SOUTHEAST UNIVERSITY MLSys

# OPKV: A High-Throughput Plugin-Driven Framework for Recallable Sparsity in Paged KV Cache Systems

**Huazheng Lao**[1], Xiaofeng Li[1], Rui Xu[1], Long Chen[1*], Xia Zhu[1], Jinquan Zhang[2]

[1]Southeast University [2]Guangdong University of Technology

# Motivations

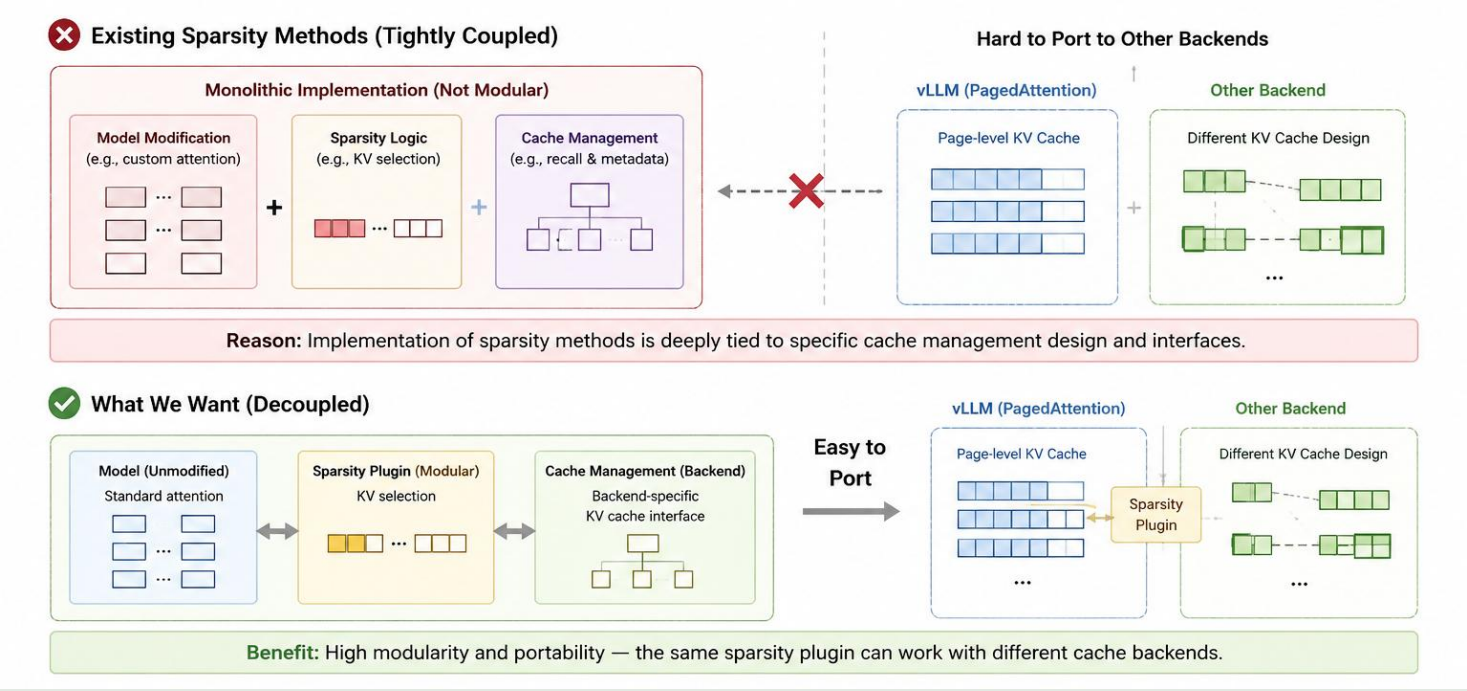
## Challenge 1

### Model-Cache Coupling

Sparsity logic tightly coupled with

- attention implementation
- KV cache manager
- model architecture

→ Poor portability and invasive changes



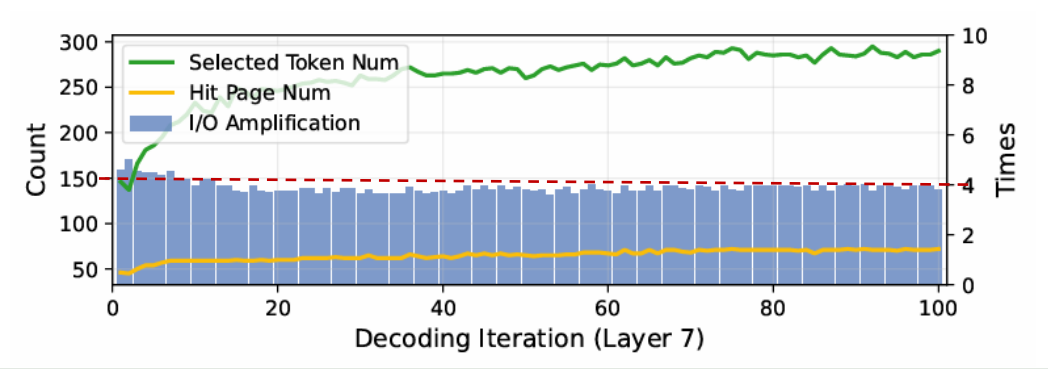
# Motivations

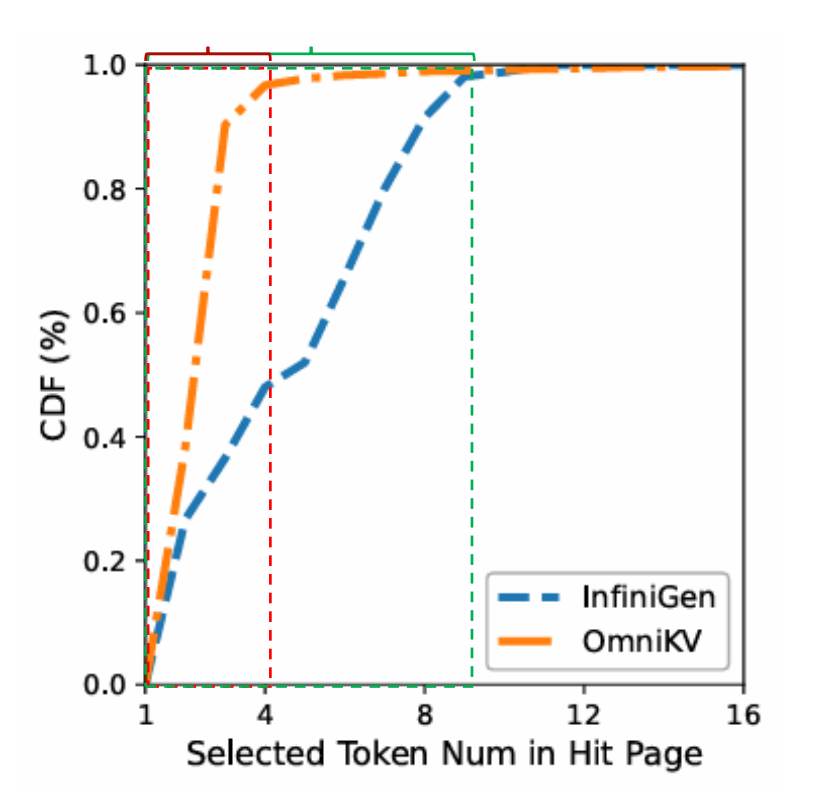
