EVAS INTELLIGENCE 奕行智能

Dynamic Tile Scheduling for AI Accelerators via TISA

Guanghui Song[1] Xiaoqiang Dan[2] Hui Liu[2] Jie Zhao[1] etc.

[1]Hunan University

[2]EVAS Intelligence

May 13, 2026

Outline

Outline

Heterogeneous execution units across vendors

Deep learning's rapid growth has driven the development of increasingly sophisticated accelerators. Representative platforms include GPUs (both NVIDIA and AMD), TPUs, Ascend, and Tenstorrent. These systems employ heterogeneous-unit architectures that integrate tensor, vector, and DMA engines to exploit massive parallelism, as exemplified in Table below.

Vendor	**Tensor Units**	**Vector ALUs**	**DMA Engines**
NVIDIA	Tensor core	CUDA core	TMA
AMD	Matrix	SIMD	SDMA
TPU	MXU	V-ALU	async DMA
Ascend	Cube	VU	on-chip DMA
Tenstorrent	SFPU	FPU	on-chip DMA

Underutilization from static scheduling

Unfortunately, today's accelerators overwhelmingly rely on static scheduling, a compile-time approach where execution orders are fixed before runtime.

The static scheduling suffers from four fundamental limitations:

- Compile-time complexity and engineering burden.
- Inadequate abstraction granularity.
- Inability to adapt to runtime variability.
- Historical precedent: static approaches fail under uncertainty.

Dynamic scheduling provides a path forward, but applying it at arbitrary instruction granularity would be impractical for deep learning accelerators.

Why is tile-level selected?

The tile abstraction provides the sweet spot: each tile encapsulates a semantically coherent computation with well-defined data ranges and resource requirements.

This granularity enables the runtime to:

- detect dependencies precisely.
- arbitrate heterogeneous resources safely.
- adapt execution in response to runtime conditions

Tile-granular scheduling thus combines the adaptivity of dynamic systems with the semantic safety of compile-time reasoning, forming the foundation of our design.

Outline

Structural contention and data dependencies in FlashAttention-3

Dynamically scheduling tiled computations must reason about two forms of conflict:

- **Structural contention:** Tensor and vector units are distinct but complementary resources. The two GEMM stages exclusively occupy tensor units, while the Softmax stage uses vector units. Static schedulers could exploit this heterogeneity within an iteration: while the vector units execute S_i, tensor units can concurrently begin the next GEMM.
- **Data dependencies:** However, three tiled operations are linked by producer-consumer edges. S_i depends on the outputs of $M0_i$, and $M1_i$ depends on the outputs of S_i. These dependencies enforcement introduces an *implicit barrier* between iterations: the next iteration $(i{+}1)$ cannot begin until all dependent results from iteration i are completed.

Implication: Implicit synchronization and missed inter-iteration concurrency

Although structural heterogeneity allows tensor and vector units to run in parallel, the implicit synchronization across iterations forces them to do so only within a single iteration.

Different Tiles execution manners of FlashAttention-3

Sequential execution of Tiles

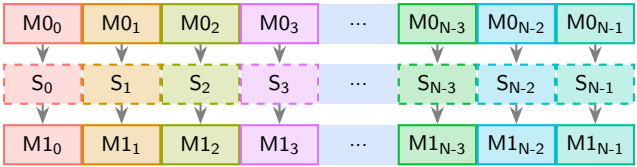


Figure: After tiling, the three operations involved in the attention pattern are fused and executed iteratively along the horizontal direction. Each iteration (shown in a distinct color) comprises three tiles: solid-outlined boxes represent tensor-unit executions, and dashed-outlined boxes represent vector-unit softmax computations. Vertical arrows indicate intra-iteration data dependencies.

Different Tiles execution manners of FlashAttention-3

Sequential execution of Tiles

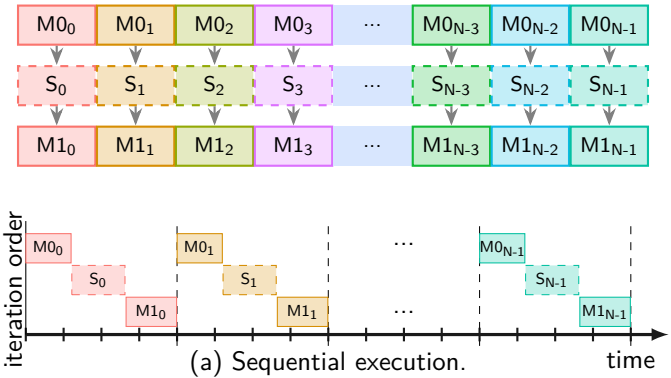


Figure: Vertical dashed lines denote synchronization barriers between iterations imposed by static scheduling.

