Rethinking AI Acceleration: From Data to Metadata-Centric Computing

2026.06.24

Gi-Ho Park, Chiwon Han
Sejong University

MPSoC-2026,
Victoria, BC, Canada

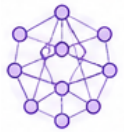
Models are getting bigger, and compression techniques are evolving

1. Models keep getting larger

Driven by better capability and performance

AlexNet
2012
~60M params

Transformer(BERT)
2018
~340M params



GPT-3
2020
175B params

GPT-4
2023
~ 1.8T params

GPT-4o / Llama3 / Gemini 1.5
2024~
10B ~ 1T+params

Bigger models

- → Higher accuracy
- → Higher compute & memory cost

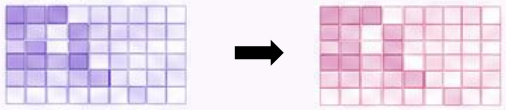
2. Compression techniques have advanced

Two major directions

Quantization (Lower precision)

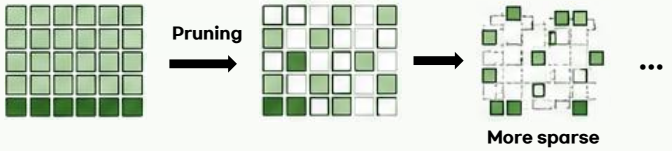
Represent weights/activations with fewer bits

Example: FP32 32-bit → FP16 16-bit → INT8 8-bit → INT4/FP4 4-bit → ...



Pruning / Sparsity (Remove unimportant data)

Remove less important weights or connections



Less data → Less compute & memory

3. However, these techniques come with a trade-off

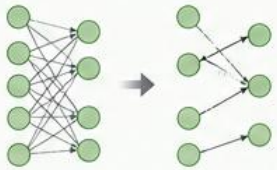
reduce computation and memory requirements VS inevitably reduce accuracy

Quantization

Lower precision → less information
→ Quantization Error

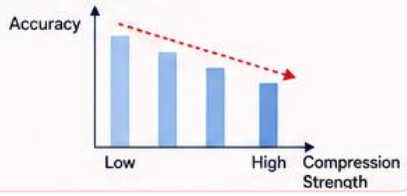
Pruning / Sparsity

Removing weights
→ Loss of useful information



Common Outcome

Accuracy degradation



Challenge

How to maintain high accuracy while reducing computation and memory footprint?

High accuracy requires fine-grained, multi-compression → complexity and irregularity is inevitable

Goal

Reduce Accuracy Loss with Compression

How to achieve it?

Fine-grained Compression
(block / channel / weight level)

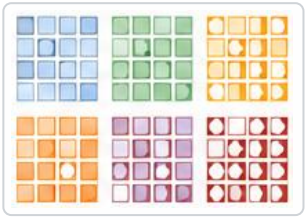
+

Multi-compression (mixed precision)

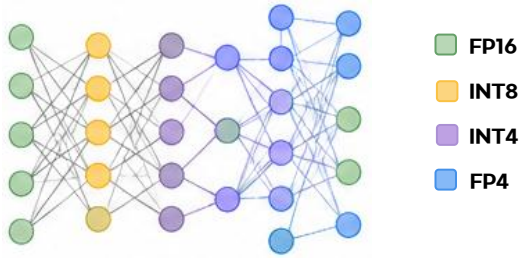
FP16 INT8 INT4 FP4 ...

Result: Complex Model Structure
(improves the accuracy-compression trade-off, have significant irregularity)

Varying sparsity per block



Multiple precisions in model



Not Hardware-Friendly

Unpredictable Memory Access

Unknown amount,
Unknown location

Workload Imbalnce

Varying # of non-zeros
→ load imbalance

Hard Accumulation Control

Irregular destinations
of partial sums

Hard Memory Hierarchy Control

Irregular data placement
across hierarchy

Achieving high accuracy often means introducing complexity that is not very hardware-friendly.

High accuracy requires fine-grained, multi-compression → complexity is inevitable → Metadata-guided Execution

Goal

Reduce Accuracy Loss with Compression

How to achieve it?

Fine-grained Compression
(block / channel / weight level)

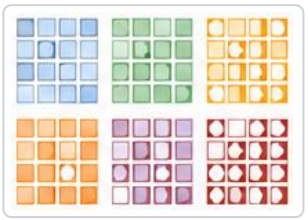
+

Multi-compression (mixed precision)

FP16 INT8 INT4 FP4 ...

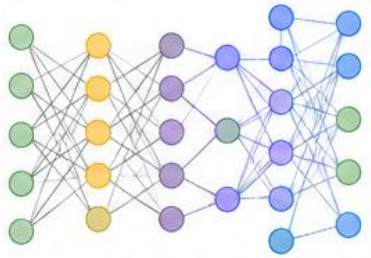
Result: Complex Model Structure (improves the accuracy-compression trade-off, have significant irregularity)

Varying sparsity per block



nnz: 8
nnz: 32
nnz: 64
nnz: 96

Multiple precisions in model



FP16
INT8
INT4
FP4

Not Hardware-Friendly

Unpredictable Memory Access

Unknown amount, Unknown location

Workload Imbalnce

Varying # of non-zeros → load imbalance

Hard Accumulation Control

Irregular destinations of partial sums

Hard Memory Hierarchy Control

Irregular data placement across hierarchy

Key is to control the complexity/irregularity using well-designed metadata (index)

Complex Model
(Fine-grained, Multi-compression)

FP16
INT8
INT4
FP4
...

Metadata (Index)

Position Info
(sparsity pattern)

Quantization Info
(precision per block/channel)

Address / Mapping Info
(dataflow control)

Efficient Execution (Hardware-Friendly)

- Predictable memory access
- Balanced computation
- Efficient accumulation
- Optimized memory hierarchy

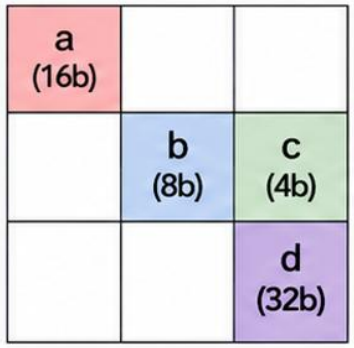
High Accuracy & Efficient Inference

What if we use metadata to help hardware understand and manage complexity? (Instead of treating metadata as an overhead)

→ This idea led us to metadata-guided execution

Metadata for Efficient Execution (MEE)
Providing Useful Information for Hardware via Metadata

Compressed Block (Data only)



Data alone is not enough!

- ? Which data should be loaded?
- ? How should each value be interpreted?
- ? How many operations are required?

