

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

MCCSys Workshop, ISCA 2026

Raleigh, USA, June 28th 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: co-optimize data and compute-loop partition
- PIM-Specific Code Optimizer: performs PIM-specific performance optimizations
- Coupled Predictor: selects the best configuration across joint compute-loop and data partition
- Intermediate Representation: a data-centric IR 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

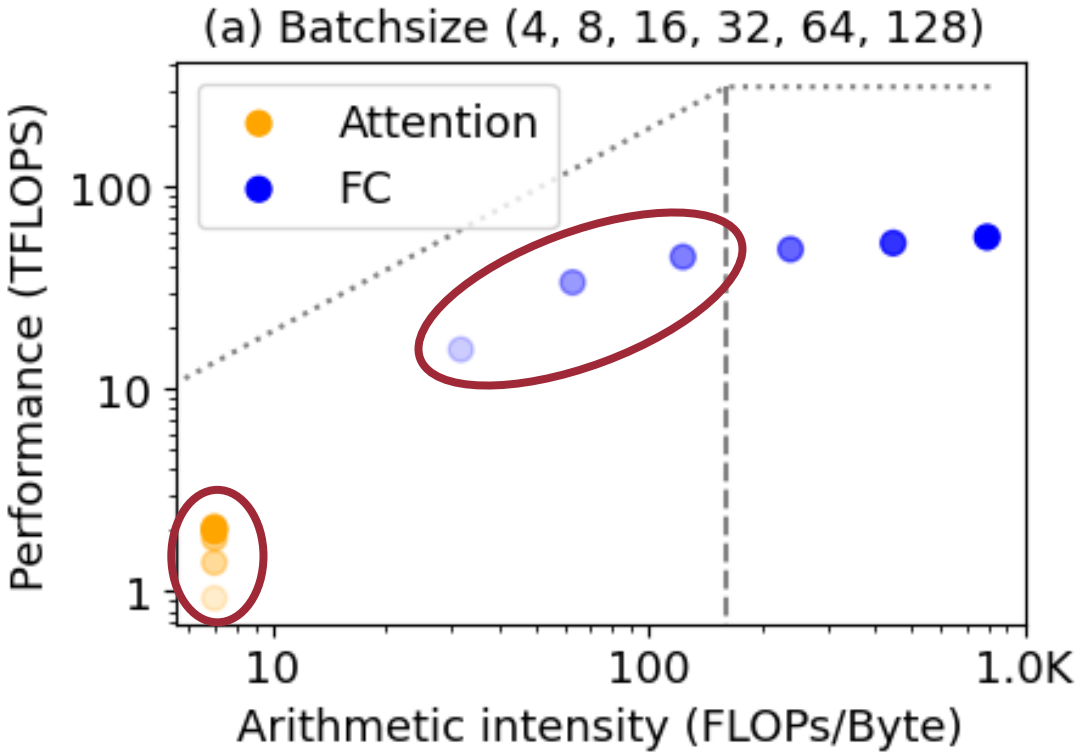
Our Design

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 small batch sizes

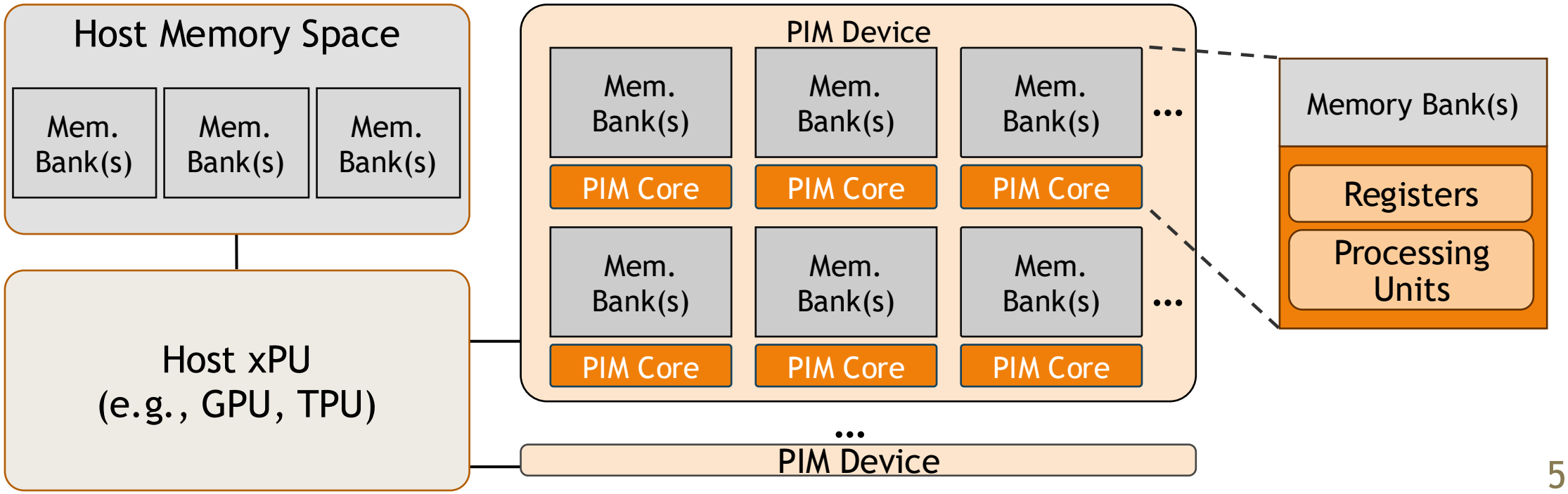


OPT-30B Inference kernels on NVIDIA A100 GPU

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

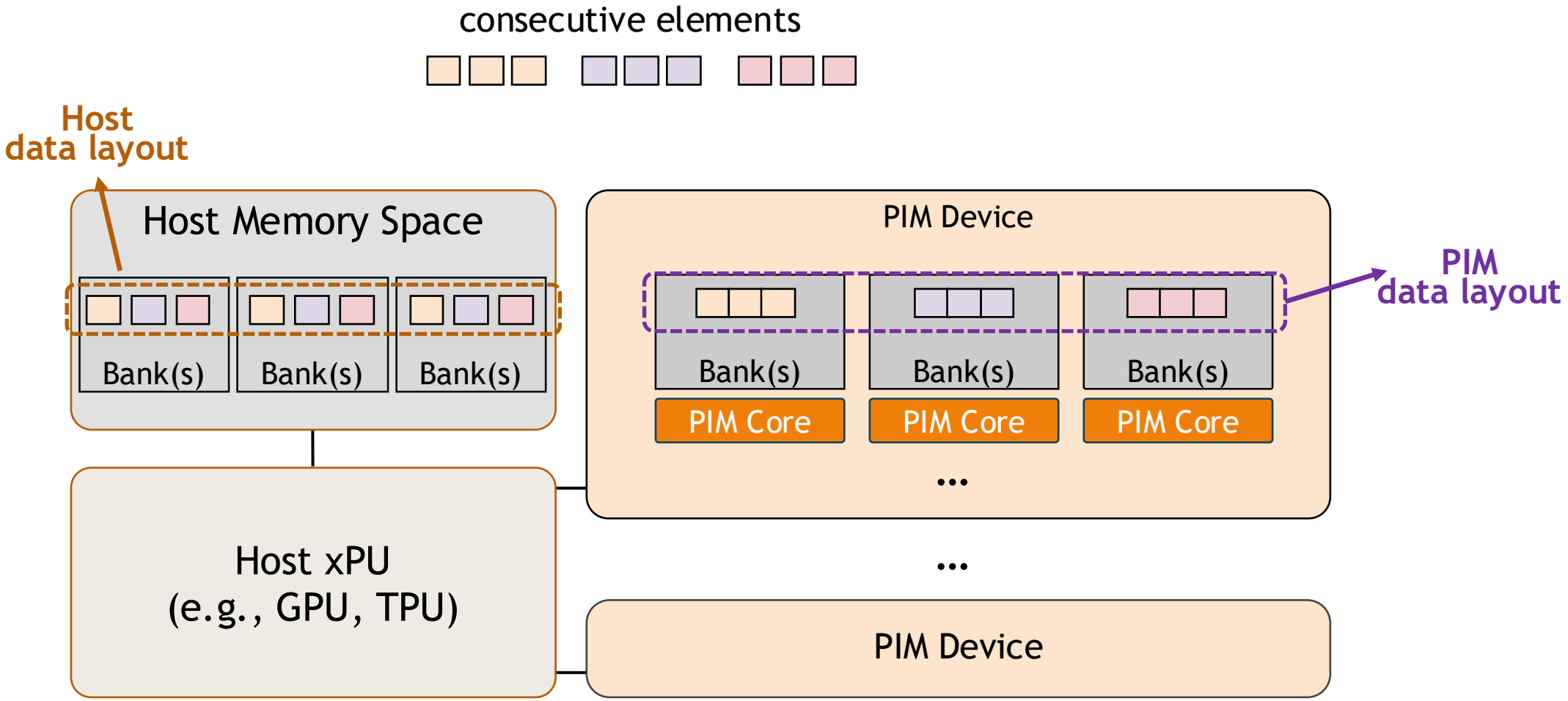
PIM Alleviates Data Movement Bottlenecks

- 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 cores) close to one (or a few) memory bank(s)
  - Immense aggregated memory bandwidth & high-levels 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