Different Tiles execution manners of FlashAttention-3

Dual-staged pipelined execution of Tiles

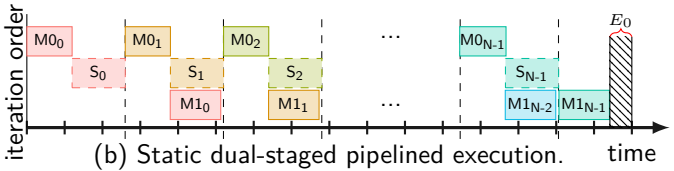


Figure: Vertical dashed lines denote synchronization barriers between iterations imposed by static scheduling. Shaded regions (E_x) represent latency saved through improved scheduling.

Different Tiles execution manners of FlashAttention-3

Dual-staged pipelined execution of Tiles

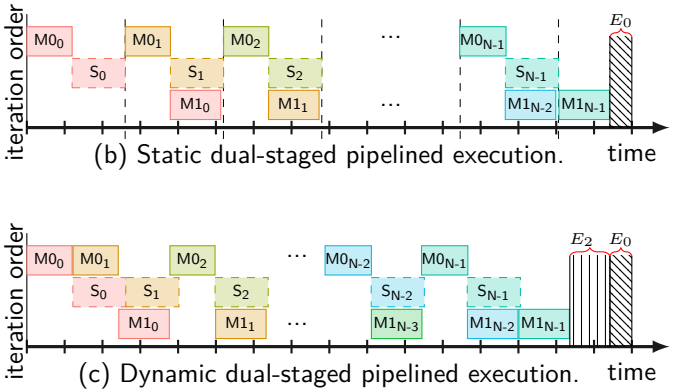


Figure: Vertical dashed lines denote synchronization barriers between iterations imposed by static scheduling. Shaded regions (E_x) represent latency saved through improved scheduling.

Different Tiles execution manners of FlashAttention-3

Triple-staged pipelined execution of Tiles

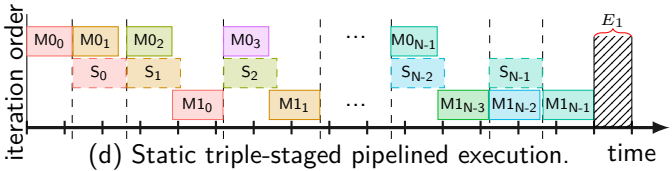


Figure: Vertical dashed lines denote synchronization barriers between iterations imposed by static scheduling. Shaded regions (E_x) represent latency saved through improved scheduling.

Different Tiles execution manners of FlashAttention-3

Triple-staged pipelined execution of Tiles

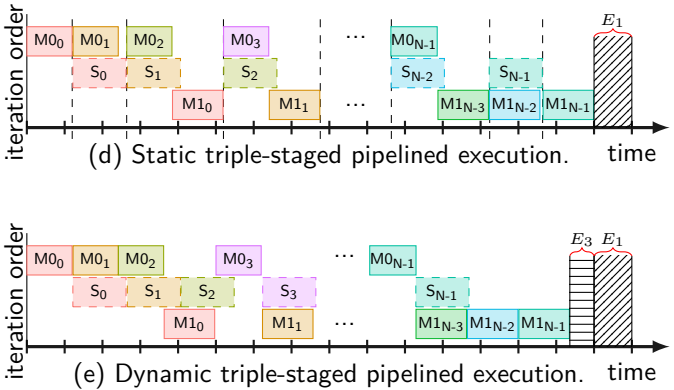
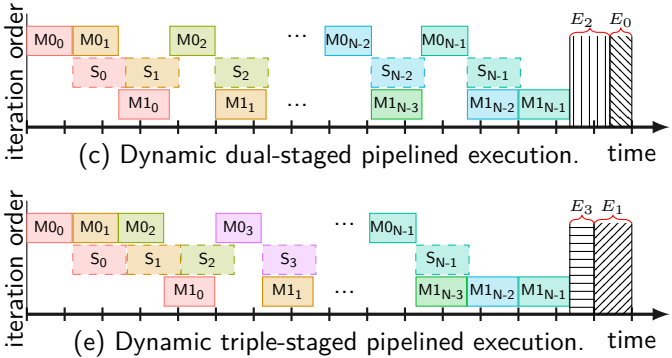


Figure: Vertical dashed lines denote synchronization barriers between iterations imposed by static scheduling. Shaded regions (E_x) represent latency saved through improved scheduling.