Metadata for Efficient Execution (MEE)

Regular Size Index

Bitmap (Position)	1 0 0 0 1 1 0 0 1	9 bit (fixed)
Quantization Info	0 1 1	3 bit (fixed)
# of Non-zero (Count)	0 1 0 0	4 bit (fixed)
Quantized Data	a b c d	64 bit (variable)

Position index

1 0 0 0 1 1 0 0 1

- Transfer only required operands to memory using a bitmap
- Perform only required computations using a bitmap

Quantization info (Array size)

bit size

0 1 1

Compute block dimensions and memory footprint from array size and quantization bitwidth

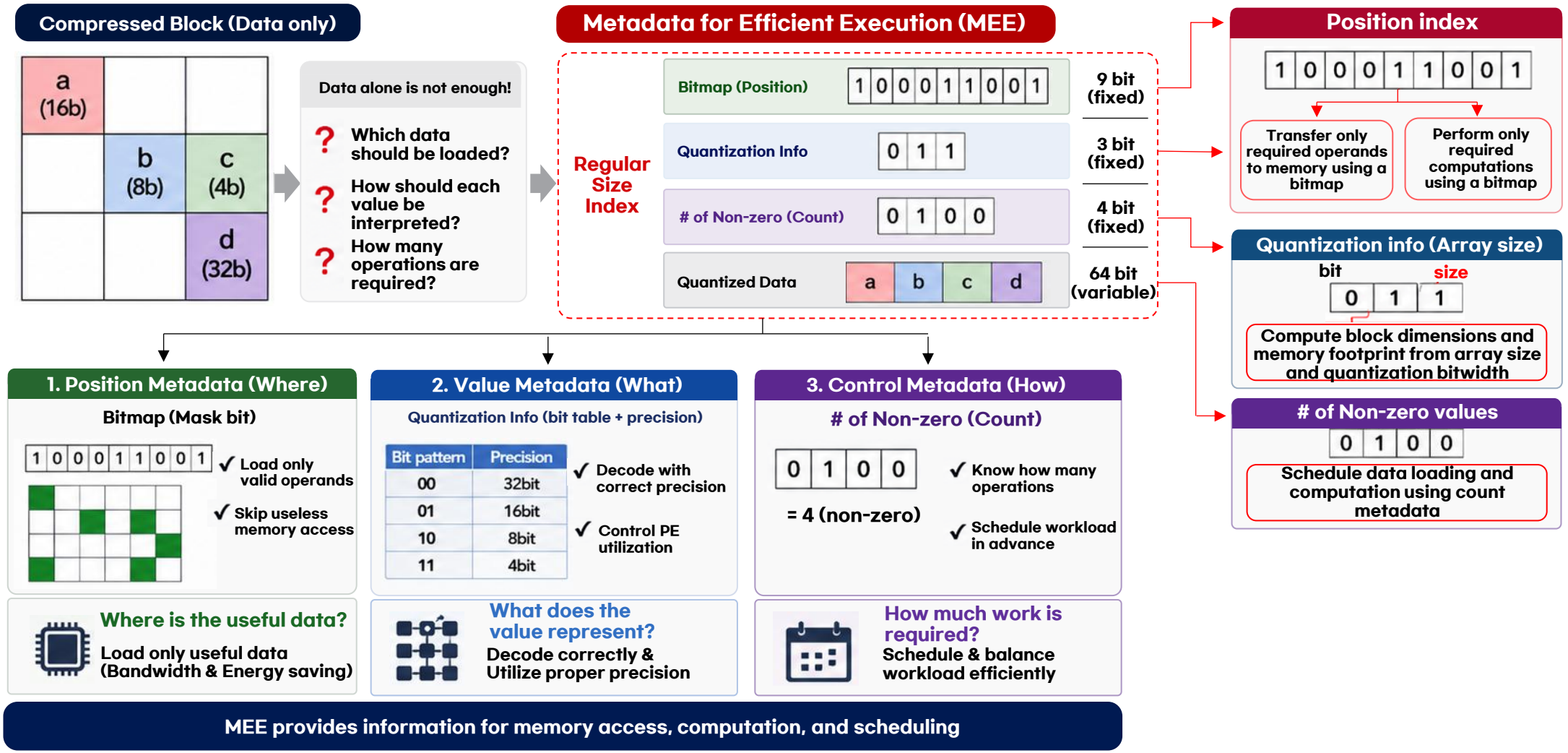
# of Non-zero values

0 1 0 0

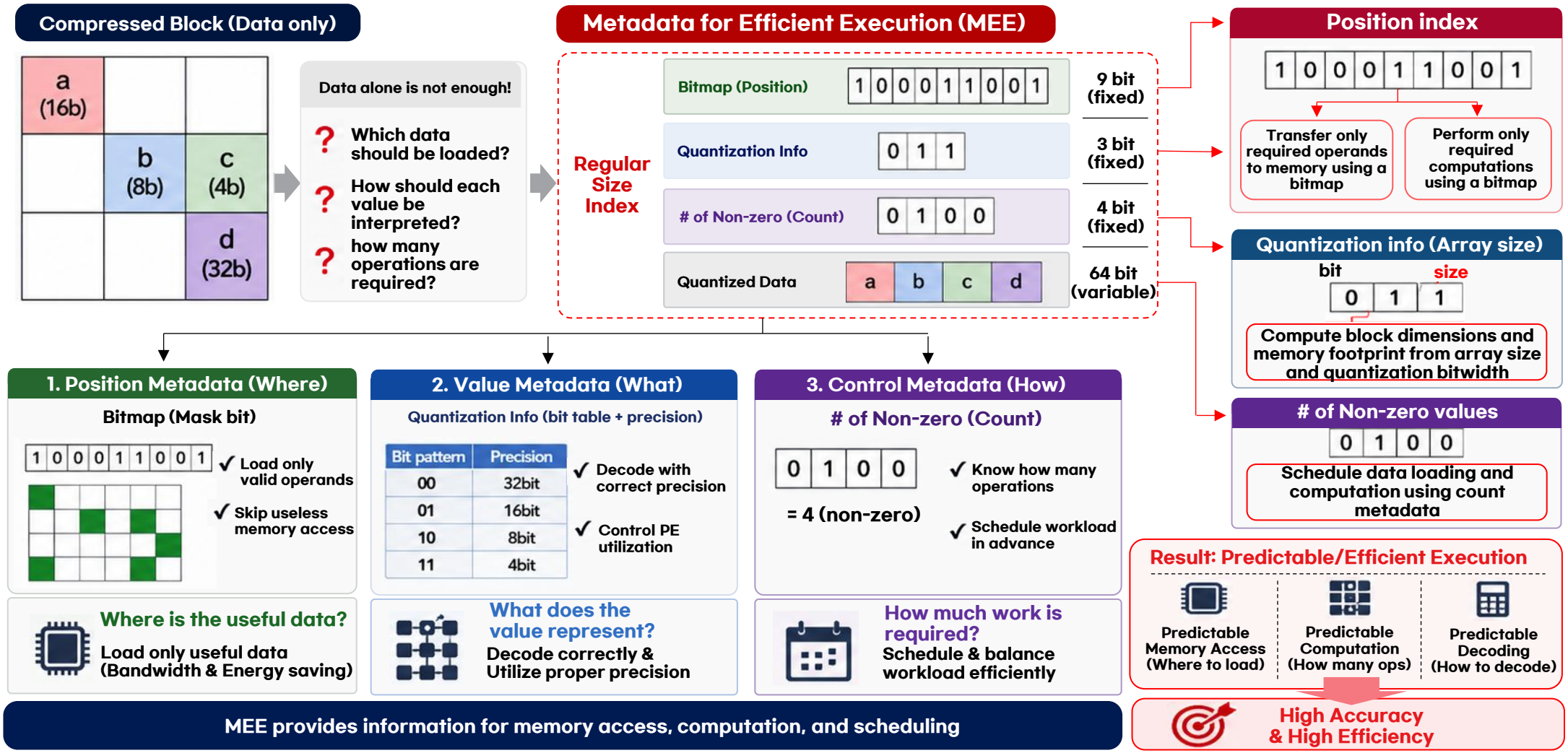
Schedule data loading and computation using count metadata