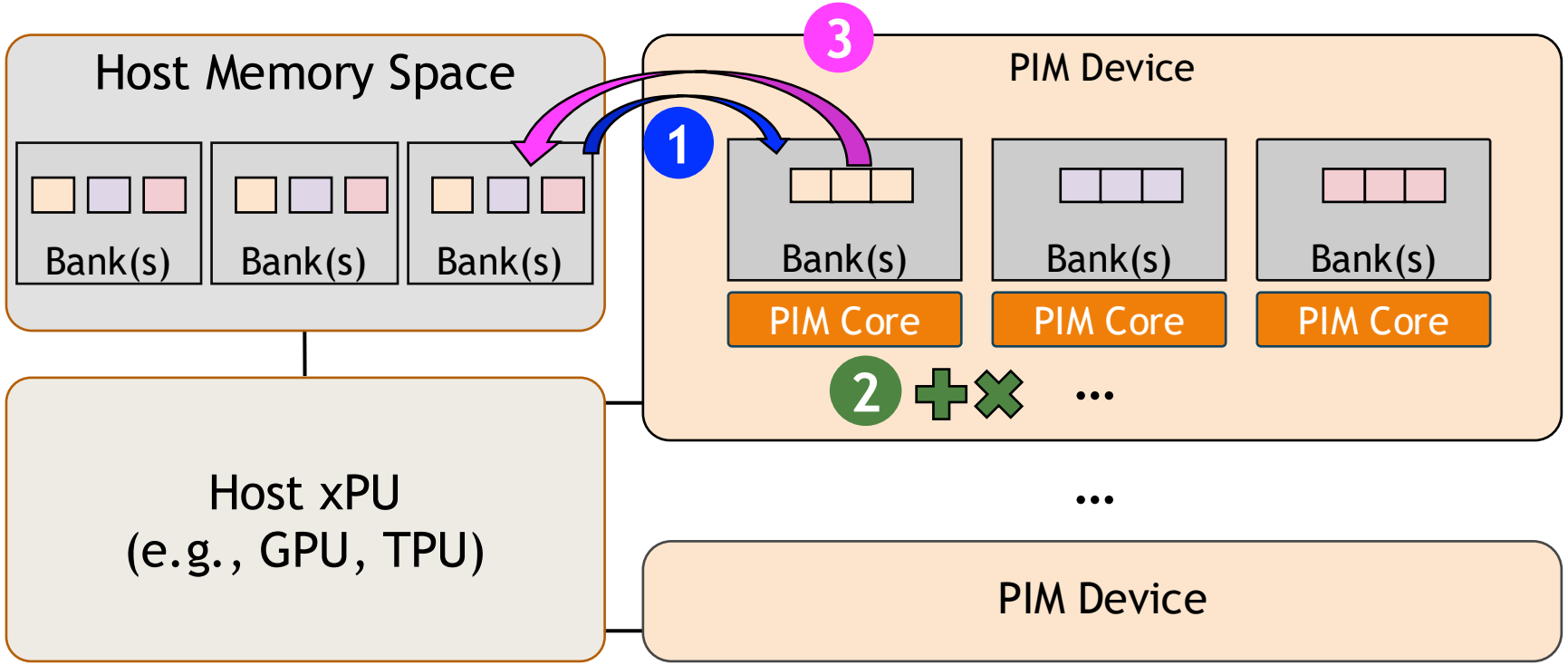


Three Steps of ML Kernel Execution on PIM Systems

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

Our Design

Evaluation

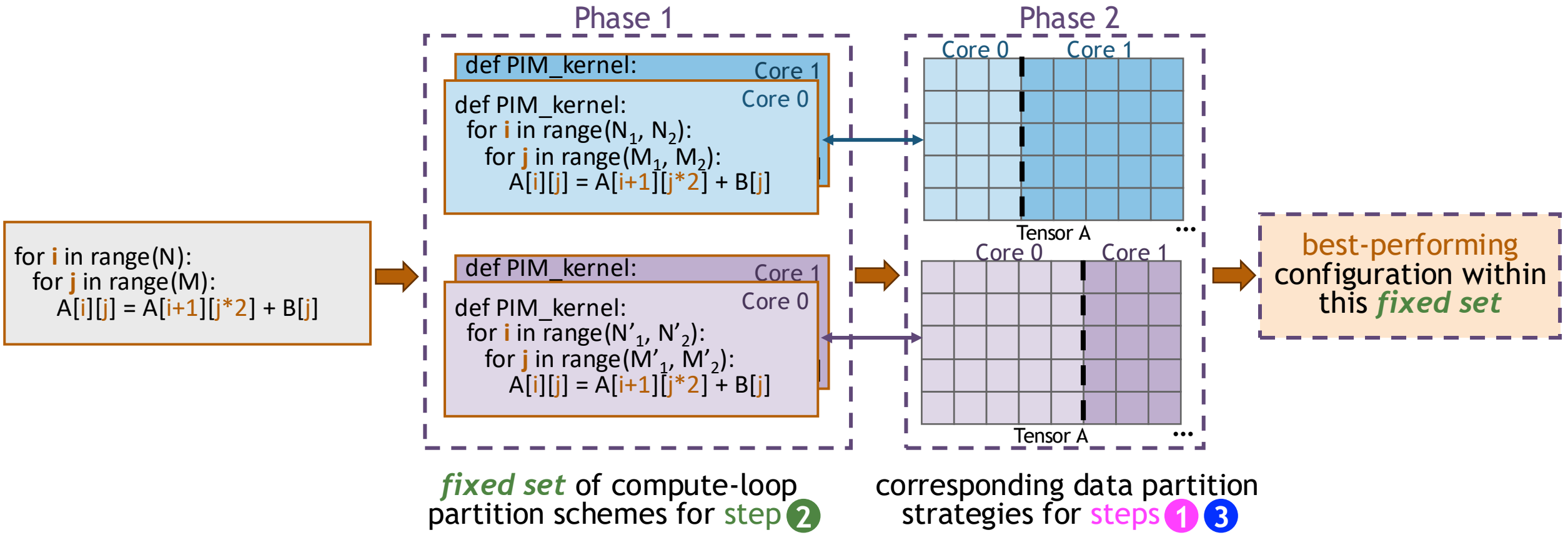
Limitations of Existing PIM Compilers (I)

- 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
- 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

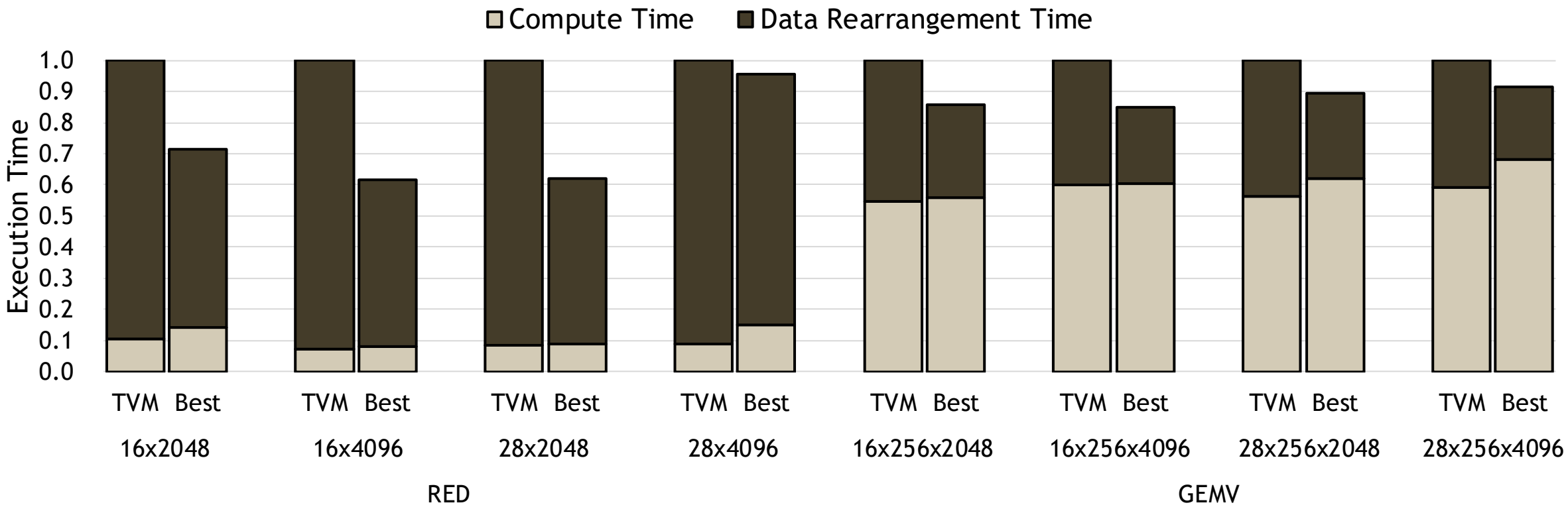
Limitations of Existing PIM Compilers (II)

- Limitation 2: Existing compilers follow a compute-centric approach, i.e., largely ignore data rearrangement costs during the compute-loop optimization step



Limitations of Existing PIM Compilers (II)

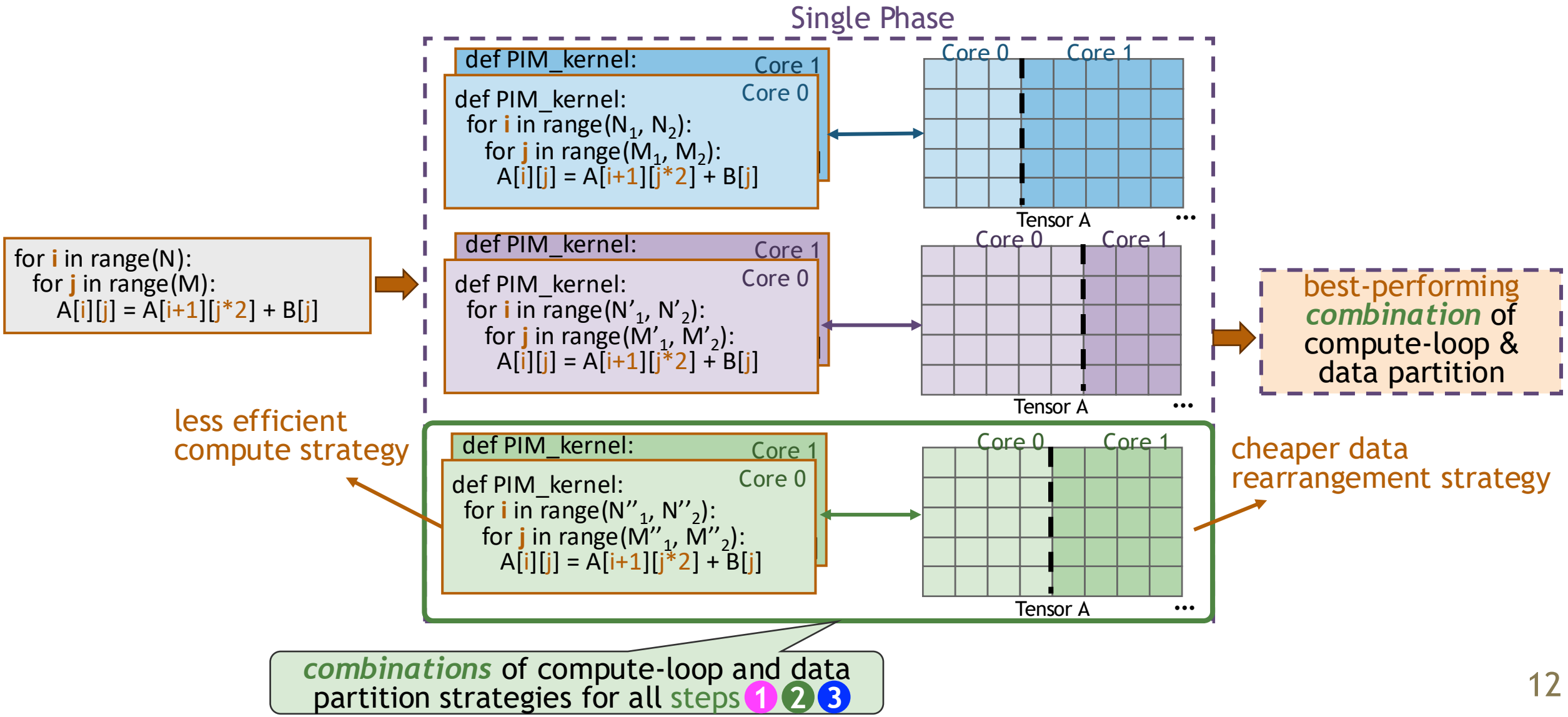
- Performance comparison on a PIM system:
  - Compute-centric TVM-based compiler (TVM) (following ATiM [ISCA'24]) vs
  - Manually-tuned best-performing implementation (Best)



Slightly worse compute time could achieve much better data rearrangement time, resulting in 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

