DUAL-BLADE: Dual-Path NVMe-Direct KV-Cache Offloading for Edge LLM Inference

Bodon Jeong[1,3], Hongsu Byun[1], Youngjae Kim[1], Weikuan Yu[2], Kyungkeun Lee[3], Jihoon Yang[1], Sungyong Park[1,†]
[1]*Sogang University*, Seoul, Republic of Korea, [2]*Florida State University*, FL, USA, [3]*Samsung Electronics Co.*
{bd91jeong, byhs, youkim, yangjh, parksy}@sogang.ac.kr, wyu3@fsu.edu, kavin.lee@samsung.com

***Abstract*—The increasing deployment of Large Language Model (LLM) inference on edge AI systems demands efficient execution under tight memory budgets. A key challenge arises from Key–Value (KV) caches, which often exceed available device memory. Although NVMe-based offloading offers scalable capacity, existing file-based designs rely heavily on the kernel page cache, leading to cache thrashing, unpredictable latency, and high software overhead under memory pressure. We present DUAL-BLADE, a *dual-path KV residency framework* that dynamically assigns KV tensors to either a page-cache path or an NVMe-direct path based on runtime memory availability. The NVMe-direct path bypasses the filesystem by mapping KV tensors to contiguous logical block address (LBA) regions, enabling low-overhead direct storage access. DUAL-BLADE further incorporates adaptive pipeline parallelism to overlap storage I/O with GPU DMA, improving inference throughput. Our evaluation shows that DUAL-BLADE substantially mitigates I/O bottlenecks, reducing prefill and decode latency by up to 33.1% and 42.4%, respectively, while improving SSD utilization by 2.2× across diverse memory budgets.**

***Index Terms*—LLM Inference, KV Cache Offloading, Edge AI Systems, NVMe SSD, Page Cache, Kernel Bypass.**

I. INTRODUCTION

Driven by privacy concerns and the need for low-latency local inference, LLM serving is increasingly migrating from cloud infrastructure to edge AI systems, including on-premise and embedded deployments [1]–[4]. These edge AI systems are typically single-GPU platforms operating under tight memory budgets (8–32 GB DRAM), often adopting unified memory where the CPU and GPU share the same DRAM pool [5]. Under these constraints, 7–8B parameter models have emerged as the dominant choice [6]–[8], enabling applications such as AI assistants, long-form document analysis, multimodal search, and log parsing [9]–[12]. Unlike cloud datacenters with elastic resources, however, edge AI systems must operate strictly within fixed physical memory limits, making efficient memory management a central challenge for LLM inference.

On edge AI systems, GPU device memory is often near saturation after loading model parameters. Some systems further spill a portion of model weights into host DRAM [13], [14], reducing DRAM slack for runtime states. Under these conditions, when the KV-cache, which stores per-token key/value vectors for attention to avoid recomputation [15], exceeds GPU memory, it is pushed into the same host DRAM pool [14], [16], competing with co-located workloads. The KV-cache size varies dynamically with context length and batch size, and can exhaust both GPU memory and the remaining host DRAM budget, creating a structural bottleneck for modern LLM inference on edge AI systems.

As a result, host DRAM is often insufficient to accommodate the growing KV-cache, making a DRAM–SSD memory-tiering hierarchy for the KV-cache increasingly important. Accordingly, many inference serving platforms have begun integrating NVMe SSDs to extend memory capacity [14], [17], [18]. NVMe SSDs provide terabytes of capacity at a fraction of the cost of DRAM, offering a viable solution for hosting massive KV-caches on single-GPU systems [14], [19]–[21]. However, naively mapping these SSDs via standard file systems introduces severe software-stack overheads and I/O inefficiencies, preventing full utilization of device bandwidth.

Motivation. A common memory tiering method in LLM inference systems is to use the OS page-cache via file-backed mapping (`mmap`) [14], [17]. However, during the *prefill phase*, massive writes exert pressure on the page-cache, triggering synchronous write-back to reclaim space, leading to write stalls. Furthermore, in the *decode phase*, the auto-regressive pattern of repeatedly reading the accumulated KV-cache causes severe *page-cache thrashing*, where cached pages are evicted before reuse. As a result, the effective page-cache hit ratio collapses, crippling I/O performance (§III-A).

Moreover, the software overhead traversing the virtual file system (VFS), filesystem, block layer, and device driver introduces critical latency that hinders full utilization of the NVMe SSD (§III-B). This long-standing bottleneck is widely recognized in the storage domain, motivating numerous kernel-bypass solutions to eliminate redundant layers [22]–[25].

Another critical issue is the disruption of sequential locality at the SSD device level. Since the Linux multi-queue block layer (`blk-mq`) distributes requests across multiple NVMe queues, the originally contiguous KV-cache stream becomes fragmented, resulting in an interleaved and non-sequential arrival order at the SSD controller (§III-C).

Key Design and Contributions. The motivation above reveals three fundamental bottlenecks in existing KV-cache offloading designs: (i) page-cache thrashing caused by indiscriminate `mmap`-based placement, (ii) excessive software overhead along the kernel storage stack, and (iii) loss of sequential locality at the NVMe device level. DUAL-BLADE directly addresses these challenges through three complementary designs:

- **Dual-Path KV Residency.** DUAL-BLADE decouples the I/O path into a page-cache path and an NVMe-direct path. KV tensor groups that fit in the dynamically estimated page-

† Corresponding author.

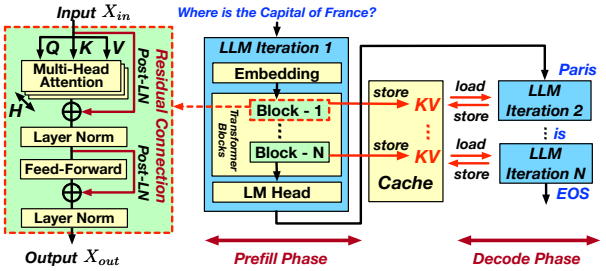


Fig. 1: LLM transformer architecture [15].