## Challenge 2

### Token-Page Mismatch

Sparse recall selects tokens but PagedAttention manages pages(tens of tokens)

→ Severe I/O amplification




"vital few"

# Motivations

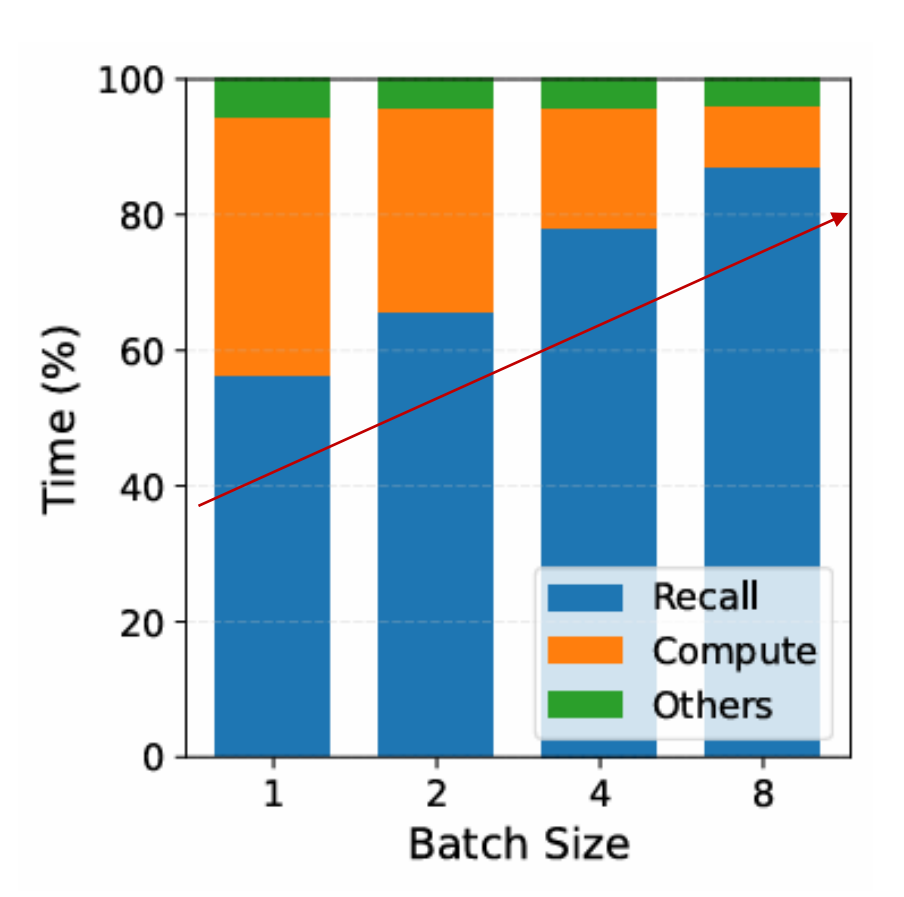
## Challenge 3

### Recall Overhead Explosion

Recall overhead grows with batch size

→ Prefetch overlap breaks

→ Throughput saturates at high batch




The proportion of overheads

# OPKV Design

## System Overview

### Model-Sparsity-Cache Unified Architecture

Key Innovations:

#### 1. Plugin Interface

Decouples sparsity logic from model & cache management

#### 2. Object Reaggregation (OP Blocks)

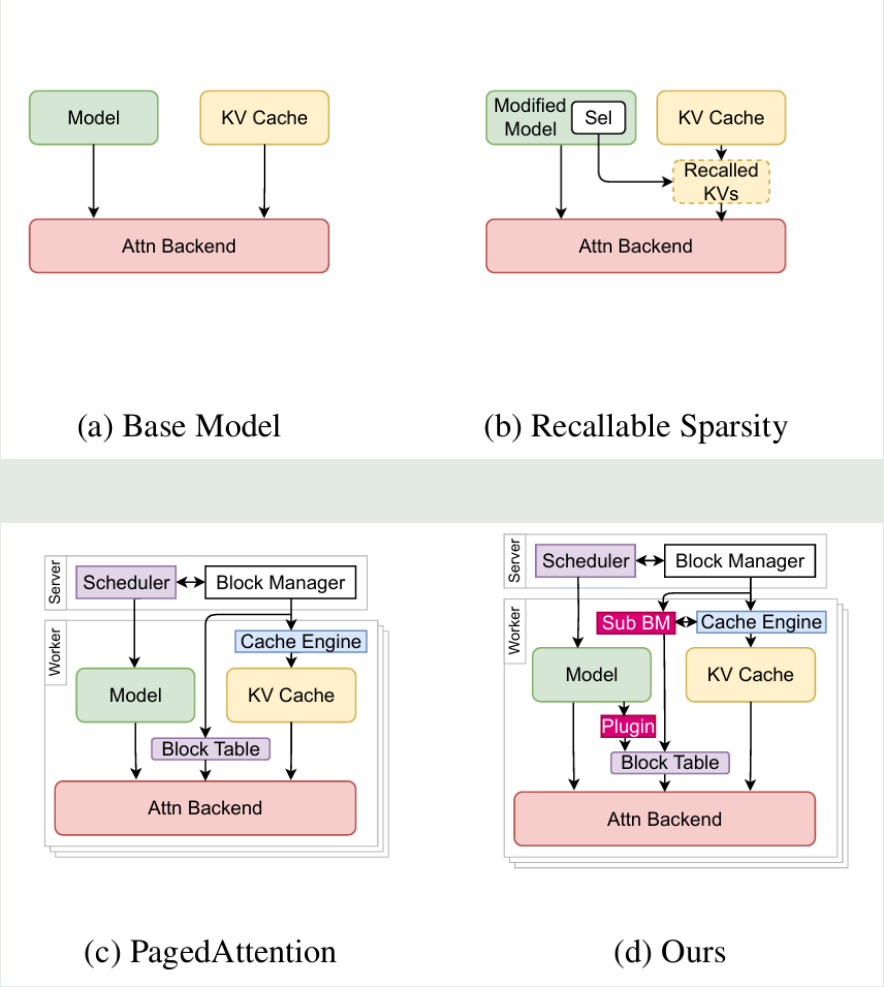
Mitigates spatial discreteness → reduces I/O amplification