Different Tiles execution manners of FlashAttention-3

Triple-staged VS. Dual-staged



Dynamic tile scheduling does not rely on rigid pipeline phase definitions.

$E_0 + E_2 = E_1 + E_3$ illustrates the equivalent latency savings achievable by dual-stage or triple-stage dynamic tile scheduling without relying on rigid pipeline stage definitions.

Outline

Our Framework: Software and Hardware Semantic Bridge

Key Insight:*Information Preservation*

Unlike static scheduling which loses semantics and enforces synchronization across iterations, we try to restore runtimevisible semantics. So, we design TISA, a semanticspreserving abstraction that forms a bridge between software decomposition and hardware scheduling.

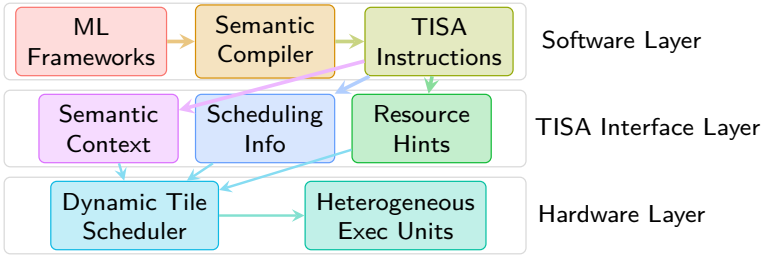


Figure: The overall architecture of our integrated framework.

Outline

TISA Instruction Structure Definition

Field	Type/Constraints	Description
TISA_Inst = (OpType, Operands, Attributes, UnitMap)		
OpType	{GEMM, SOFTMAX, ...}	Semantic identifier
Operands	$[op_1, op_2, ...]$	Array of Operand
	$\|outs\| \leq 3, \|ins\| \leq 7$	Hardware constraints
Attributes	Op Attrs,Schedule Params	Reorder constraints, sync requirements
UnitMap	(unit, quantity, affinity)	Resource specification

- Operand = (TileShape, TileMem, AccessType)
  - TileShape is Symbolic/Parametric, represents Computational bounds
  - TileMem Composed of (base, scope), represents Memory specification
  - AccessType has {R, W, RW}, represents Access pattern
- TileMem = (base, scope)
  - base means Address, represents Symbolic/Constant address
  - scope has {Private,Local,Shared} , represents Memory hierarchy level

Outline

1. **Semantic Routing:** Incoming TISA instructions are parsed for their `OpType`, `UnitMap`, and dependency metadata, then routed to the appropriate waiting queue (WQ) of each target unit. Each WQ preserves operator semantics and provides a local view of ready candidates per unit type.

Semantics-Aware Dynamic Tile Scheduling

The scheduling mechanism

1. **Semantic Routing:** Incoming TISA instructions are parsed for their `OpType`, `UnitMap`, and dependency metadata, then routed to the appropriate waiting queue (WQ) of each target unit. Each WQ preserves operator semantics and provides a local view of ready candidates per unit type.
2. **Dependency Resolution:** The scheduler periodically selects a ready window W from each WQ and checks for semantic hazards against the unit's in-flight table $\mathcal{F}_u$. Only instructions passing dependency and resource checks are promoted to the issue queue (IQ). This step enforces correctness while enabling out-of-order admission across units.

Semantics-Aware Dynamic Tile Scheduling

The scheduling mechanism

1. **Semantic Routing:** Incoming TISA instructions are parsed for their `OpType`, `UnitMap`, and dependency metadata, then routed to the appropriate waiting queue (WQ) of each target unit. Each WQ preserves operator semantics and provides a local view of ready candidates per unit type.
2. **Dependency Resolution:** The scheduler periodically selects a ready window W from each WQ and checks for semantic hazards against the unit's in-flight table $\mathcal{F}_u$. Only instructions passing dependency and resource checks are promoted to the issue queue (IQ). This step enforces correctness while enabling out-of-order admission across units.
3. **Adaptive Issue:** IQ entries are issued to hardware execution pipelines once their dependencies are cleared.