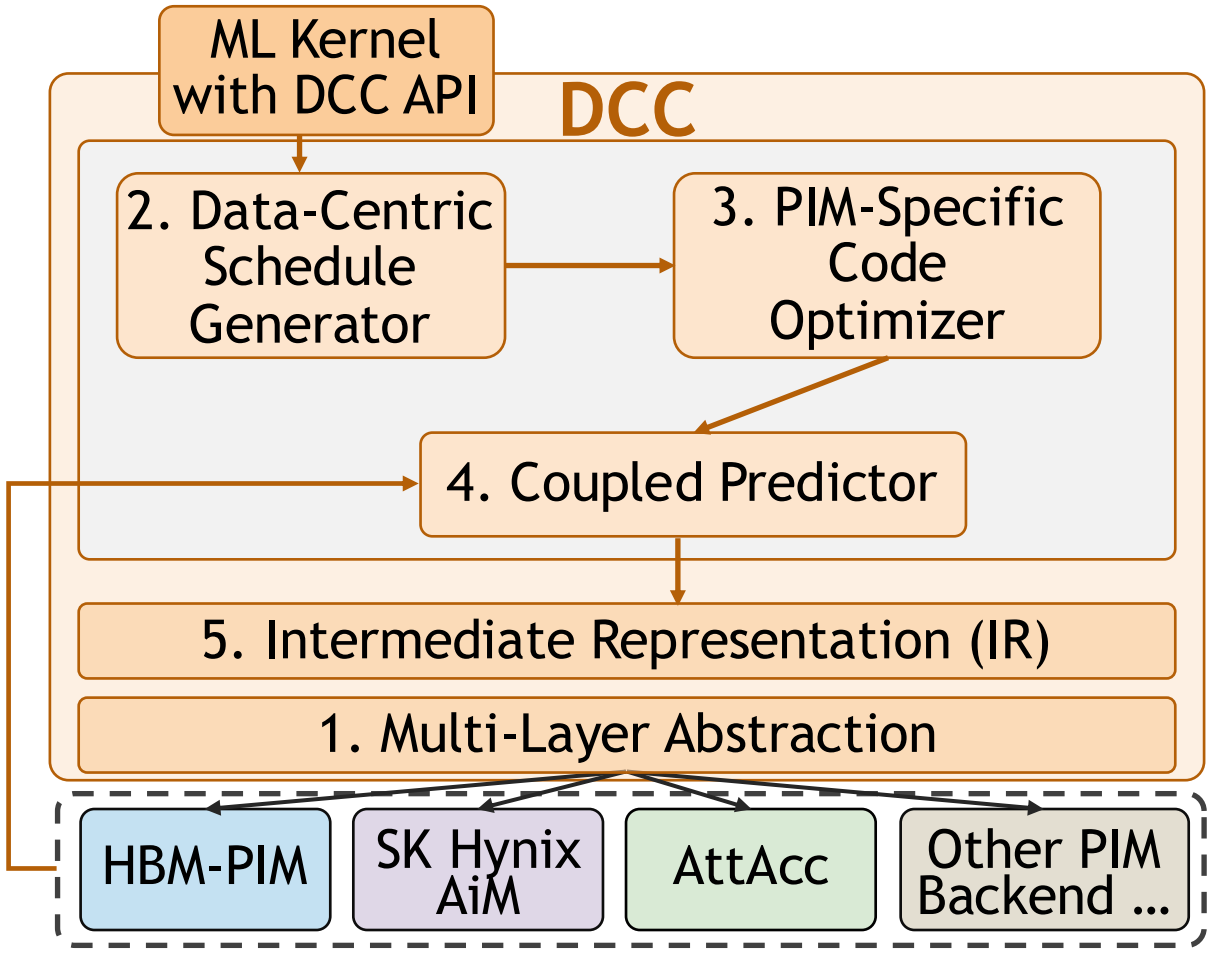
Motivation

Our Design

Evaluation

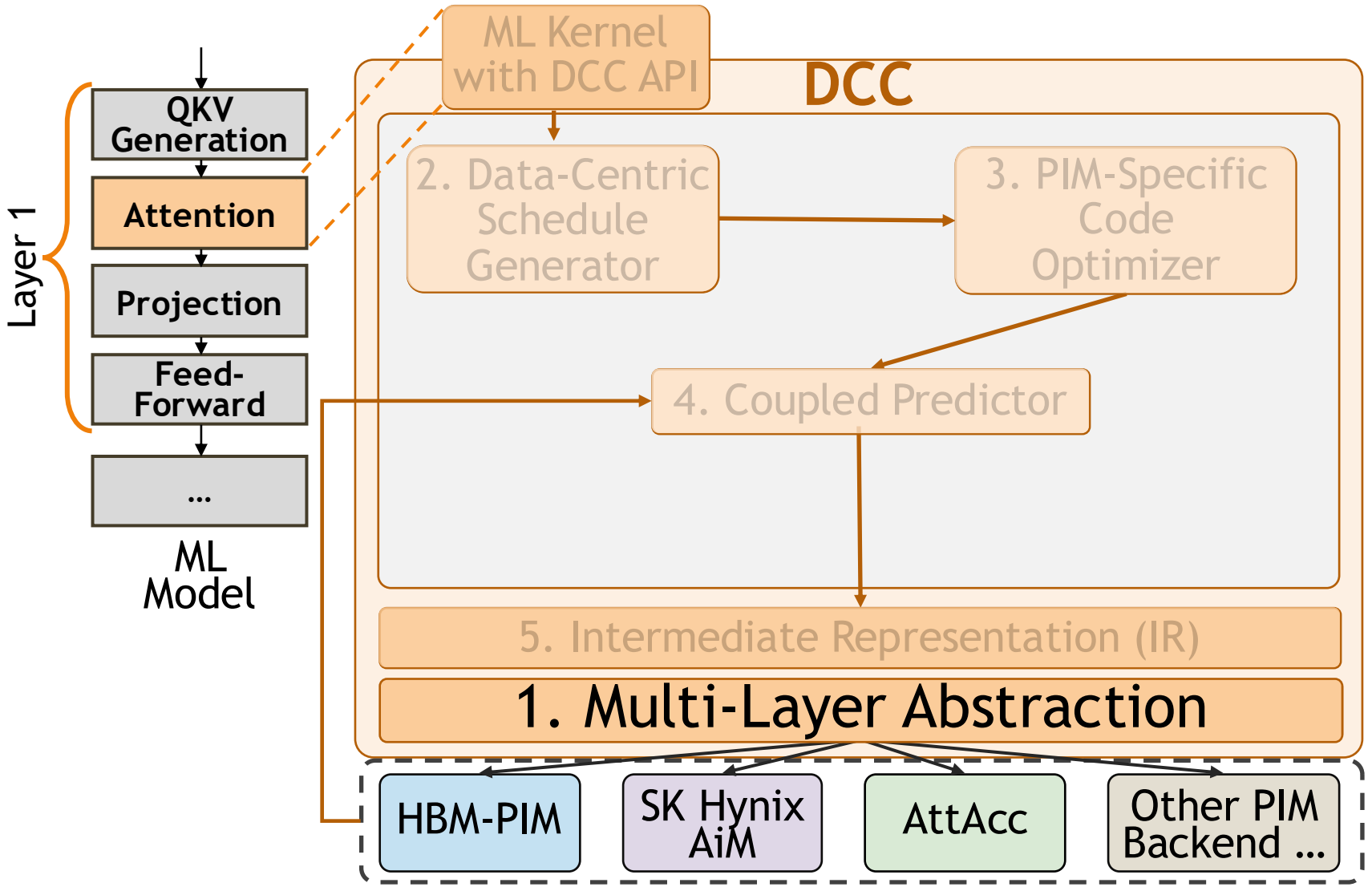
DCC Overview & 5 Key Components

→Our data-centric compilation(DCC) is the first data-centric ML compiler for PIM systems that jointly co-optimizes compute-loop and data partition strategies into a unified tuning process



- ☑ Multiple PIM backends
- ☑ Any ML tensor kernel
- ☑ Low compilation time
- ☑ Near-optimal performance

DCC Overview & 5 Key Components



Goal

→ Enable support for multiple current and future PIM backends

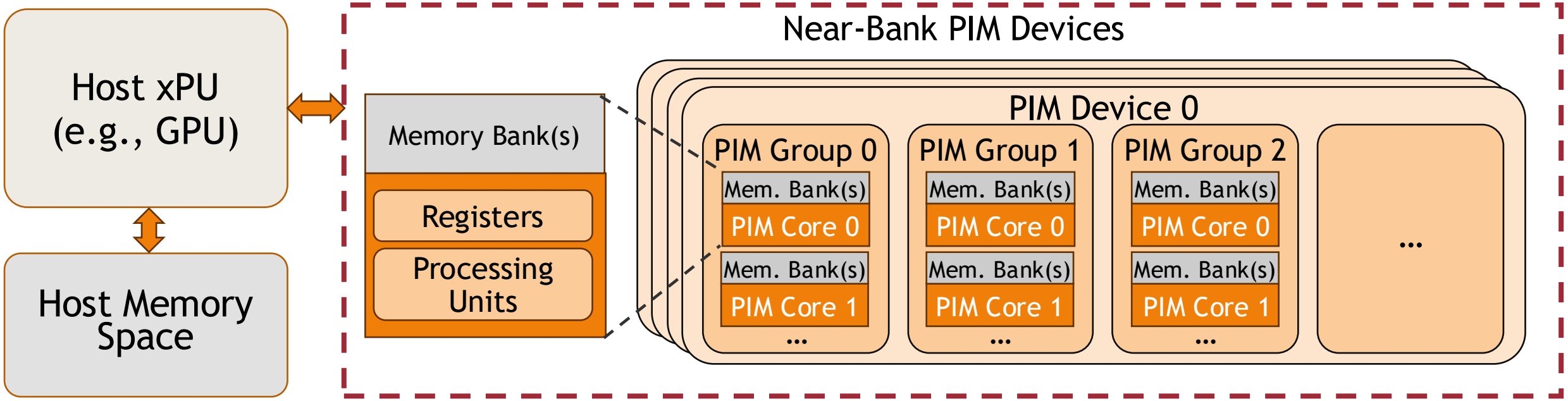
1. Multi-Layer Abstraction for Near-Bank PIM Systems

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

a) System level: Host xPU, Host memory spaces and multiple PIM devices.

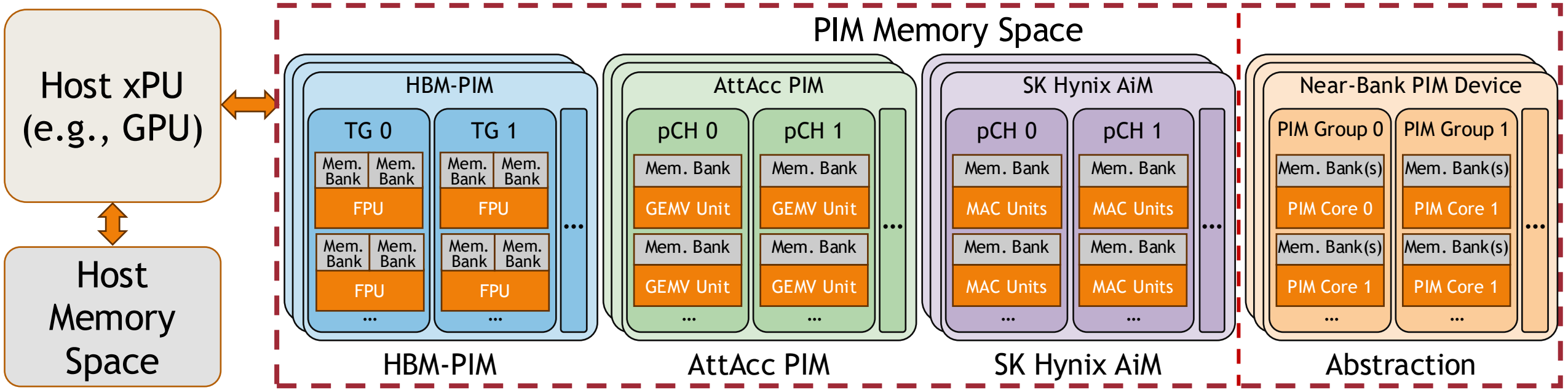
b) PIM group level: multiple PIM cores (and their local memory banks) belonging to the same memory channel or rank