#### 3. Hot Page Hit

Exploits temporal locality → reduces CPU-GPU traffic

#### 4. Sub Block Manager (Sub BM)

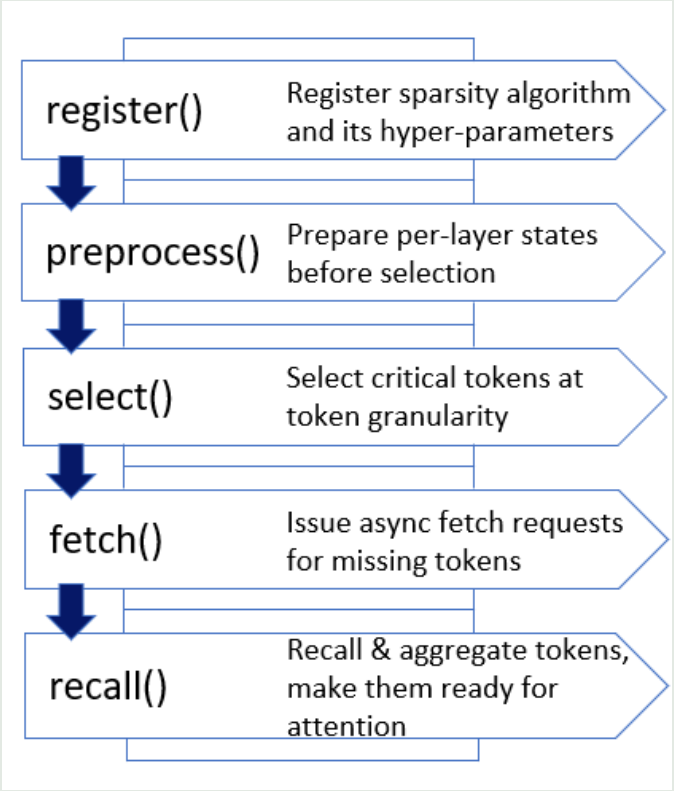
Local metadata → millisecond-level page retrieval



# OPKV Design

## Plugin Interface

### Sparsity – Cache Management Decoupling




Five standardized callbacks to implement any recallable sparsity method

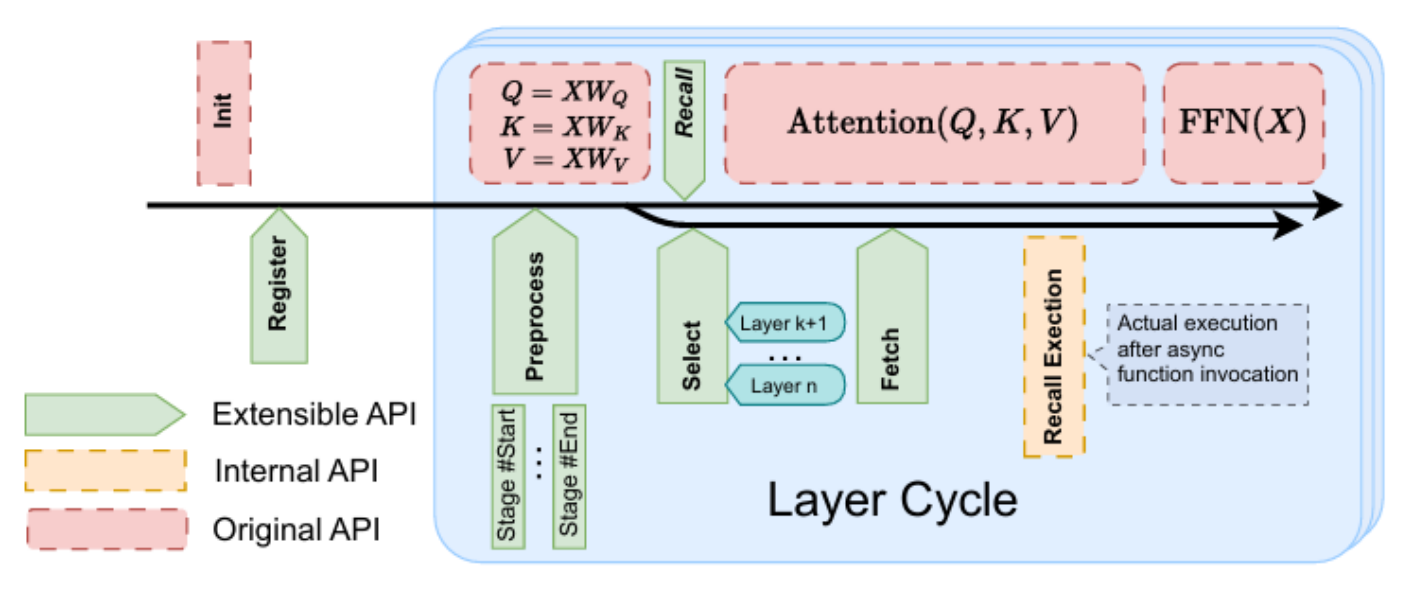
```python
class RecallableSparsityPlugin:

    def register(self, attn_layer):
        layer_id = self._register(attn_layer)
        return layer_id

    def preprocess(self, layer_id, stage, context):
        if stage == Stage.Start:
            return self._process_start(layer_id, context)
        if stage == Stage.End:
            return self._process_end(layer_id, context)
        ...

    def select(self, layer_id, context):
        sel_token_idx_by_layer = self._select(
            layer_id, context)
        return sel_token_idx_by_layer

    def fetch(self, sel_token_idx_by_layer, context):
        requests = self._build(
            sel_token_idx_by_layer, context)
        outputs = [
            sub_block_manager.async_fetch(requests)
            for sub_block_manager in context.block_manager
        ]
        err_code = self._handle_err(outputs)
        return err_code

    def recall(self, layer_id, context):
        outputs = [
            sub_block_manager.recall(layer_id)
            for sub_block_manager in context.block_manager
        ]
        err_code = self._handle_err(outputs)
        block_table = self._handle_result(outputs)
        return err_code, block_table
```