MEE provides information for memory access, computation, and scheduling through
- **a bitmap tells us where the non-zero values are**
- **quantization metadata tells us how to interpret the values**
- **the non-zero count tells us how much work needs to be done**

Metadata for Efficient Execution (MEE): Metadata for Memory Access, Computing, and Scheduling

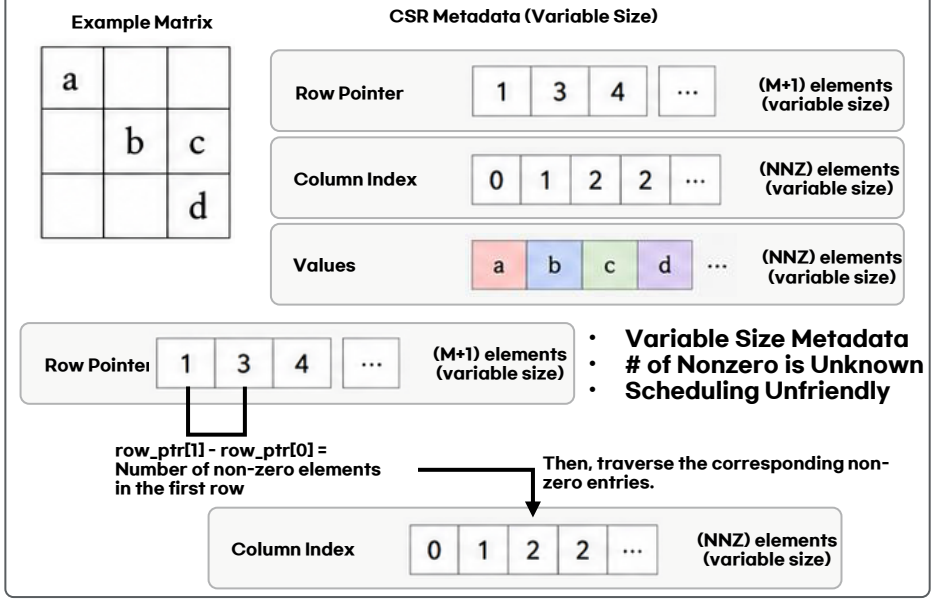


Metadata for Efficient Execution (MEE): Predictable Execution through Metadata



Fixed-Size Metadata itself for Efficient Processing Predictable and Regular Fetching for Metadata Itself

CSR (Compressed Sparse Row)



- Variable Size Metadata
- # of Nonzero is Unknown
- Scheduling Unfriendly

VS

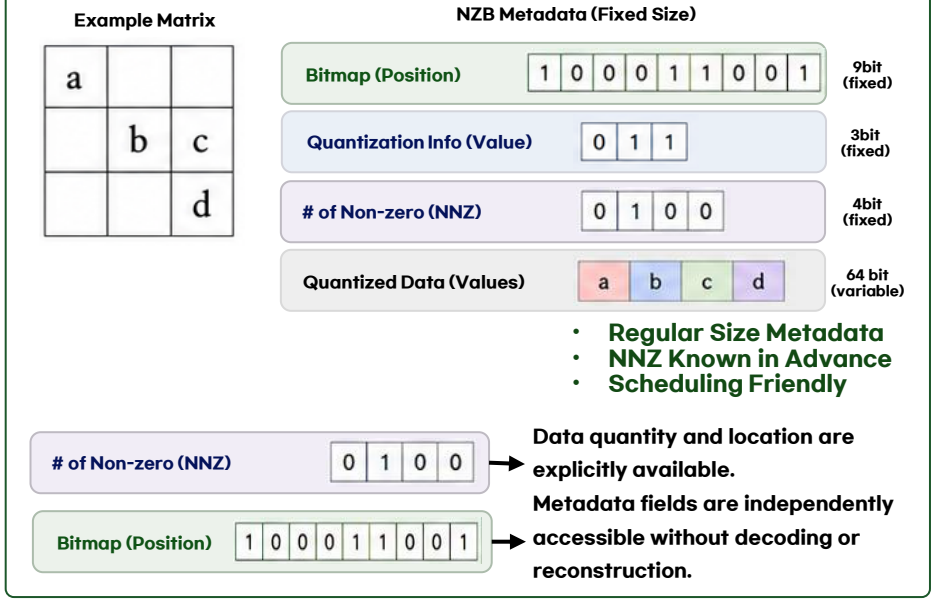
Metadata Size

(Length varies with sparsity)

NNZ information

(Total number of non-zero elements)

MEE (Bitmap + Quant + NNZ)



- Regular Size Metadata
- NNZ Known in Advance
- Scheduling Friendly

Data quantity and location are explicitly available.
Metadata fields are independently accessible without decoding or reconstruction.

MEE itself is regular. Unlike CSR, the metadata size does not change with sparsity. bitmap, quantization information, and NNZ count all have fixed sizes.

Fixed-Size Metadata for Efficient Processing Predictable Execution Enables High Efficiency

CSR (Compressed Sparse Row)

Example Matrix

a		
	b	c
		d

CSR Metadata (Variable Size)

Row Pointer	1 3 4 …	(M+1) elements (variable size)
Column Index	0 1 2 2 …	(NNZ) elements (variable size)
Values	a b c d …	(NNZ) elements (variable size)

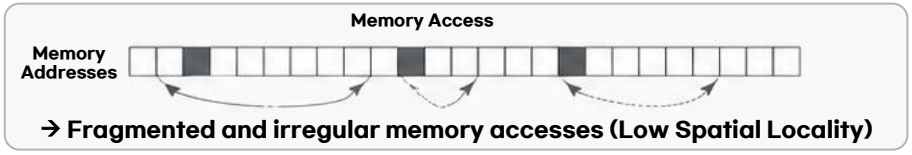
Row Pointer 1 3 4 … (M+1) elements (variable size)

row_ptr[1] - row_ptr[0] = Number of non-zero elements in the first row

Then, traverse the corresponding non-zero entries.