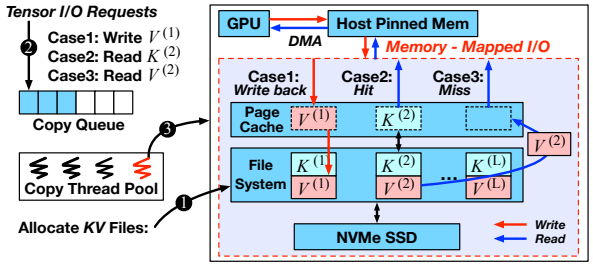


Fig. 2: KV-cache disk offload workflow [14].

cache budget are placed on the page-cache path to avoid thrashing and sustain high hit ratios, while remaining tensors are routed to the NVMe-direct path for direct access to contiguous NVMe logical block addresses (LBAs) (§IV-A).

- **NVMe-direct with Sequential-LBA Placement.** To minimize software overhead and maximize SSD utilization, DUAL-BLADE bypasses the kernel storage stack on the NVMe-direct path. It further enforces sequential LBA placement for KV tensors, preserving device-level sequentiality and reducing Flash Translation Layer (FTL) and controller scheduling overhead (§IV-B).
- **Adaptive Pipeline Parallelism.** In addition, DUAL-BLADE incorporates adaptive pipeline parallelism to overlap KV-cache storage I/O with GPU DMA. By dynamically coordinating I/O and computation across threads under bandwidth contention, this design effectively hides residual I/O latency and maximizes end-to-end inference throughput (§IV-C).

We implemented the proposed design on a modern disk-offloading-based inference system, FlexLLMGen [14]. The evaluation results demonstrate that our approach reduces end-to-end inference latency by up to 33.1% in the prefill phase and 42.4% in the decode phase compared to baseline methods, while boosting SSD utilization by up to 2.2× and maintaining robust performance under diverse memory constraints.

II. BACKGROUND

A. LLM Transformer Inference and KV-Cache

Figure 1 shows an LLM built from stacked transformer blocks. Each block has two sublayers, multi-head attention (MHA) and a feed-forward network (FFN), each followed by a residual connection and layer normalization. MHA maps input X to Query (Q), Key (K), and Value (V), computes attention from Q–K similarity, and weights V to form a contextual representation for each token. The computation runs in parallel over H heads, and their outputs are combined to produce the MHA result. This then goes to the FFN, yielding the block output X_{out} for the next block.

LLM inference has two phases, *Prefill* and *Decode*. In prefill, all prompt tokens pass through the embedding, transformer stack, and language modeling head to produce the first token (*LLM Iteration 1*). That token becomes the input to decode. Decode proceeds in an auto-regressive manner, generating the next token from the previous one (*LLM Iteration 2…N*). Generation stops at a maximum length or an EOS token.

During each decode step, the model *loads* all prior KV pairs from the KV-cache to compute attention for the current token, then *stores* the newly generated KV pair back into the KV-cache. This avoids recomputing past KVs at every step, making KV caching a standard technique in modern LLM inference. However, KV-cache size depends on context length and batch size and can readily exceed GPU memory.

B. Memory Tiering for Efficient KV-Cache Management

When KV-cache exceeds GPU memory, LLM inference systems offload it to host DRAM and, increasingly, to NVMe SSDs as an extended memory tier. To illustrate KV-cache offloading in practice across diverse inference platforms, we describe FlexLLMGen [14], a recent LLM serving system that enables high-throughput single-GPU inference by tiering the KV-cache between host DRAM and NVMe SSDs.

As shown in Figure 2, the system first creates $2L$ files for $K^{(i)}$ and $V^{(i)}$ tensors[1] per layer $i \in \{1, \ldots, L\}$ (❶). When a layer is needed during prefill or decode, an I/O request for its $K^{(i)}/V^{(i)}$ is enqueued to the *Copy Queue* (❷), and a *Copy Thread Pool* worker performs the transfer (❸). Data moves in two segments: GPU↔CPU via pinned-memory DMA, and CPU↔filesystem via memory-mapped I/O, with the OS page-cache tiering DRAM and SSD. Writes land in the page cache and flush asynchronously (*Case1: Write* $V^{(1)}$). For reads, page-cache hits return immediately (*Case2: Read* $K^{(2)}$). On a miss, the kernel fetches from disk, fills the page-cache, and then serves the data (*Case3: Read* $V^{(2)}$).

C. Memory Constraints in Edge LLM Serving

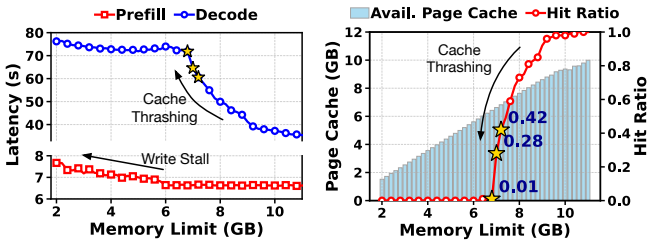
In edge AI systems, GPU memory is largely consumed by model parameters. The KV-cache size varies with context length and batch size, and can exceed GPU memory capacity. **Target Scenario.** We focus on deployments where the KV-cache outgrows GPU memory, spills into host memory, and is eventually offloaded to SSDs when the host-memory budget is tight under contention from co-located workloads. To evaluate disk offloading under controlled yet realistic memory pressure, we fix the KV-cache workload and sweep the host-memory limit, spanning regimes from severe under-provisioning (relative to the KV-cache) to ample headroom.

III. PRELIMINARY STUDY AND MOTIVATION

A. Page-Cache Thrashing in KV Workloads

We analyze *page-cache thrashing under limited host memory*. Figure 3 reports prefill/decode latency, available page-cache, and page-cache hit ratio as we vary the host memory

[1]Here, "tensor" denotes the per-layer KV component ($K^{(i)}$ or $V^{(i)}$). We use the terms interchangeably.



(a) Inference latency (b) Page-cache utilization

Fig. 3: Page-cache thrashing under host memory limits.

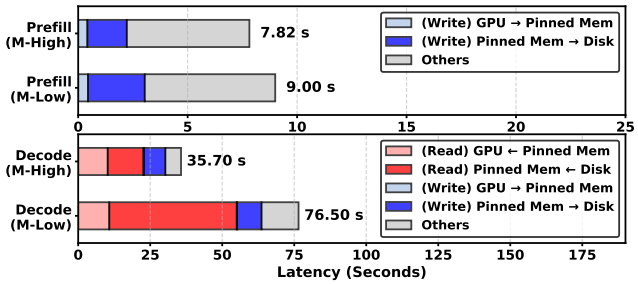


Fig. 4: Total latency breakdown for prefill and decode phases.

limit from 2 to 11 GB. We use OPT-6.7B [26] with a 512-token prompt, 32-token generation, and batch size 32. The resulting KV-cache totals 8.57–9.11 GB. For example, at a host memory limit of 11 GB, the page-cache can hold the entire KV-cache. We estimate available page-cache from cgroup stats sampled every 1 s. The page-cache hit ratio in the decode phase is the fraction of total read bytes served from the page-cache. The full experimental setup is described in §V-A.

