compilation of Machine Learning Kernels for Processing-In-Memory Architectures


Peiming Yang,

Sankeerth Durvasula, Ivan Fernandez, Mohammad Sadrosadati,

Onur Mutlu, Gennady Pekhimenko, Christina Giannoula



ISCA 2026

Raleigh, USA, July 1st 2026

# Executive Summary

Problem: Host processors and PIM cores require different data layouts, thus necessitating expensive data rearrangements in ML kernel execution on PIM cores

Prior Works: Current PIM compilation approaches lack systematic optimization of diverse ML kernels across multiple PIM backends and may largely ignore data rearrangement costs

DCC: The first data-centric ML compiler for PIM systems that jointly co-optimizes data rearrangements and compute code in a unified tuning process for multiple PIM backends

Key Components & Benefits:

- Multi-Layer PIM Abstraction: enables support for multiple PIM backends
- Data-Centric Schedule Generator: couples data and compute-loop partition strategies for co-optimization
- PIM-Specific Code Optimizer: performs PIM-specific performance optimizations
- Coupled Predictor: selects the best-performing configuration across joint data and compute-loop partition strategies
- Intermediate Representation: provides a data-centric representation that can be lowered to multiple PIM backends

Key Results: DCC improves performance by 1.7× and 1.3× over default implementation on HBM-PIM and AttAcc backends, respectively, for individual ML kernels, and by 2.0× over for LLM inference

# Talk Outline

**Background**

Motivation

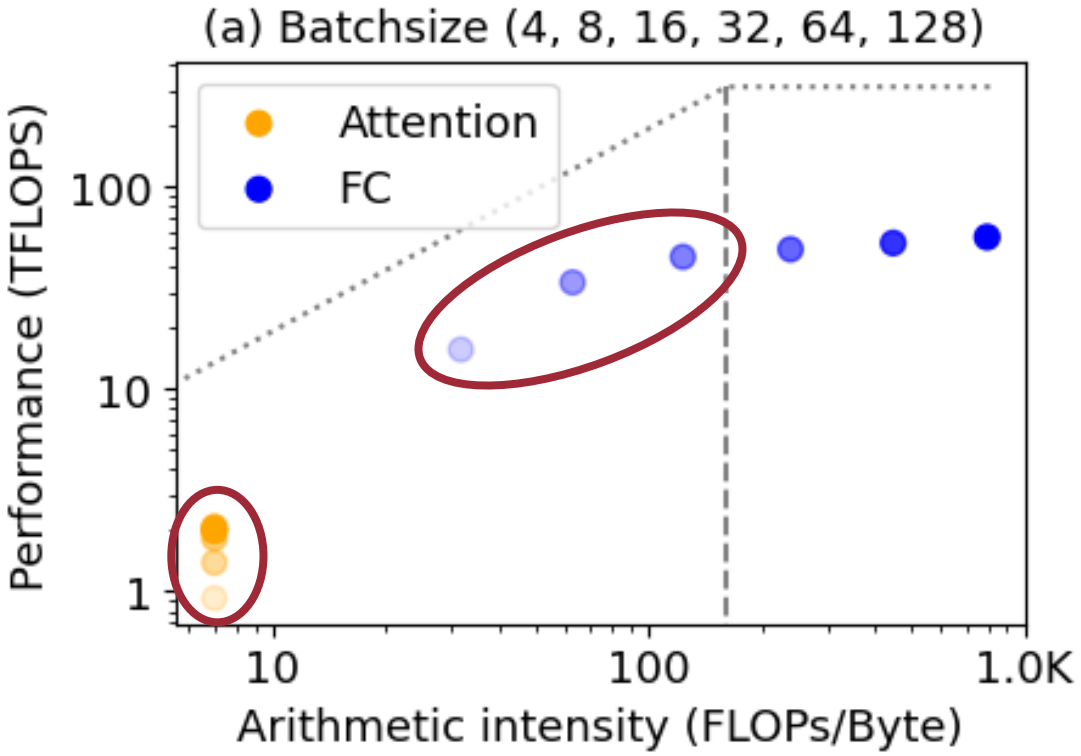
DCC Design

Evaluation

# Critical ML Kernels are Bottlenecked by Data Movement

Large Language Models (LLMs):

- Attention → always memory-bandwidth-bound
- Fully-Connected (FC) → is memory-bandwidth-bound for certain batch sizes

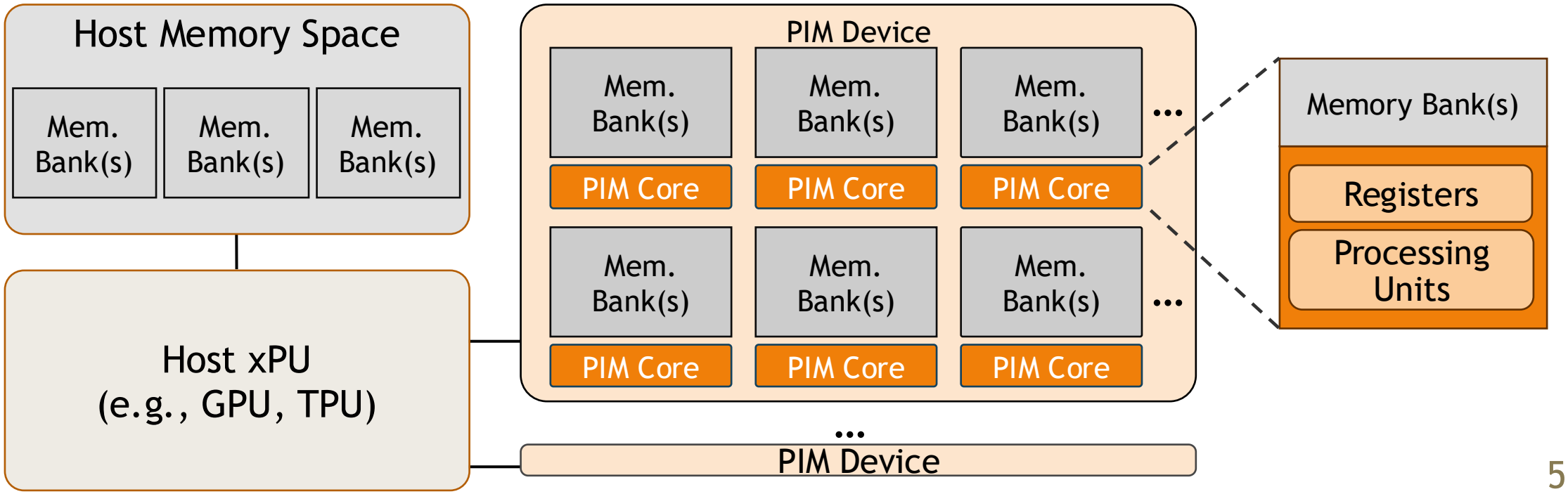



**OPT-30B Inference kernels on NVIDIA A100 GPU**

Source: Y. He, et al., "PAPI: Exploiting Dynamic Parallelism in LLM Decoding with a PIM-enabled Computing System", ASPLOS'24

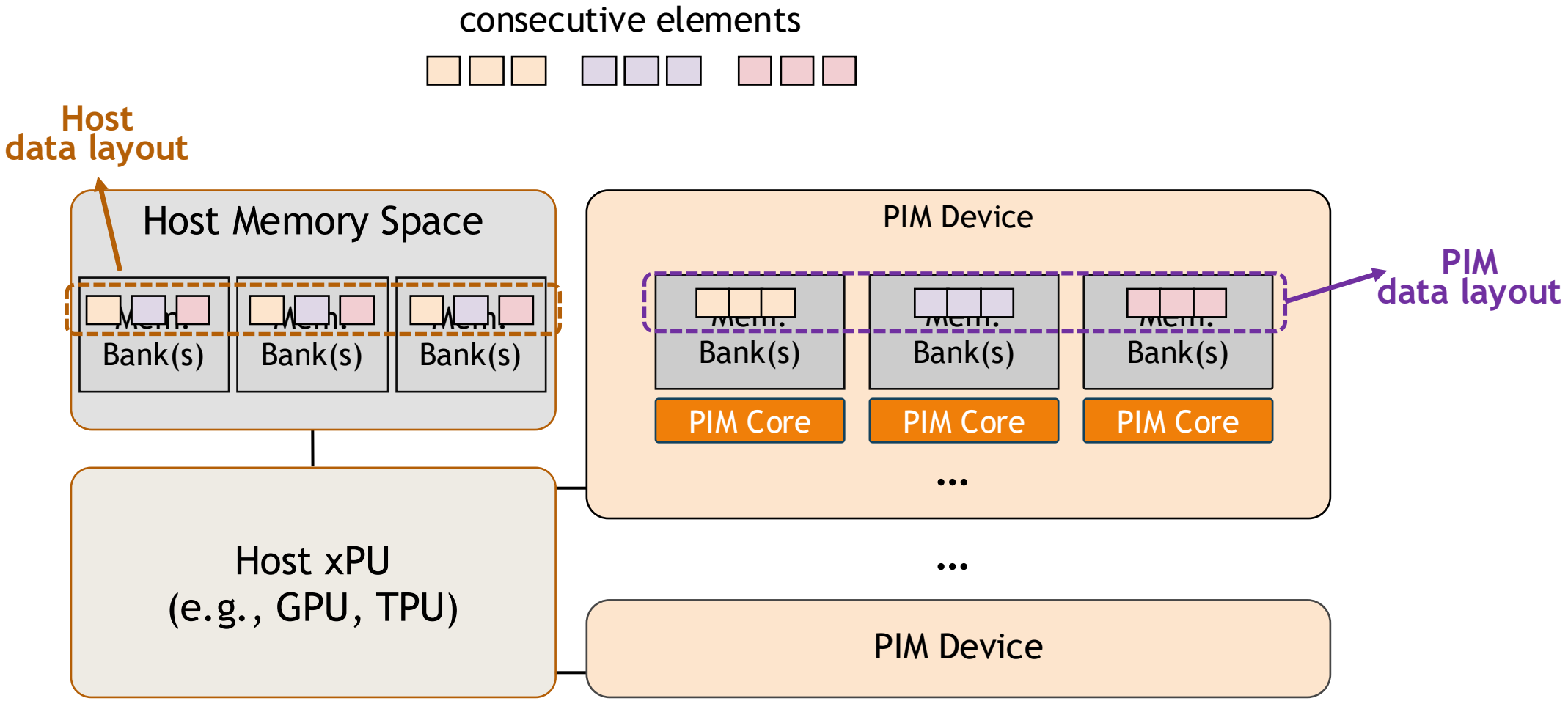
# PIM Alleviates Data Movement Bottlenecks

- Processing-In-Memory (PIM) places processing units near memory arrays
- Prior works (HBM-PIM [ISCA'21], AttAcc [ASPLOS'24], PAPI [ASPLOS'24]) design near-bank PIM systems for ML kernels
- Near-bank PIM systems:
  - Integrated with high-performance processor – Host xPU (e.g., GPU, TPU)
  - Multiple PIM-enabled memory devices
  - Low-power processing units (PIM core) close to one (or a few) memory bank(s)
  - Immense aggregated memory bandwidth & High-levels of execution parallelism



# Host xPU vs PIM Cores Require Different Data Layouts

- Host processors: consecutive elements are distributed across memory banks
- PIM cores: consecutive elements are placed within local memory banks
  - PIM cores can only access data from their local memory banks

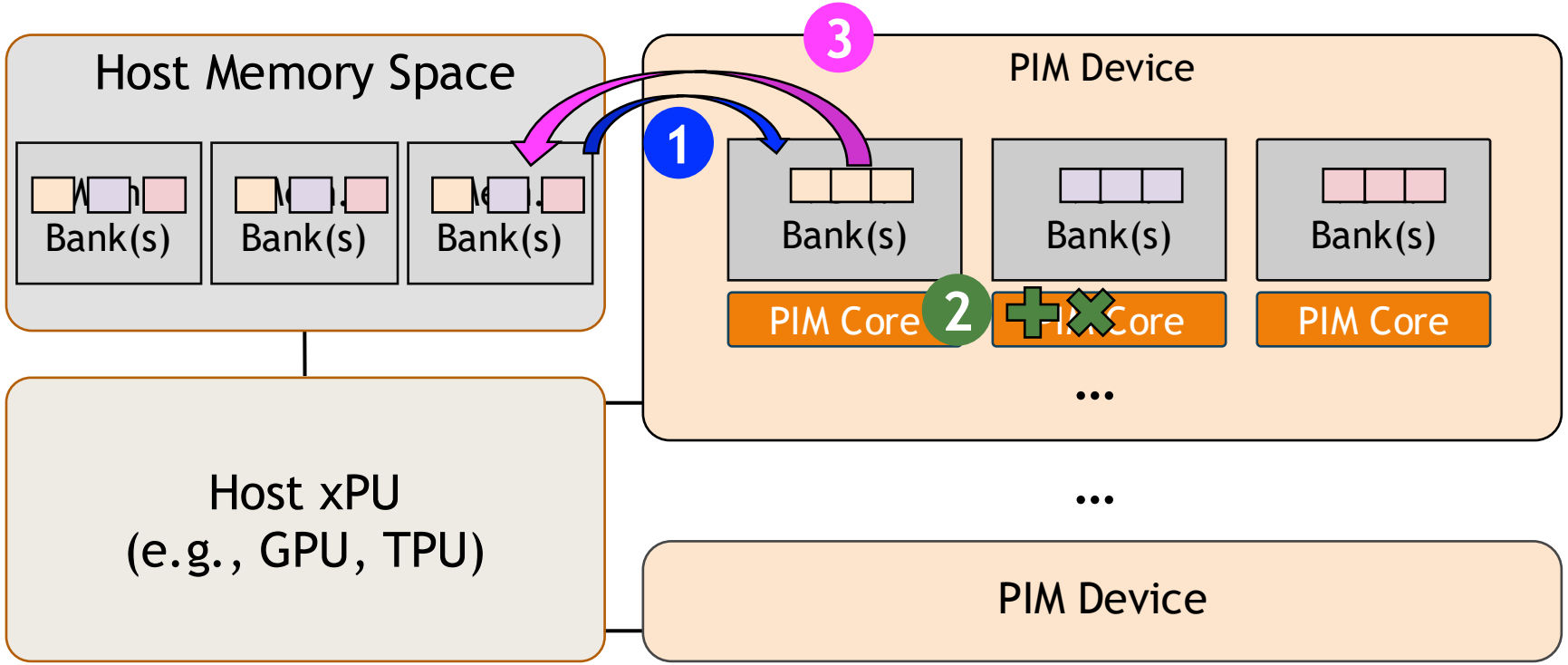


# Three Steps of ML Kernel Execution on PIM Systems

Step 1. Input data rearrangements to place consecutive elements within the same banks for local PIM core processing

Step 2. Computation execution on PIM cores

Step 3. Output data rearrangements to merge partial results from step (2) or prepare output data for Host access