# OPKV Design

## Plugin Workflow

### Async Compute-Recall Overlap

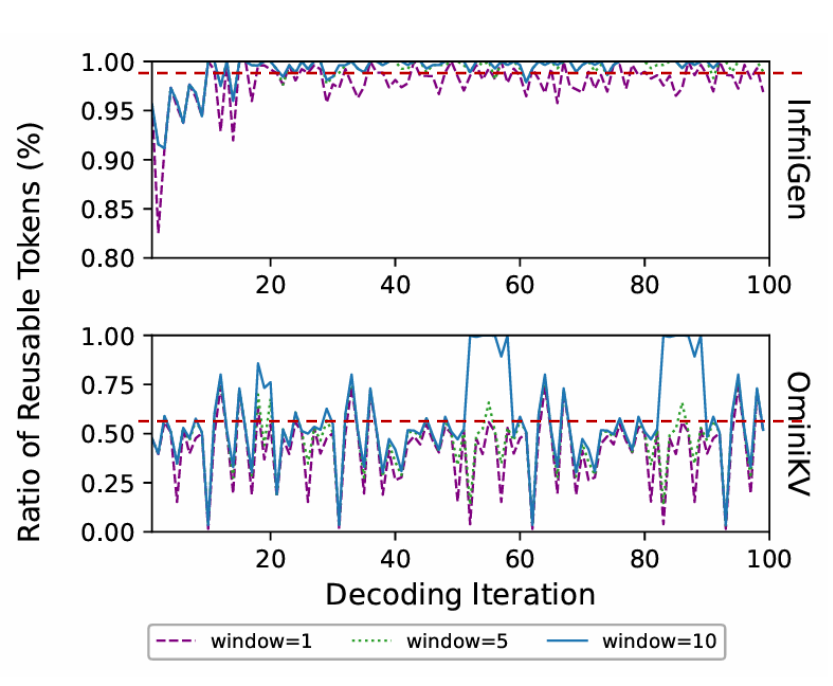
- Preprocess → Select → Fetch → (async) → Recall
- Recall and attention computation are overlapped
- Multiple layers can be managed in parallel



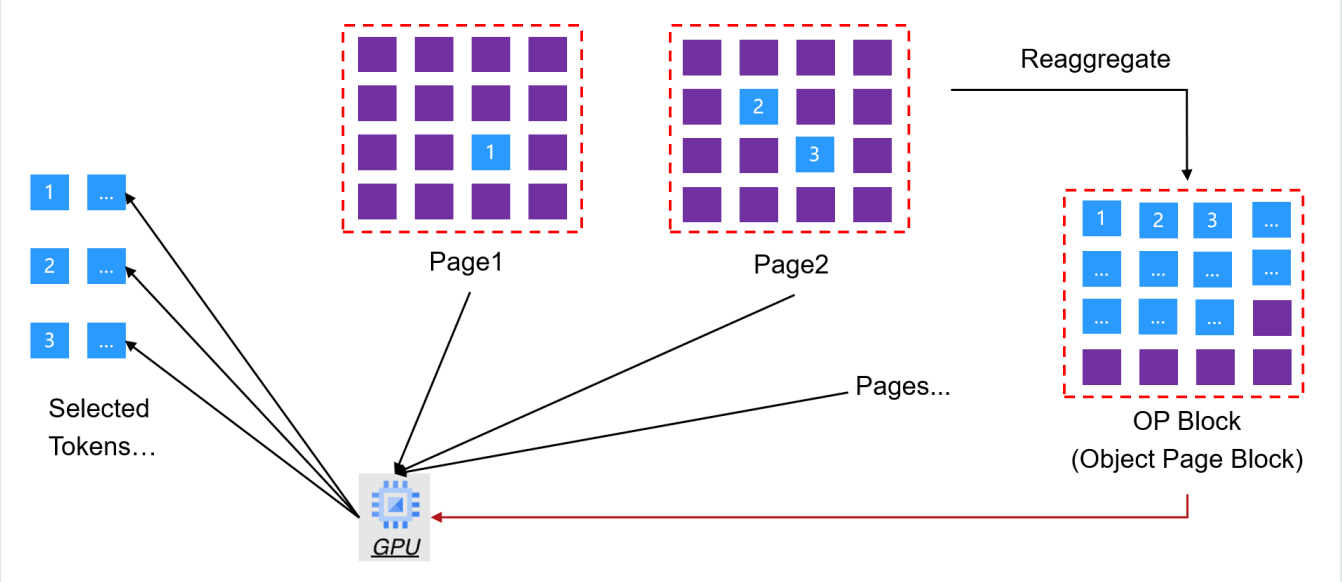
# OPKV Design

## KV Recall

### Object Reaggregation (OP Blocks)




temporal locality in selected tokens
→aggregated pages are likely to be reused in the short term