Prefill is write-heavy. The monotonic latency rise in Figure 3(a) once memory drops below $\approx$ 6 GB follows directly from page-cache behavior. A smaller page-cache forces synchronous evictions before background write-back, stalling write I/O and increasing prefill latency.

Decode is read-heavy. For each generated token, the system rereads the entire accumulated KV-cache. Thus, the reusable working set starts right after prefill and increases each iteration as decode appends new KVs. With 10–11 GB, the page-cache can hold the full KV-cache, yielding $\approx$ 100% hit ratio and low latency. As the host memory limit falls below the KV working set (8.57–9.11 GB), one might expect the page-cache hit ratio to decline gradually as misses increase. Instead, we observe a sharp cliff. At the starred points in Figure 3(b), the page-cache hit ratio collapses from 42% to $<$ 1%, coinciding with the spike in decode latency in Figure 3(a).

This creates a thrashing zone at 2–7 GB. The hit ratio stays $<$ 1% despite several GB of page-cache being available. To our knowledge, this is the first experimental identification of page-cache thrashing as a major bottleneck in KV offloading for LLM inference, consistent with OS literature [27]–[29].

Root Cause Analysis. The root cause is the *conflict between auto-regressive reuse and LRU-based page caching*. In decode, attention repeatedly reads the growing KV working set. When the page-cache is smaller than this working set, LRU triggers cascading evictions. Reading one tensor evicts another that will soon be reread, leaving the cache full yet ineffective and driving sustained disk I/O. Hence, a naive `mmap`-based design fails to properly tier KV data across DRAM and SSD and can collapse into page-cache thrashing, motivating strategies that structurally avoid thrashing for such KV access patterns.

B. Kernel Storage I/O Path as the Core Inference Bottleneck

Figure 4 contrasts two extremes of page-cache efficacy, M-High and M-Low, instantiated using the 11 GB and 2 GB host-memory limits in Figure 3. Beyond the obvious benefit that more page-cache hits are better, it shows that *prefill* and *decode* have fundamentally different dominant costs. Here, disk I/O refers to file-backed `mmap` I/O via the page-cache.

Prefill is a one-shot *large write* of input-token KVs. The performance difference between the two prefill cases mainly comes from the differing impact of page-cache write stalls. In M-High and M-Low, "Others" (mostly GPU compute) dominates (72% and 66%), indicating that prefill is largely compute dominated. However, disk I/O ("Pinned Mem $\rightarrow$ Disk") still accounts for a non-trivial fraction (24% and 28%).

Decode repeatedly performs *large read + small write*. The performance gap between the two decode cases is driven by their page-cache hit ratios, which are ($\approx$ 100%) in M-High and ($\approx$0%) in M-Low. In M-High and M-Low, decode accounts for 82% and 89% of end-to-end inference time. Within decode, disk I/O ("Pinned Mem $\leftrightarrow$ Disk") accounts for 56% (M-High) and 69% (M-Low), indicating that decode is largely disk I/O dominated rather than compute dominated.

Overall, disk I/O is a major cost in both prefill and decode across the two extreme memory regimes (M-High and M-Low), although it is secondary to GPU compute in prefill. This motivates optimizing the storage I/O path.

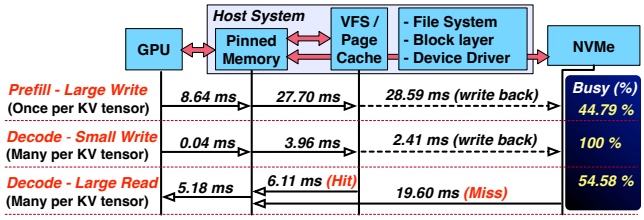


Fig. 5: Per-tensor I/O request latency breakdown for a single layer.

Figure 5 shows the latency breakdown for each K or V tensor I/O request issued within a single transformer layer. Request sizes are as follows in our setup: 128 MB (prefill write), 128–135 MB (decode read), and 256 KB (decode write). We measure the NVMe *busy ratio* (utilization), the fraction of time the device is actively processing I/O, using `bpftrace` [30] at the block layer by correlating submission and completion events and accounting for overlaps and idle gaps across requests over the measurement window.

On write-back or a page-cache miss, requests traverse the full kernel software stack (VFS$\rightarrow$filesystem$\rightarrow$block layer$\rightarrow$driver), accumulating latency that is substantial on a per-request basis. Decode write (256 KB) reaches 100% busy because the kernel issues it as a single I/O (e.g., one `bio`). In contrast, prefill writes (128 MB) and decode reads (128–135 MB) are chunked at the block layer. Each chunk pays full-

stack overhead, leaving device idle gaps between commands and yielding low busy ratios (45% and 55%) [24], [31], [32]. This underutilizes NVMe hardware. Eliminating per-chunk overhead via a lighter I/O path to maximize device busy time is therefore a key optimization target.

C. Logical-to-Physical Access Disparity in KV-Cache I/O

The KV-cache workload is flash-friendly. Logically (from the tensor view), it is dominated by large sequential writes (prefill) and large sequential reads (decode).

(a) Tensor Access (b) LBA Access (c) Submission Queue

Fig. 6: Logical (tensor-level) and physical (device-level) access patterns. The x-axis denotes the I/O index in submission order.

Figure 6(a) illustrates the linearized logical address over tensor access order, exhibiting near-perfect sequentiality in both the prefill and decode phases. Decode writes are minimal and appear only as sparse points. In contrast, the device-level logical block address (LBA) stream breaks this sequentiality, as shown in Figure 6(b), where prefill writes remain strictly increasing as an idealized sequential pattern, whereas decode reads split into multiple interleaved streams.