# Talk Outline

- Background
- **Motivation**
- DCC Design
- Evaluation

# Limitations of Existing PIM Compilers (I)

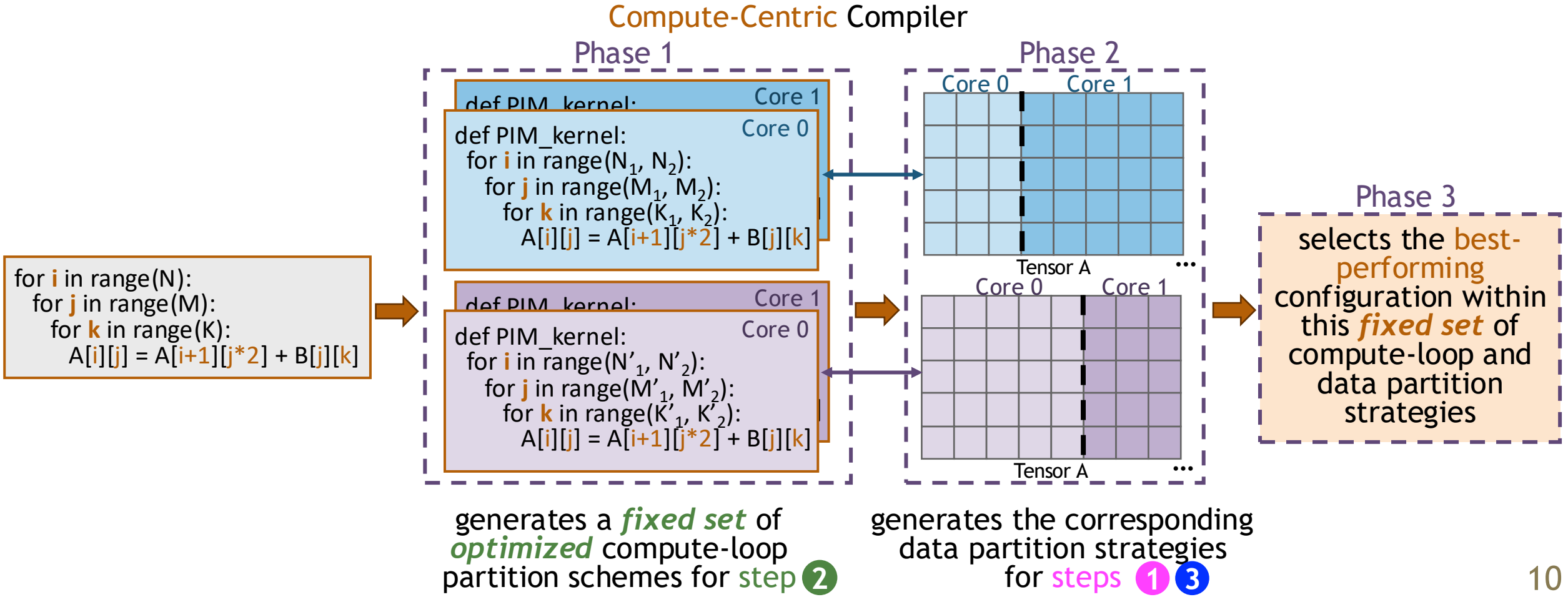
- Limitation 1: Existing compilers support a single PIM backend and/or optimize for a single ML kernel

- PIM-DL [ASPLOS'24] → supports primarily UPMEM PIM backend & only GEMM kernel
- ATiM [ISCA'24] → supports only UPMEM PIM backend
- SimplePIM [PACT'23] → supports only UPMEM PIM backend
- Cinm [ASPLOS'24] → supports primarily UPMEM PIM backend
- PIMFlow [CGO'23] → supports only AiM PIM backend & only convolution kernel

# Limitations of Existing PIM Compilers (II)

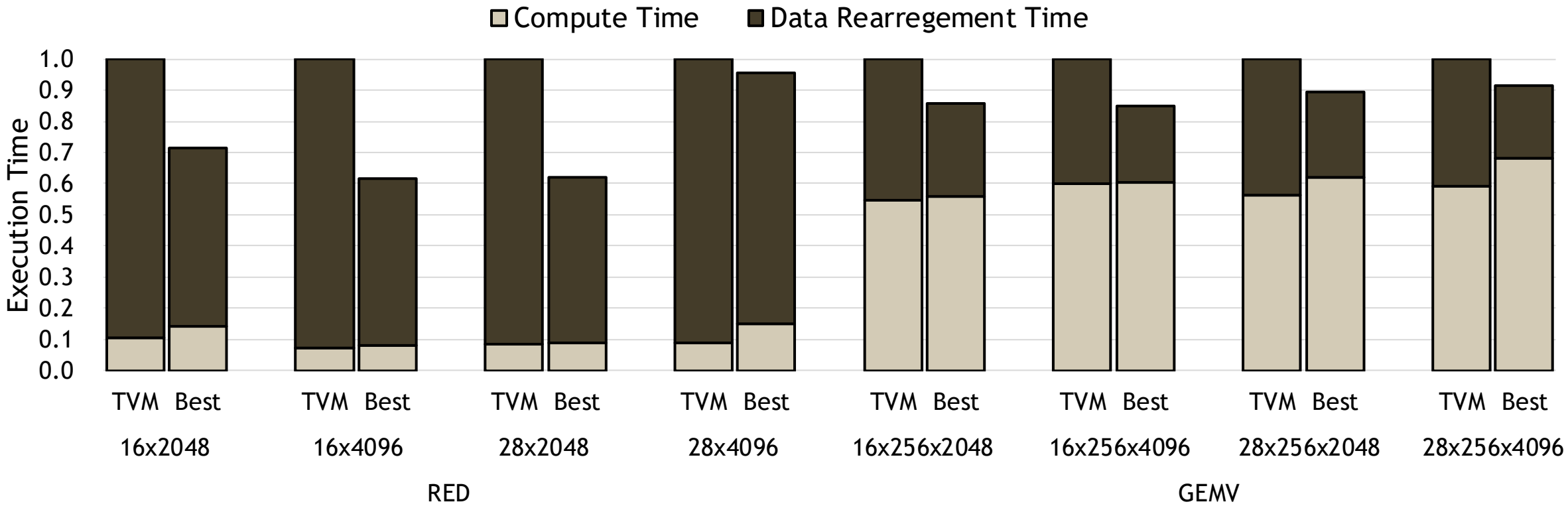
- Limitation 2: Existing compilers follow a compute-centric approach, and largely ignore data rearrangements costs during the compute-loop optimization step

→ Treat data rearrangement costs as secondary, despite contributing significantly to total execution time



# Limitations of Existing PIM Compilers (II)

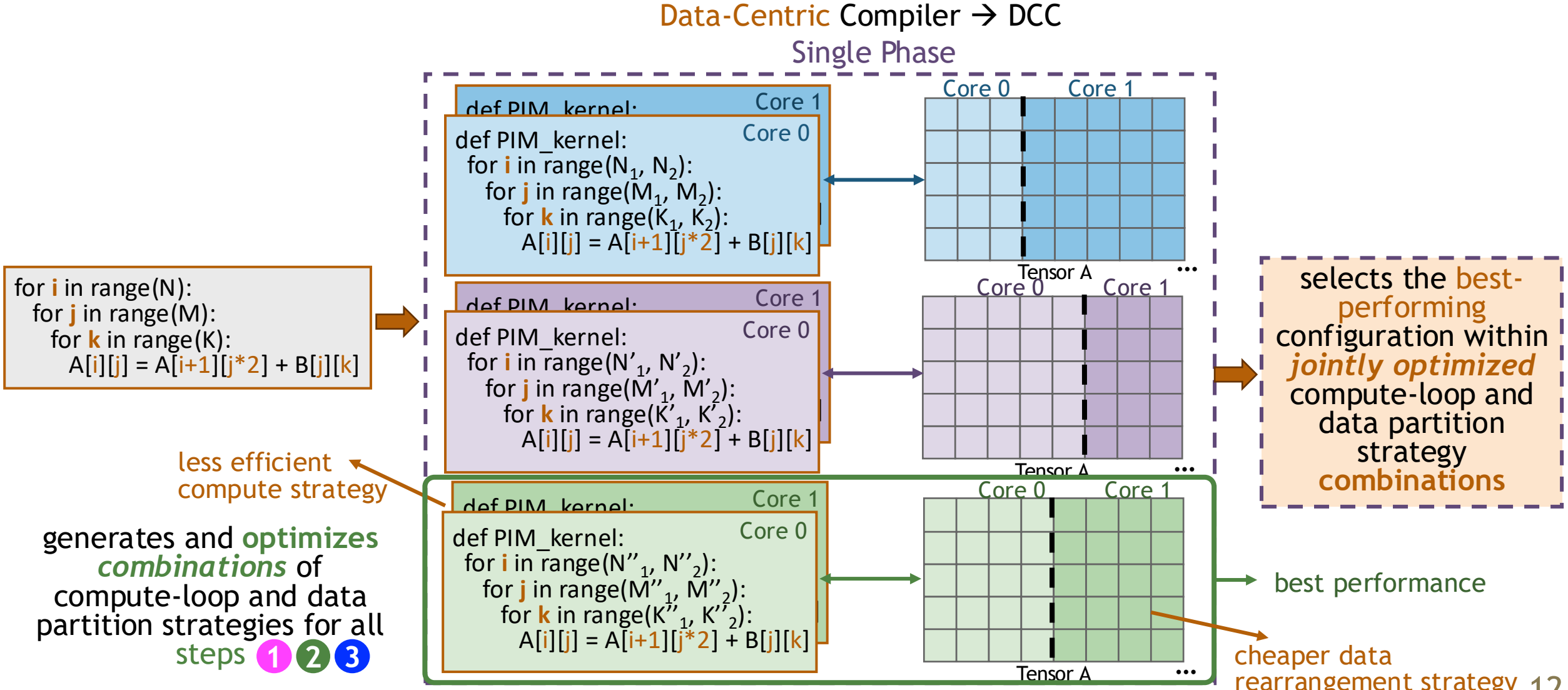
- Performance comparison on a PIM system:
  - Compute-centric TVM-based compiler (TVM) (following ATiM [ISCA'24]) vs
  - Manually-tuned best-performing implementation (Best)




Slightly worse compute time could achieve much better data rearrangement time, thus resulting much better end-to-end kernel performance