Column Index 0 1 2 2 … (NNZ) elements (variable size)

- Variable Size Metadata
- # of Nonzero is Unknown
- Scheduling Unfriendly



→ Fragmented and irregular memory accesses (Low Spatial Locality)

CSR leads to unpredictable execution with inefficiency

Difficult to predict data loading volume | Difficult to estimate computational workload | Inefficient scheduling and resource allocation

→ Low Utilization, High Energy, Accuracy Degradation

VS

- Metadata Size (Length varies with sparsity)
- NNZ information (Total number of non-zero elements)
- Scheduling information (Limited load/compute predictability)
- Memory Access pattern

MEE (Bitmap + Quant + NNZ)

Example Matrix

a		
	b	c
		d

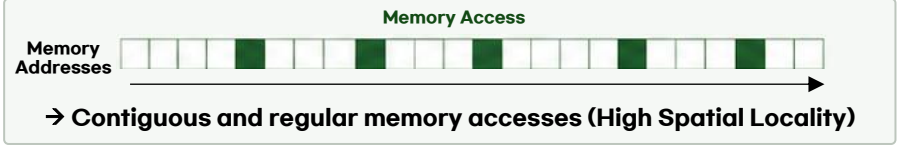
NZB Metadata (Fixed Size)

Bitmap (Position)	1 0 0 0 1 1 0 0 1	9bit (fixed)
Quantization Info (Value)	0 1 1	3bit (fixed)
# of Non-zero (NNZ)	0 1 0 0	4bit (fixed)
Quantized Data (Values)	a b c d	64 bit (variable)

- Regular Size Metadata
- NNZ Known in Advance
- Scheduling Friendly

# of Non-zero (NNZ) 0 1 0 0 → Data quantity and location are explicitly available.

Bitmap (Position) 1 0 0 0 1 1 0 0 1 → Metadata fields are independently accessible without decoding or reconstruction.



→ Contiguous and regular memory accesses (High Spatial Locality)

MEE enables predictable execution with high efficiency

Accurate Data Load Prediction | Accurate Computational Workload Prediction | Efficient Scheduling and Resource Allocation

→ High Utilization, Low Energy, High Accuracy

MEE leverages fixed-size metadata and provide explicit NNZ information to determine Where, What, and How Many before execution.

→ Predictable Memory Access, Efficient Computation, and Effective Scheduling

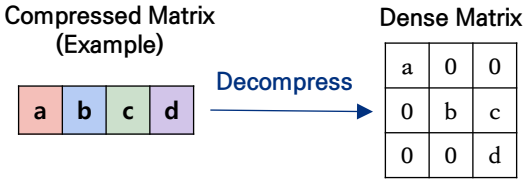
VS

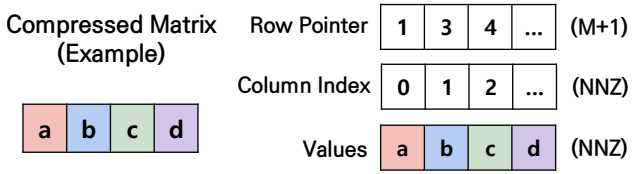
CSR only tells us where the non-zero values are.

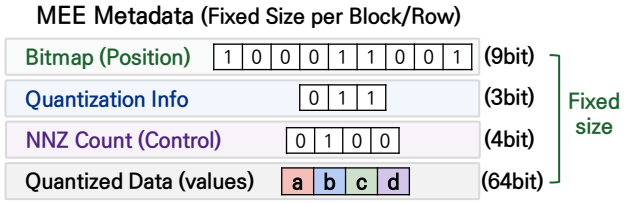
Three Approaches to Processing Compressed Matrices

Processing Compressed Matrices : From Decompression → Irregular Access → Regular Metadata Access

Data Representation	1. Conventional (Decompress & Compute)	2. CSR/COO/CSC (Compressed Sparse Fotmat)	3. MEE (New Metadata Format)
	Decompress to Dense, Then Compute	Compute on Compressed Data, But Irregular Access & Index Overhead	Compute on Compressed Data, With Regular Metadata Access







Three Approaches to Processing Compressed Matrices
Execution Flow Comparison for Compressed Matrices

Processing Compressed Matrices : From Decompression → Irregular Access → Regular Metadata Access

	1. Conventional (Decompress & Compute)	2. CSR/COO/CSC (Compressed Sparse Fotmat)	3. MEE (New Metadata Format)
Data Representation	**Decompress to Dense, Then Compute** Compressed Matrix (Example): a b c d Decompress → Dense Matrix: a 0 0 / 0 b c / 0 0 d	**Compute on Compressed Data, But Irregular Access & Index Overhead** Compressed Matrix (Example): a b c d Row Pointer: 1 3 4 ... (M+1) Column Index: 0 1 2 ... (NNZ) Values: a b c d (NNZ)	**Compute on Compressed Data, With Regular Metadata Access** MEE Metadata (Fixed Size per Block/Row) Bitmap (Position): 1 0 0 0 1 1 0 0 1 (9bit) Quantization Info: 0 1 1 (3bit) NNZ Count (Control): 0 1 0 0 (4bit) Quantized Data (values): a b c d (64bit) Fixed size
Hardware Execution Path	1. Load Compressed Data 2. Decompress (Expand to Dense) 3. Compute (Dense MAC) 4. Accumulate & Write Back Decompression Overhead (Extra Memory, Bandwidth, Energy)	1. Load Row Pointer: 1 3 4 ... 2. Read Row Info&Compute Row Range: Start = row_ptr[i], End = row_ptr(i+1) 3. Load Column Index (Irregular): 0 1 2 ... 4. Load Values Using Index: a b c d 5. Compute (Sparse MAC) 6. Accumulate & Write Back **Overheads** • Irregular index access (Col/Row Pointer) • Row range calculation (prefix-sum/subtraction) • Many small memory transactions • Complex control & scheduling	1. Load Metadata (Regular) 2. Decode Metadata (No Extra Calc.) 3. Load Values (Sequential): a b c d 4. Compute (Sparse MAC) 5. Accumulate & Write Back **Benefits** • Regular, fixed-size metadata access • No index deference per element • No row-range computation • Sequential data access • Simple control & easy scheduling