c) PIM core level: the processing unit (core) and its local memory banks

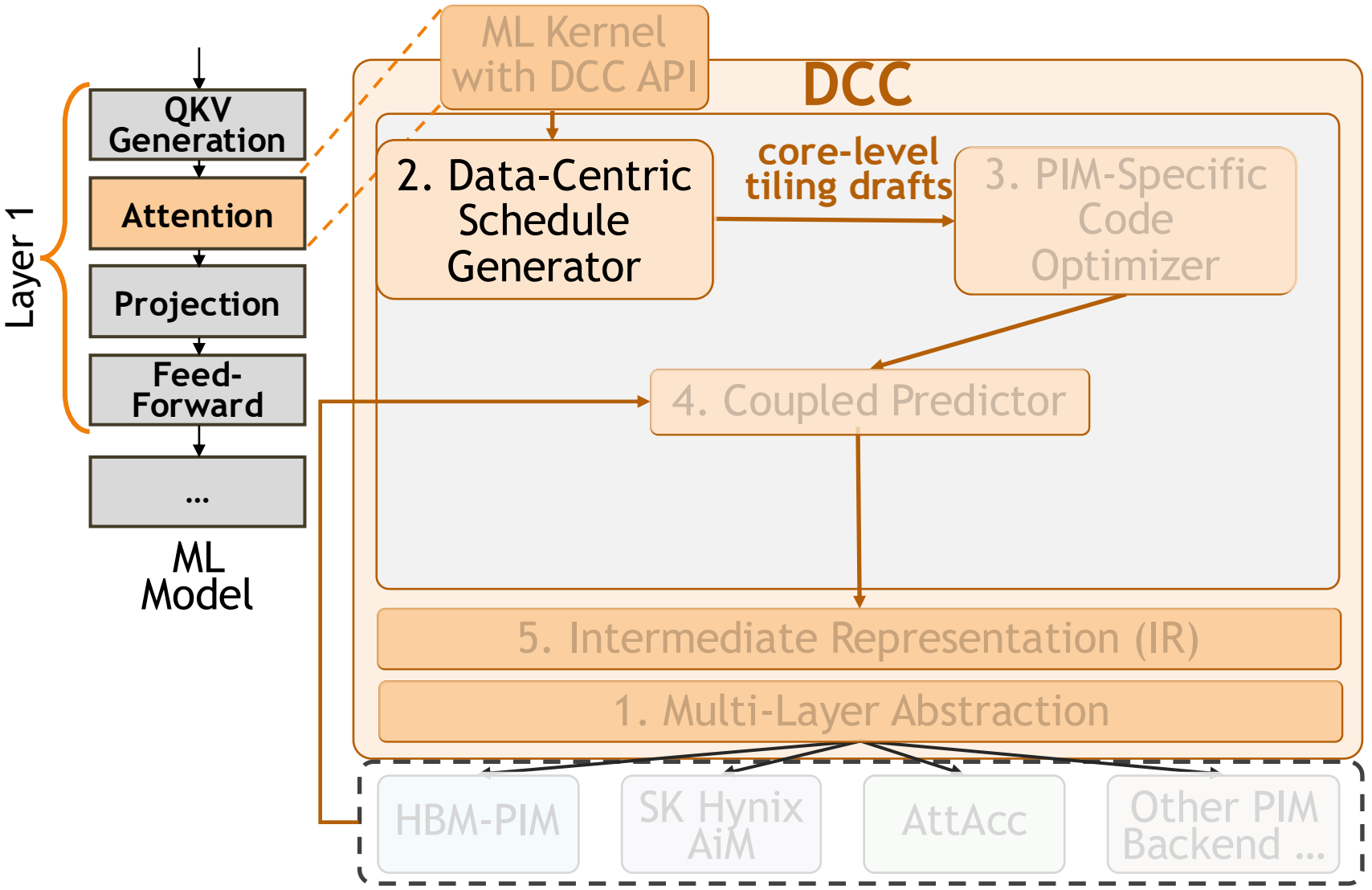


Three Example PIM Backends into DCC's Abstraction

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



DCC Overview & 5 Key Components



Goal

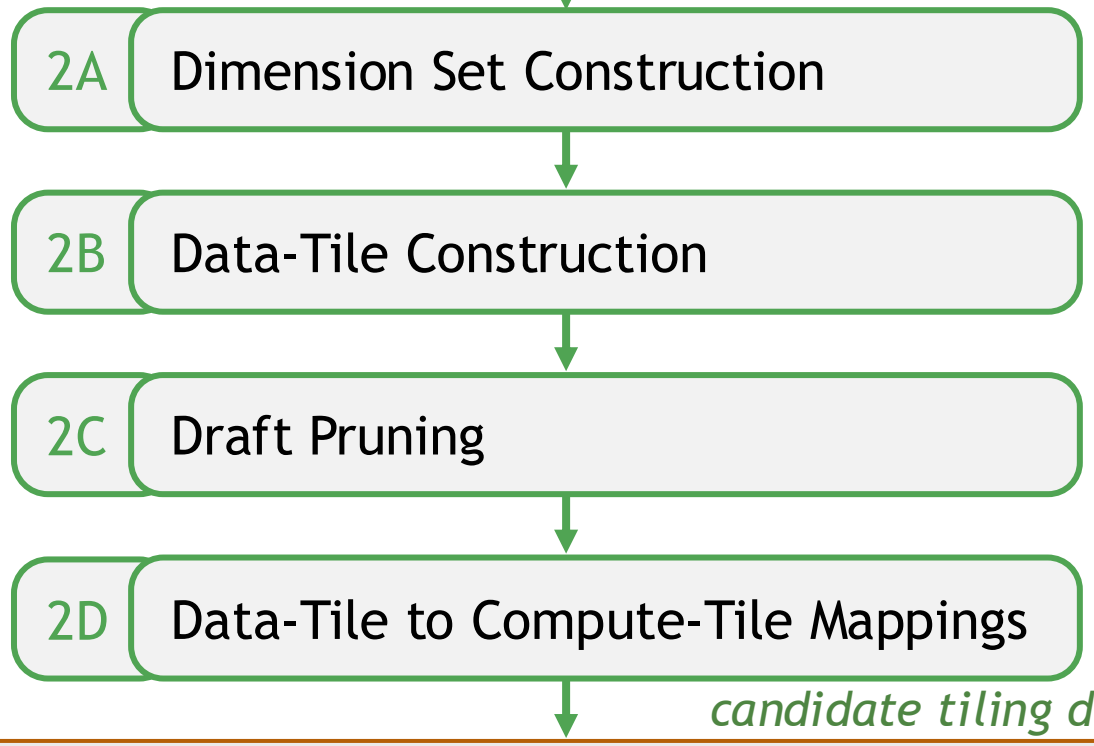
→ Generate candidate combinations of data partition strategies and compute-loop partition schemes for each PIM core

candidate tiling drafts

2. Data-Centric Schedule Generator

Goal

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



candidate tiling drafts

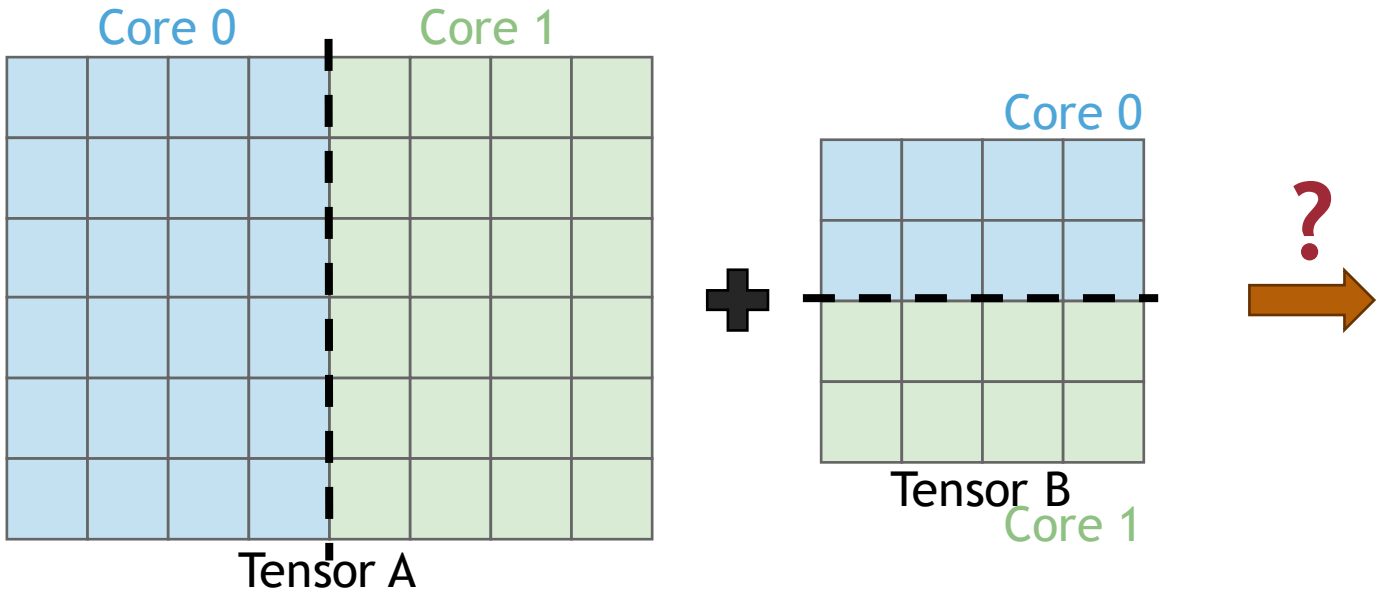
Draft 0. PIM Core 0: [(A[0:3][0:2], B[0:1][0:4] ...), (i ∈ {0,1,2}), (j ∈ {0}), (k ∈ {0,1,2,3})], ...
Draft 1. PIM Core 0: [(A[0:6][0:4], B[0:2][0:4] ...), (i ∈ {0,1,2,3,4,5}), (j ∈ {0,1}), (k ∈ {0,1,2,3})], ...
Draft 4. PIM Core 0: [(A[0:6][0:2], B[0:1][0:4] ...), (i ∈ {0,1,2,3,4,5}), (j ∈ {0}), (k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

Challenge in Data-Centric Compilation Approach

How to associate data partitions with 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]
```

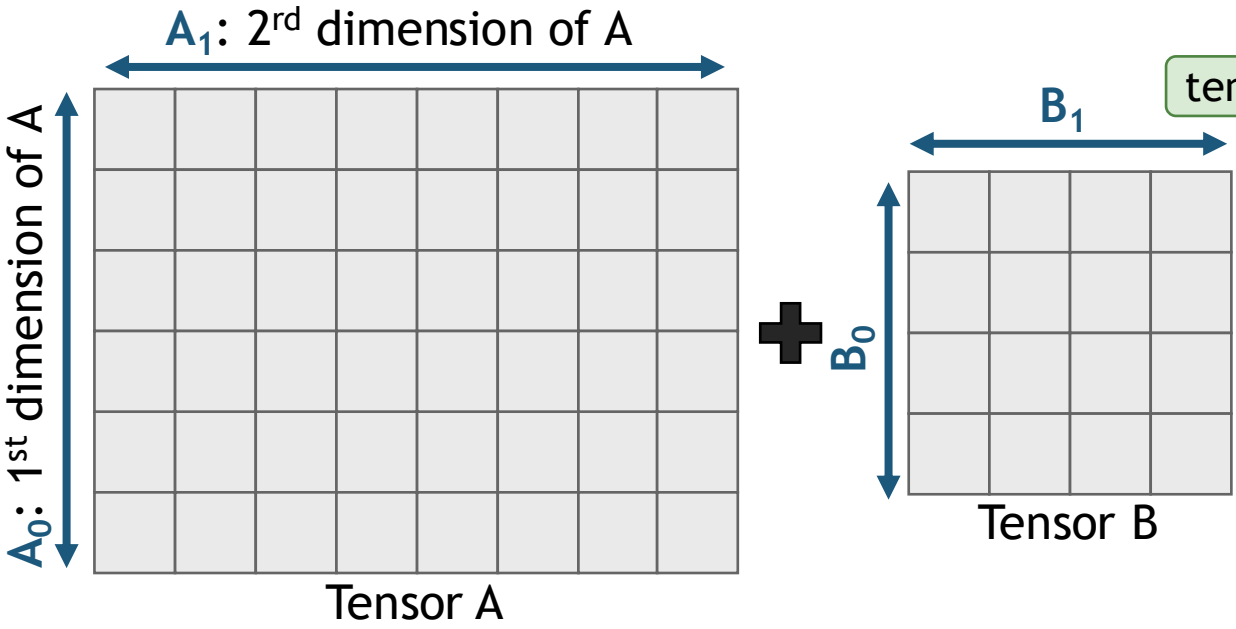
2A. Dimension Set Construction

→ associates data partitions with loop partitions

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

3 loop variables

```
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)

tensor dimension

corresponding mapping function

$A_0 \rightarrow F(i) = i$
$A_0 \rightarrow F(i) = i+1$ — i loop variable

$A_1 \rightarrow F(j) = j$
$A_1 \rightarrow F(j) = j*2$
$B_0 \rightarrow F(j) = j$ — j loop variable

$B_1 \rightarrow F(k) = k$ — k loop variable

2B. Data-Tile Construction

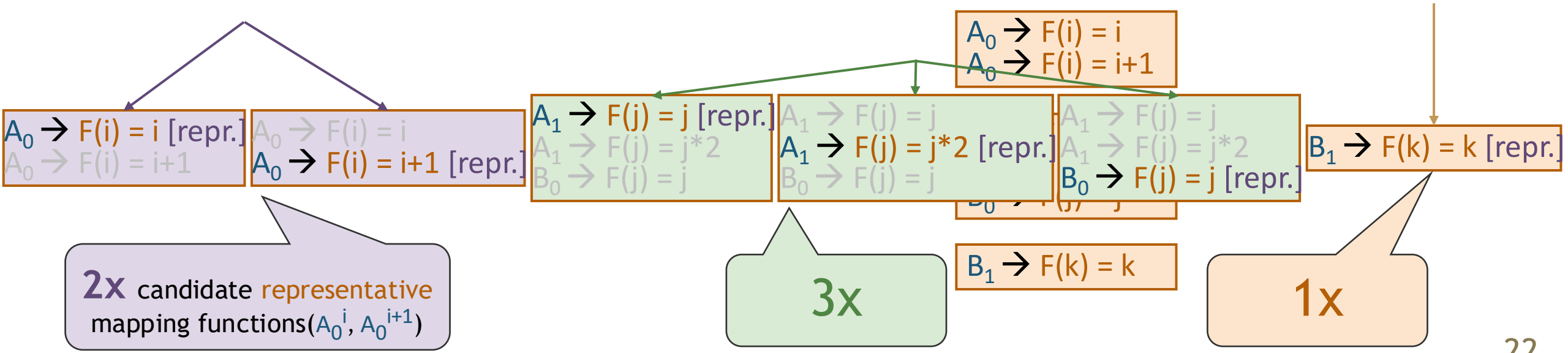
→ performs an exhaustive search to generate all data partitions

- To decide the dimension to partition, DCC searches for representative 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 dimension sets (one for each loop variable)

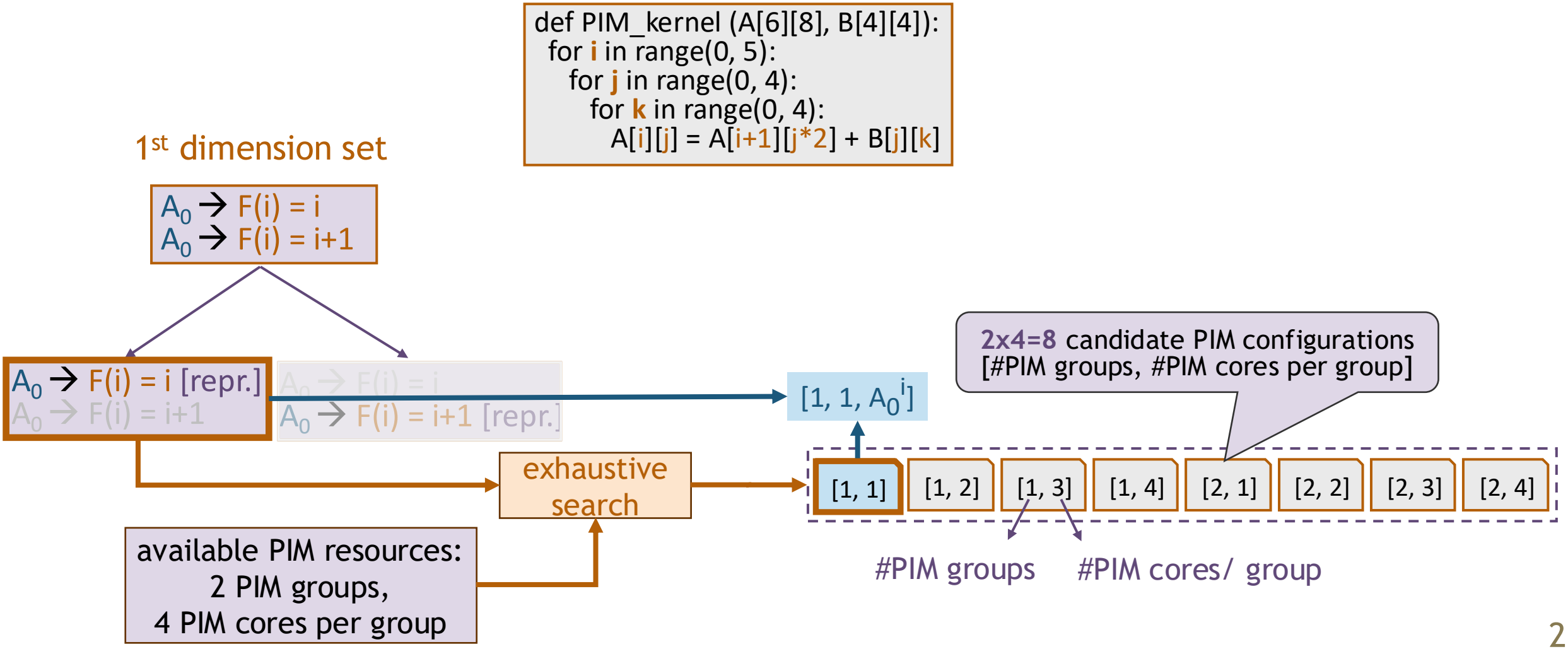
3 dimension sets
(one for each loop variable)