# Co-Optimization of Compute-Loop and Data Partitioning is Needed

Data rearrangement and compute code optimization are *interdependent*



# Talk Outline

Background

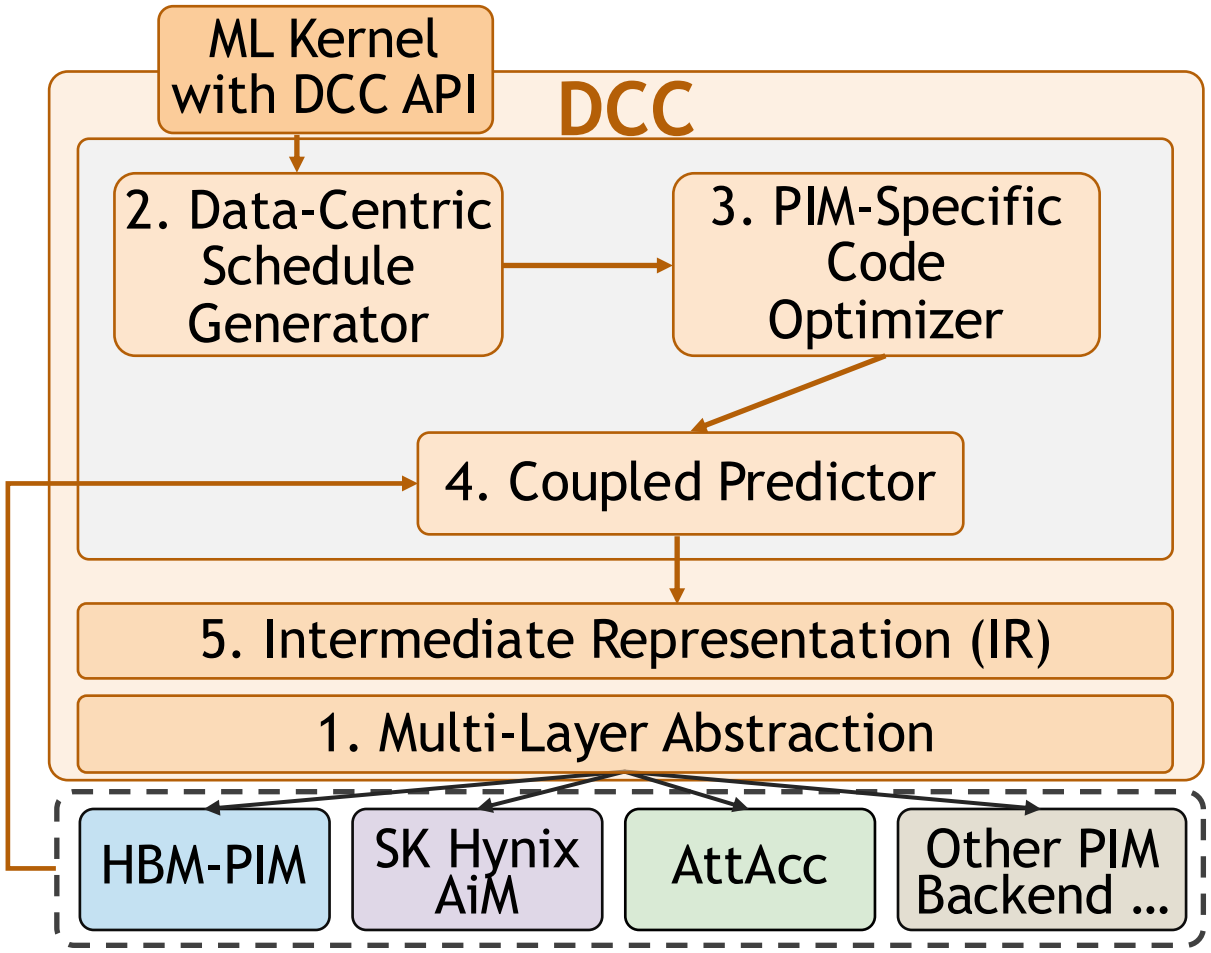
Motivation

**DCC Design**

Evaluation

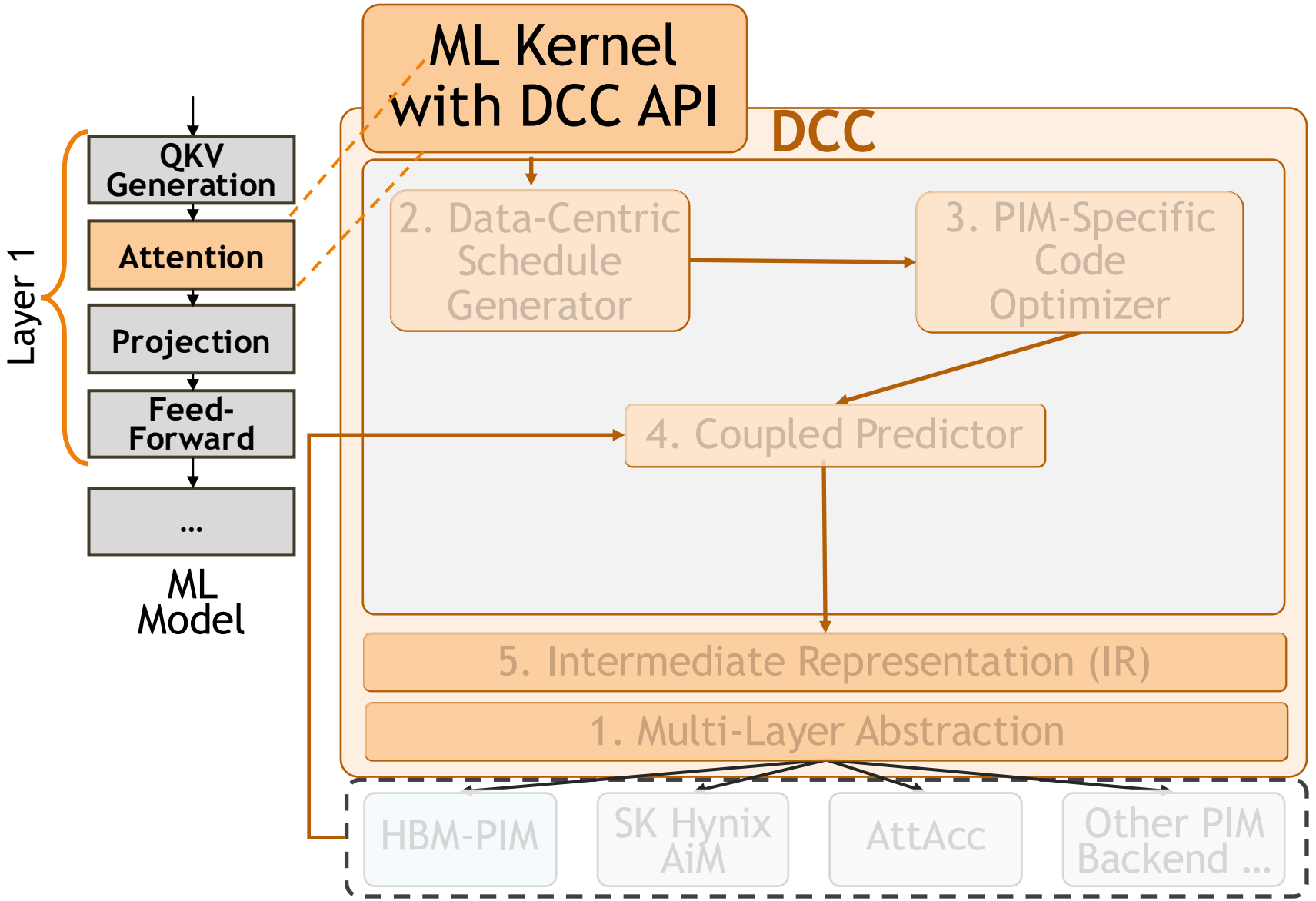
# DCC Overview & 5 Key Components

→The first data-centric ML compiler for PIM systems that jointly co-optimizes compute-loop and data partition strategies into a unified tuning process




- DCC includes 5 key components and:
- ☑ supports multiple PIM backends
- ☑ supports any ML kernel
- ☑ performs PIM-specific optimizations
- ☑ is extensible for backend-specific and backend-agnostic optimizations
- ☑ has low compilation time
- ☑ consistently delivers near-optimal performance

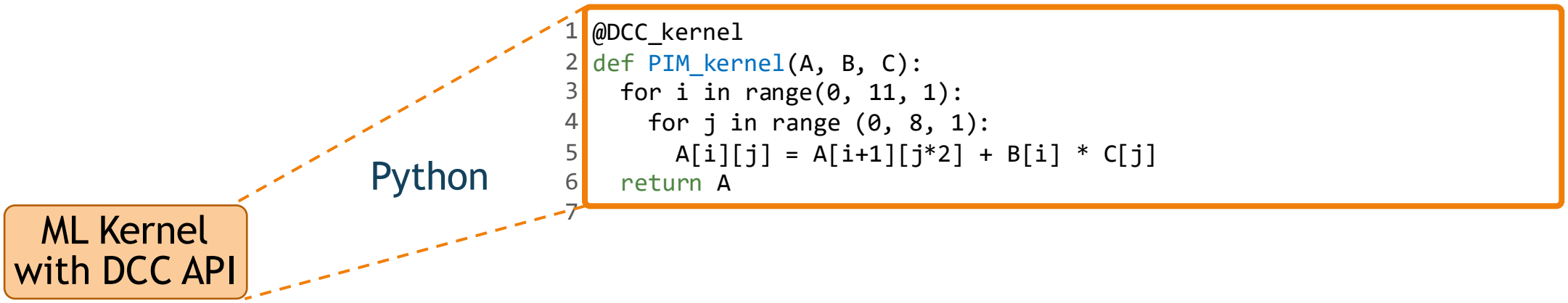
# DCC Overview & 5 Key Components



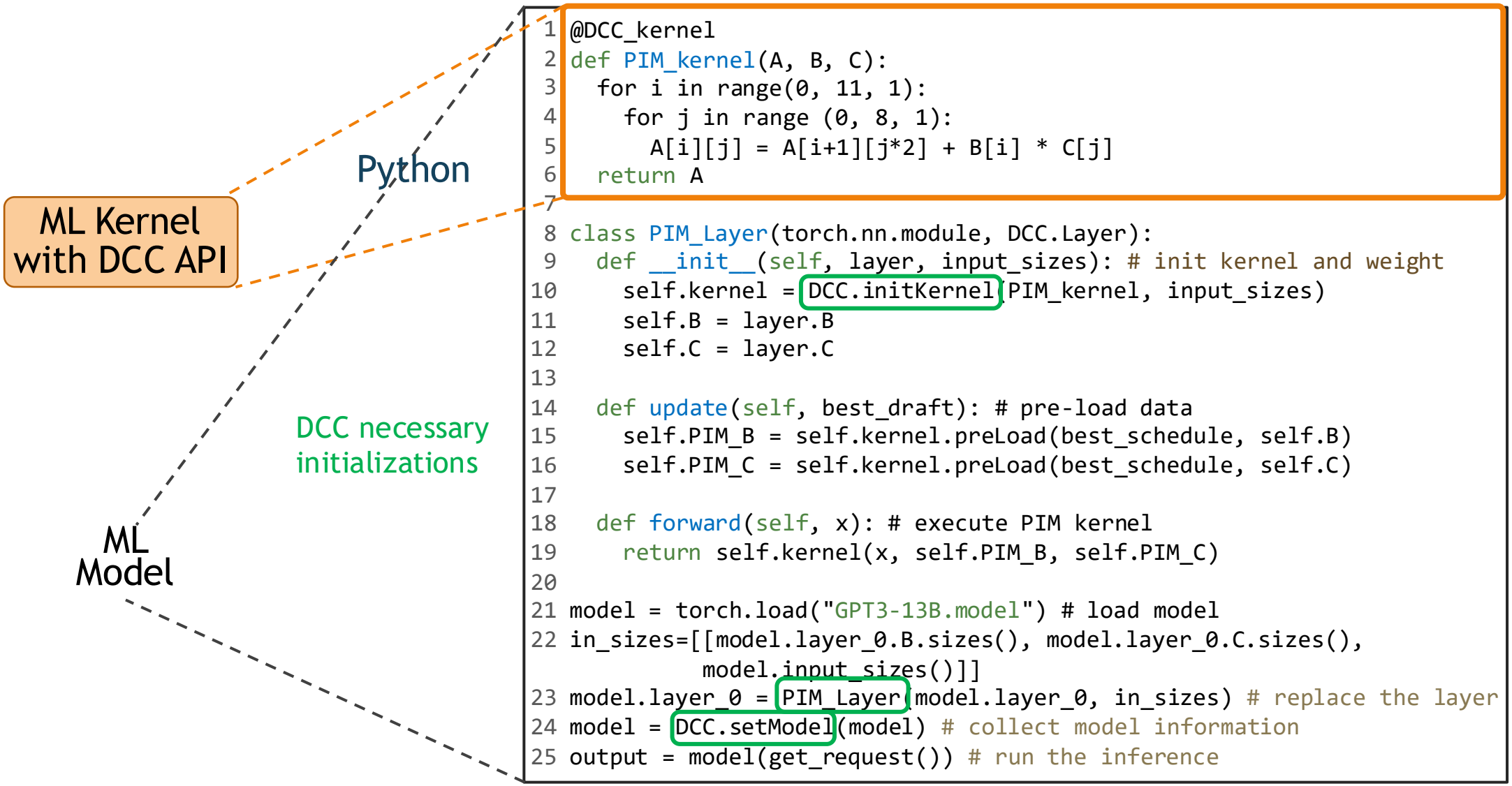

**Goal**

→ Provide a high-level (Python) interface to deploy ML kernels for PIM execution

# DCC Frontend Programming Interface - Python

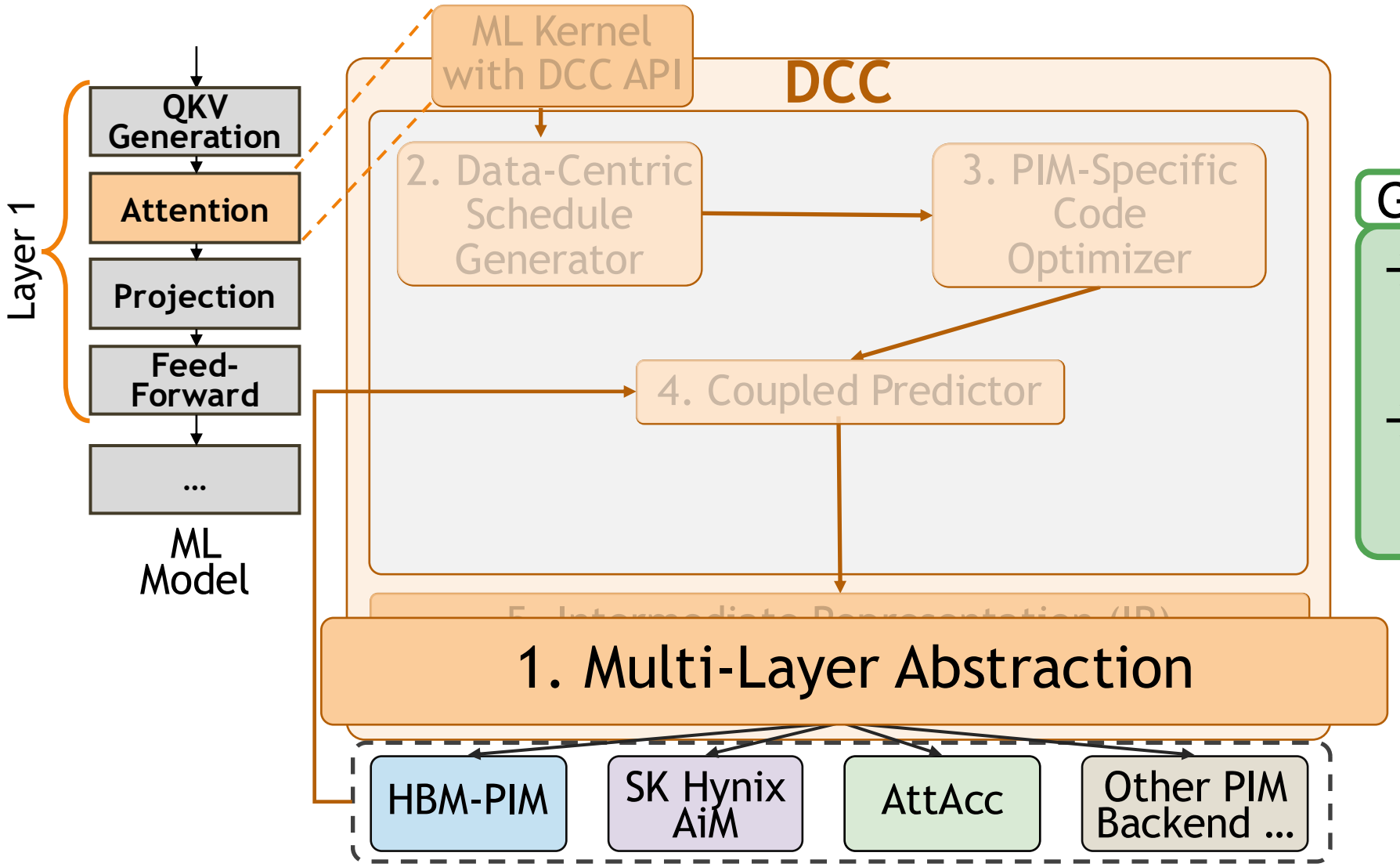


# DCC Frontend Programming Interface - Python




```python
@DCC_kernel
def PIM_kernel(A, B, C):
  for i in range(0, 11, 1):
    for j in range (0, 8, 1):
      A[i][j] = A[i+1][j*2] + B[i] * C[j]
  return A

class PIM_Layer(torch.nn.module, DCC.Layer):
  def __init__(self, layer, input_sizes): # init kernel and weight
    self.kernel = DCC.initKernel(PIM_kernel, input_sizes)
    self.B = layer.B
    self.C = layer.C

  def update(self, best_draft): # pre-load data
    self.PIM_B = self.kernel.preLoad(best_schedule, self.B)
    self.PIM_C = self.kernel.preLoad(best_schedule, self.C)

  def forward(self, x): # execute PIM kernel
    return self.kernel(x, self.PIM_B, self.PIM_C)

model = torch.load("GPT3-13B.model") # load model
in_sizes=[[model.layer_0.B.sizes(), model.layer_0.C.sizes(),
          model.input_sizes()]]
model.layer_0 = PIM_Layer(model.layer_0, in_sizes) # replace the layer
model = DCC.setModel(model) # collect model information
output = model(get_request()) # run the inference
```