higher page hit ratio & lower I/O amplification

# OPKV Design

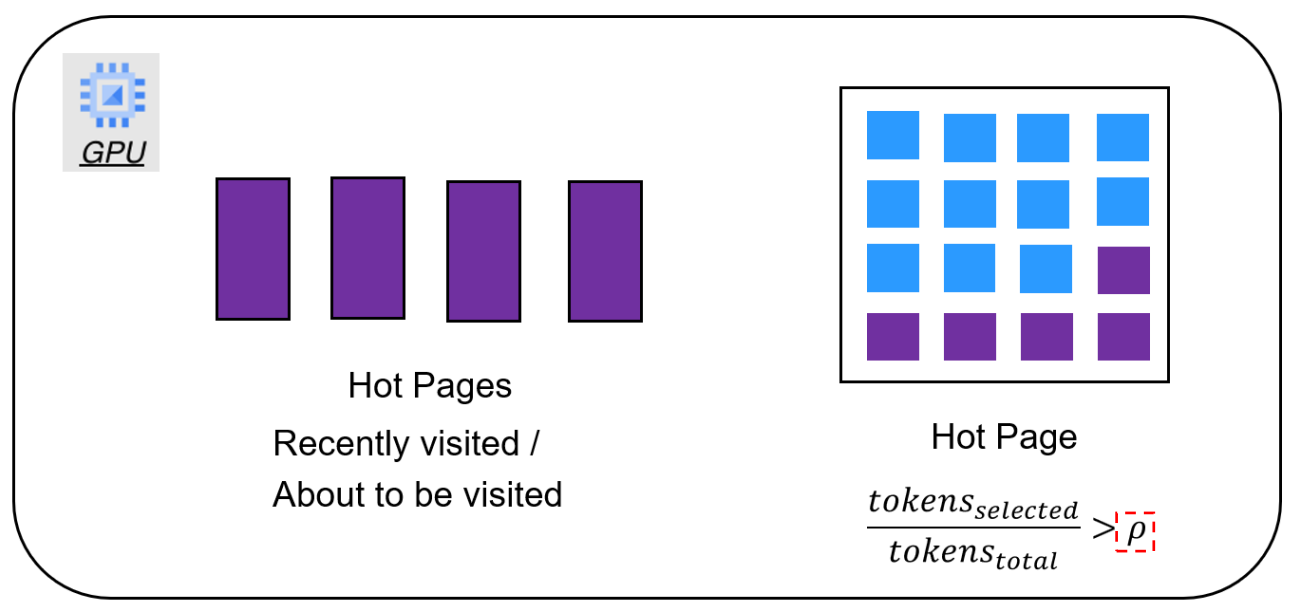
## KV Recall

### Hot Page Hit

Utilization of temporal locality

Page Hit Threshold ρ (tunable):

- Higher ρ → only hottest pages → higher GPU memory utilization, but increased reaggregation overhead
- Lower ρ → keep more pages → lower reaggregation overhead, but higher GPU memory usage

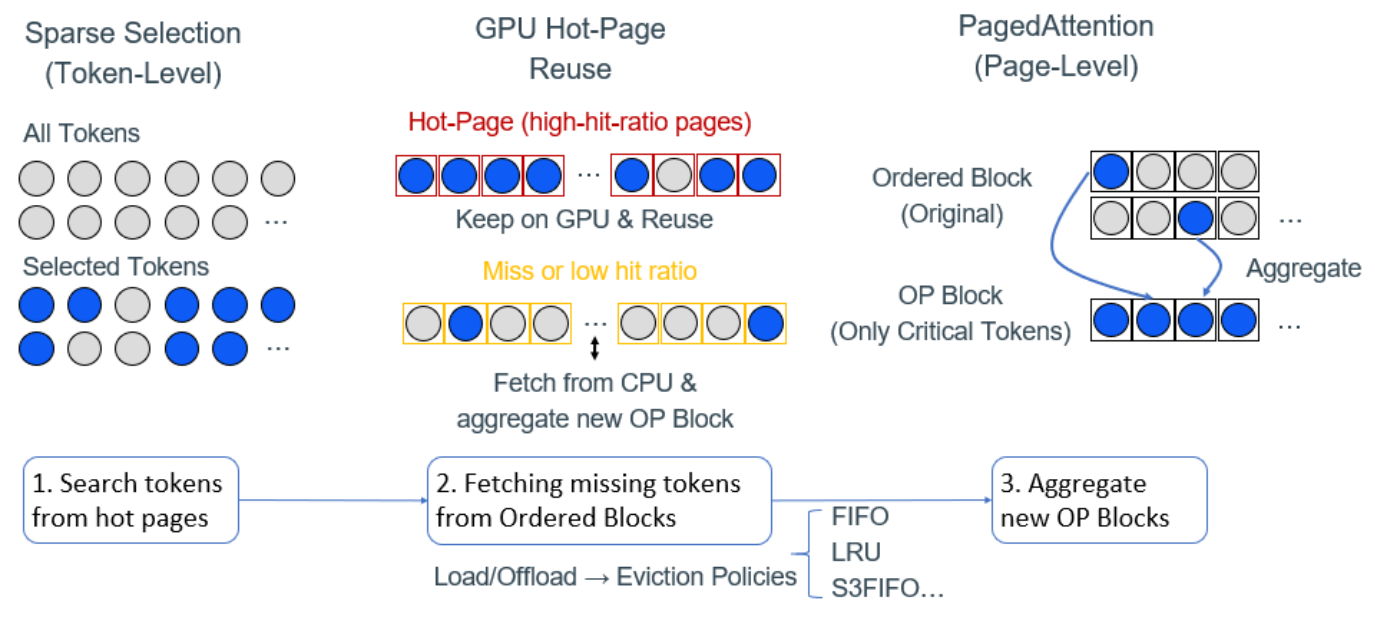


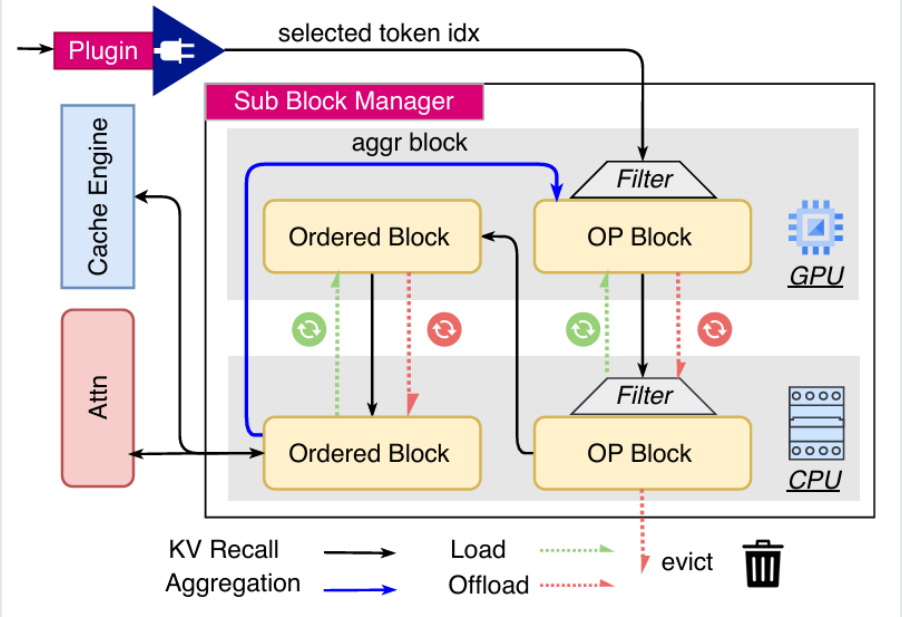

lower I/O overhead (pages swap in/out)