Three Approaches to Processing Compressed Matrices Regular Metadata Enables Predictable Execution

Processing Compressed Matrices : From Decompression → Irregular Access → Regular Metadata Access

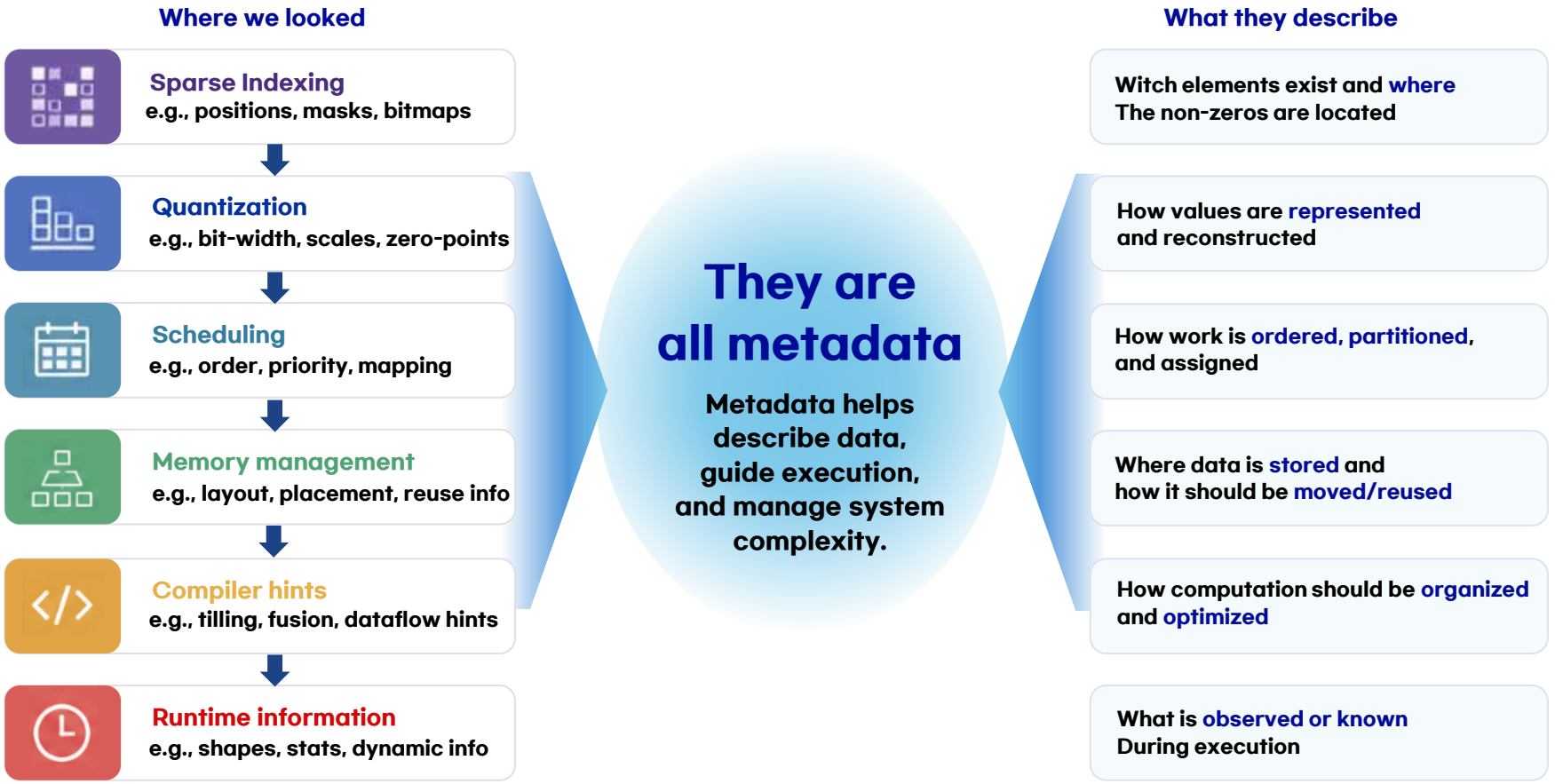
	1. Conventional (Decompress & Compute)	2. CSR/COO/CSC (Compressed Sparse Fotmat)	3. MEE (New Metadata Format)
Data Representation	**Decompress to Dense, Then Compute** Compressed Matrix (Example): a b c d Decompress → Dense Matrix: a 0 0 / 0 b c / 0 0 d	**Compute on Compressed Data, But Irregular Access & Index Overhead** Compressed Matrix (Example): a b c d Row Pointer: 1 3 4 ... (M+1) Column Index: 0 1 2 ... (NNZ) Values: a b c d (NNZ)	**Compute on Compressed Data, With Regular Metadata Access** MEE Metadata (Fixed Size per Block/Row) Bitmap (Position): 1 0 0 0 1 1 0 0 1 (9bit) Quantization Info: 0 1 1 (3bit) NNZ Count (Control): 0 1 0 0 (4bit) Quantized Data (values): a b c d (64bit) Fixed size
Hardware Execution Path	1. Load Compressed Data 2. Decompress (Expand to Dense) 3. Compute (Dense MAC) 4. Accumulate & Write Back Decompression Overhead (Extra Memory, Bandwidth, Energy)	1. Load Row Pointer: 1 3 4 ... 2. Read Row Info&Compute Row Range: Start = row_ptr[i], End = row_ptr(i+1) 3. Load Column Index (Irregular): 0 1 2 ... 4. Load Values Using Index: a b c d 5. Compute (Sparse MAC) 6. Accumulate & Write Back **Overheads** • Irregular index access (Col/Row Pointer) • Row range calculation (prefix-sum/subtraction) • Many small memory transactions • Complex control & scheduling	1. Load Metadata (Regular) 2. Decode Metadata (No Extra Calc.) 3. Load Values (Sequential): a b c d 4. Compute (Sparse MAC) 5. Accumulate & Write Back **Benefits** • Regular, fixed-size metadata access • No index deference per element • No row-range computation • Sequential data access • Simple control & easy scheduling
Characteristics	• Must restore to dense format • Additional decompression cost (time, energy, memory) • High memory traffic (read compressed + write dense) • Simple compute, but inefficient overall	• Compute on compressed data (no decompression) • But requires index access for every non-zero element • Row pointer calculation needed for each row • Irregular memory access →low locality, hard to predict • Additional control logic and complex scheduling	• Compute on compressed data without decompression • Metadata has fixed size and regular structure • No per-element idx access, no row range calculation • Sequential memory access → high locality • Highly predictable, simple control, efficient scheduling
Key Issue	Decompression overhead and unnecessary memory traffic for zeros	Irregular metadata access and index computation cause low predictability and inefficiency	Regular metadata access enables predictable, efficient and high-performance execution

Summary

1) Decompression cost is high → 2) Still compressed, but irregular access & index computation hurt performance → 3) Compressed + Regular Metadata → Predictable, Efficient and Scalable

What surprised us?