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



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

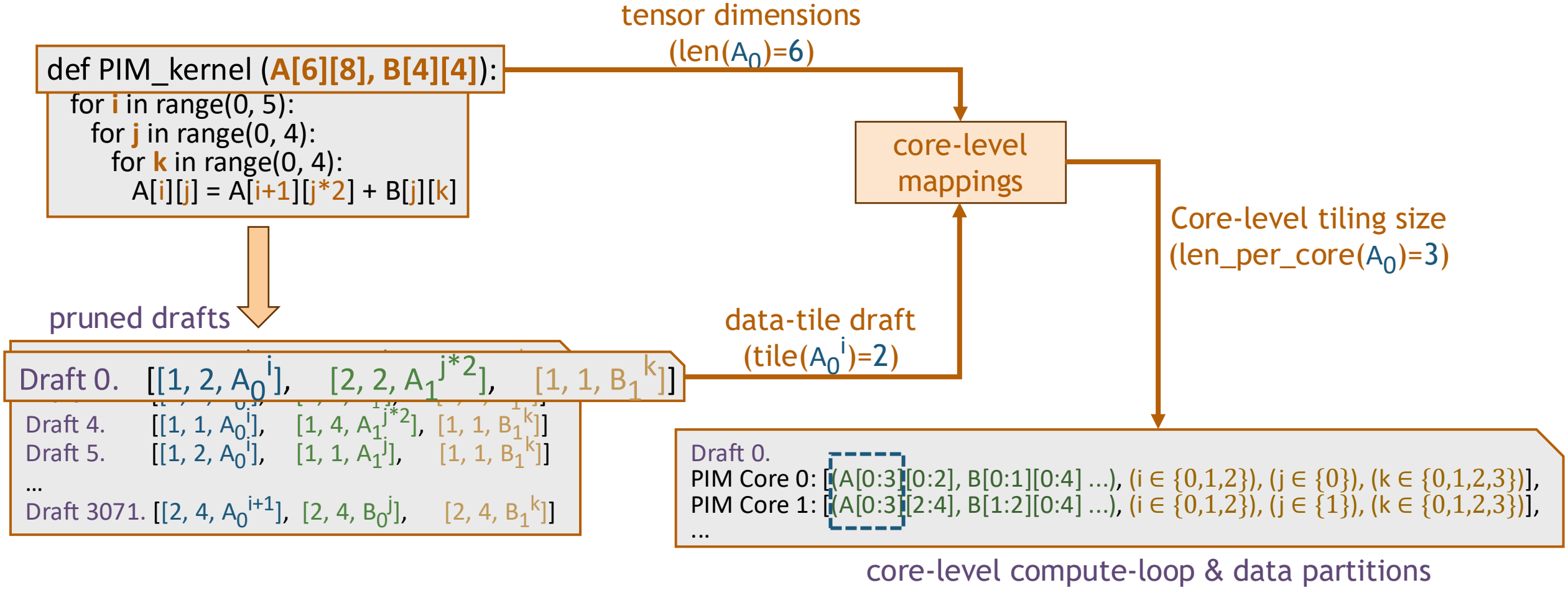
pruned by ~90%
(only ~10% retained)

pruned drafts

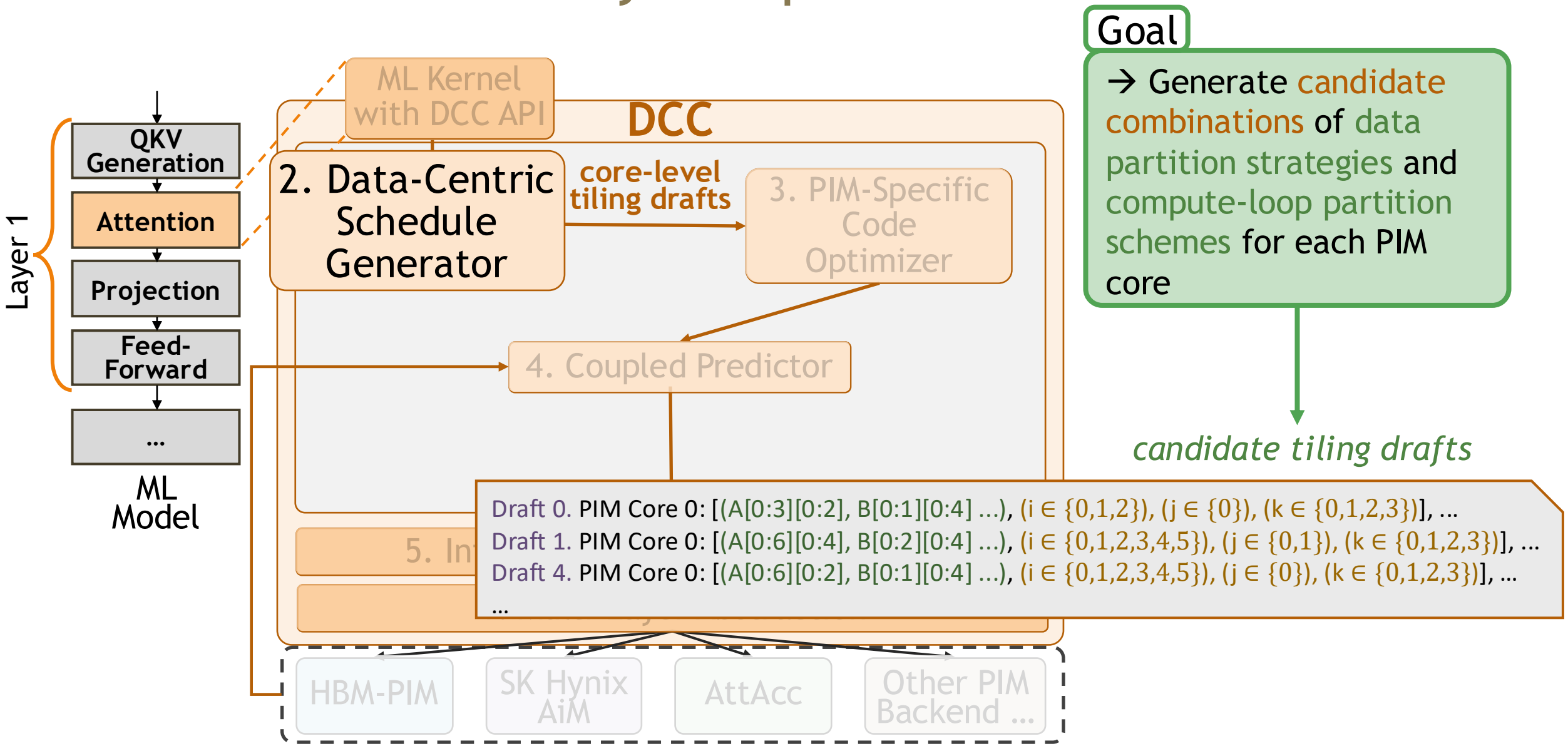
Draft 0.	[[1, 1, A_0^i],	[1, 1, A_1^j],	[1, 1, B_1^k]]
Draft 1.	[[1, 1, A_0^i],	[1, 2, A_1^j],	[1, 1, B_1^k]]
Draft 4.	[[1, 1, A_0^i],	[1, 4, A_1^{j*2}],	[1, 1, B_1^k]]
Draft 5.	[[1, 2, A_0^i],	[1, 1, A_1^j],	[1, 1, B_1^k]]
…			
Draft 3071.	[[2, 4, A_0^{i+1}],	[2, 4, B_0^j],	[2, 4, B_1^k]]

2D. Data-Tile to Compute-Tile Mappings

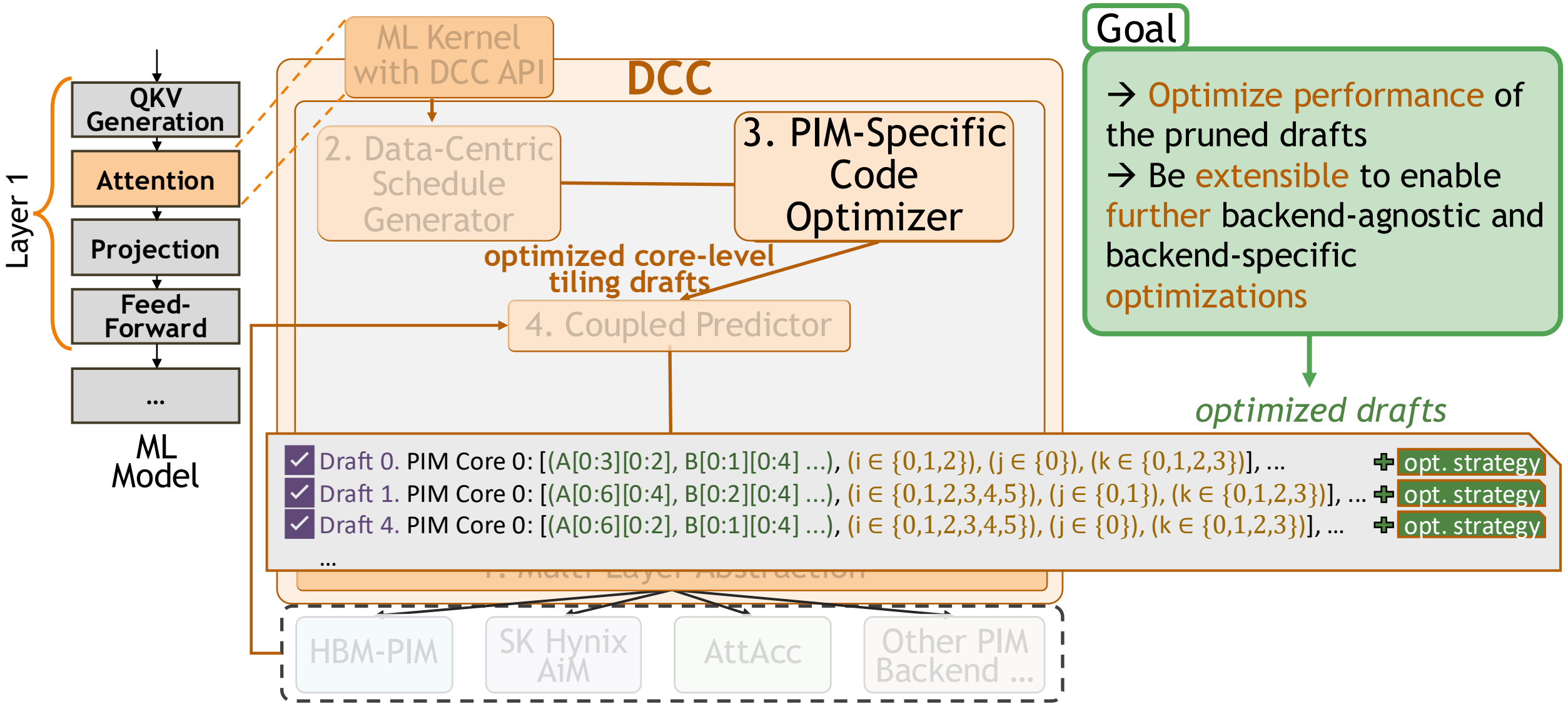
→ constructs compute-loop partitions and data partitions for each PIM core for all drafts