# OPKV Design

## KV Recall

### Workflow




token-level sparsity ~ page-level cache management

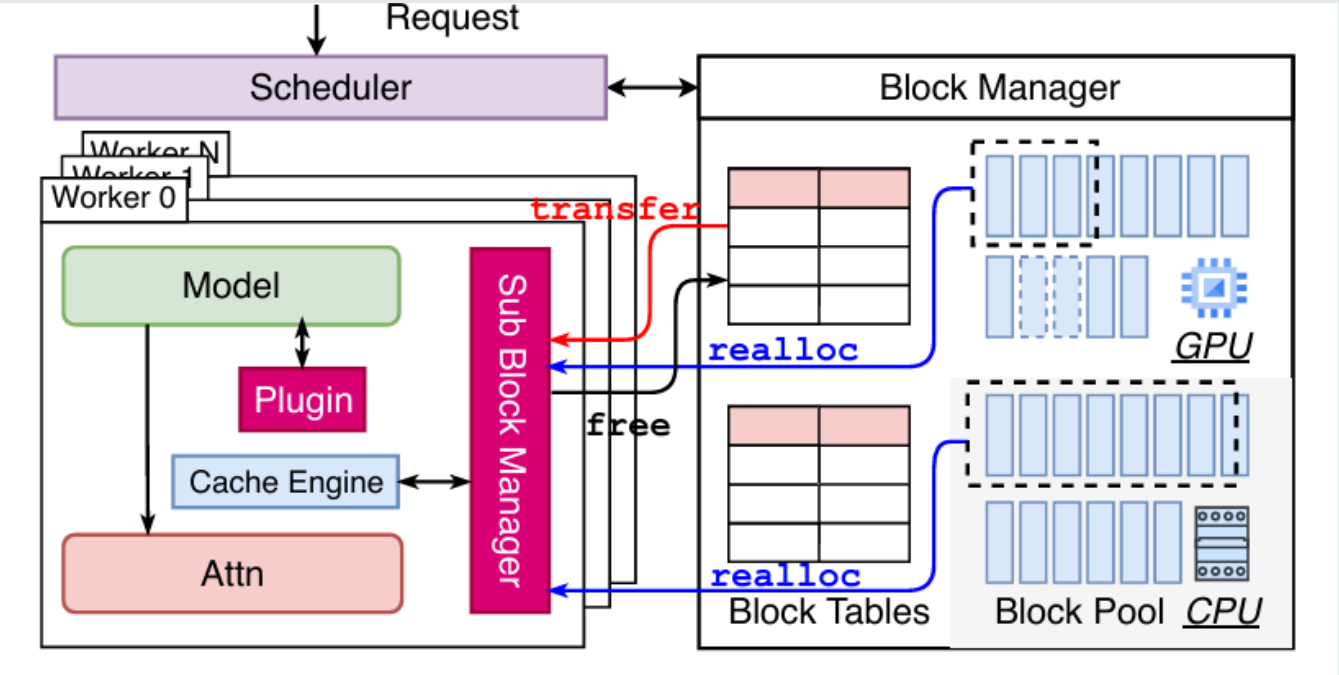
# OPKV Design

## Sub Block Manager

### Local metadata management

PagedAttention updates metadata per iteration, but recall happens per layer.

- Handles per-layer recall within the GPU process
- No RPC overhead on the critical path
- Millisecond-level page retrieval & cache eviction




Per-worker local metadata manager

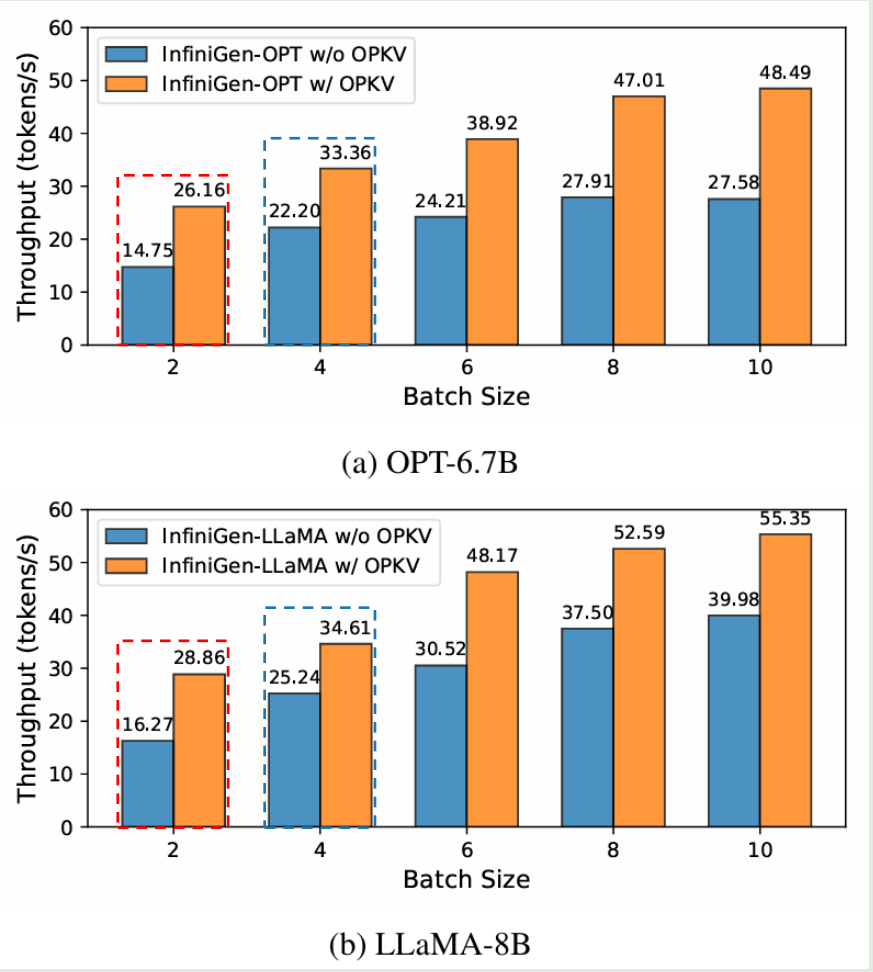
# Evaluation

## Case Study

### InfiniGen End-to-End Performance

Decoding throughput improvement
Batch size: 2-8

- OPT-6.7B: 1.77x, 1.50x, 1.60x, 1.68x, 1.75x
- LLaMA-8B: 1.77x, 1.37x, 1.57x, 1.39x, 1.38x