Root Cause Analysis. The root cause appears in the NVMe submission-queue (SQ) ID distribution of Figure 6(c). Writes concentrate in a few queues driven by a small set of kernel threads [33], whereas reads fan out across many queues. Page-cache misses trigger parallel submissions from application and kernel threads [34]. Thus a logically single sequential read becomes, after the Linux multi-queue block layer (`blk-mq`), an interleaved mix of short sequential fragments.

While `blk-mq` schedulers improve throughput for complex I/O mixes, their kernel-stack and scheduling overhead (on the μs scale) on low-latency NVMe SSDs can be counter-productive for a single sequential stream [24], [35], [36]. Multi-queueing is generally beneficial, but domain-specific sequential workloads such as KV-cache access can saturate the device with a single thread. Preserving end-to-end sequentiality to the NVMe controller reduces internal work (e.g., FTL mapping, reordering). As shown in §III-B, when I/O dominates, shaving even μs-level overhead accumulates and reduces total inference time. Maintaining a strictly sequential submitted-LBA pattern is therefore a sound optimization goal.

D. Opportunity: NVMe-direct Path (Kernel-bypass)

The KV-cache workload for LLM inference is fundamentally different from general file I/O. First, its access pattern is predictable and largely sequential. The prefill phase involves sequential writes, while the decode phase requires sequential reads with appended writes. There are no overwrites and tensors are not shared concurrently. Moreover, the inference stack can predetermine the entire KV footprint and on-disk layout before execution. Second, the KV-cache is temporal and ideally resides in DRAM. Consequently, standard filesystem guarantees such as persistence, consistency, and metadata durability are unnecessary. Leveraging these characteristics, we employ a lightweight mechanism called NVMe-direct that bypasses the filesystem to handle offloaded KV data efficiently.

TABLE I: I/O Path Comparison (128KB Seq. Read/Write, QD=32)

Metric	ext4 File I/O	`io_uring_cmd`	SPDK
Path Depth	Deep (VFS/ext4)	**Short** (Kernel-bypass)	Short (User Driver)
Mechanism	Standard File I/O	**Passthrough**	User-space Polling
Write Latency (Avg. / 99.99th)	668 μs / 3.50 ms	**662 μs / 1.62 ms**	662 μs / 1.14 ms
Read Latency (Avg. / 99.99th)	351 μs / 17.8 ms	**340 μs / 1.19 ms**	339 μs / 0.86 ms

In our implementation, we use the *NVMe-direct path* as a kernel-bypass channel that issues device-native NVMe commands via `io_uring_cmd` [25]. To justify this, we compare it with SPDK [23] and standard filesystem I/O in Table I. As shown, both bypass methods significantly outperform the filesystem, particularly in tail latency. While SPDK offers slightly lower latency, it incurs high complexity and resource consumption due to busy-polling and core binding. In contrast, `io_uring_cmd` achieves comparable performance through a standard kernel interface. Thus, we adopt `io_uring_cmd` for its balance of performance and ease of implementation, noting that our design is extensible to SPDK.

IV. Design and Implementation

From §III, KV-cache offloading faces three bottlenecks: (i) page-cache thrashing, (ii) kernel-stack overhead, and (iii) tensor–LBA mismatch. DUAL-BLADE is a high-performance offloading architecture for edge AI systems under tight memory budgets that addresses them with the following components.

- **Dual-Path KV Residency.** Partition KV tensors into a page-cache path (Group 1) and an NVMe-direct path (Group 2) that bypasses the kernel storage stack. This separation avoids page-cache thrashing during decode, while also preventing write stalls during prefill.
- **NVMe-direct with Sequential-LBA Placement.** Accelerate the NVMe-direct path by issuing device-native NVMe I/O that bypasses the kernel stack, aiming to (i) increase NVMe utilization by eliminating software overhead, and (ii) reduce NVMe controller latency by enforcing a strictly sequential LBA submission pattern.
- **Adaptive Pipeline Parallelism.** To further maximize inference performance, in multi-threaded configurations where storage bandwidth becomes saturated, dynamically enforce a staggered start on trailing threads to overlap storage I/O and GPU DMA, thereby mitigating resource contention.

A. Dual-Path KV Residency

The dual-path KV residency assigns KV tensors of hidden layers to two I/O path groups. Consequently, during the prefill

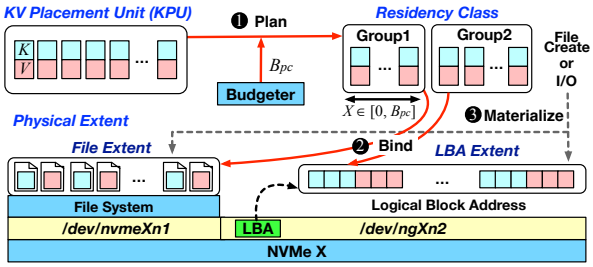


Fig. 7: High-level architecture of dual-path KV cache residency.

and decode phases, each tensor is processed via the I/O stack corresponding to its assigned path.

Figure 7 outlines this workflow. At initialization, each KV pair is abstracted as a *KV Placement Unit* (KPU). A budgeter computes the usable page-cache and classifies each KPU into two classes: Group 1 (page-cache path) and Group 2 (NVMe-direct path) (❶). Each group is then bound to a physical address space, mapping Group 1 to filesystem extents and Group 2 to NVMe-namespace LBA extents (❷). This binding remains logical until the first access, at which point it is materialized as an actual file or block (❸). The details of the three steps, **Plan**, **Bind**, and **Materialize**, are as follows.

Plan. First, the budgeter estimates the available page-cache budget B_{pc} under system and cgroup limits:

$$M^* \leftarrow \min(M_{avail},\ M_{max} - M_{anon+shmem}) \quad (1)$$

$$B_{pc} \leftarrow \max(0,\ M^* - N_{threads} \cdot M_{pin}) \quad (2)$$

Pinned memory (M_{pin}) is reserved exclusively for GPU DMA usage by each copy-thread, matching the size of a single K or V tensor (KPU) of a hidden layer. Consequently, the total reservation $N_{threads} \cdot M_{pin}$ constitutes a constant DRAM overhead for inference serving, distinct from the page-cache.

Algorithm 1 uses a knob $X \in [0, B_{pc}]$ to partition each hidden layer's KPU pair $(K^{(i)},\ V^{(i)})$ into Group 1 or Group 2. The knob X sets the total bytes admitted to the page-cache path. We first compute how many layers n_1 fit (line 1), then assign KPU pairs from layer 1 to layer n_1 to Group 1 (x_i=1) and the rest to Group 2 (x_i=0) (lines 2–4).