# DCC Overview & 5 Key Components




**Goal**

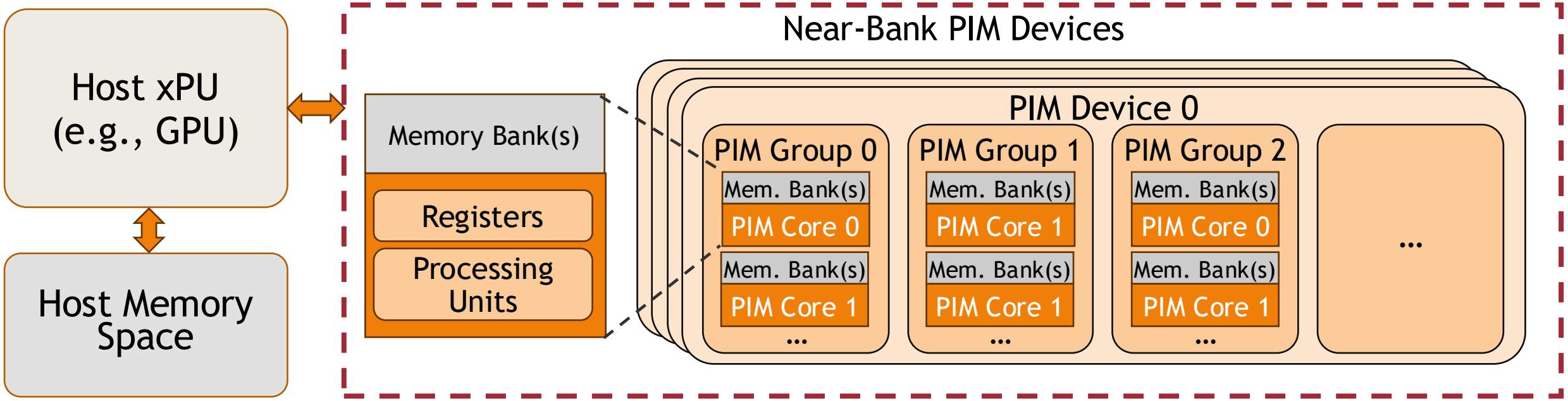
→ Enable support for multiple current and future PIM backends

→ Enable coarse-grained and fine-grained performance optimizations

# 1. Multi-Layer Abstraction for Near-Bank PIM Systems

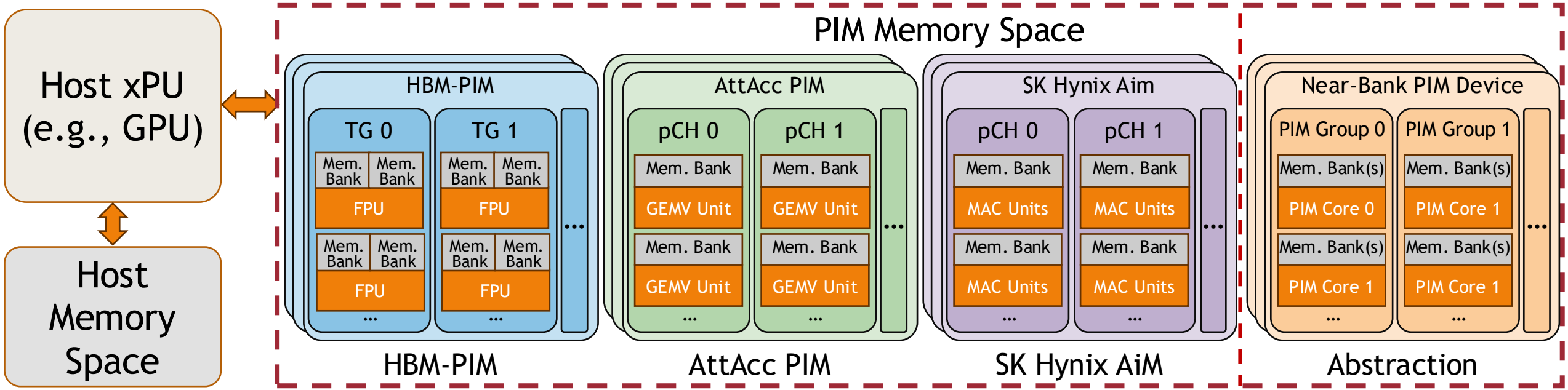
We map PIM memory hierarchy into a compute hierarchy of 3 levels:

a) System level: Host xPU, Host memory spaces and multiple PIM devices
b) PIM group level: multiple PIM cores (and their local memory banks) belonging to the same memory channel or rank
c) PIM core level: the processing unit (core) and its local memory banks

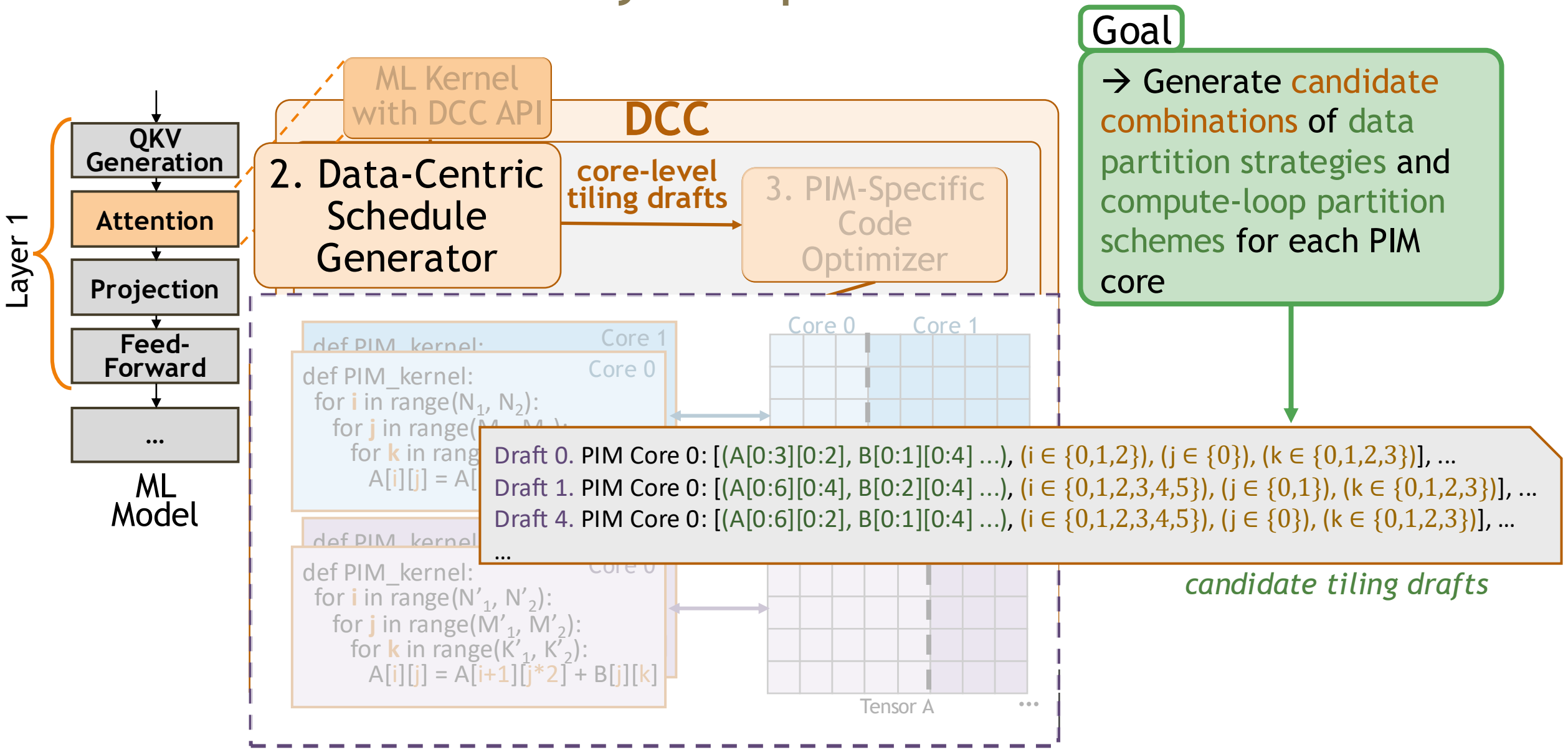


# Three Example PIM Backends into DCC's Abstraction

- ✓ **HBM-PIM:** Lee et al., "Hardware architecture and software stack for PIM based on commercial DRAM technology", ISCA 2021 - Samsung
- ✓ **AttAcc PIM:** Lee et al., "AttAcc! Unleashing the Power of PIM for Batched Transformer-based Generative Model Inference", ASPLOS 2024
- ✓ **SK Hynix AiM:** Lee et al., "Newton: A DRAM-Maker's Accelerator-in-Memory (AiM) Architecture for Machine Learning", MICRO 2020 – SK Hynix



# DCC Overview & 5 Key Components



# 2. Data-Centric Schedule Generator

**Goal**

→ Generate **candidate** data partition strategies, and derive corresponding compute-loop partition schemes for each PIM core

**2A** Dimension Set Construction

Associates *data partitions* with *compute-loop partitions*

**2B** Data-Tile Construction

Generates all candidate data partitions across available PIM resources

**2C** Draft Pruning

Prunes unpromising data partition candidates to reduce tuning time

**2D** Data-Tile to Compute-Tile Mappings

Constructs core-level data partitions and maps them to their corresponding compute-loop partitions on each PIM core

*candidate tiling drafts*

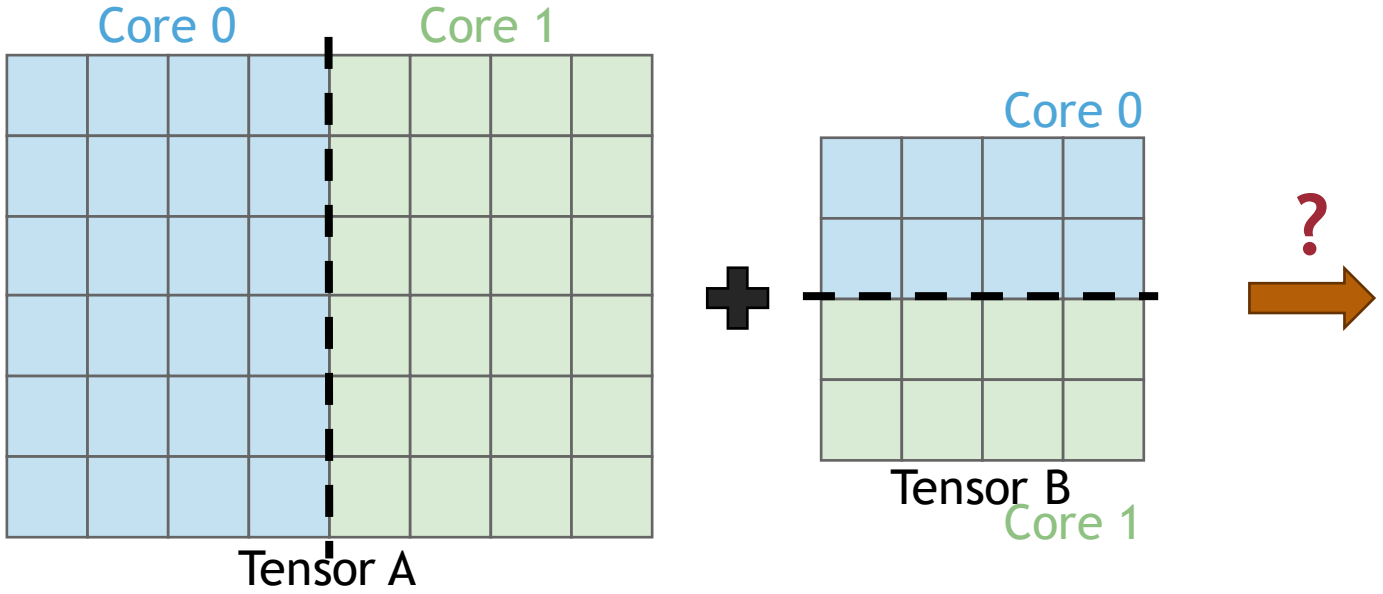
Draft 0. PIM Core 0: [(A[0:3][0:2], B[0:1][0:4] ...), (i ∈ {0,1,2}), (j ∈ {0}), (k ∈ {0,1,2,3})], ...
Draft 1. PIM Core 0: [(A[0:6][0:4], B[0:2][0:4] ...), (i ∈ {0,1,2,3,4,5}), (j ∈ {0,1}), (k ∈ {0,1,2,3})], ...
Draft 4. PIM Core 0: [(A[0:6][0:2], B[0:1][0:4] ...), (i ∈ {0,1,2,3,4,5}), (j ∈ {0}), (k ∈ {0,1,2,3})], ...
...

# Challenge in Data-Centric Compilation Approach

How to associate data partitions to compute-loop partitions?

```
def PIM_kernel (A[6][8], B[4][4]):
  for i in range(0, 5):
    for j in range(0, 4):
      for k in range(0, 4):
        A[i][j] = A[i+1][j*2] + B[j][k]
```

data partitions




compute-loop partitions

Core 0

```
def PIM_kernel:
  for i in range(?, ?):
    for j in range(?, ?):
      for k in range(?, ?):
        A[i][j] = A[i+1][j*2] + B[j][k]
```

Core 1

```
def PIM_kernel:
  for i in range(?, ?):
    for j in range(?, ?):
      for k in range(?, ?):
        A[i][j] = A[i+1][j*2] + B[j][k]
```

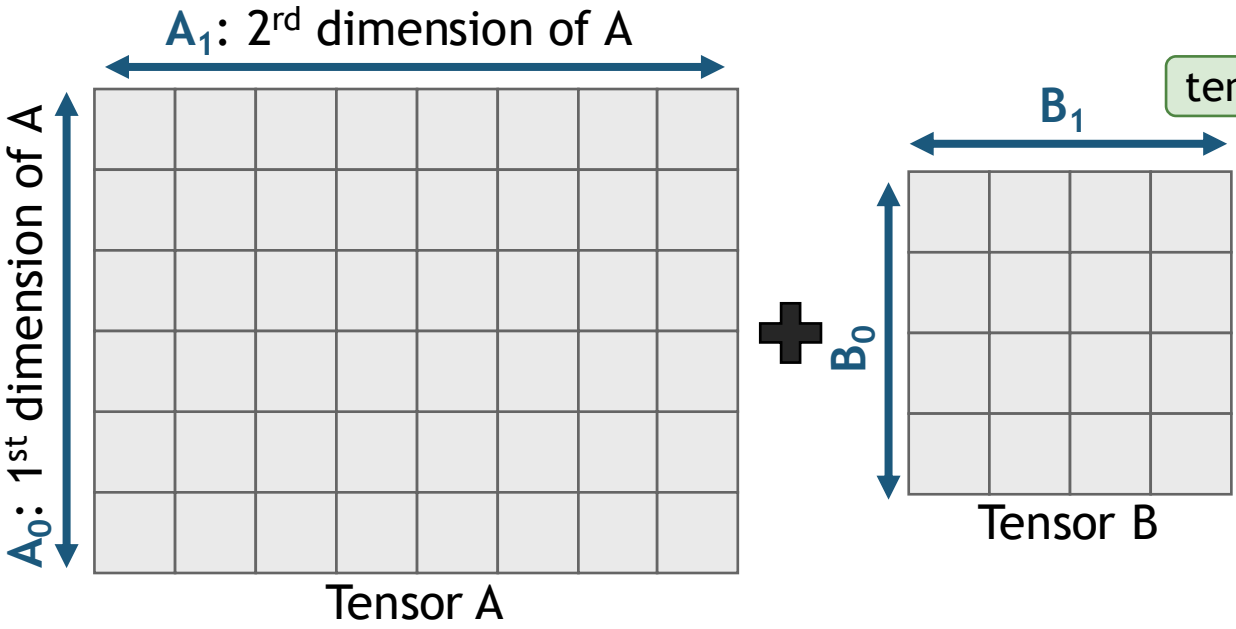
# 2A. Dimension Set Construction

→ associates data partitions with loop partitions

- Dimension Set: a set of tensor dimensions indexed by the same loop, each through its own mapping function

```
def PIM_kernel (A[6][8], B[4][4]):
  for i in range(0, 5):
    for j in range(0, 4):
      for k in range(0, 4):
        A[i][j] = A[i+1][j*2] + B[j][k]
```

3 loop variables




3 dimension sets
(one for each loop variable)

tensor dimension

corresponding mapping function

| | |
|---|---|
| $A_0 \rightarrow F(i) = i$<br>$A_0 \rightarrow F(i) = i+1$ | i loop variable |
| $A_1 \rightarrow F(j) = j$<br>$A_1 \rightarrow F(j) = j*2$<br>$B_0 \rightarrow F(j) = j$ | j loop variable |
| $B_1 \rightarrow F(k) = k$ | k loop variable |

# 2B. Data-Tile Construction

→ generates all candidate data partitions across available PIM resources

- DCC performs an exhaustive search to generate all data partitions for each dimension set

```
def PIM_kernel (A[6][8], B[4][4]):
  for i in range(0, 5):
    for j in range(0, 4):
      for k in range(0, 4):
        A[i][j] = A[i+1][j*2] + B[j][k]
```

3 dimension sets (one for each loop variable)

3 dimension sets
(one for each loop variable)

$A_0$ → F(i) = i [repr.]
$A_0$ → F(i) = i+1

$A_0$ → F(i) = i
$A_0$ → F(i) = i+1 [repr.]