Decoding throughput under different batches

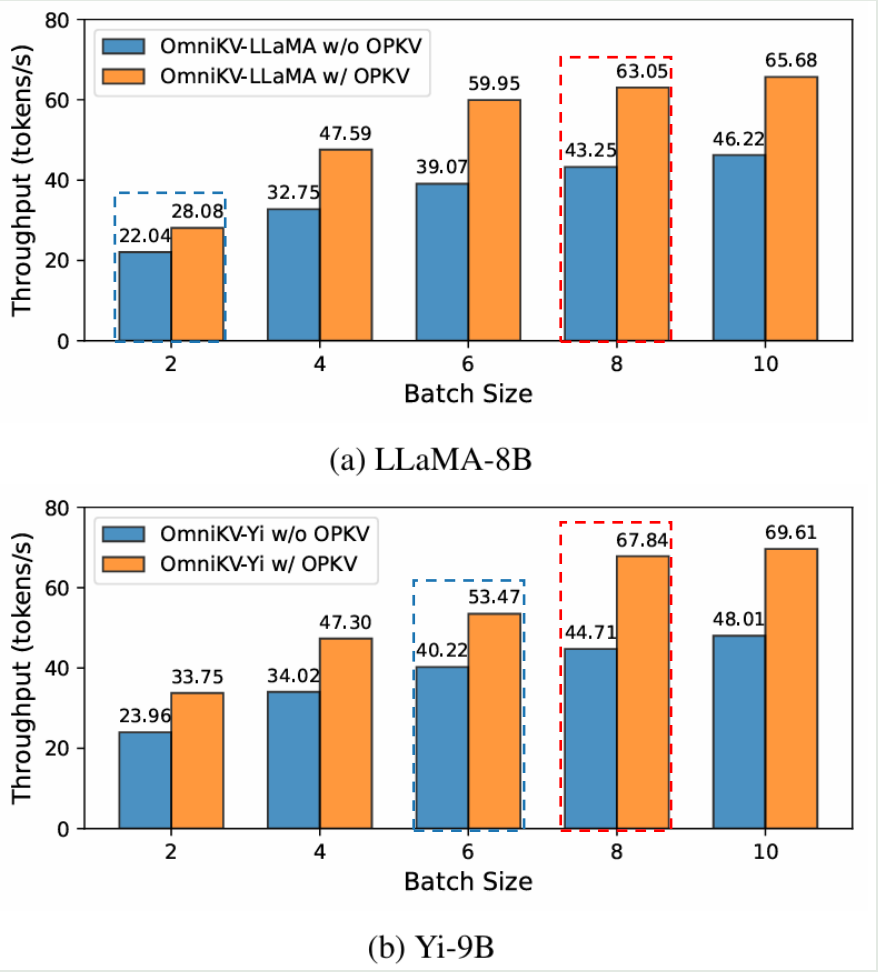
# Evaluation

## Case Study

### OmniKV End-to-End Performance

Decoding throughput improvement
Batch size: 2-8

- LLaMA-8B: 1.35x, 1.51x, 1.52x, 1.56x, 1.55x
- Yi-9B: 1.40x, 1.39x, 1.32x, 1.51x, 1.44x




Decoding throughput under different batches

# Evaluation

## Ablation Study

### Quantified Contributions

**Disable OP Block Aggregation:**

**InfiniGen: -33.06% | OmniKV: -23.51%**

→ OP Blocks are critical for reducing I/O amplification

**Disable Hot Page Reuse:**

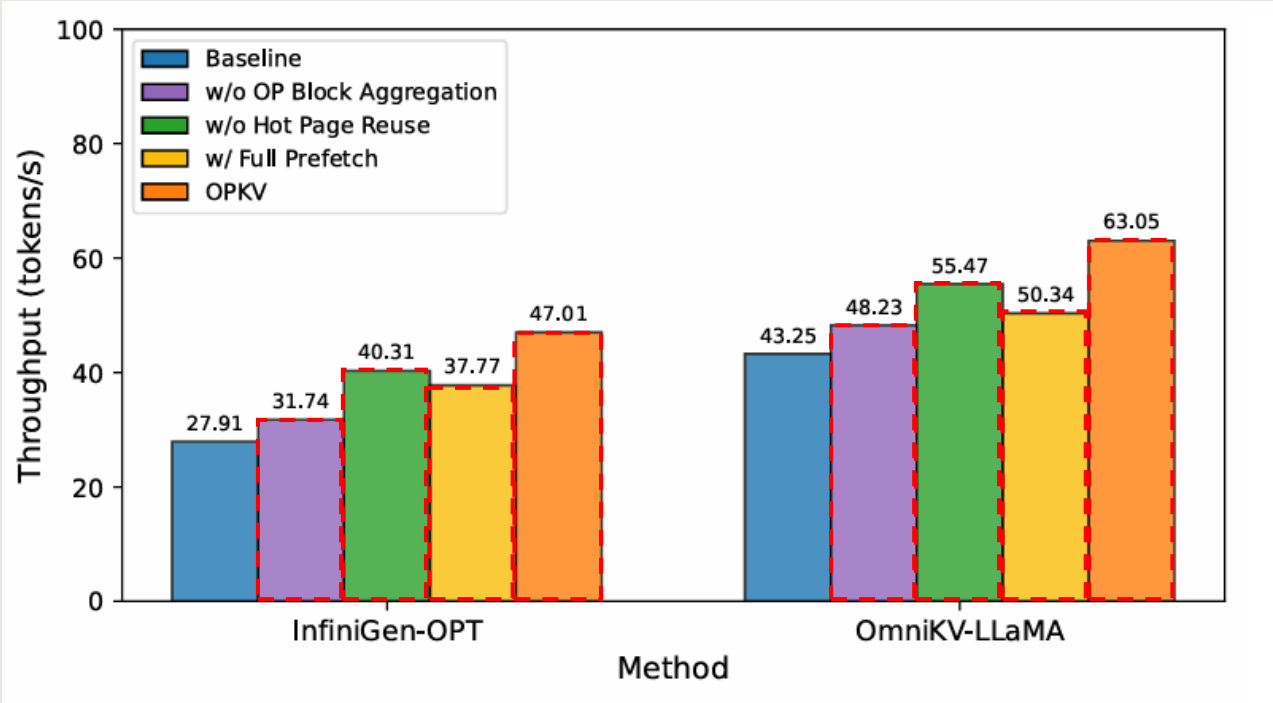
**InfiniGen: -14.25% | OmniKV: -12.02%**

→ Temporal locality is a significant source of savings

**Replace Selective with Full Prefetch:**

**InfiniGen: -19.66% | OmniKV: -20.16%**

→ Token-level selection avoids interconnect saturation




Decoding throughput under different ablation configurations (a fixed batch size of 8)