Semantics-Aware Dynamic Tile Scheduling

The scheduling mechanism

1. **Semantic Routing:** Incoming TISA instructions are parsed for their `OpType`, `UnitMap`, and dependency metadata, then routed to the appropriate waiting queue (WQ) of each target unit. Each WQ preserves operator semantics and provides a local view of ready candidates per unit type.
2. **Dependency Resolution:** The scheduler periodically selects a ready window W from each WQ and checks for semantic hazards against the unit's in-flight table $\mathcal{F}_u$. Only instructions passing dependency and resource checks are promoted to the issue queue (IQ). This step enforces correctness while enabling out-of-order admission across units.
3. **Adaptive Issue:** IQ entries are issued to hardware execution pipelines once their dependencies are cleared.
4. **Feedback:** Upon completion, the corresponding $\mathcal{F}_u$ entry is retired, dependent instructions are notified, and per-unit scheduling priorities are adaptively updated based on observed contention or latency. This feedback mechanism continuously tunes overlap and unit utilization at runtime.

Semantics-Aware Dynamic Tile Scheduling

Per-unit semantic scheduling

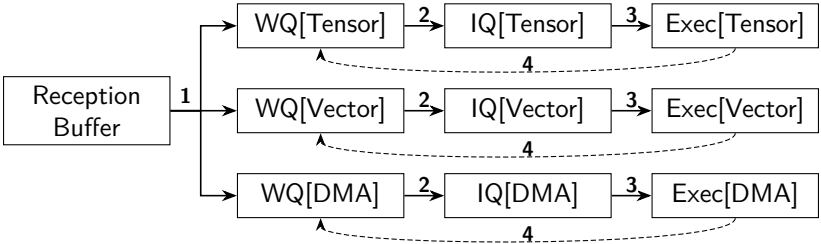


Figure: Decentralized queues localize dependency checking and prevent unrelated blocking across heterogeneous units. WQ: waiting queue; IQ: issue queue.

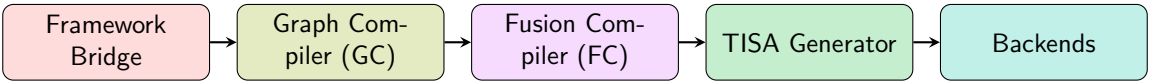
Algorithm 1: Semantic Conflict Detection

Input: Instruction I with semantic annotations, in-flight semantic table $\mathcal{F}_u$
foreach $r \in \mathcal{F}_u$ **do**
    if *not same_scope(*I, r*)* **then**
        continue ;
    if *semantic_compatibility(*$I.OpType$, $r.OpType$*)* **then**
        // Semantically compatible operations can overlap
        continue ;
    if *memory_range_overlap(*I, r*)* ***and*** *true_dependency(*I, r*)* **then**
        if *cannot_reorder_safely(*I, r*)* **then**
            // True semantic conflict detected
            return true ;
// Safe to execute in parallel
return false ;

Outline

Semantics-Preserving Compiler Stack

End-to-end compilation flow preserving semantic information



- **Framework Bridge:** Ingests models from PyTorch, JAX, and TensorFlow using torchXLA frontend, exporting framework graphs to XLA or StableHLO dialects.
- **Graph Compiler (GC):** MLIR-based compiler consuming StableHLO IR, performing architecture-aware optimizations. Emits software-scheduled tile graph that exposes legal overlap opportunities.
- **Fusion Compiler (FC):** Specializes fused subgraphs into TISA-compatible operators using custom TISA dialect, which is abstract representation of TISA instruction.
- **TISA Generator:** Provides virtual tile-level instruction set unifying multiple hardware backends.
- **Backends:**
  - **TISA-NPU**: Targets Epoch hardware with full dynamic scheduling support.
  - **TISA-CPU**: Emits CPU kernels for functional validation, retaining TISA semantics.