Prior work ranks KV entries by reuse or importance. Although our mechanism treats all tensors uniformly, the design is pluggable. For instance, a ranker can drive X such that only top-ranked tensors occupy the page-cache path, while the others are routed to the NVMe-direct path.

Bind & Materialize. *Bind* maps residency classes to storage. Group 1 KPUs are assigned to filesystem extents (using `mmap`), and Group 2 KPUs to NVMe namespace LBA extents (detailed in §IV-B). A fixed DRAM reservation is brittle under time-varying memory pressure (risking OOM), whereas the OS page-cache via `mmap` provides best-effort DRAM caching.

To isolate paths, we separate the NVMe space using partitions or distinct namespaces. During inference, each request is dispatched by the binary decision x_i, effectively *materializing* the logical binding. For read and write operations, Group 1 utilizes the page-cache under filesystem control, whereas Group 2 directly accesses reserved LBA blocks via NVMe-direct.

Algorithm 1: KPU Residency Planner (parameterized by X)

Input: The set of KPU for all layers $\{(K^{(i)}, V^{(i)})\}_{i=1}^{L}$,
Size of a single KPU (either K or V) S_{kpu}, knob $X \in [0, B_{pc}]$
Output: A binary decision $\{x_i\}_{i=1}^{L}$ where $x_i \in \{0, 1\}$

1 $n_1 \leftarrow \min(\lfloor X/(2 \cdot S_{kpu}) \rfloor, L)$;
2 **for** $i \leftarrow 1$ **to** L **do**
// Residency Class: Group 1, page-cache path
3 **if** $i \le n_1$ **then** $x_i \leftarrow 1$; FILEBIND($K^{(i)}$); FILEBIND($V^{(i)}$) ;
// Residency Class: Group 2, NVMe-direct path
4 **else** $x_i \leftarrow 0$; LBABIND($K^{(i)}$); LBABIND($V^{(i)}$) ;
5 **return** $\{x_i\}$

B. NVMe-direct with Sequential-LBA Placement

The NVMe-direct design accelerates Group 2 KPUs (§IV-A) by removing kernel storage stack overheads and the tensor–LBA mismatch. The key idea is to bypass the kernel-stack (including the filesystem) and map tensors to a contiguous LBA range in the NVMe namespace.

Precondition. The tensor size defines the minimum I/O unit and must be an integer multiple of the device LBA size (typically 512 B or 4 KiB). The minimum tensor layout is the single-token (S=1) K or V tensor with shape $(1, B \cdot H, D)$, where $S, B, H,$ and D denote sequence length, batch size, heads, and per-head dimension. For OPT models [37], this alignment generally holds when $B \ge 1$ (Table II). For edge cases (e.g., OPT-13B ≈10 KiB, which is not a 4 KiB multiple), an even B is selected to ensure alignment.

LBA Bind. The <*Bind*> process in Figure 8 maps Group 2 KPUs onto a contiguous namespace extent via a hash map $\mathcal{M}$ from tensor ID n to its start LBA and block count:

$$\mathcal{M}:\ n\ \rightarrow\ \langle\, lba_{start},\ n_{blocks}\,\rangle \quad (3)$$

$$\text{extent}(n)\ =\ \big[\, lba_{start}(n),\ lba_{start}(n) + n_{blocks}(n)\,\big) \quad (4)$$

The binding adheres to three invariants: (i) *Alignment* (tensor I/O unit is a multiple of LBA size), (ii) *Disjointness* (extents do not overlap), and (iii) *Contiguity* (each next tensor begins where the previous ends). Only the first tensor's lba_{start} is user-specified. Others follow:

$$n_{blocks}(n) = \text{bytes(tensor } n) \,/\, lba_{size} \quad (5)$$

$$lba_{start}(n{+}1) = lba_{start}(n) + n_{blocks}(n) \quad (6)$$

Here, bytes() is the byte size of the tensor. For instance, setting $lba_{start}(t\_531\_k) = 2048$ sequentially determines the start LBAs of t_532_v, t_533_k, ... Thus all Group 2 KPUs occupy one large contiguous extent, preserving the KV tensor's logical sequential stream at the NVMe controller.

GPU↔NVMe Co-DMA over Host Pinned Memory. In the case of the page-cache path (Group 1), write operations follow a 3-hop sequence: (1) GPU→pinned memory (DMA), (2) pinned memory→page-cache (`memcpy`), and (3) page-cache→NVMe (DMA). Read operations follow the inverse

TABLE II: Minimal tensor I/O unit for a single KV component (K or V), FP16/BF16. General formula: bytes = $B \times H \times D \times 2$.

Model	H	D	Formula (bytes)	Size (KiB)
OPT-1.3B	32	64	$B \times 32 \times 64 \times 2$	**4** KiB × B
OPT-6.7B	32	128	$B \times 32 \times 128 \times 2$	**8** KiB × B
OPT-13B	40	128	$B \times 40 \times 128 \times 2$	**10** KiB × B

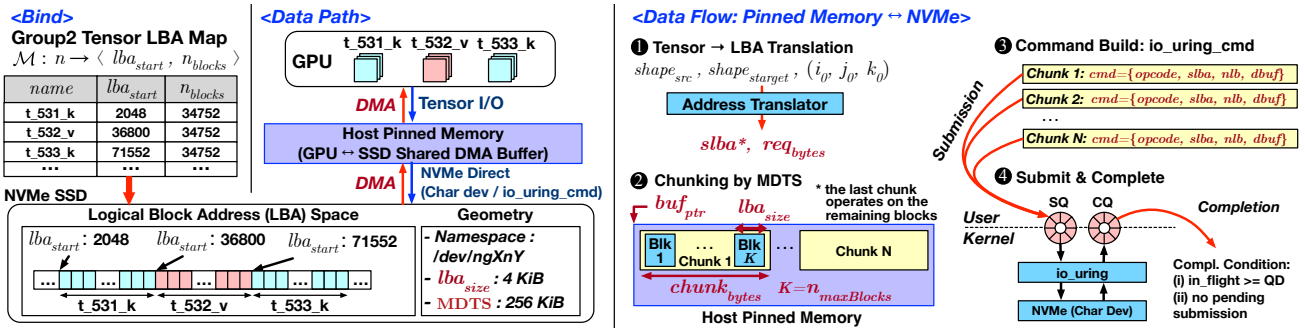


Fig. 8: Architecture of the NVMe-direct path: Bind, Data Path, and Data Flow.