# Evaluation

## Sensitivity Analysis

### Parameters

**Block Size:**

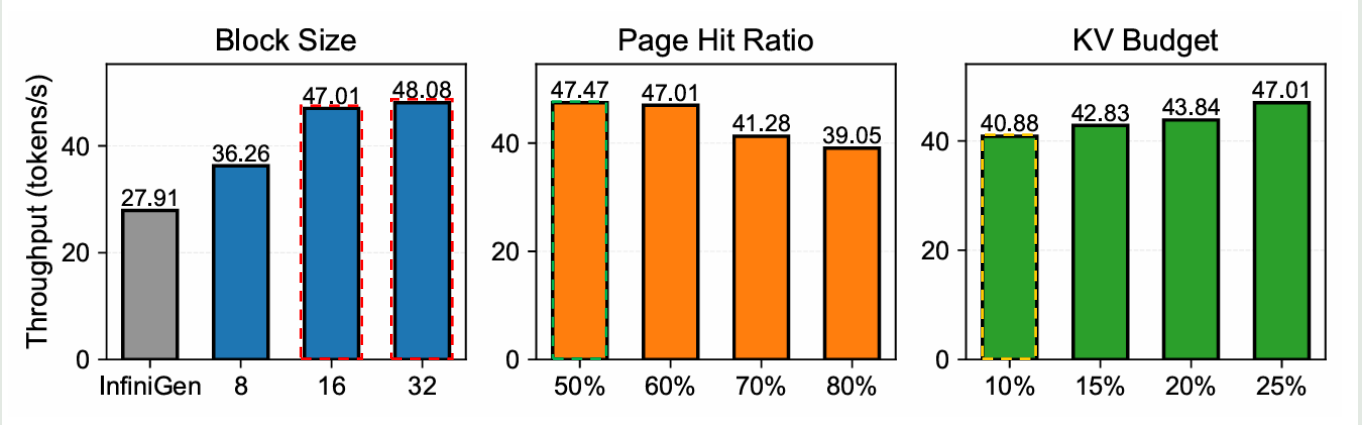
16–32 optimal – smaller increases overhead, larger amplifies I/O

**Page Hit Ratio:**

Lower improves reuse but adds I/O amplification

**KV Budget:**

Even 10% → 46.47% throughput gain vs. prototype




InfiniGen decoding throughput with a fixed batch size of 8 on OPT-6.7B

# Evaluation

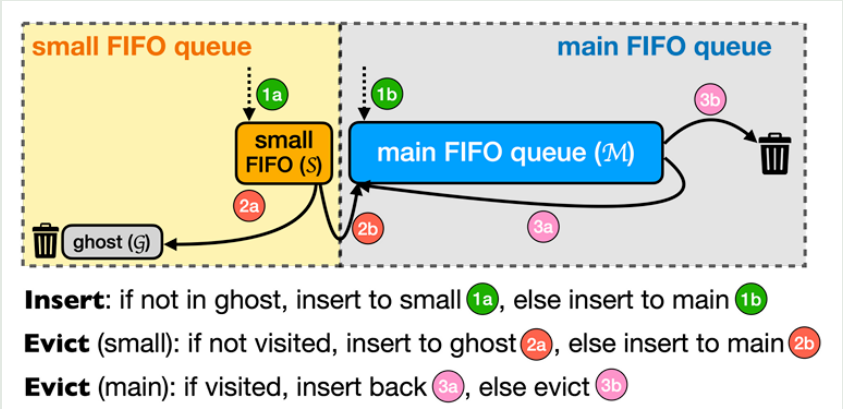
## Eviction Policy Analysis

### New Eviction Policies are anticipated

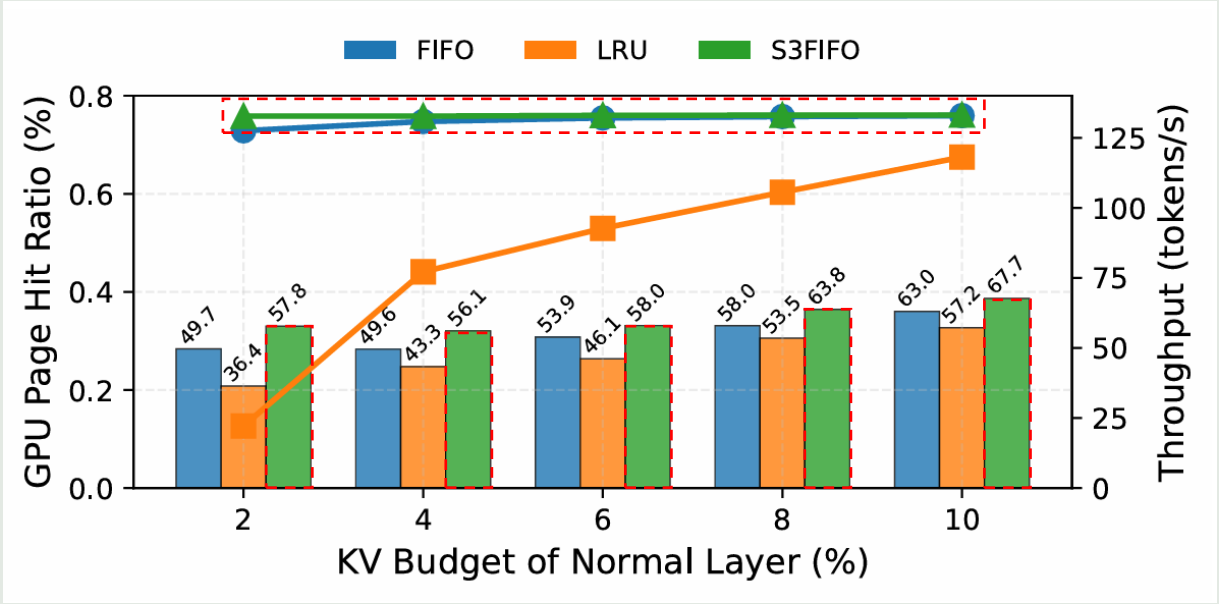
Eviction: S3FIFO[1] outperforms LRU/FIFO

Quick demotion of one-hit objects

+33% throughput at 17% KV budget




[1] Yang J, Zhang Y, Qiu Z, et al. FIFO queues are all you need for cache eviction[C]//Proceedings of the 29th Symposium on Operating Systems Principles. 2023: 130-149.




OmniKV decoding throughput with a fixed batch size of 8 on LLaMA-8B

# Conclusion

## Our Contributions

- First framework to bridge token-level recallable sparsity and page-level KV cache management.
- Plugin-driven design enables rapid integration of new sparsity algorithms with minimal framework intrusion.
- Achieves up to 1.8x decoding throughput improvement across models, batch sizes and methods.

## Future Work

- Explore adaptive scheduling strategies to transparently enable sparsity plugins for mixed regular & long-context requests.
- Maximize throughput under SLOs by tuning the recall–computation trade-off.

# Q&A