Case Study: FlashAttention-3 Pseudocode

```
//CUDA fa3 pseudocode
tma_load_q(s_Q,Q);
tma_load_k_transpose(s_K,K);
warpgroup_fence_producer();
wgmma::mma_sync(s_P,s_Q,s_K);
wgmma::wait();
softmax_warpgroup(s_S,s_P,state);
for (int j = 0; j < Tc; j++) {
 if (j < Tc - 1) {
  tma_load_k_transpose(s_K_next,K_next);
  warpgroup_barrier_arrive();
  wgmma::mma_async(s_S_next,s_Q,s_K_next);
 }
 tma_load_v(s_V,V);
 warpgroup_barrier_wait();
 wgmma::mma_sync(s_R,s_S,s_V);
 if (j < Tc - 1) {
  wgmma::wait();
  softmax_warpgroup(s_S_next,s_S_next,state_next)
      ;
 }
 wgmma::wait();
 rescale_warpgroup(s_O,s_R,state,state_next);
 warpgroup_commit_batch();
 update_carousel_index();
}
tma_store_o(O,s_O);
warpgroup_epilogue();
```

```
//TISA fa3 pseudocode
tisa::load<de>(s_Q,Q);
tisa::load_transpose<de>(s_K,K);
tisa::gemm<me>(s_P,s_Q,s_K);
tisa::softmax<ve>(s_S,s_P,state);
for (int j = 0; j < Tc; j++) {
 if (j < Tc - 1) {
  tisa::load_transpose<de>(s_K_next,K_next);
  tisa::gemm<me>(s_S_next,s_Q,s_K_next);
 }
 tisa::load<de>(s_V,V);
 tisa::gemm<me>(s_R,s_S,s_V);
 if (j < Tc - 1) {
  tisa::softmax<ve>(s_S_next,s_S_next,state_next);
 }
 tisa::rescale<ve>(s_O,s_R,state,state_next);
 update_next_index();
}
tisa::store<de>(O,s_O);
```

Figure: Comparison of FlashAttention-3 CUDA and TISA Pseudocode

Outline

Experimental Platform Specifications

Table: Experimental platform specifications.

Platform	Epoch	H100	A100	Xeon 6348
Arch	X	Hopper	Ampere	x86_64
Cores	32 Cores	132 SMs	108 SMs	56 Cores
Compute	256T FP16	989T FP16	312T FP16	9.2T FP32
Memory	48GB GDDR6	80GB HBM3	40GB HBM2	512GB DDR4
Bandwidth	1TB/s	3.35TB/s	1.55TB/s	204.8GB/s

Table 1 summarizes the platform specifications. For NVIDIA GPUs, we use driver 550.127.0, CUDA 12.1, cuDNN 9.1.0, TensorRT 10.3.0 (for ResNet50/BERT-Base), and TensorRT-LLM 0.12.0 (for GPT-J/LLaMA2/DeepSeek-R1).

Experimental Results on the Epoch Platform

Execution Latency on Epoch

Table: Execution cycles (smaller is better), M: million cycles; N: Naive; S: Static; D: Dynamic

Model	Naive	Static	Dynamic	S vs N	D vs N	D vs S
ResNet50	8.98M	8.72M	5.92M	1.03×	1.52×	1.47×
BERT	13.16M	11.94M	7.37M	1.10×	1.79×	1.62×
GPTJ(oneblk)	14.44M	13.54M	8.30M	1.07×	1.74×	1.63×
LLaMA2(oneblk)	25.85M	15.29M	13.47M	1.69×	1.92×	1.14×

Experimental Results on the Epoch Platform

Cross-unit Overlapping Latency on Epoch

Table: Cross-unit overlapping cycles(larger is better), M: million cycles; N: Naive; S: Static; D: Dynamic

Model	Naive	Static	Dynamic	S vs N	D vs N	D vs S
ResNet50	1.09M	1.81M	2.84M	1.66×	2.60×	1.57×
BERT	0.41M	1.45M	7.50M	3.54×	18.29×	5.17×
GPTJ(oneblk)	0.14M	1.01M	4.57M	7.21×	32.64×	4.52×
LLaMA2(oneblk)	0.53M	10.27M	11.30M	19.38×	21.32×	1.10×

Comparison with GPU Baselines

End-to-End AI Model Performance Analysis (Epoch VS. A100)

Table: Execution latency comparison across platforms (smaller is better)