As we continued working on compression-related research, including sparse representations and quantization, we started noticing a common pattern.



Across the entire stack-from model to hardware to runtime
We are already surrounded by metadata.

What is Metadata?

Metadata is information ABOUT data that that helps efficient and correct execution.

Metadata Type	What it Describes	Example	Illustration (Example)
Sparse Indexing Metadata	Where non-zero values exist	CSR / COO / CSC (Index Format) (e.g., CSR)	Sparse Matrix: 1 0 0 2 0 / 0 3 0 0 0 / 4 0 5 0 6 → CSR Representation: Values 1 2 3 4 5 6 (non-zero values); Col Indices 0 3 1 0 2 4 (column indices); Row Ptr 0 2 3 6 (row pointers)
Quantization Metadata	How values should be interpreted	Quantization (Scale, Zero-point, Bit-width)	Quantization parameters for a block/channel: Scale = 0.03125 | Zero-point = 128 | Bit-width = 4-bit
Control Metadata	How work should be assigned and executed.	NNZ Count (Non-Zero Count)	Number of non-zero elements: 128, 64, 256, ...
Scheduling Metadata	When, where, and in what order work should be executed	Task Mapping (Priority, Mapping, Dependency)	Mapping of tasks to processing elements (PEs): Task 1 → PE3; Task 2 → PE7; Task 3 → PE1; Task 4 → PE5; ...
Memory Metadata	Where determine where data resides and how it should move	Placement / Location (Reuse, Layout)	Data placement across memory hierarchy: DRAM → SRAM → Buffer → Register

Metadata is used in many area as essential information for describing WHERE data resides, WHAT it represents, and HOW computation proceeds.

Metadata for Efficient Execution in AI Systems
From Overhead to First-Class Design Object

Position Metadata (Where)

Describes where data is

CSR | COO | MEE | ...

+

Value Metadata (What)

Describes what the data means or how it should be interpreted.

Scale | Zero-point | Outlier | ...

+

Control Metadata (How)

Describes how execution should proceed.

Dependency | Schedule | Placement | ...

=

Metadata-Centric Computing

Treating metadata as a first-class design object.

Metadata describes how data is accessed, interpreted, and executed.

This perspective motivates us to think about Metadata-Centric Computing.

What makes metadata effective?
3 simple design principles learned from MEE design

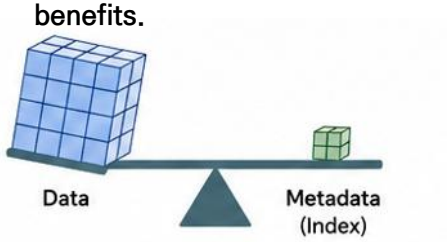
Metadata is stored together with the data

Compressed & Quantized Data + Metadata (Index)

Good metadata enables efficient, predictable, and hardware-friendly execution.

1 Metadata should be much smaller than the data itself.

Metadata overhead can outweigh its benefits.



Size Constraint

|Metadata| ≪ |Data|

Example

Quantization (INT4)

Data (weights)	100%
Metadata (scale, zp)	~1–3%

Pruning (sparse)

Data (non-zero values)	100%
Metadata (indices/pattern)	~5–20%

2 Accessed before actual data, so, it should be predictable and regular

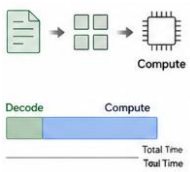
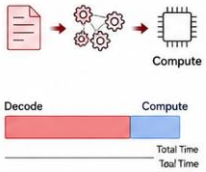
Irregular metadata structures make prefetching difficult and introduce additional overhead.

- ✓ Must be accessed before data
- ✓ Size and number should be predictable
- ✓ Regularity enables efficient prefetching, scheduling and data access

Good (Predictable)	Bad (Irregular)
Fixed-size Fixed-count per unit	Varying-size Varying-count per unit
	
• Easy to schedule • Easy to prefetch	• Hard to schedule • Unpredictable access

3 Metadata decoding should be simple

If the metadata requires excessive processing, the decoding overhead may become larger than the computation itself.

Good (Low-complexity decode)	Bad (High-complexity decode)
	
Decode overhead ≪ Compute	Decode overhead ≥ Compute

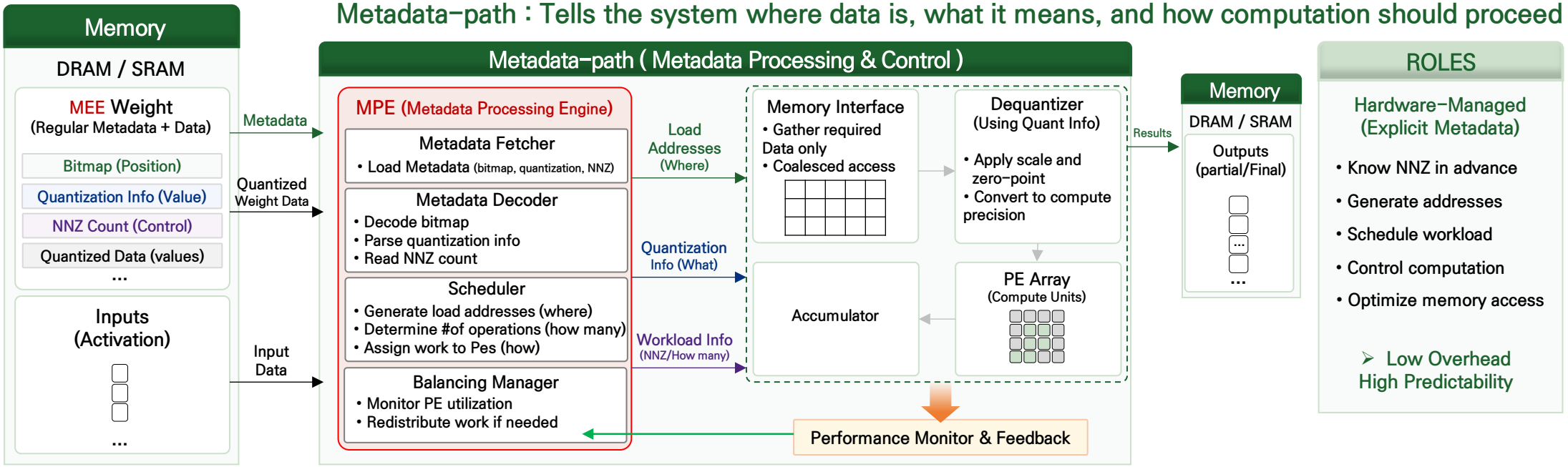
Rule of Thumb

- Simple and fast to decode
- Low memory access for decoding
- Limited branching and irregularity

MEE (with bitmap structure) is compact, fixed-size, predictable, and simple to decode

⇒ These principles extend beyond MEE, may serve as useful guidelines for designing metadata structures in many AI systems.