2x candidate representative mapping functions($A_0^i$, $A_0^{i+1}$)

$A_1$ → F(j) = j [repr.]
$A_1$ → F(j) = j*2
$B_0$ → F(j) = j

$A_0$ → F(i) = i
$A_0$ → F(i) = i+1

$A_1$ → F(j) = j
$A_1$ → F(j) = j*2 [repr.]
$B_0$ → F(j) = j

$A_1$ → F(j) = j
$A_1$ → F(j) = j*2
$B_0$ → F(j) = j [repr.]

$B_1$ → F(k) = k

3x

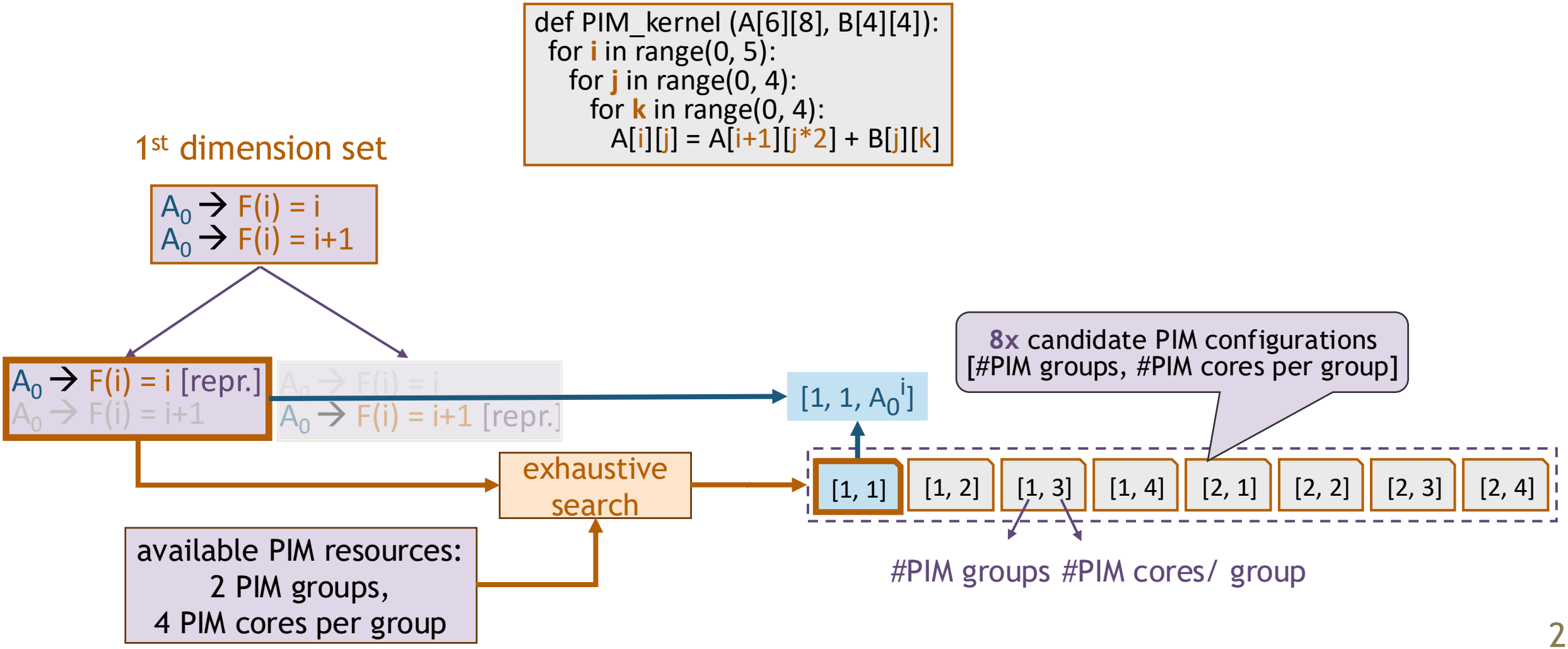
$B_1$ → F(k) = k [repr.]

1x

# 2B. Data-Tile Construction

→ generates all candidate data partitions across available PIM resources

- DCC performs an exhaustive search to generate all data partitions for each dimension set



# 2B. Data-Tile Construction

→ generates all candidate data partitions across available PIM resources

- DCC performs an exhaustive search to generate all data partitions for each dimension set

```
def PIM_kernel (A[6][8], B[4][4]):
  for i in range(0, 5):
    for j in range(0, 4):
      for k in range(0, 4):
        A[i][j] = A[i+1][j*2] + B[j][k]
```

available PIM resources:
2 PIM groups,
4 PIM cores per group

data-tile drafts

| | | | |
|---|---|---|---|
| Draft 0. | [[1, 1, $A_0^i$], | [1, 1, $A_1^j$], | [1, 1, $B_1^k$]] |
| Draft 1. | [[1, 1, $A_0^i$], | [1, 2, $A_1^j$], | [1, 1, $B_1^k$]] |
| Draft 2. | [[1, 1, $A_0^i$], | [1, 2, $B_0^j$], | [1, 1, $B_1^k$]] |
| Draft 3. | [[1, 1, $A_0^i$], | [1, 2, $A_1^{j*2}$], | [1, 2, $B_1^k$]] |
| Draft 4. | [[1, 1, $A_0^i$], | [1, 4, $A_1^{j*2}$], | [1, 1, $B_1^k$]] |
| Draft 5. | [[1, 2, $A_0^i$], | [1, 1, $A_1^j$], | [1, 1, $B_1^k$]] |
| … | | | |
| Draft 3070. | [[2, 4, $A_0^{i+1}$], | [2, 4, $B_0^j$], | [2, 3, $B_1^k$]] |
| Draft 3071. | [[2, 4, $A_0^{i+1}$], | [2, 4, $B_0^j$], | [2, 4, $B_1^k$]] |

(2x8) x (3x8) x (1x8)
= 3072 drafts

1st dimension set:
1 PIM group
1 PIM core per group
repr. mapping function: $A_0$ → F(i) = i

2nd dimension set:
1 PIM group
1 PIM core per group
repr. mapping function: $A_1$ → F(j) = j

3rd dimension set:
1 PIM group
1 PIM core per group
repr. mapping function: $B_1$ → F(k) = k

# 2C. Draft Pruning

→ prunes unpromising data-tile drafts to reduce tuning time

- DCC applies 3 pruning rules to significantly reduce the search space

```
def PIM_kernel (A[6][8], B[4][4]):
  for i in range(0, 5):
    for j in range(0, 4):
      for k in range(0, 4):
        A[i][j] = A[i+1][j*2] + B[j][k]
```

available PIM resources:
2 PIM groups,
4 PIM cores per group

typically $10^3$ – $10^6$ drafts

data-tile drafts

| | | | |
|---|---|---|---|
| Draft 0. | [[1, 1, $A_0^i$], | [1, 1, $A_1^j$], | [1, 1, $B_1^k$]] |
| Draft 1. | [[1, 1, $A_0^i$], | [1, 2, $A_1^j$], | [1, 1, $B_1^k$]] |
| Draft 2. | [[1, 1, $A_0^i$], | [1, 2, $B_0^j$], | [1, 1, $B_1^k$]] |
| Draft 3. | [[1, 1, $A_0^i$], | [1, 2, $A_1^{j*2}$], | [1, 2, $B_1^k$]] |
| Draft 4. | [[1, 1, $A_0^i$], | [1, 4, $A_1^{j*2}$], | [1, 1, $B_1^k$]] |
| Draft 5. | [[1, 2, $A_0^i$], | [1, 1, $A_1^j$], | [1, 1, $B_1^k$]] |
| … | | | |
| Draft 3070. | [[2, 4, $A_0^{i+1}$], | [2, 4, $B_0^j$], | [2, 3, $B_1^k$]] |
| Draft 3071. | [[2, 4, $A_0^{i+1}$], | [2, 4, $B_0^j$], | [2, 4, $B_1^k$]] |

pruning

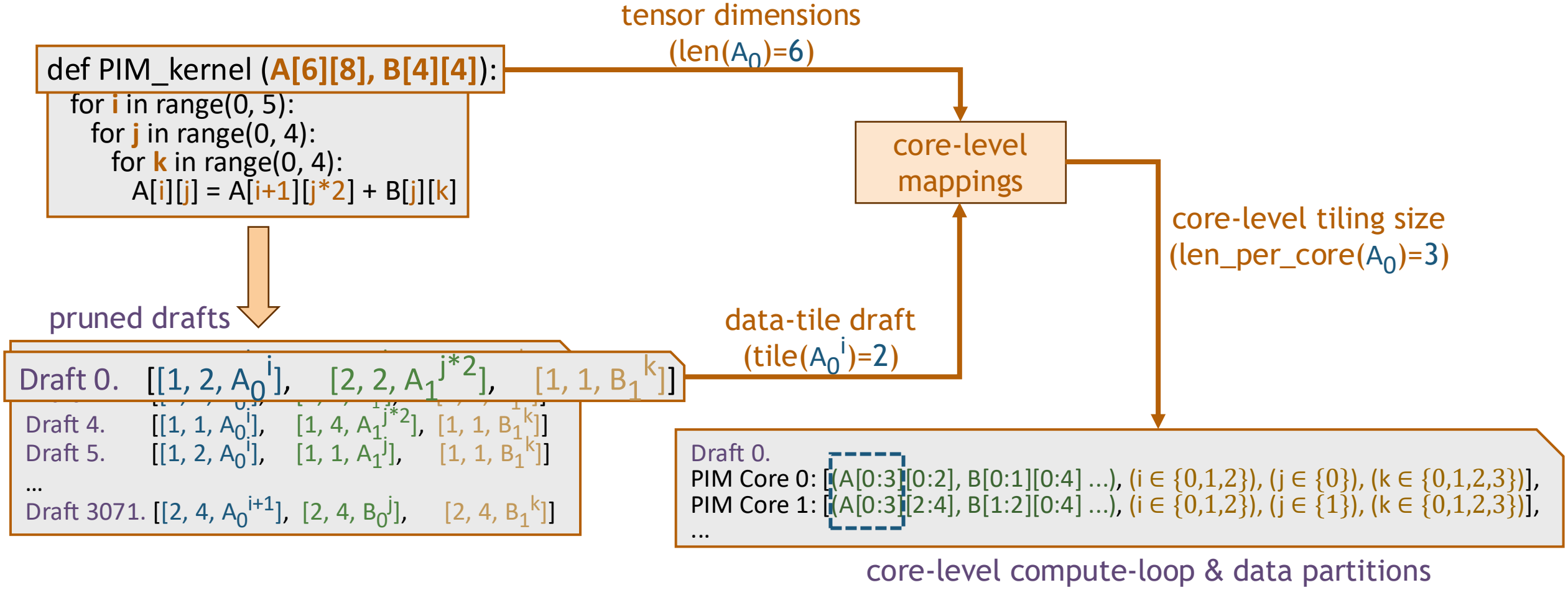
pruned by ~95%
(only ~5% retained)

pruned drafts

| | | | |
|---|---|---|---|
| Draft 0. | [[1, 1, $A_0^i$], | [1, 1, $A_1^j$], | [1, 1, $B_1^k$]] |
| Draft 1. | [[1, 1, $A_0^i$], | [1, 2, $A_1^j$], | [1, 1, $B_1^k$]] |
| Draft 4. | [[1, 1, $A_0^i$], | [1, 4, $A_1^{j*2}$], | [1, 1, $B_1^k$]] |
| Draft 5. | [[1, 2, $A_0^i$], | [1, 1, $A_1^j$], | [1, 1, $B_1^k$]] |
| … | | | |
| Draft 3071. | [[2, 4, $A_0^{i+1}$], | [2, 4, $B_0^j$], | [2, 4, $B_1^k$]] |

# 2D. Data-Tile to Compute-Tile Mappings

→ constructs compute-loop partitions and data partitions for each PIM core for all drafts



# 2D. Data-Tile to Compute-Tile Mappings

→ constructs compute-loop partitions and data partitions for each PIM core for all drafts

```
def PIM_kernel (A[6][8], B[4][4]):
  for i in range(0, 5):
    for j in range(0, 4):
      for k in range(0, 4):
        A[i][j] = A[i+1][j*2] + B[j][k]
```

pruned drafts

Draft 0. [[1, 1, $A_0^i$], [1, 1, $A_1^j$], [1, 1, $B_1^k$]]
Draft 1. [[1, 1, $A_0^i$], [1, 2, $A_1^j$], [1, 1, $B_1^k$]]
Draft 4. [[1, 1, $A_0^i$], [1, 4, $A_1^{j*2}$], [1, 1, $B_1^k$]]
Draft 5. [[1, 2, $A_0^i$], [1, 1, $A_1^j$], [1, 1, $B_1^k$]]
...
Draft 3071. [[2, 4, $A_0^{i+1}$], [2, 4, $B_0^j$], [2, 4, $B_1^k$]]

core-level tiling drafts

Draft 0.
PIM Core 0: [(A[0:3][0:2], B[0:1][0:4] ...), (i ∈ {0,1,2}), (j ∈ {0}), (k ∈ {0,1,2,3})],
PIM Core 1: ...

Draft 1.
PIM Core 0: [(A[0:6][0:4], B[0:2][0:4] ...), (i ∈ {0,1,2,3,4,5}), (j ∈ {0,1}), (k ∈ {0,1,2,3})],
PIM Core 1: ...