Model	Configuration	Epoch	A100	Speedup
ResNet50	FP16, batch=128, 224*224	6.2ms	9.3ms	1.50×
BERT-Base	FP16, batch=64, seq=128	7.5ms	9.8ms	1.31×
GPT-J-6B	FP16, batch=1, seq=512, prefill	29.9ms	37.3ms	1.25×
LLaMA2-13B	FP16, batch=1, seq=512, prefill	54.0ms	77.1ms	1.43×
DeepSeek-R1-16B	BF16, batch=50, seq=100, prefill	213.5ms	412.3ms	1.93×
DeepSeek-R1-16B	BF16, batch=50, seq=700, decode	51.2ms	69.0ms	1.35×

Comparison with GPU Baselines

FlashAttention-3 Performance Analysis (Epoch VS. H100)

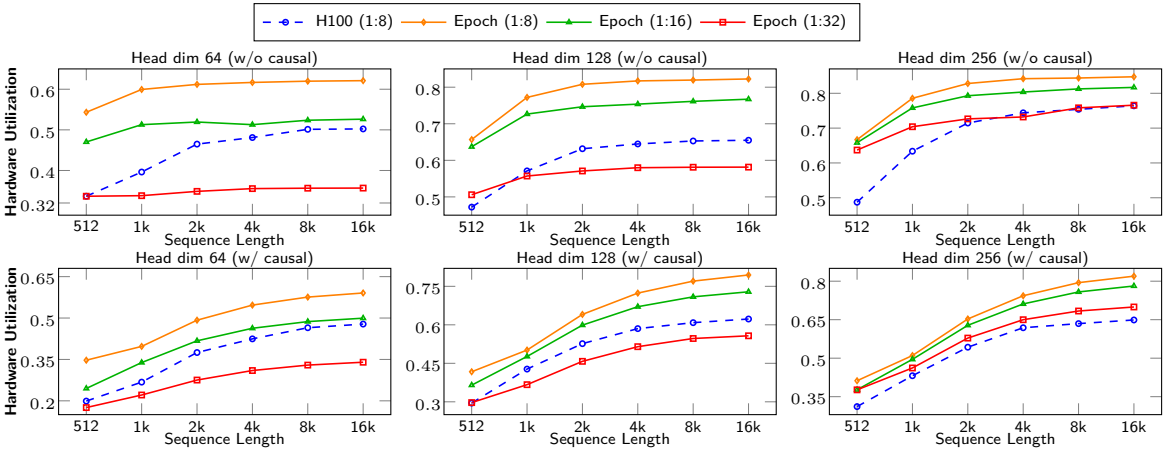


Figure: Performance comparison of FlashAttention-3 implementations on Epoch and NVIDIA H100. Hardware utilization is measured across varying sequence lengths and head dimensions.

Portability Experimental

Execution Time on CPU

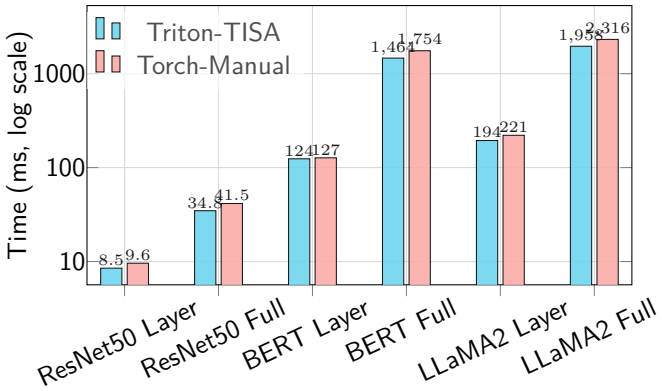


Figure: Comparison of execution times (in milliseconds) between Triton-TISA and Torch-Manual.

Outline

Summary

We addressed underutilization rooted in static scheduling by co-designing three pieces that expose runtime-visible scheduling semantics: a tile-level instruction set (TISA), a semantics-aware dynamic tile scheduler, and a semantics-preserving compiler. This makes operator/tile boundaries, typed dependencies with readiness, resource intents, and tile memory ranges directly consumable at runtime for safe overlap and reordering across heterogeneous units.

Across ResNet50, BERT, GPT-J, LLaMA2 and DeepSeek-R1 on Epoch as well as NVIDIA H100 and A100 GPUs, the approach delivers 1.52–1.92$\times$ over the baselines; on FlashAttention-3 it achieves approximately 26.4% higher matrix-unit utilization in the mainstream head-dim-128.

Outline

EVAS INTELLIGENCE | 奕行智能

Thank You

Questions are welcome