Algorithm 2: Tensor-Index–to–LBA Translation

Input: $name$ **(tensor id)**, $shape_{src}=(f_0, f_1, f_2)$ **(source tensor shape)**, $shape_{tgt}=(d_0, d_1, d_2)$ **(target tensor shape)**, (i_0, j_0, k_0) **(offset indices in target)**, e **(bytes per element, e.g., 2 for FP16/BF16)**, lba_{size} **(bytes)**

Output: $slba^\star$, req_{bytes}

1 **Precondition (as stated in the section).** The minimal tensor I/O unit is an integer multiple of lba_{size} and placement starts on unit boundaries. Thus req_{bytes} and $offset_{bytes}$ are multiples of lba_{size}.

// lookup from the tensor→LBA map

2 $(lba_{start}, \_) \leftarrow \mathcal{M}[name]$

// row-major element offset and byte offset

3 $offset_{elem} \leftarrow ((i_0 \cdot d_1 + j_0) \cdot d_2 + k_0)$

4 $offset_{bytes} \leftarrow offset_{elem} \cdot e$

// anchor LBA and requested payload size

5 $slba^\star \leftarrow lba_{start} + (offset_{bytes}/lba_{size})$

6 $req_{bytes} \leftarrow (f_0 \cdot f_1 \cdot f_2) \cdot e$

7 **return** $(slba^\star, req_{bytes})$

sequence (or 2-hop path on a page-cache hit).

Conversely, the <*Data Path*> in Figure 8 illustrates the NVMe-direct path (Group 2) for tensor I/O between the GPU and NVMe. By bypassing the page-cache, this path removes `memcpy` and forms a symmetric 2-hop flow. Specifically, writes follow GPU→buffer→NVMe `WRITE`, while reads follow NVMe `READ`→buffer→GPU. We use a Co-DMA scheme, where the GPU and NVMe share the same per-thread pinned buffer, which is already allocated for GPU DMA.

Three factors validate this design: (i) *DMA buffer* constraints are aligned, meaning buffers are page-locked and 4 KiB-aligned, allowing both the GPU (via IOMMU/IOVA) and NVMe (via PRP/SGL) to access the same buffer. (ii) *Exclusive ownership* ensures copy-threads use disjoint pinned regions and operate unidirectionally (read or write) per tensor. (iii) *Dual views* enable the same memory to serve as a structured tensor to the GPU and a byte array to NVMe.

Data Flow: Pinned Memory ↔ NVMe. To execute direct reads or writes on NVMe via `io_uring_cmd`, the high-level tensor I/O request must be translated into device-specific native NVMe I/O parameters. In the <*Data Flow*> of Figure 8, a copy thread translates tensor info into the target read/write location (LBA) and request size (❶). Algorithm 2 looks up lba_{start} in $\mathcal{M}$ (line 2), computes the row-major byte offset from shape_{target} and indices (i_0, j_0, k_0) (lines 3–4), then derives $slba^\star$ and req_{bytes} from shape_{source} (lines 5–6).

Next, the copy-thread chunks req_{bytes} to fit within the NVMe controller's *maximum data transfer size* (MDTS), the per-command payload cap (e.g., 256 KB) (❷). To maximize throughput while preserving lba_{size} alignment, the chunk size is set to the largest multiple of lba_{size} fitting within the MDTS. Consequently, the number of chunks and maximum blocks per chunk are derived as:

$$chunk_{bytes} = \text{align\_down}(\text{MDTS}, lba_{size}) \tag{7}$$

$$n_{chunks} = \left\lceil \frac{req_{bytes}}{chunk_{bytes}} \right\rceil, \; n_{maxBlocks} = \frac{chunk_{bytes}}{lba_{size}} \tag{8}$$

For each chunk $N \in [1, n_{chunks}]$, the copy-thread constructs an NVMe native command (❸) with: (i) *opcode* $\in \{\texttt{READ}, \texttt{WRITE}\}$, (ii) $(nsid, slba, nlb)$ denoting the namespace ID, the starting LBA, and the logical block count (where nlb is 0-based, transferring $nlb + 1$ blocks), and (iii) the host data buffer $dbuf$ pointing to the pinned-memory address for chunk N. The per-chunk parameters are:

$$slba = slba^\star + (N-1) \times n_{maxBlocks} \tag{9}$$

$$nlb = \min\left(n_{maxBlocks}, n_{remains}\right) - 1 \tag{10}$$

$$dbuf = buf_{ptr} + (N-1) \times chunk_{bytes} \tag{11}$$

Per-chunk NVMe commands are submitted asynchronously to the submission queue (SQ), and completions arrive on the completion queue (CQ) (❹). The copy-thread harvests completion queue entries (CQEs) when in-flight requests reach the queue depth (QD) or when draining remaining chunks. A tensor transfer request, whether read or write, is finalized once all its constituent chunk CQEs are collected.

Dataset Management–Deallocate (TRIM). On context teardown, allocated LBA space is reclaimed via NVMe *Dataset Management (DSM) deallocate*. For each tensor, we query $\mathcal{M}$ for $(lba_{start}, n_{blocks})$ and submit TRIM via `io_uring_cmd`. This command explicitly hints to the NVMe controller that the specified LBA range no longer contains valid data, enabling efficient and rapid deallocation of large, contiguous KV ranges.

Although TRIM can trigger internal garbage collection (GC) potentially affecting latency, since concurrent requests are processed as batched tensors or scheduled serially in a single-GPU system, our architecture effectively eliminates LBA fragmentation. Consequently, the device maintains a near-ideal Write Amplification Factor (WAF) close to 1, rendering GC overhead negligible.