Draft 4.
PIM Core 0: [(A[0:6][0:2], B[0:1][0:4] ...), (i ∈ {0,1,2,3,4,5}), (j ∈ {0}), (k ∈ {0,1,2,3})],
PIM Core 1: ...

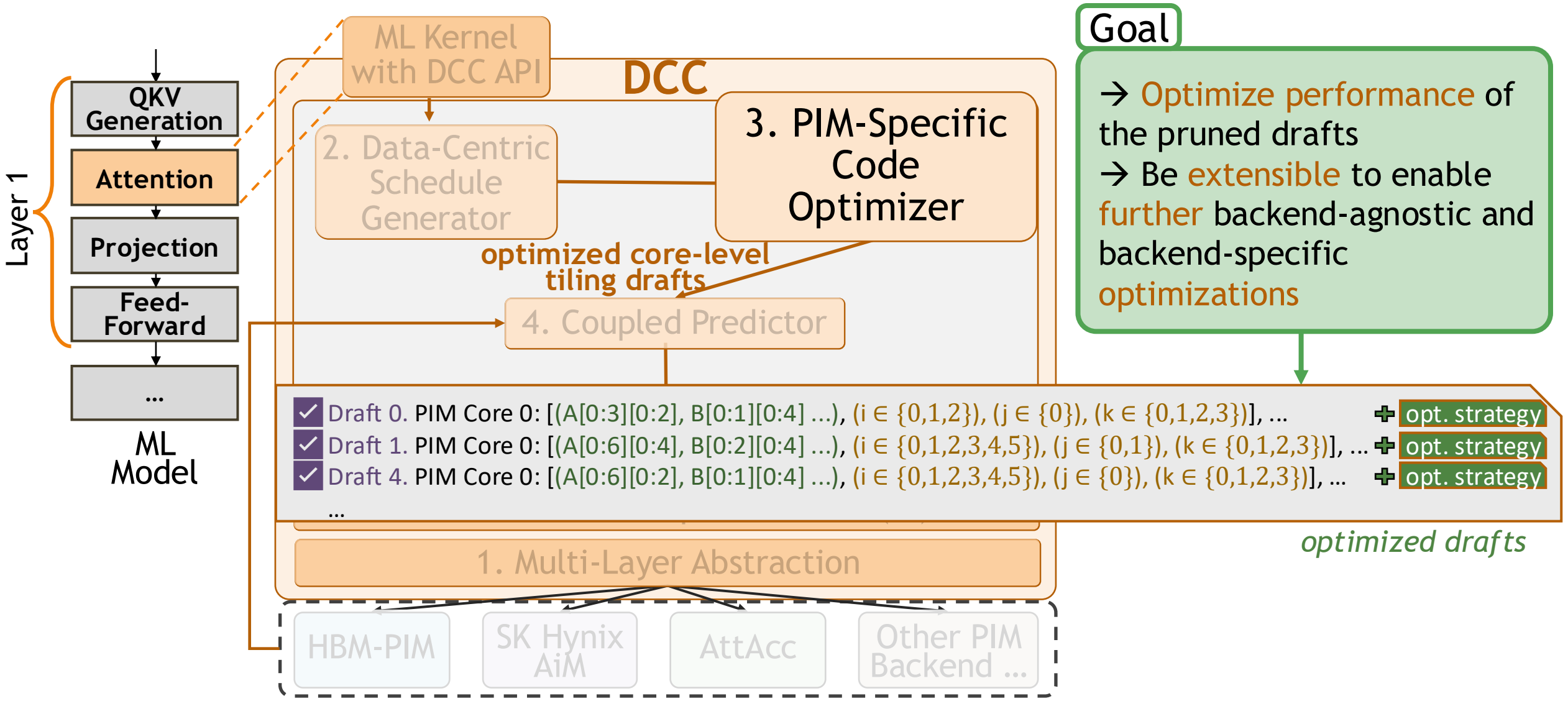
Draft 5.
PIM Core 0: [(A[0:3][0:8], B[0:4][0:4] ...), (i ∈ {0,1,2}), (j ∈ {0,1,2,3}), (k ∈ {0,1,2,3})],
PIM Core 1: ...

...

Draft 3071.
PIM Core 0: [(A[0:1][0:1], B[0:1][0:1] ...), (i ∈ {0}), (j ∈ {0}), (k ∈ {0})],
PIM Core 1: ...

core-level compute-loop partitions and data partitions for all dimension sets of **all drafts**