DCC Overview & 5 Key Components



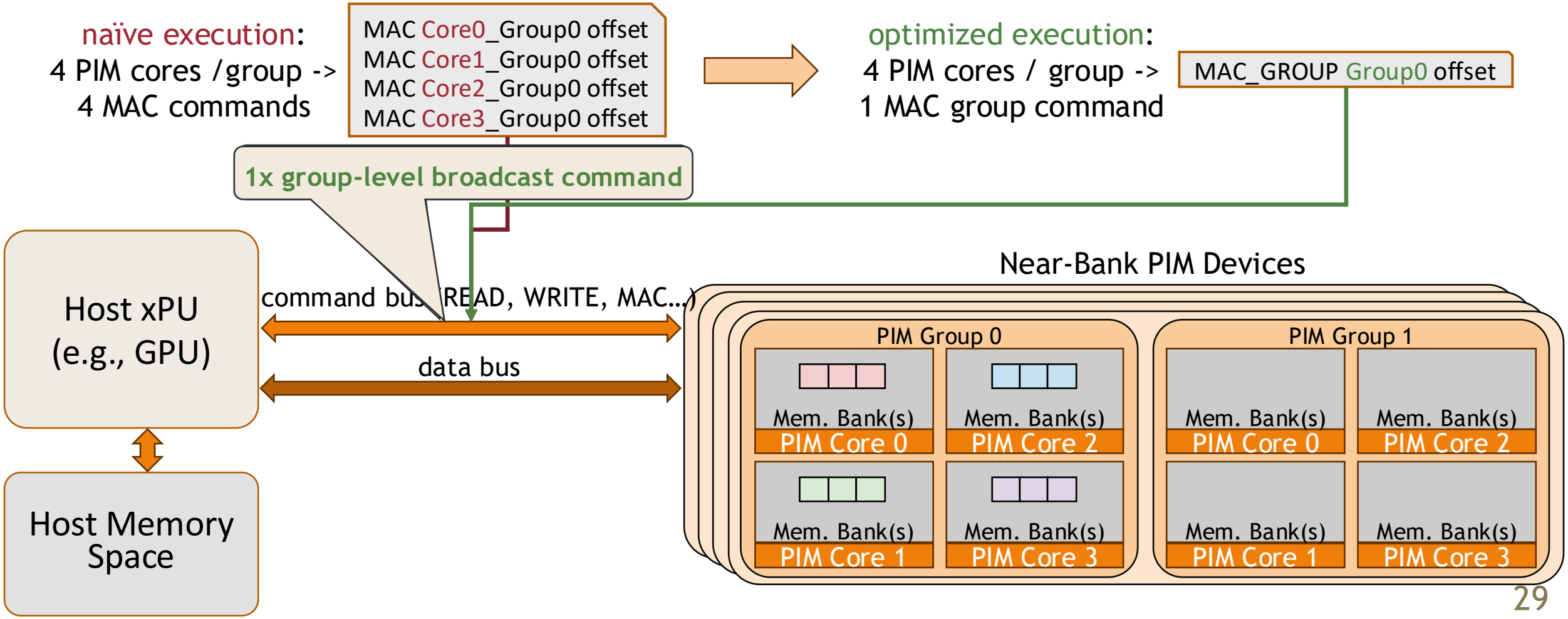
DCC Overview & 5 Key Components



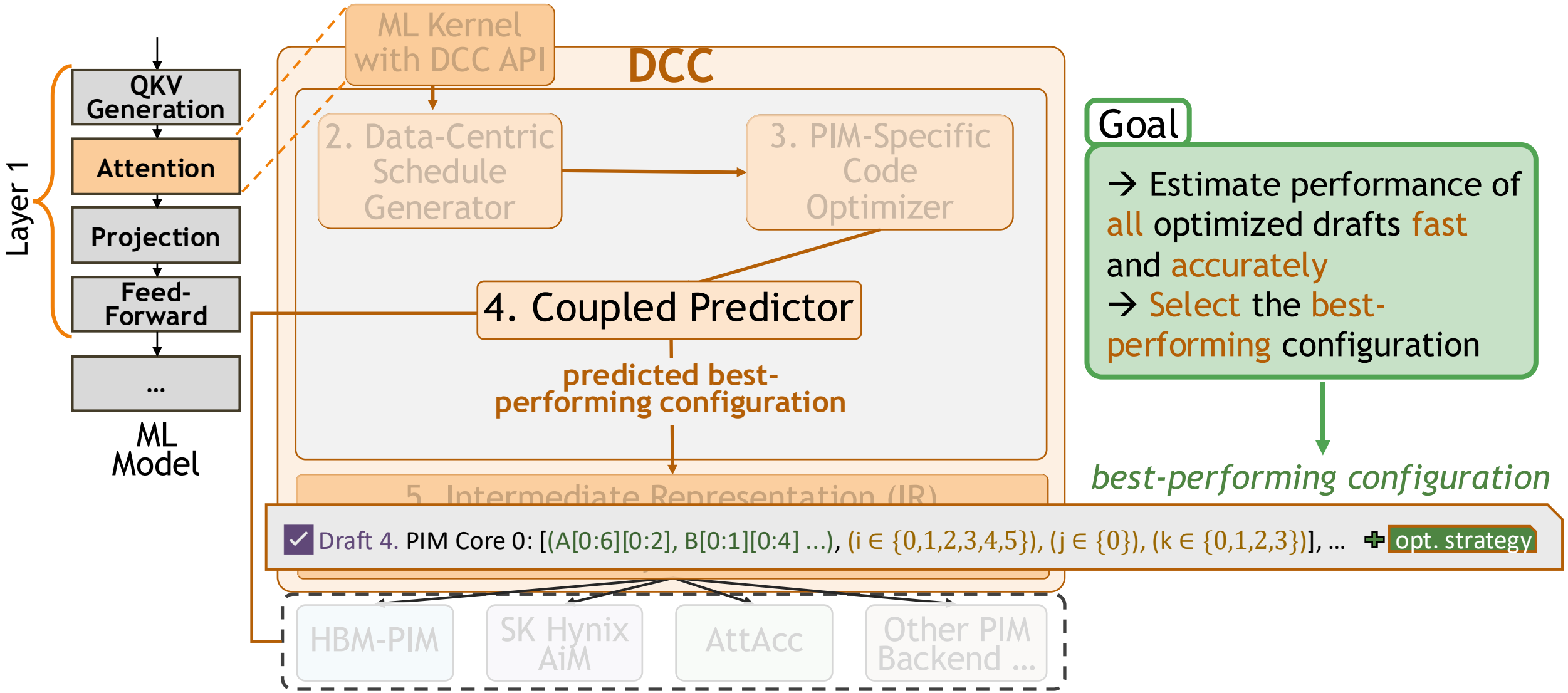
3. PIM-Specific Code Optimizer

→ optimizes performance of drafts & is extensible for further optimizations

- **Example: group-level broadcast commands in DCC**



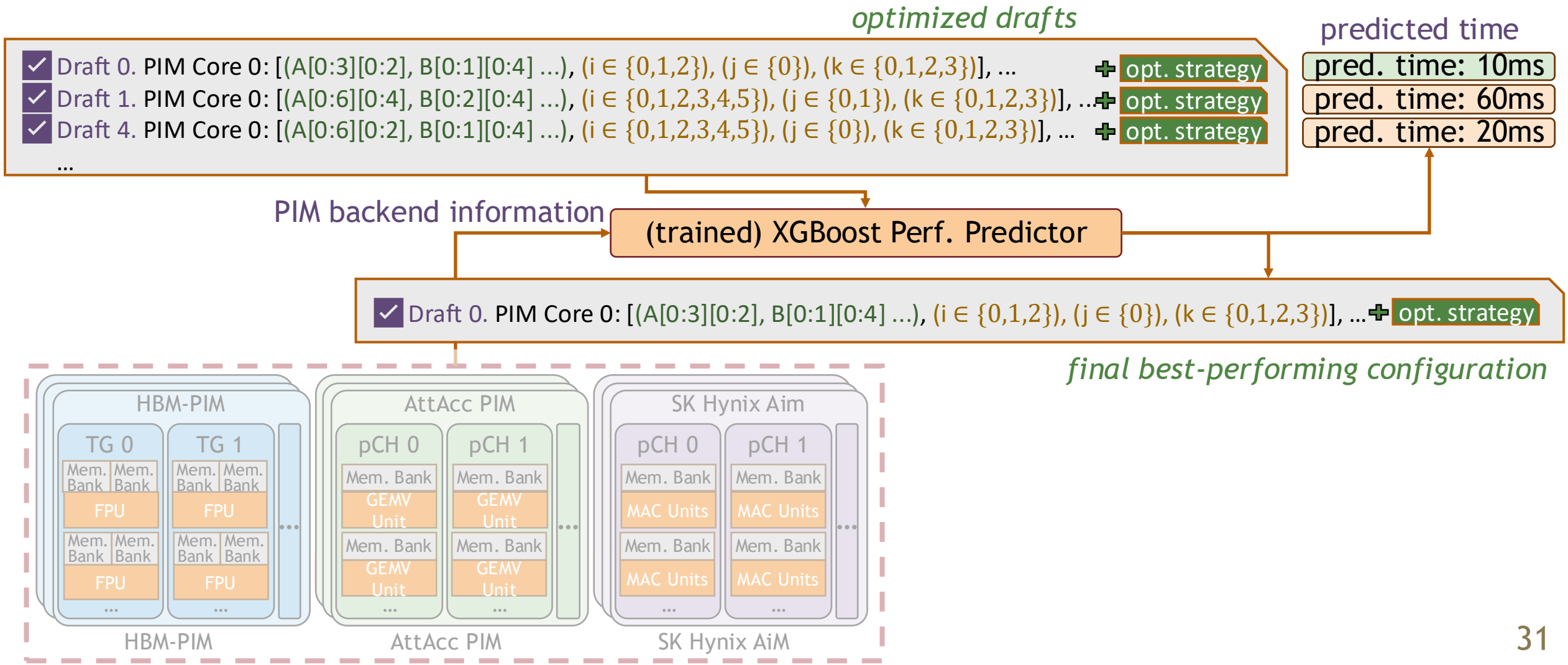
DCC Overview & 5 Key Components



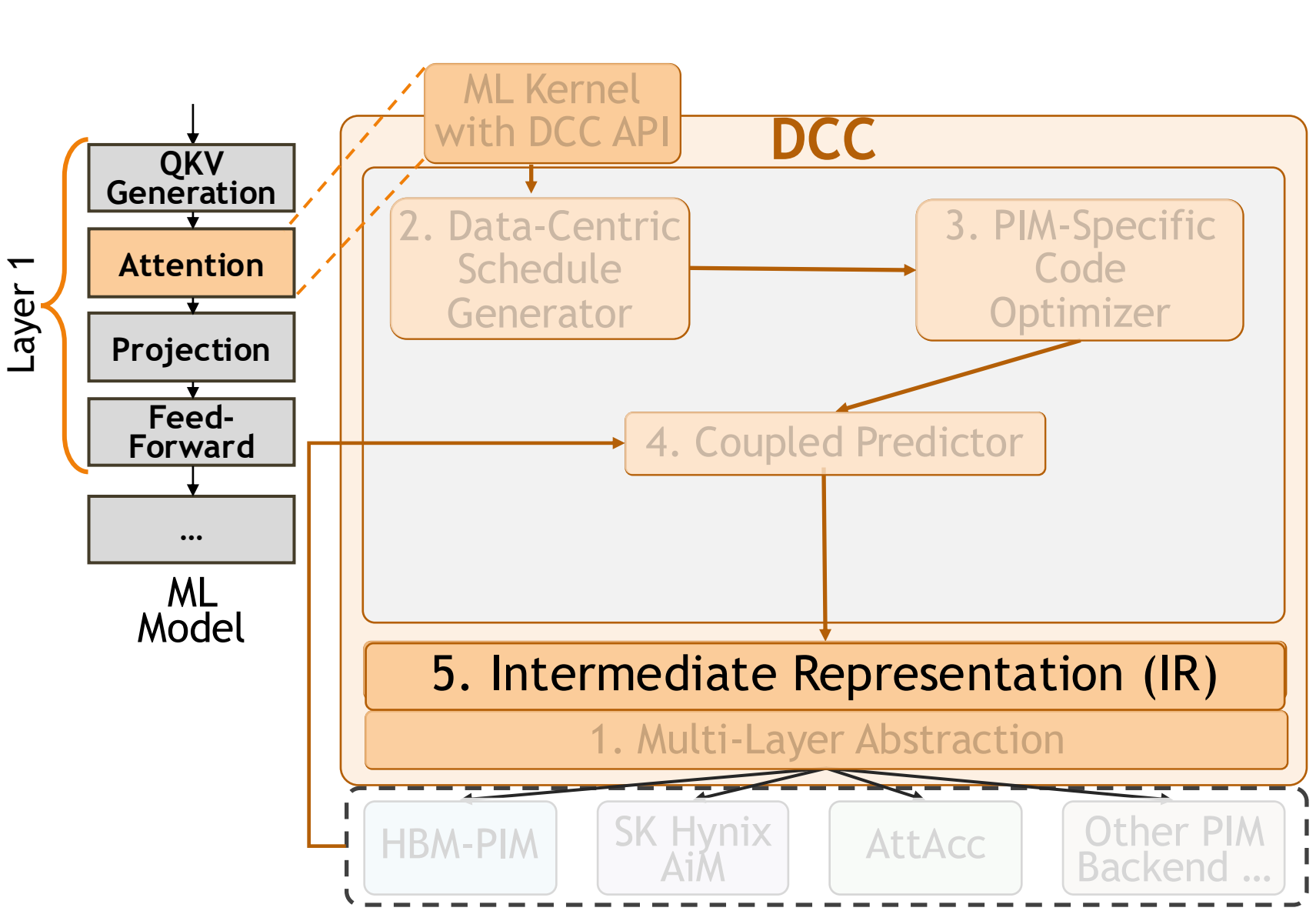
4. Coupled Predictor

→ 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

Data-centric DCC IR

```
@hardware_platform[GROUPS=2, CORES=4]
func @example(%a:<6x8xfp32>, %b:<4x4xfp32>): <6x8xfp32>
 ……
 complex_dimset %ds_j {
   %dim[0] = %a.dim[1] -> j,
   %dim[1] = %a.dim[1] -> j*2 [representative],
   %dim[2] = %c.dim[0] -> j
 }
 ……
 parallel_PIM (GROUPS, CORES) {
   loop (i, j) in (%ct_i, %ct_j, %ct_k) {
%a_core[%i, %j] = %a_core[%i+1, %j*2] + %b_core[%j][%k]
   }
 }
 %a_out = rearrange %a_core -> %a [to_host]
 ……
```