C. Pipeline Parallelism between GPU DMA and Storage I/O

Building on the dual-path and NVMe-direct design, this section presents a pipeline parallelism to maximize throughput by overlapping GPU DMA and storage I/O in a multi copy-

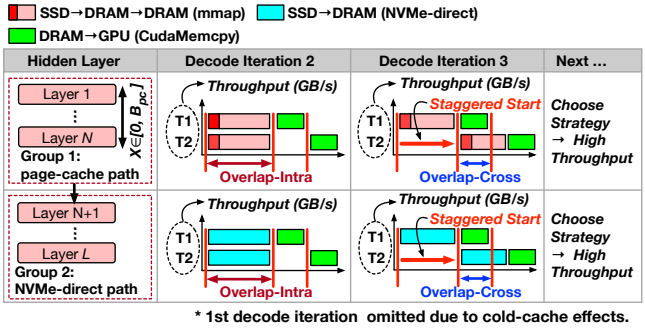


Fig. 9: Adaptive pipeline parallelism with two overlap strategies.

thread environment. This dynamically selects between two pipeline strategies, *overlap-intra* and *overlap-cross*.

Parallelism Types and Constraints. In a multi copy-thread setting, two threads can process the K and V tensors of each layer concurrently (e.g., thread 1 for K and thread 2 for V). We categorize the parallelism into two types. (i) *Overlap-Intra* overlaps I/O operations within the same hardware path (e.g., parallel NVMe reads), which improves bandwidth utilization but may introduce contention when the storage is saturated. While NVMe SSDs support parallel I/O via multiple hardware queues, copy streams issued by multiple threads are effectively serialized on the GPU [38]. (ii) *Overlap-Cross* overlaps I/O across different hardware paths (e.g., NVMe reads and GPU DMA), enabling effective latency hiding by utilizing independent hardware and PCIe resources.

Adaptive Pipeline Strategy Selection. Because GPU DMA and storage I/O naturally overlap during the prefill phase, we focus on the decode phase. Figure 9 illustrates the comparison and selection of two pipeline strategies for each path (Group 1 and Group 2) during the decode phase.

(1) Warm-up: To exclude transient latency caused by cold page-cache effects in the Group 1 path, the first decode iteration is omitted from measurement. *(2) Iteration 2 (Overlap-Intra):* Storage reads from two copy-threads are parallelized, while GPU H2D DMA is executed serially. This maximizes storage bandwidth utilization when the storage is not saturated. *(3) Iteration 3 (Overlap-Cross):* A staggered start delays the storage read on thread T2, overlapping it with GPU DMA issued by thread T1 on a different hardware path. Under saturated storage bandwidth, this mitigates contention and improves overall pipeline throughput. *(4) Strategy Selection:* The throughput of the two iterations is compared for each group, and the higher-performing strategy is fixed for subsequent decoding iterations. The adaptive pipeline incurs only a bounded overhead from lightweight throughput monitoring and at most one trial iteration for strategy comparison. This constant cost is amortized as generation length increases.

V. Evaluation

A. Experimental Setup

Experimental Setup. All experiments were conducted on a single-GPU system equipped with an Intel Core Ultra 7 265K processor (20 cores at 3.8 GHz), 16 GB of host memory, and an NVIDIA RTX 5060 Ti (16 GB). The software environment includes Ubuntu 24.04.2, Linux 6.8.0, liburing 2.5, and PyTorch 2.8.0 (+CUDA 12.8). We employed FlexLLMGen [14] (commit `004ffef`) as the experimental framework.

To assess device-dependent effects and investigate how storage media characteristics impact end-to-end inference latency, we evaluate two NVMe SSDs with distinct classes and I/O constraints: SSD A (Samsung PM9D3a, PCIe Gen5, 4 KiB LBA, 256 KiB MDTS) and SSD B (Samsung 990 PRO, PCIe Gen4, 512 B LBA, 2 MiB MDTS). This comparison validates whether our architectural benefits persist across different hardware generations and internal controller specifications, as LBA size and MDTS directly constrain NVMe-direct alignment and per-command transfer granularity.

Configuration. We evaluated the OPT-6.7B model [26]. In FlexLLMGen, only the KV-cache is offloaded, while parameter offloading is disabled. Unless otherwise noted, the prompt length is 512 tokens, the generation length is 32 tokens, and the batch size is 32. Although we evaluated various parameter settings, the trends remain consistent. Therefore, we report this representative case. We use two copy-threads, and additional threads provide no benefit and typically remain idle.

Comparison. We evaluate the effects of three design choices. CachePolicy-Only isolates the impact of the *KV Residency Policy*. NVMe-direct-Only evaluates the benefit of the *NVMe-direct* I/O path. DUAL-BLADE combines both designs as a dual-path NVMe-direct approach. Baseline corresponds to vanilla FlexLLMGen. The detailed configurations are summarized in Table III. The third key component, *Pipeline Parallelism*, is applied in a multi copy-thread environment to CachePolicy-Only, NVMe-direct-Only, and DUAL-BLADE.

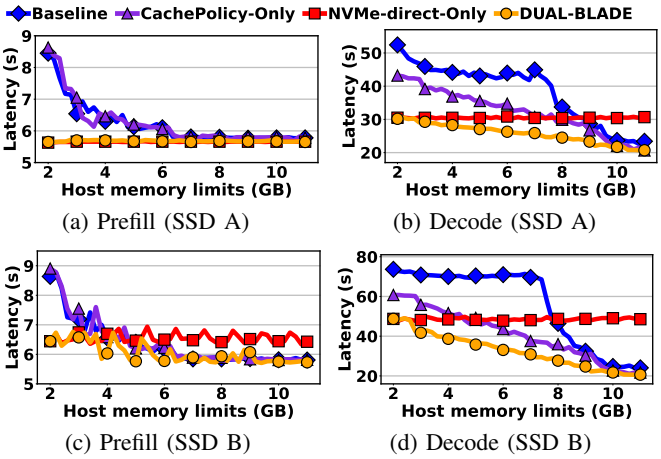
B. LLM Serving Evaluation: Prefill and Decode

To evaluate end-to-end inference performance, we measured prefill and decode latencies across all comparison groups. We executed the workload repeatedly under varying host memory limits (2–11 GB) to capture page-cache dynamics. In these experiments, we employed a multi copy-thread configuration and evaluated two distinct SSDs, SSD A and SSD B, to verify performance consistency, as shown in Figure 10.