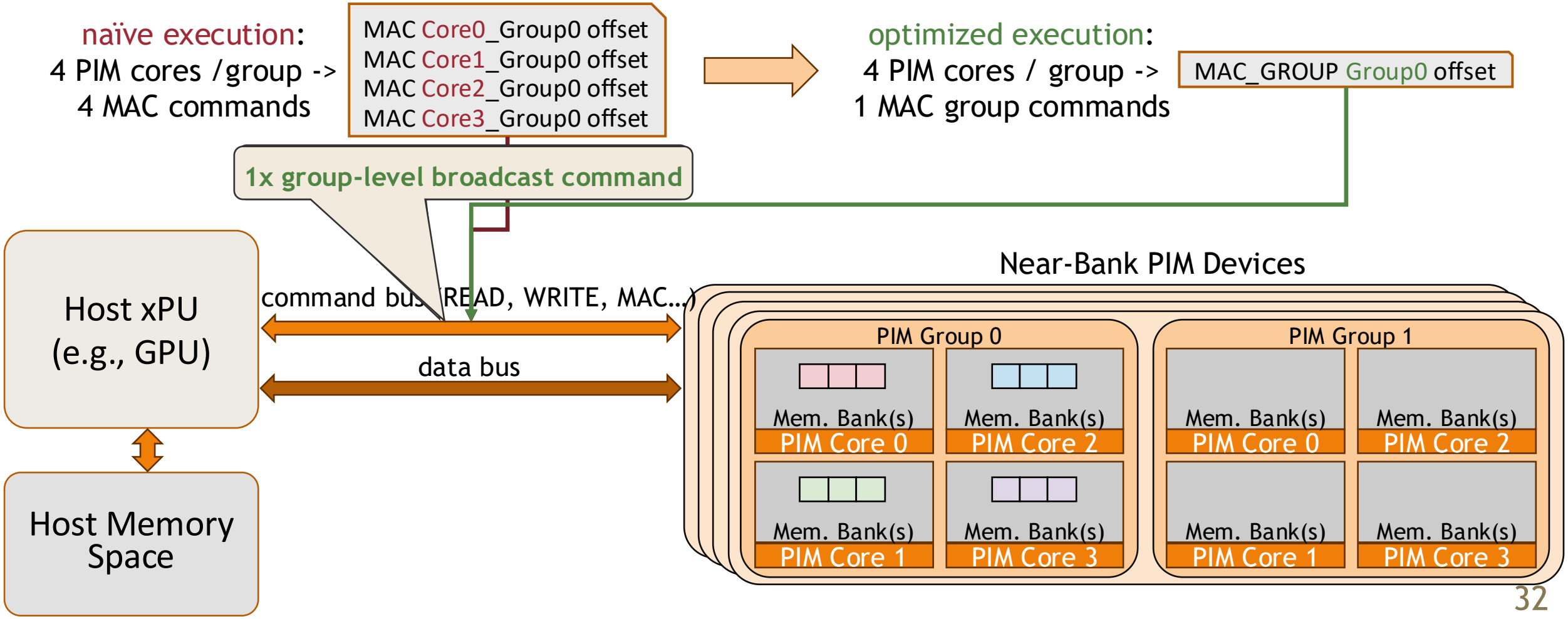
# DCC Overview & 5 Key Components



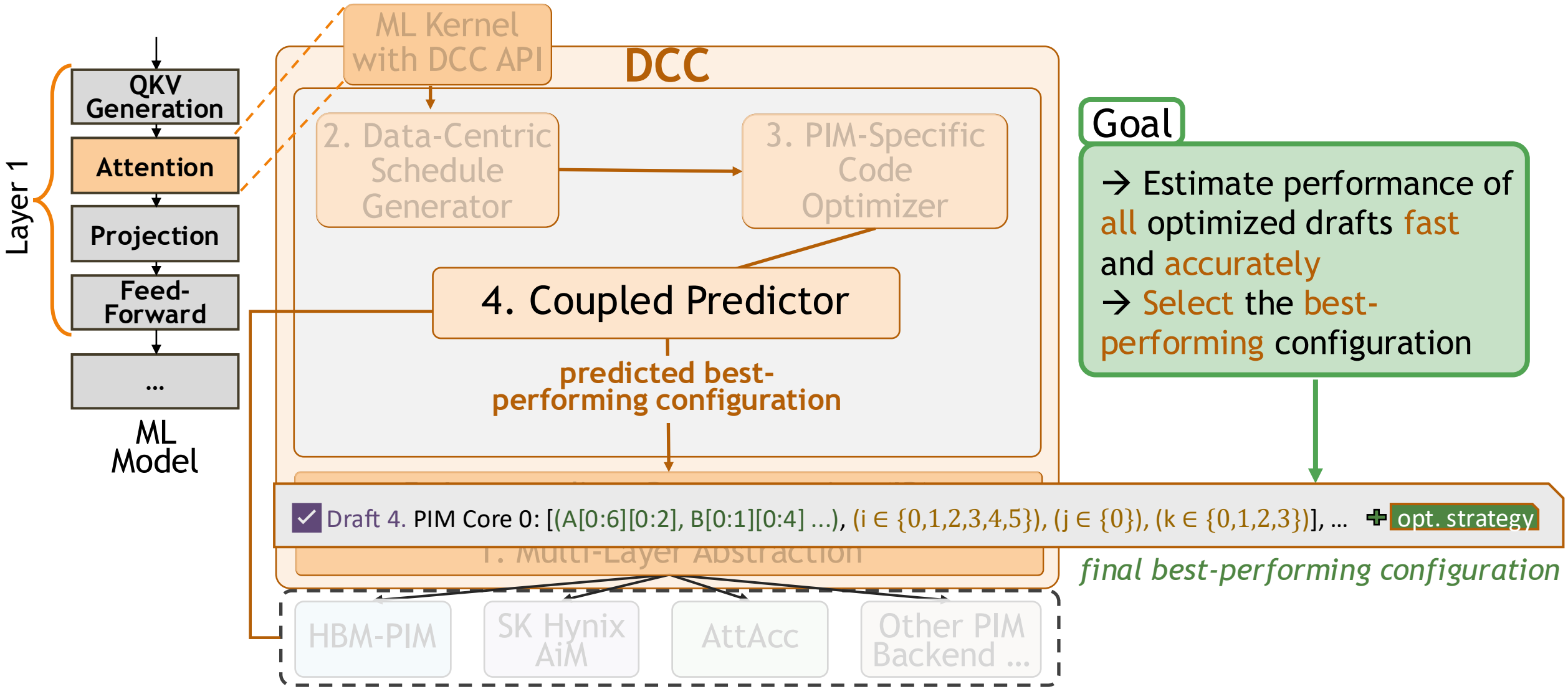
# 3. PIM-Specific Code Optimizer

→ optimizes performance of drafts & is extensible for further optimizations

- Example: group-level broadcast commands in DCC



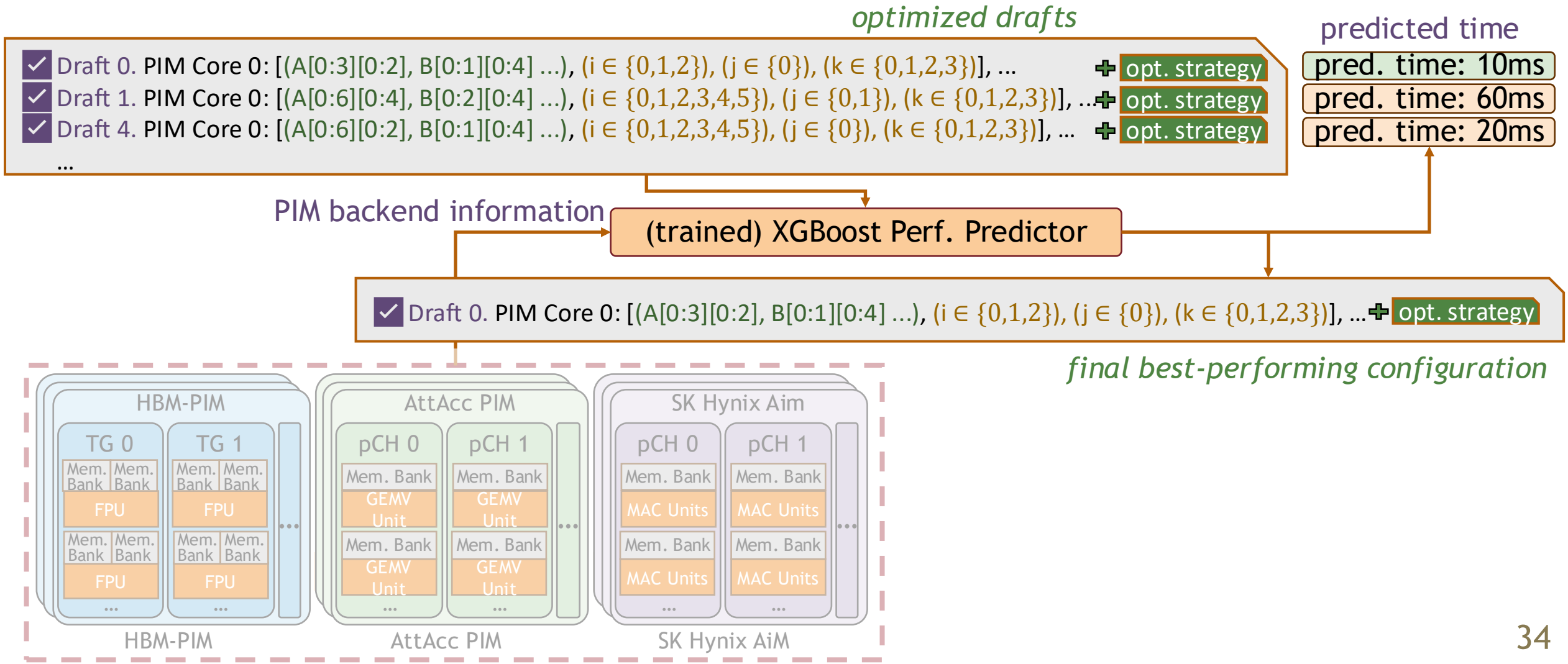
# DCC Overview & 5 Key Components



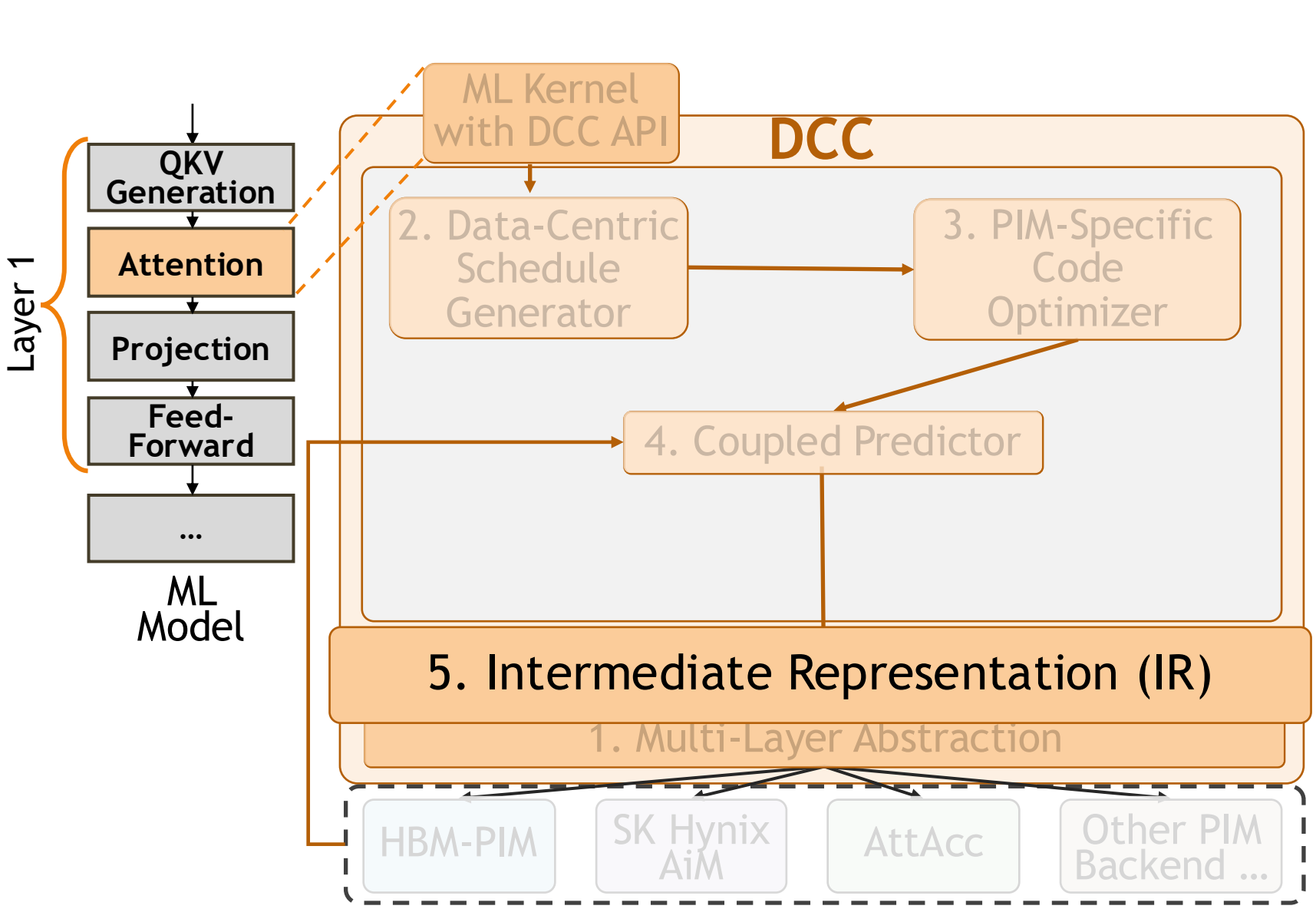
# 4. Coupled Predictor

→ accurately estimates performance of drafts & selects the best-performing configuration

- DCC uses a learning-based performance predictor to enable fast & accurate predictions



# DCC Overview & 5 Key Components




**Goal**

→ Capture optimized data partition strategies and compute-loop partition schemes

→ Enable support for multiple current & future PIM backends

```
%a_core[%i,%j] =
%a_core[%i+1,%j*2] + %b_core[%j][%k]
…
```

*DCC IR*

```
%reg_a=load %a_core[%i+1,%j*2]
%reg_sum=ADD %b_core[%j][%k],%reg_a
…
```

*backend PIM instructions*

# 5. Intermediate Representation (IR)

→ captures data and compute-loop partition strategies & enables multi-PIM-backend support

```
@DCC_kernel
def example(A, B, C):
  for i in range(0, 5):
    for j in range(0, 4):
      for k in range(0, 4):
        A[i][j] = A[i+1][j*2] + B[j][k]
  return A
```

2. Data-Centric Schedule Generator

3. PIM-Specific Code Optimizer

4. Coupled Predictor

*best-performing config.*

```
1  @hardware_platform[GROUPS=2, CORES=4]
2  func @example(%a:<6x8xfp32>, %b:<4x4xfp32>): <6x8xfp32>
3    complex_dimset %ds_i {
4      %dim[0] = %a.dim[0] -> i,
5      %dim[1] = %a.dim[0] -> i+1 [representative] }
6    %ds_i.range = (0, 5, 1)
7    complex_dimset %ds_j {
8      %dim[0] = %a.dim[1] -> j,
9      %dim[1] = %a.dim[1] -> j*2 [representative],
10     %dim[2] = %c.dim[0] -> j }
11   %ds_j.range = (0, 4, 1)
12   complex_dimset %ds_k {
13     %dim[0] = %b.dim[1] -> k }
14   %ds_k.range = (0, 4, 1)
15   data_tile %dt_i: (1,2) -> (1,2,3)
16   data_tile %dt_j: (2,2) -> (2,2,1)
17   data_tile %dt_k: (1,1) -> (1,1,4)
18   compute_tile %ct_i: (1,2,6) -> [[{0,1,2}, {3,4,5}]]
19   compute_tile %ct_j: (2,1,4) -> [[{0}], [{1}], [{2}], [{3}]]
20   compute_tile %ct_k: (1,1,4) -> [[{0,1,2,3}]]
21   tensor_indices %tt_i: %ct_i -> [[{0,1,2,3}, {3,4,5,6}]]
22   tensor_indices %tt_j: %ct_j -> [[{0}], [{1,2}], [{2,4}], [{3,6}]]
23   tensor_indices %tt_k: %ct_k -> [[{0,1,2,3}]]
24   %a_core = rearrange %a -> %tt_i, %tt_j [to_PIM_core]
25   %b_core = rearrange %b -> %tt_i, %tt_k [to_PIM_core]
26   parallel_PIM (GROUPS, CORES) {
27     loop (i, j) in (%ct_i, %ct_j, %ct_k) {
28   %a_core[%i, %j] = %a_core[%i+1, %j*2] + %b_core[%j][%k]
29   }}
30   %a_out = rearrange %a_core -> %a [to_host]
31   ret %a_out
```

DCC IR

*backend PIM instructions*

AttAcc PIM Instructions

```
%reg_a = load %a_core[%i+1, %j*2]
%reg_b = load %b_core[%j][%k]
%reg_sum = ADD %reg_b, %reg_a
store %reg_sum -> %a_core[%i, %j]
```

HBM-PIM Instructions

```
%reg_a = load %a_core[%i+1, %j*2]
%reg_sum = ADD %b_core[%j][%k], %reg_a
store %reg_sum -> %a_core[%i, %j]
```

# Talk Outline

Background

Motivation

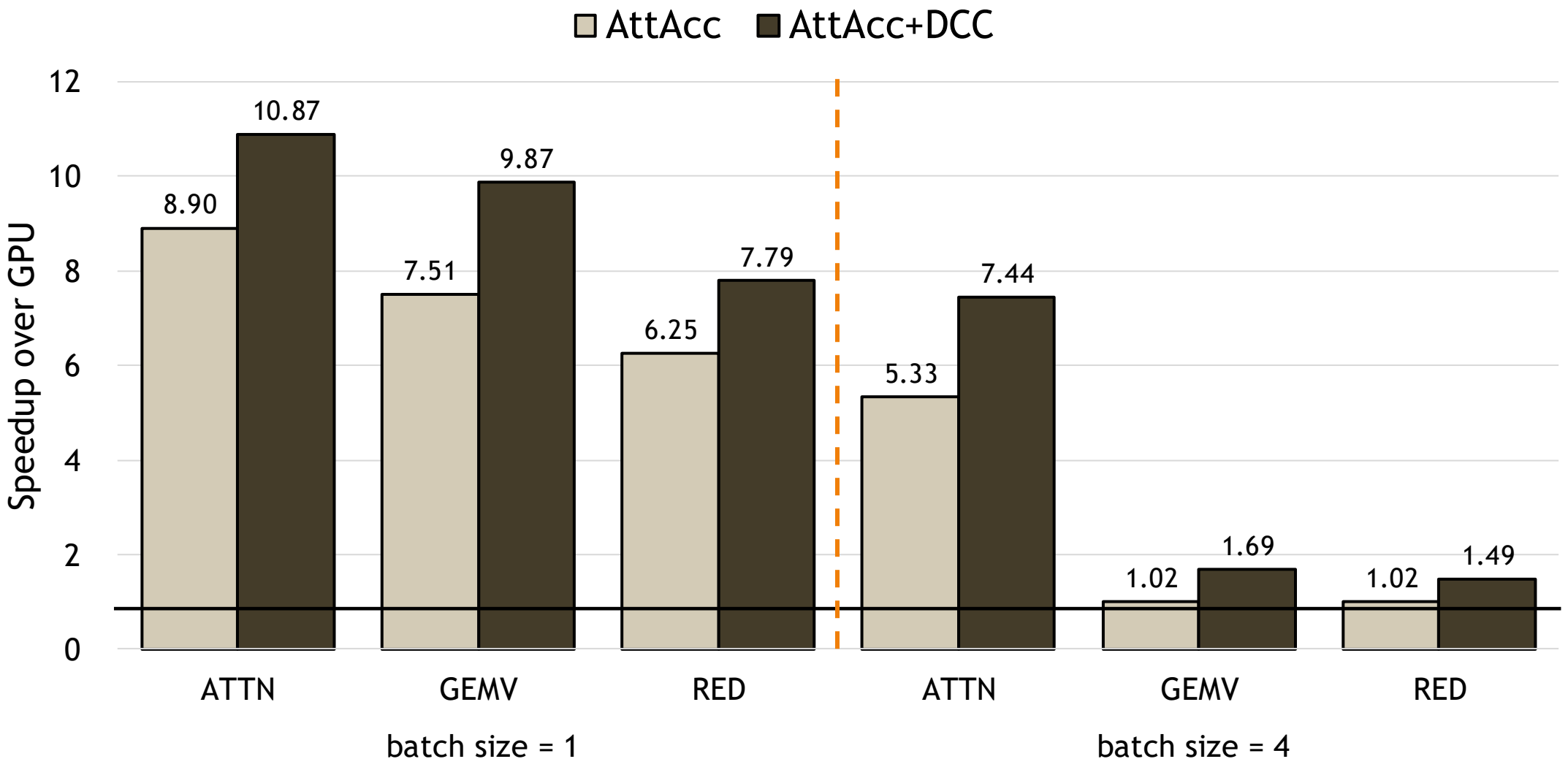
DCC Design

**Evaluation**

# Evaluation Methodology

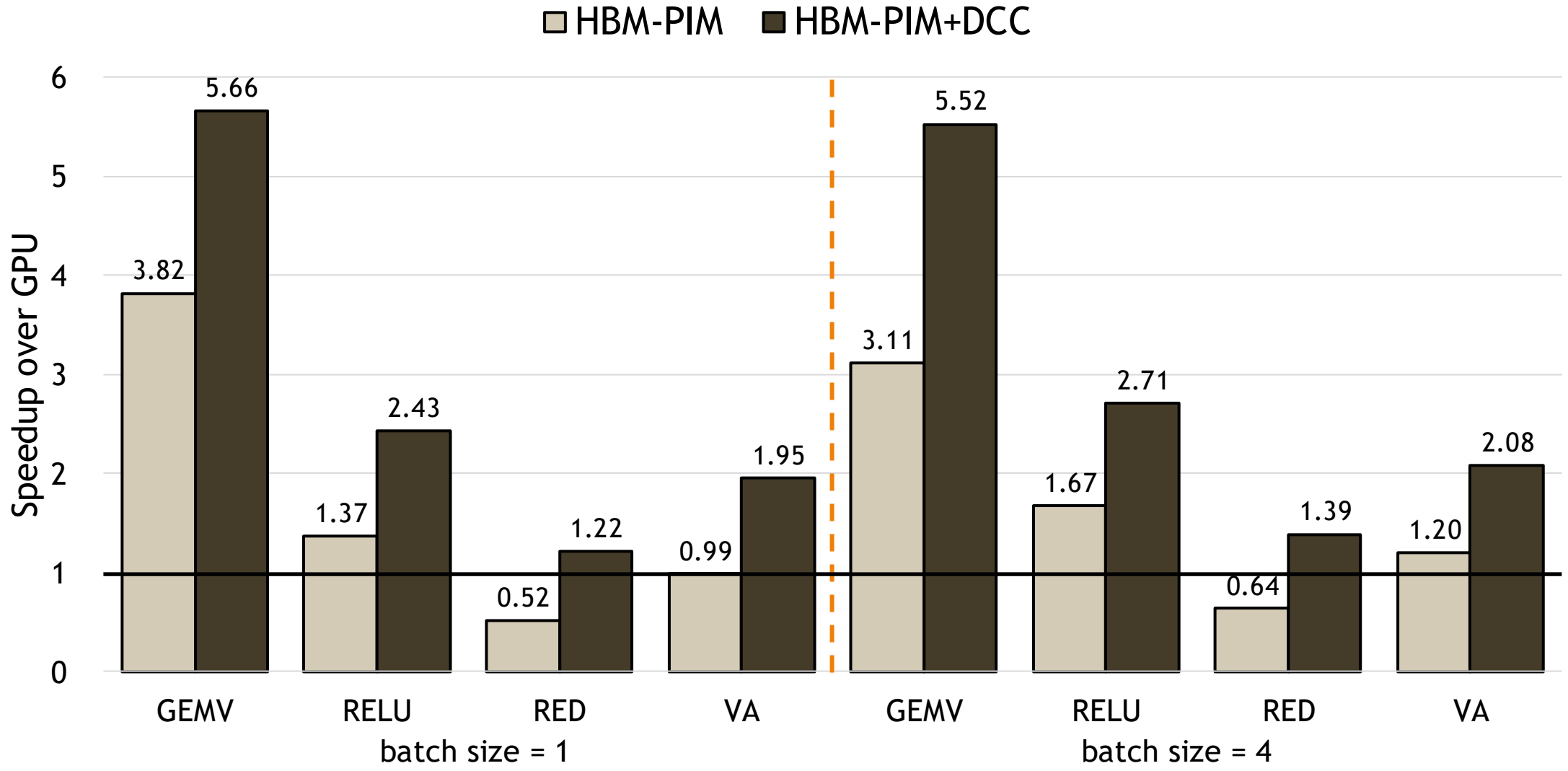
- Simulation Methodology:
  - Modified simulator using AttAcc [ASPLOS'24] and Ramulator 2.0 [IEEE CAL'23]
  - NVIDIA A100 GPU with 5 PIM-enabled HBM3 devices (80GB memory in total)
- 2× PIM Backends:
  - AttAcc [ASPLOS'24]
  - HBM-PIM [ISCA'21]
- 5× ML Kernels (kernel evaluation):
  - AttAcc: GEMV, RED, ATTN
  - HBM-PIM: GEMV, RED, VA, RELU
- 3× ML Models (end-to-end inference evaluation): GPT3-13B, LLAMA2-33B, MT-NLG-310B
- Comparison Points:
  - GPU: GPU-only execution (NVIDIA A100)
  - AttAcc: Original open-source AttAcc PIM implementation
  - AttAcc+DCC: AttAcc PIM backend with DCC compilation
  - HBM-PIM: HBM-PIM backend using AttAcc's data partition strategy
  - HBM-PIM+DCC: HBM-PIM backend with DCC compilation