One Possible Metadata-Centric Accelerator Architecture



key idea is to separate metadata processing from data processing.

Instead of treating metadata as a passive byproduct, explicitly process metadata and use it to guide execution.

Metadata Processing Engine (MPE) reads metadata first, extracts useful information, and helps orchestrate execution

One Possible Metadata-Centric Accelerator Architecture
Metadata-Guided Execution Flow

Metadata-path : a guide map for execution

Memory
DRAM / SRAM
NZB Weight
(Regular Metadata + Data)
- Bitmap (Position)
- Quantization Info (Value)
- NNZ Count (Control)
- Quantized Data (values)
- ...

Inputs
(Activation)
...

Metadata
Quantized Weight Data
Input Data

Metadata-path (Metadata Processing & Control)

MPE (Metadata Processing Engine)
- Metadata Fetcher
  - Load Metadata (bitmap, quantization, NNZ)
- Metadata Decoder
  - Decode bitmap
  - Parse quantization info
  - Read NNZ count
- Scheduler
  - Generate load addresses (where)
  - Determine #of operations (how many)
  - Assign work to Pes (how)
- Balancing Manager
  - Monitor PE utilization
  - Redistribute work if needed

Load Addresses (Where)
Quantization Info (What)
Workload Info (NNZ/How many)

Memory Interface
- Gather required Data only
- Coalesced access

Dequantizer
(Using Quant Info)
- Apply scale and zero-point
- Convert to compute precision

Accumulator

PE Array
(Compute Units)

Performance Monitor & Feedback

Results

Memory
DRAM / SRAM
Outputs
(partial/Final)
...

ROLES

Hardware-Managed
(Explicit Metadata)
- Know NNZ in advance
- Generate addresses
- Schedule workload
- Control computation
- Optimize memory access

➢ Low Overhead
High Predictability

Key Difference

Conventional (CSR/COO, etc.)	Data-Driven, CPU-Controlled Metadata is implicit or absent → CPU computes & controls everything
MEE (Bitmap+Quant+NNZ)	Metadata-Driven, Hardware-Managed Metadata is implicit or regular → Metadata guides & orchestrates execution

How Metadata Enables Predictable Execution

Predictable Memory Access Patterns (Where)
- Bitmap → Know locations
- Only necessary data is loaded
- Coalesced access

Predictable Computation (What / How many)
- Quant Info → Know precision
- NNZ Count → Know *# of ops*
- Workload known in advance

Predictable Scheduling (How)
- Scheduler assigns work to PEs
- Balanced execution
- High utilization

MEE Metadata-path Realizes

Performs CPU-like control-path functions within the accelerator hardware

Enables predictable, metadata-driven high-efficiency acceleration

Metadata as a guide map for execution to help determine where to load, what to compute, and how many operations should be performed

1. Position (Bitmap) → Where

1 0 0 0 1 1 0 0 1 — Generate Addresses (valid data)

Valid (load) Invalid (skip)

Reduce memory traffic & energy

2. Quantization Info → What

0 1 1

bit-width	Scale	Zero-point
4bit	0.125	−7

Provide Decode & Precision information

Correctly decode values & select proper precision

3. Count (NNZ) → How many

0 1 0 0 (= 4) — Determine *# of operations*

Total non-zero = 4

Schedule workload & balance PEs

4. Execution Flow

[1] MPE reads metadata
[2] Generate load addresses & schedule
[3] Load required data from memory
[4] Decode, compute, accumulate in PEs
[5] Write back results

Predictable, efficient, and high-utilization execution

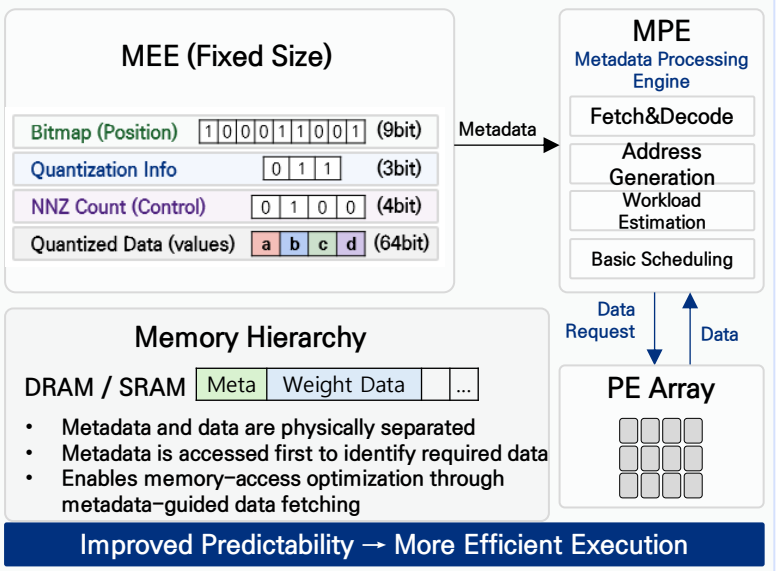
Outcome
- High PE Utilization
- Low Memory Traffic
- Low Energy
- High Accuracy

Metadata helps determine where to load, what to compute, and how many operations should be performed,
→ Predictable Execution with High Efficiency and Accuracy