AttAcc PIM Instructions

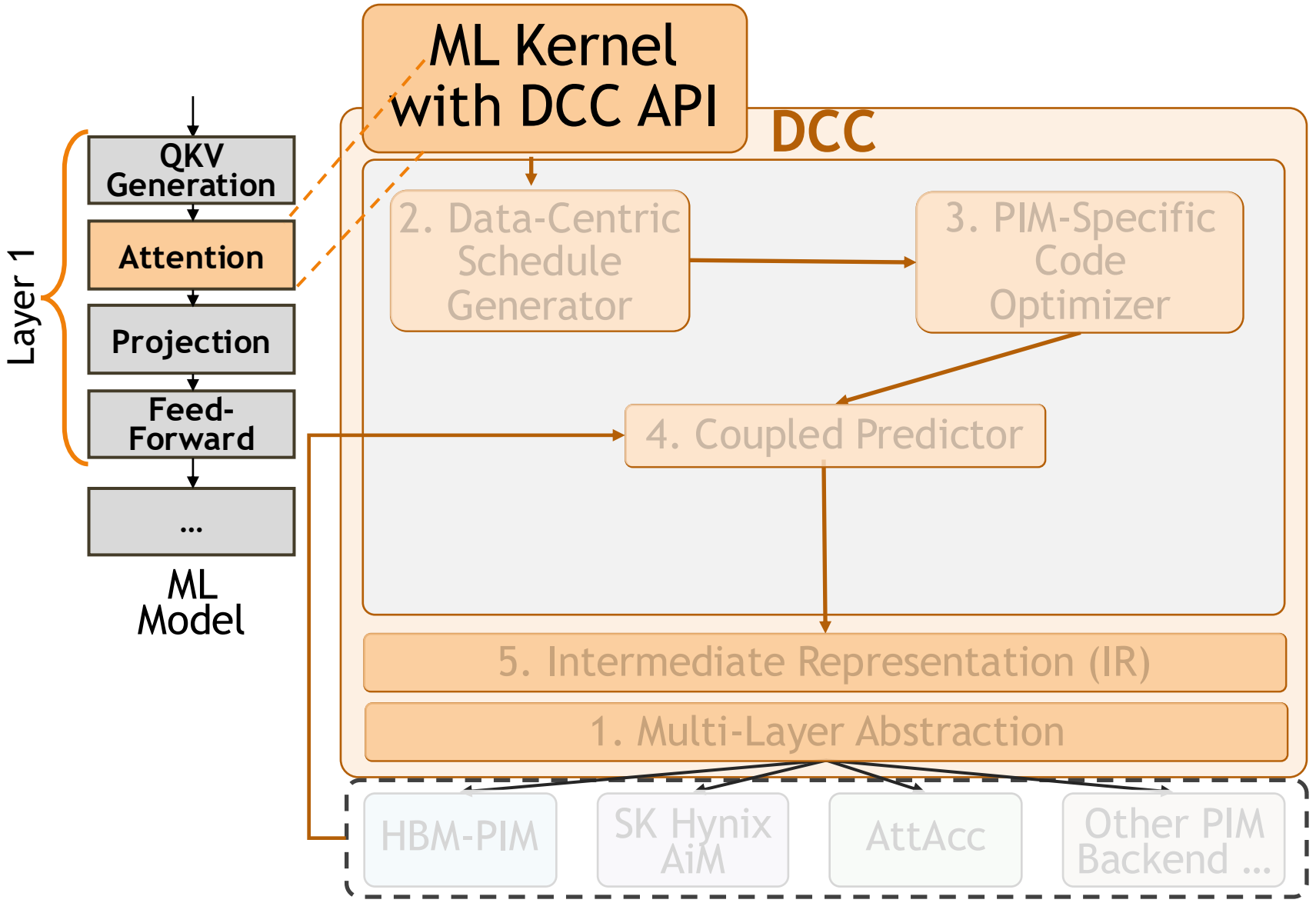
```
%reg_a = load %a_core[%i+1, %j*2]
%reg_b = load %b_core[%j][%k]
%reg_sum = ADD %reg_b, %reg_a
```