# ML Kernel Performance on AttAcc PIM Backend




DCC on AttAcc improves performance of individual ML kernels by up to 1.7× and 1.3× on average over AttAcc with its fixed data partitioning method

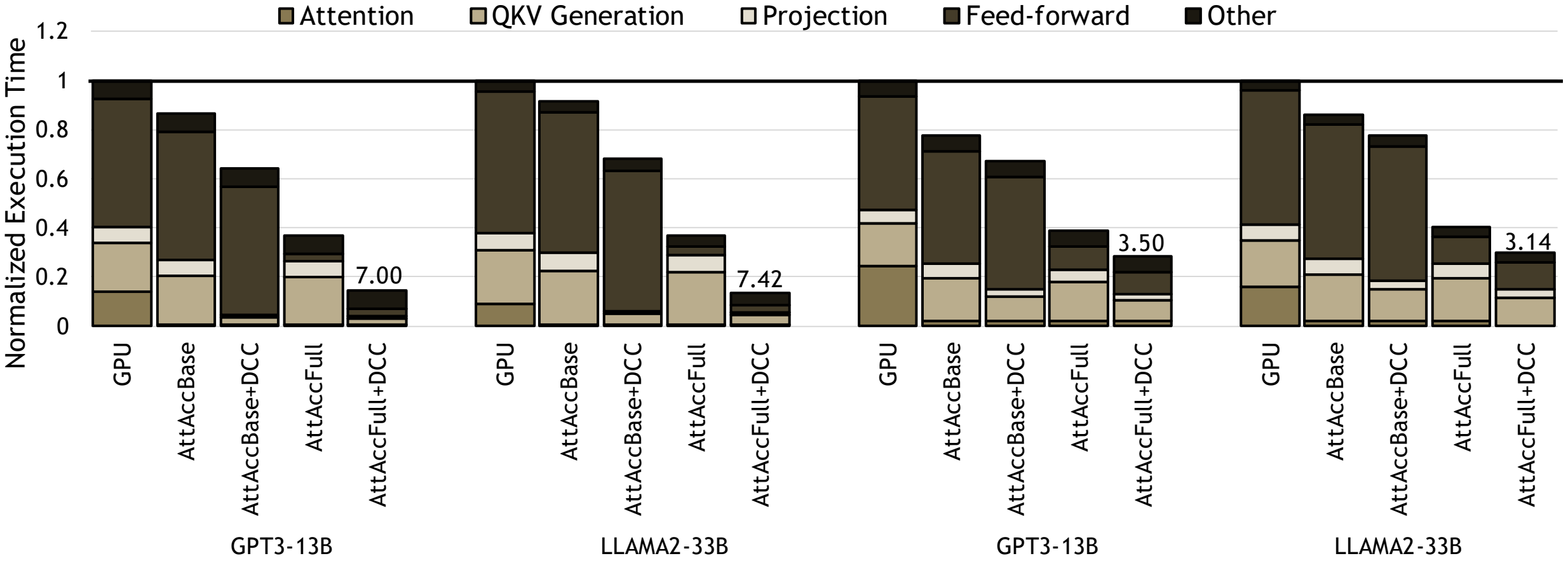
# ML Kernel Performance on HBM-PIM PIM Backend




DCC on HBM-PIM improves performance of individual ML kernels by up to 2.1× and 1.7× on average over HBM-PIM with its fixed data partitioning method

# LLM Inference Performance on AttAcc PIM Backend

- $\text{AttAcc}_{\text{Base}}$: AttAcc PIM fully running attention on PIM cores
- $\text{AttAcc}_{\text{Full}}$: AttAcc PIM running both attention and a part of feed-forward on PIM cores

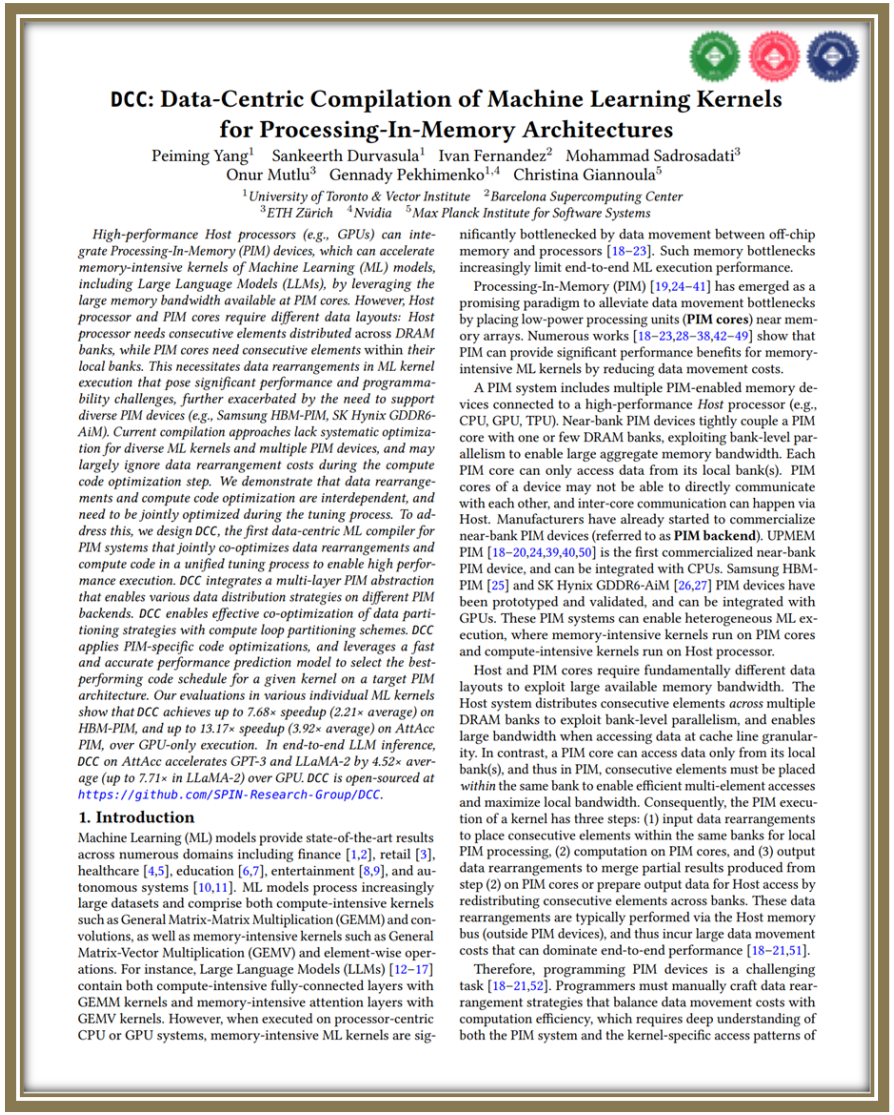



DCC accelerates LLM inference by up to 2.7× and 2.0× on average over AttAcc with its default fixed partitioning method

# More Evaluations in the Paper

- Detailed ML kernel analysis with various tensor sizes and batch sizes
- Scalability analysis on large batch sizes
- Detailed LLM inference results with various input token, output token, and batch sizes
- Large-scale LLM inference performance with FP8 and FP16 precisions
- Energy consumption evaluation on LLM inference with various input token, output token, and batch sizes
- Compilation time analysis on various ML kernels, LLMs and PIM hardware resources
- Effectiveness of the DCC's pruning mechanisms
- Accuracy, compilation analysis and performance analysis of the DCC predictor
- Performance analysis of DCC in LLM inference with dynamic tensor sizes

# DCC is Open Source & Awarded with all 3 Eval. Badges



**DCC: Data-Centric Compilation of Machine Learning Kernels for Processing-In-Memory Architectures**

Peiming Yang[1] Sankeerth Durvasula[1] Ivan Fernandez[2] Mohammad Sadrosadati[3]
Onur Mutlu[3] Gennady Pekhimenko[1,4] Christina Giannoula[5]
[1]University of Toronto & Vector Institute [2]Barcelona Supercomputing Center
[3]ETH Zürich [4]Nvidia [5]Max Planck Institute for Software Systems

*High-performance Host processors (e.g., GPUs) can integrate Processing-In-Memory (PIM) devices, which can accelerate memory-intensive kernels of Machine Learning (ML) models, including Large Language Models (LLMs), by leveraging the large memory bandwidth available at PIM cores. However, Host processor and PIM cores require different data layouts: Host processor needs consecutive elements distributed across DRAM banks, while PIM cores need consecutive elements within their local banks. This necessitates data rearrangements in ML kernel execution that pose significant performance and programmability challenges, further exacerbated by the need to support diverse PIM devices (e.g., Samsung HBM-PIM, SK Hynix GDDR6-AiM). Current compilation approaches lack systematic optimization for diverse ML kernels and multiple PIM devices, and may largely ignore data rearrangement costs during the compute code optimization step. We demonstrate that data rearrangements and compute code optimization are interdependent, and need to be jointly optimized during the tuning process. To address this, we design DCC, the first data-centric ML compiler for PIM systems that jointly co-optimizes data rearrangements and compute code in a unified tuning process to enable high performance execution. DCC integrates a multi-layer PIM abstraction that enables various data distribution strategies on different PIM backends. DCC enables effective co-optimization of data partitioning strategies with compute loop partitioning schemes. DCC applies PIM-specific code optimizations, and leverages a fast and accurate performance prediction model to select the best-performing code schedule for a given kernel on a target PIM architecture. Our evaluations in various individual ML kernels show that DCC achieves up to 7.68× speedup (2.21× average) on HBM-PIM, and up to 13.17× speedup (3.92× average) on AttAcc PIM, over GPU-only execution. In end-to-end LLM inference, DCC on AttAcc accelerates GPT-3 and LLaMA-2 by 4.52× average (up to 7.71× in LLaMA-2) over GPU. DCC is open-sourced at https://github.com/SPIN-Research-Group/DCC.*

## 1. Introduction

Machine Learning (ML) models provide state-of-the-art results across numerous domains including finance [1,2], retail [3], healthcare [4,5], education [6,7], entertainment [8,9], and autonomous systems [10,11]. ML models process increasingly large datasets and comprise both compute-intensive kernels such as General Matrix-Matrix Multiplication (GEMM) and convolutions, as well as memory-intensive kernels such as General Matrix-Vector Multiplication (GEMV) and element-wise operations. For instance, Large Language Models (LLMs) [12–17] contain both compute-intensive fully-connected layers with GEMM kernels and memory-intensive attention layers with GEMV kernels. However, when executed on processor-centric CPU or GPU systems, memory-intensive ML kernels are significantly bottlenecked by data movement between off-chip memory and processors [18–23]. Such memory bottlenecks increasingly limit end-to-end ML execution performance.

Processing-In-Memory (PIM) [19,24–41] has emerged as a promising paradigm to alleviate data movement bottlenecks by placing low-power processing units **(PIM cores)** near memory arrays. Numerous works [18–23,28–38,42–49] show that PIM can provide significant performance benefits for memory-intensive ML kernels by reducing data movement costs.

A PIM system includes multiple PIM-enabled memory devices connected to a high-performance *Host* processor (e.g., CPU, GPU, TPU). Near-bank PIM devices tightly couple a PIM core with one or few DRAM banks, exploiting bank-level parallelism to enable large aggregate memory bandwidth. Each PIM core can only access data from its local bank(s). PIM cores of a device may not be able to directly communicate with each other, and inter-core communication can happen via Host. Manufacturers have already started to commercialize near-bank PIM devices (referred to as **PIM backend**). UPMEM PIM [18–20,24,39,40,50] is the first commercialized near-bank PIM device, and can be integrated with CPUs. Samsung HBM-PIM [25] and SK Hynix GDDR6-AiM [26,27] PIM devices have been prototyped and validated, and can be integrated with GPUs. These PIM systems can enable heterogeneous ML execution, where memory-intensive kernels run on PIM cores and compute-intensive kernels run on Host processor.

Host and PIM cores require fundamentally different data layouts to exploit large available memory bandwidth. The Host system distributes consecutive elements *across* multiple DRAM banks to exploit bank-level parallelism, and enables large bandwidth when accessing data at cache line granularity. In contrast, a PIM core can access data only from its local bank(s), and thus in PIM, consecutive elements must be placed *within* the same bank to enable efficient multi-element accesses and maximize local bandwidth. Consequently, the PIM execution of a kernel has three steps: (1) input data rearrangements to place consecutive elements within the same banks for local PIM processing, (2) computation on PIM cores, and (3) output data rearrangements to merge partial results produced from step (2) on PIM cores or prepare output data for Host access by redistributing consecutive elements across banks. These data rearrangements are typically performed via the Host memory bus (outside PIM devices), and thus incur large data movement costs that can dominate end-to-end performance [18–21,51].

Therefore, programming PIM devices is a challenging task [18–21,52]. Programmers must manually craft data rearrangement strategies that balance data movement costs with computation efficiency, which requires deep understanding of both the PIM system and the kernel-specific access patterns of

Paper

DCC

GitHub

https://github.com/SPIN-Research-Group/DCC

# Conclusion

DCC: The first data-centric ML compiler for PIM systems that jointly co-optimizes data rearrangements and compute code in a unified tuning process and supports any ML tensor kernel and multiple PIM backends

DCC

Key Contributions:

- ✓ Enables support for multiple PIM backends
- ✓ Couples data and compute-loop partition strategies for co-optimization
- ✓ Performs PIM-specific performance optimizations
- ✓ Provides the best-performing configuration across joint data and compute-loop partition strategies
- ✓ Integrates a data-centric intermediate representation that can be lowered to multiple PIM backends

GitHub

Paper

Key Results: DCC improves performance by 1.7× and 1.3× over default implementation on HBM-PIM and AttAcc backends, respectively, for individual ML kernels, and by 2.0× over AttAcc default implementation for LLM inference

# DCC

# Data-Centric Compilation of Machine Learning Kernels for Processing-In-Memory Architectures


Peiming Yang,

Sankeerth Durvasula, Ivan Fernandez, Mohammad Sadrosadati,

Onur Mutlu, Gennady Pekhimenko, Christina Giannoula



ISCA 2026

Raleigh, USA, July 1st 2026