Baseline Metadata-Centric Execution Architecture

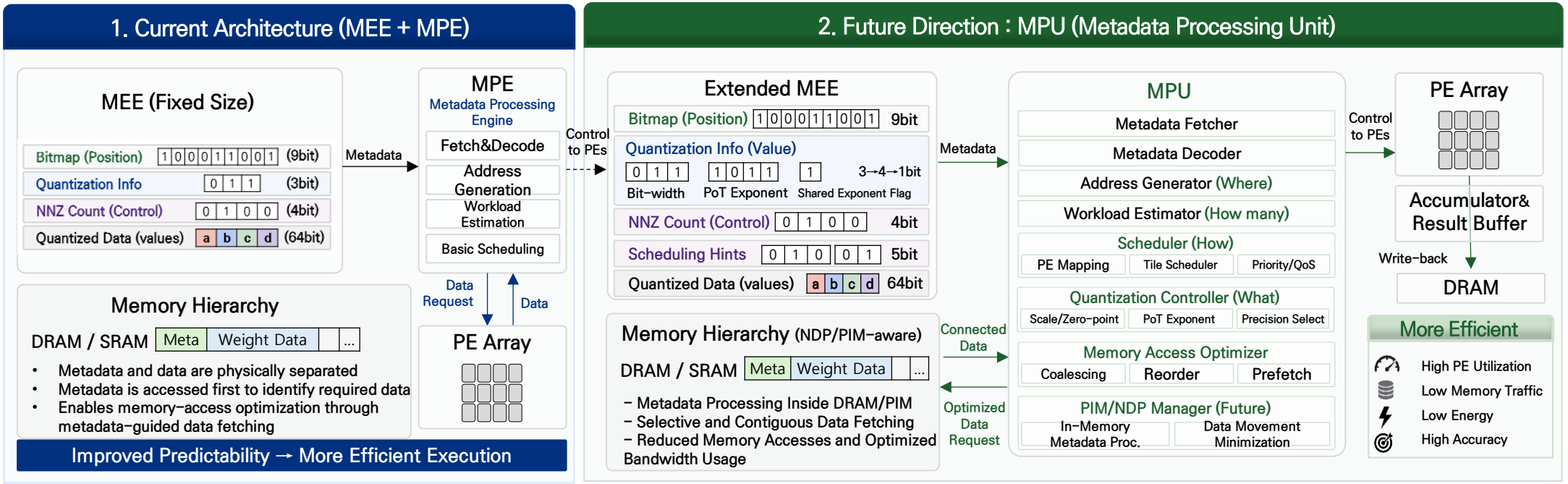
From MEE to MPU : Metadata-Centric Computing for Predictable and Efficient AI Acceleration

1. Current Architecture (MEE + MPE)



Toward Metadata-Centric Computing with MPU

From MEE to MPU : Metadata-Centric Computing for Predictable and Efficient AI Acceleration



From Metadata Processing to Metadata-Centric Computing

From MEE → MPU → Metadata-Centric Computing: A new computing paradigm? – an open question

1. Current Architecture (MEE + MPE)

MEE (Fixed Size)

Bitmap (Position)	1 0 0 0 1 1 0 0 1	(9bit)
Quantization Info	0 1 1	(3bit)
NNZ Count (Control)	0 1 0 0	(4bit)
Quantized Data (values)	a b c d	(64bit)

Metadata

MPE
Metadata Processing Engine
- Fetch&Decode
- Address Generation
- Workload Estimation
- Basic Scheduling

Control to PEs

Data Request ↓ ↑ Data

PE Array

Memory Hierarchy

DRAM / SRAM: Meta | Weight Data | ...

- Metadata and data are physically separated
- Metadata is accessed first to identify required data
- Enables memory-access optimization through metadata-guided data fetching

Improved Predictability → More Efficient Execution

2. Future Direction : MPU (Metadata Processing Unit)

Extended MEE

Bitmap (Position)	1 0 0 0 1 1 0 0 1	9bit
Quantization Info (Value)	0 1 1 / 1 0 1 1 / 1	3→4→1bit
	Bit-width / PoT Exponent / Shared Exponent Flag	
NNZ Count (Control)	0 1 0 0	4bit
Scheduling Hints	0 1 0 0 1	5bit
Quantized Data (values)	a b c d	64bit

Metadata

MPU
- Metadata Fetcher
- Metadata Decoder
- Address Generator (Where)
- Workload Estimator (How many)
- Scheduler (How): PE Mapping | Tile Scheduler | Priority/QoS
- Quantization Controller (What): Scale/Zero-point | PoT Exponent | Precision Select
- Memory Access Optimizer: Coalescing | Reorder | Prefetch
- PIM/NDP Manager (Future): In-Memory Metadata Proc. | Data Movement Minimization

Control to PEs

PE Array

Accumulator& Result Buffer

Write-back ↓ DRAM

Memory Hierarchy (NDP/PIM-aware)

DRAM / SRAM: Meta | Weight Data | ...

- Metadata Processing Inside DRAM/PIM
- Selective and Contiguous Data Fetching
- Reduced Memory Accesses and Optimized Bandwidth Usage

Connected Data →
← Optimized Data Request

More Efficient
- High PE Utilization
- Low Memory Traffic
- Low Energy
- High Accuracy

Why CSR is inefficient (vs. MEE)

Row Pointer	1	3		4	... (M+1)
Column Index	0	1	2	2	... (NNZ)
Values	a	b	c	d	... (NNZ)

Problems
- No Prior Knowledge of Workload Size
- Poor Memory Access Locality
- Complex Metadata Processing
- Variable-Length Metadata Structure

MEE Advantages
- ✓ Fixed-size metadata enables efficient metadata access
- ✓ Explicitly stores request-related information such as NNZ
- ✓ Provides operation-related information, including quantization parameters
- ✓ Extensible Metadata with Technique-Specific Scheduling Hints

PoT-aware Quantization Support

Example (Pot Quantization)

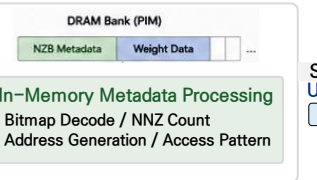
Scale = 2^{-3}
(PoT Element = −3)

Multiply by 2^{-3}
= Shift Right by 3

NZB Value Field

Easily extensible by adding new metadata fields for emerging techniques

Toward PIM / NDP with MPU



- **In-/Near-Memory Metadata Processing** → Minimizes unnecessary data movement
- Selective Data Transfer → Reduces bandwidth usage, latency, and energy consumption
- The MPU acts as a "metadata brain" between memory and the accelerator

Metadata-Centric Computing as Next Opportunity for AI Systems?

Evolution of Computing Paradigms for AI Systems

Compute-Centric

Focus

Arithmetic Operations

Key Metric

FLOPS / Throughput

Data-Centric

Focus

Data Movement & Memory Hierarchy

Key Metric

Bandwidth / Capacity / Locality

Metadata-Aware

Focus

Arithmetic Operations

Key Metric

Metadata Overhead / Efficiency

Metadata-Centric

Focus

Arithmetic Operations

Key Metric

Metadata-Driven System Efficiency

As AI systems grow in scale and complexity, metadata increasingly determines
Where data is, what it means, and how computation should proceed.

Is Metadata-Centric Computing The Next Opportunity for AI Systems?

I plan to continue thinking about this direction and would appreciate your feedback.

From **optimizing computation and data movement** to unlocking the potential of metadata

Acknowledgements

- This work was supported by the grant of the National Research Foundation of Korea (NRF) (No. NRF-2021R1A2C2014557 and No. 2021R1A4A1031864) and Institute of Information & communications Technology Planning & Evaluation (IITP) (No. RS-2022-00167143)

- This research was supported by the Commercialization Promotion Agency for R&D Outcomes(COMPA) funded by the Ministry of Science and ICT(MSIT). (RS-2025-02311436, IP Enhancement and Commercialization for AI Model Lightweighting Technology to Promote the Business Expansion of Low-Cost AI Accelerators)

THANK YOU