HBM-PIM Instructions

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

backend PIM instructions

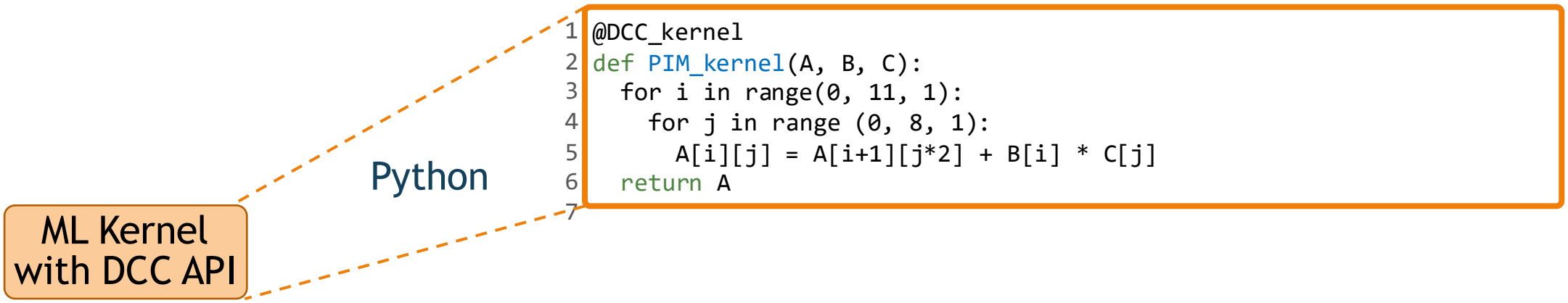
Easy-to-Use Python API



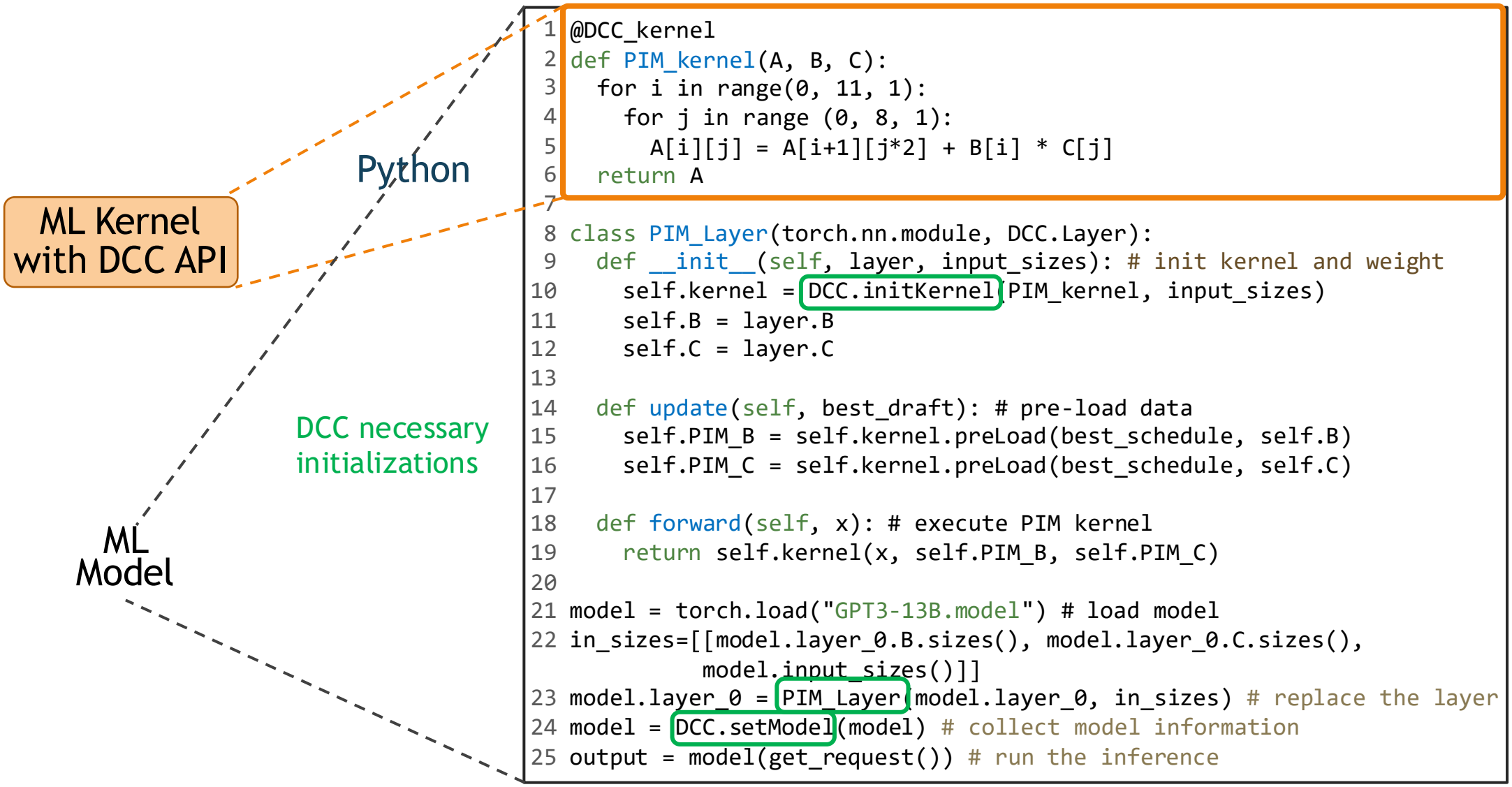
Goal

→ 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
```

Talk Outline

Background

Motivation

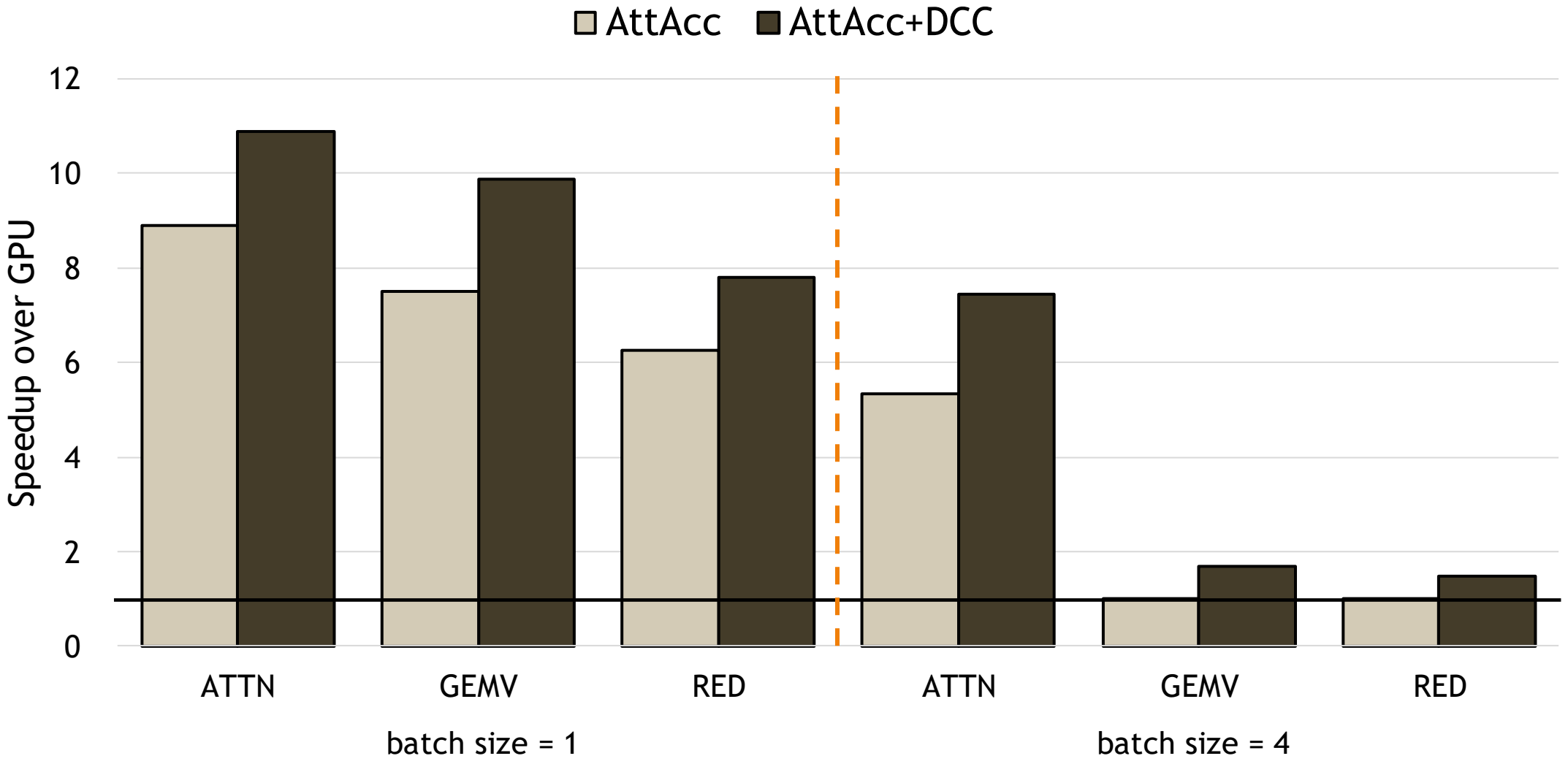
Our Design

Evaluation

Evaluation Methodology

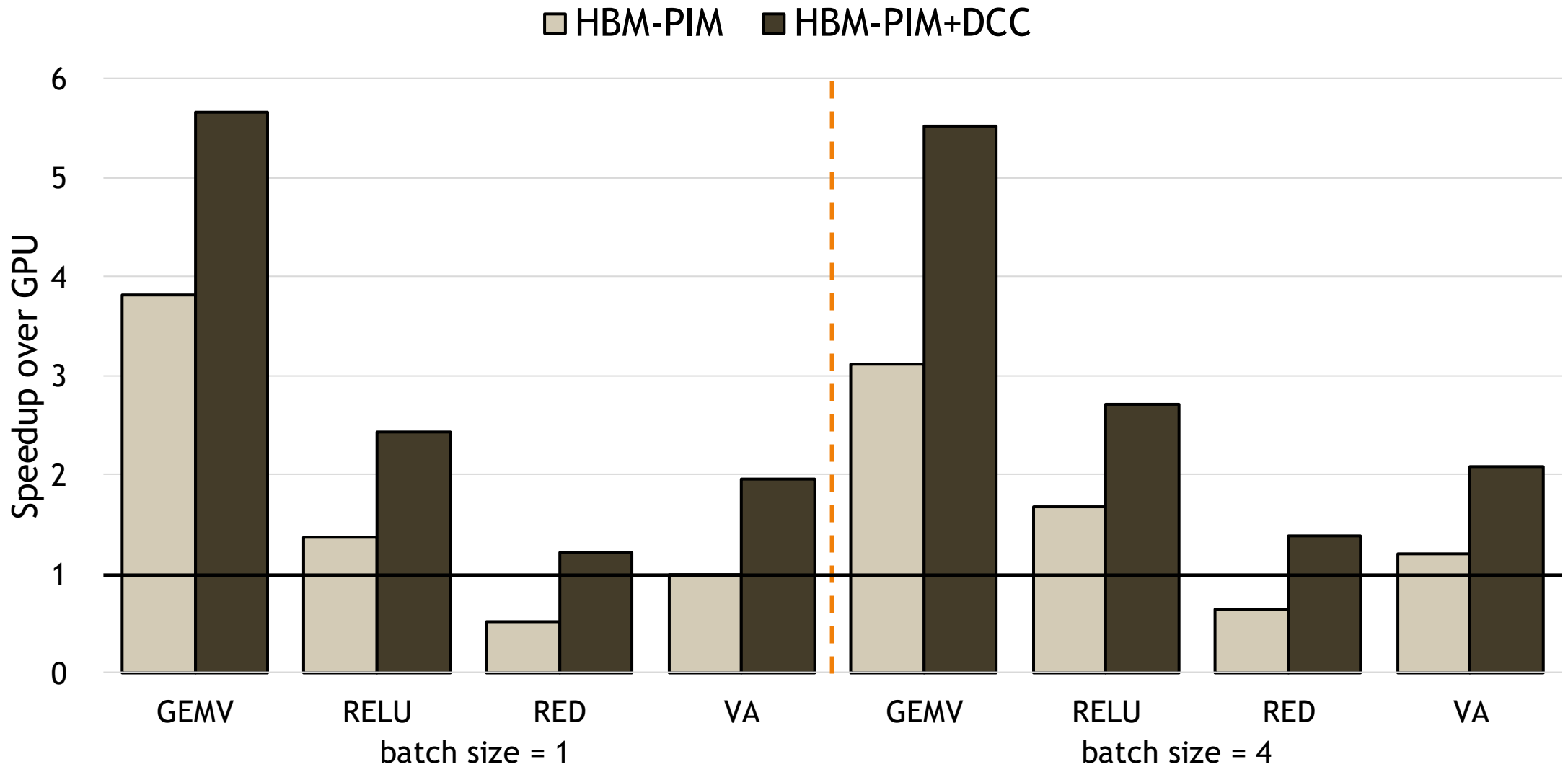
- Simulation Methodology:
  - Modified simulator using AttAcc [ASPLOS'24] and Ramulator 2.0 [IEEE CAL'23]
  - NVIDIA A100 GPU with 5 PIM-enabled HBM3 devices (80GB memory in total)
- Hardware: 2× PIM Backends:
  - AttAcc [ASPLOS'24]
  - HBM-PIM [ISCA'21]
- Workload: 5× ML Kernels & 3 ML models :
  - Kernel evaluation: GEMV, RED , ATTN, VA, RELU
  - End-to-end inference: 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

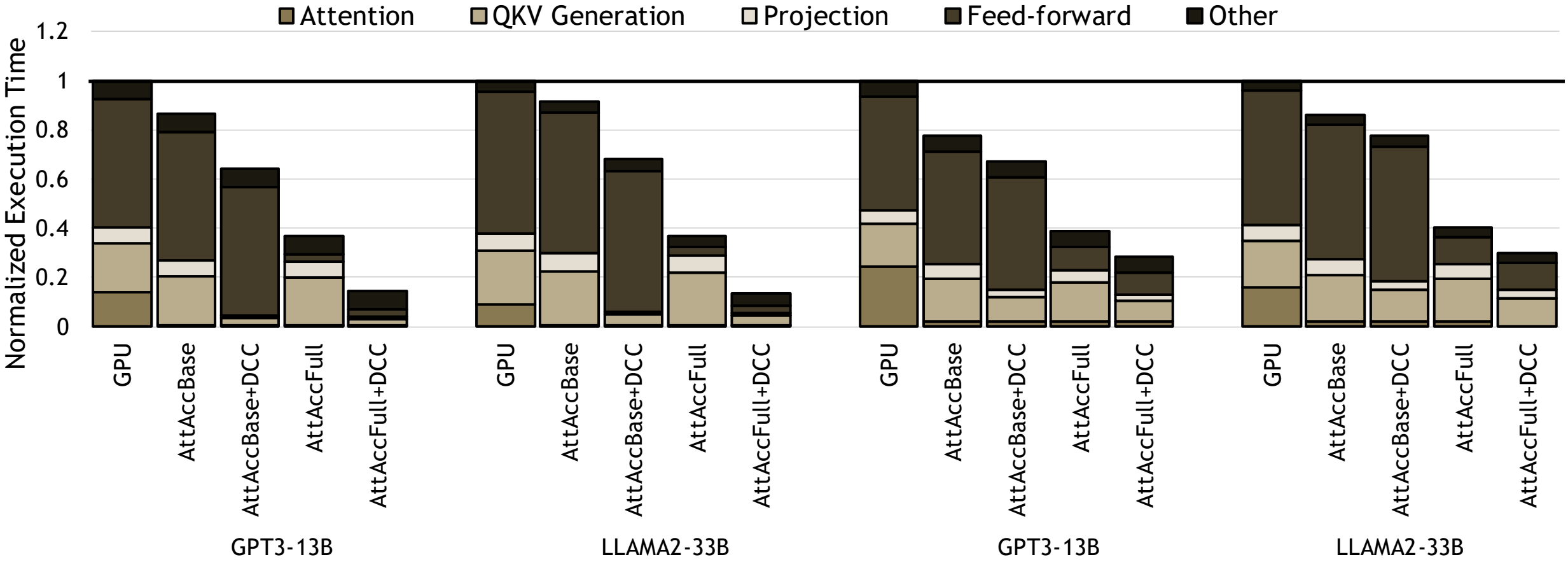
ML Kernel Performance on HBM-PIM PIM Backend



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

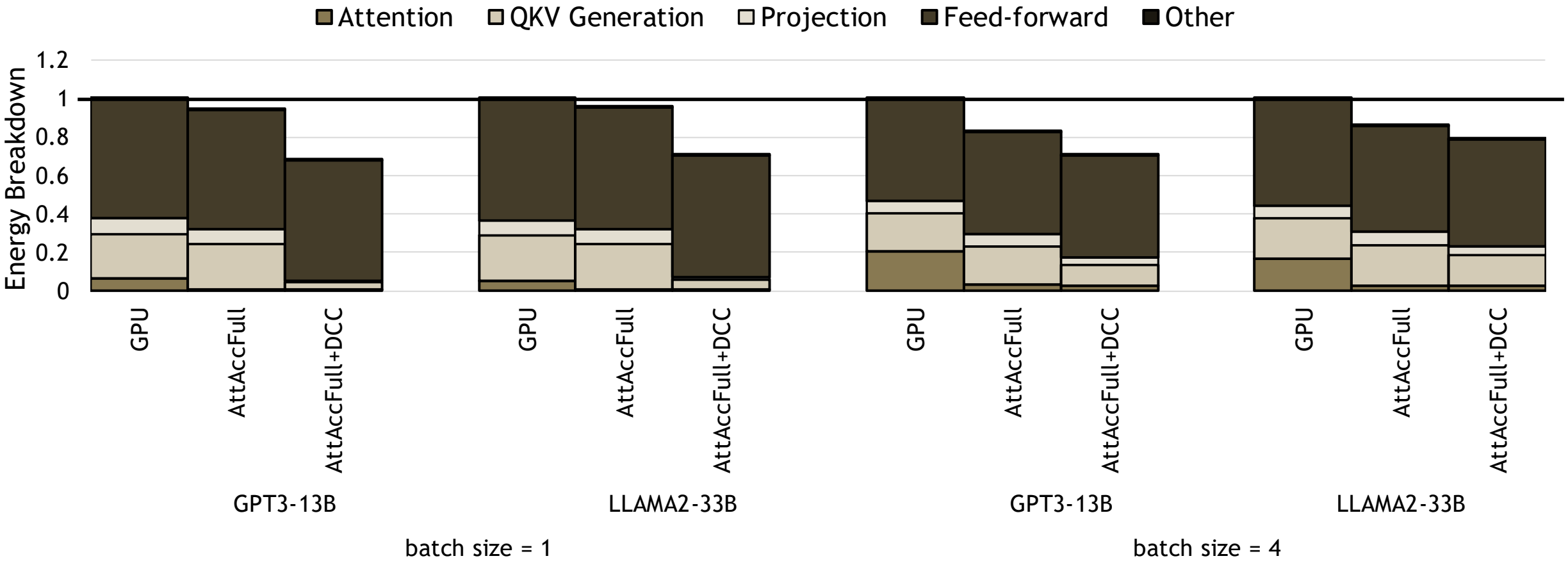
- $\text{AttAcc}_{\text{Base}}$: AttAcc PIM running attention on PIM cores
- $\text{AttAcc}_{\text{Full}}$: AttAcc PIM running both attention and a part of feed-forward on PIM cores



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

LLM Inference Energy Consumption on AttAcc PIM

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



DCC on AttAcc PIM improves LLM inference energy efficiency by up to 1.5× and 1.4× on average over GPU

More Evaluations in the Paper

- Scalability analysis
- Large-scale LLM inference with FP8 and FP16
- Energy consumption
- Compilation time analysis
- Effectiveness of pruning mechanisms
- Performance analysis of the DCC predictor
- LLM inference with dynamic tensor sizes
- ……

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:

- ✓ Support for multiple PIM backends
- ✓ Couples data and compute-loop partition strategies 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

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

GitHub

Paper



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

MCCSys Workshop, ISCA 2026

Raleigh, USA, June 28th 2026















Backup Slides