We observe consistent trends across the results. First, CachePolicy-Only effectively avoids the page-cache thrashing observed in the Baseline during the decode phase by employ-

TABLE III: Evaluation configurations for KV-cache offloading

Label	Description
Baseline	All KV accesses use the page-cache path.
CachePolicy-Only	$X = B_{pc}$ (upper bound). Page-cache path only. Group 1 remains resident in the page-cache. Group 2 remains on the page-cache path, not the NVMe-direct path, and is proactively evicted via `posix_fadvise(POSIX_FADV_DONTNEED)`.
NVMe-direct-Only	$X = 0$ (lower bound). All KV accesses use the NVMe-direct path via `io_uring_cmd`, the page-cache is unused.
DUAL-BLADE	$X = B_{pc}$ (upper bound). Dual-path design: tensors up to B_{pc} use the page-cache path (Group 1) and the remainder uses the NVMe-direct path (Group 2).



(a) Prefill (SSD A) (b) Decode (SSD A)

(c) Prefill (SSD B) (d) Decode (SSD B)

Fig. 10: End-to-end inference latency of OPT-6.7B on SSD A and SSD B under varying host memory limits (lower is better).

ing the KV residency policy. Consequently, it exhibits a linear reduction in latency proportional to the host memory limit.

Second, NVMe-direct-Only, which bypasses the page-cache, exhibits constant latency regardless of host memory limits, as expected. In decode, while this stability is beneficial under memory pressure, it fails to leverage DRAM as memory limits increase, resulting in the lowest performance.

Finally, DUAL-BLADE consistently demonstrates superior performance across the full range of host-memory limits. The results confirm that it mitigates write stalls during prefill (specifically in the 2–7 GB range) and eliminates page-cache thrashing during decode. This robustness stems from the adaptive dual-path KV residency, which partitions tensors based on the available page-cache budget, routing excess tensors to the NVMe-direct path (Group 2) under tight memory constraints while increasingly leveraging the page-cache path (Group 1) as the host-memory limit increases. We next summarize the quantitative gains for each SSD. Across the 2–11 GB memory range, on SSD A, DUAL-BLADE reduces prefill latency by up to 33.1% and decode latency by 8.2–42.4%. On SSD B, it achieves prefill reductions of up to 25.4% and decode reductions of 11.7–57.8%.

Our results demonstrate that DUAL-BLADE consistently reduces latency across different storage tiers. This confirms that the standard I/O stack creates a bottleneck regardless of the underlying hardware, highlighting the universality of our solution. Next, we provide a detailed evaluation quantifying the effectiveness of each individual design component.

C. Mitigating Page-Cache Thrashing

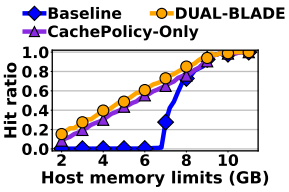


Fig. 11: Page-cache hit ratio.

Dual-path KV residency (§IV-A) allocates KV tensors to Group 1 and Group 2 to reduce page-cache thrashing and improve the hit ratio in the decode phase.

Figure 11 shows page-cache hit ratios across host-memory limits, defined as the fraction of total read bytes served from the page-cache. As noted in §III-A, Baseline exhibits a thrashing zone at 2–7 GB. By contrast, both CachePolicy-Only and DUAL-BLADE show a linear increase in hit ratios, effectively mitigating thrashing. However, DUAL-BLADE achieves superior hit ratios due to complete path separation. In contrast, CachePolicy-Only relies on `posix_fadvise` (to proactively evict data), which inevitably incurs overhead as data must traverse the page-cache.

Furthermore, DUAL-BLADE resolves the page-cache write stall issue during the prefill phase (§III-A). Conversely, CachePolicy-Only remains susceptible to these stalls, as it cannot avoid the initial write to the page-cache.

D. Maximizing NVMe Utilization and Throughput

NVMe-direct (§IV-B) shortens the host-to-device I/O path between pinned memory and the NVMe device, increasing device utilization (NVMe busy). Table IV compares per-tensor latency and NVMe busy (%) for Baseline vs. DUAL-BLADE under a 2 GB host memory limit, excluding page-cache hits for both configurations to isolate storage I/O effects.

Across SSD A/B, DUAL-BLADE lowers mean/99th latency and raises utilization (e.g., 2.2× prefill-write on SSD A), saturating device bandwidth. The efficiency gap is most pronounced for the 256 KB decode-write, where each tensor write maps to a single NVMe command and thus saturates the device (100% NVMe busy) in both configurations. Despite identical device busy, DUAL-BLADE reduces latency by over 98% on both SSDs (e.g., 4.91→0.06 ms on SSD A). This indicates that Baseline's latency is dominated by software overheads in the conventional I/O path rather than device processing.

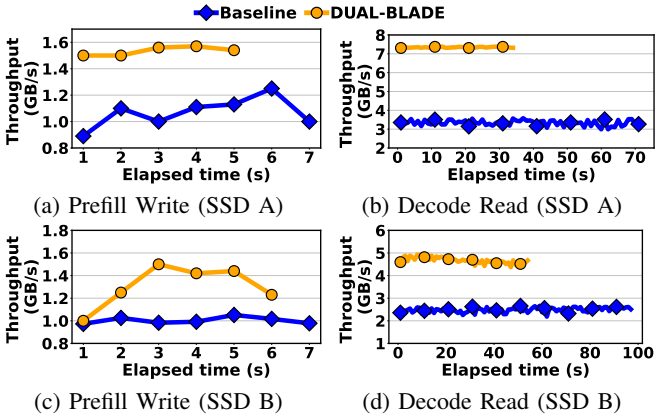
Figure 12 plots disk throughput over time for prefill-write and decode-read on SSD A and SSD B (decode-write omitted for brevity). These results provide direct evidence that the higher NVMe utilization (Table IV) of the NVMe-direct path translates into higher realized disk throughput. Specifically, on SSD A (average throughput), DUAL-BLADE outperforms Baseline by 43% in prefill-write and by 120% (2.20×) in decode-read. On SSD B, it gains 43% in prefill-write and 88% (1.88×) in decode-read over Baseline.

Collectively, these results confirm that NVMe-direct shortens the conventional kernel storage I/O stack and increases

TABLE IV: Per-tensor I/O latency (between pinned memory and NVMe) and device utilization

Config	Per-tensor I/O latency (ms)						NVMe busy (%)		
	Prefill W. avg.	Prefill W. 99th	Decode R. avg.	Decode R. 99th	Decode W. avg.	Decode W. 99th	Prefill W.	Decode R.	Decode W.
SSD A									
Baseline	35.79	109.40	18.80	79.59	4.91	8.26	44.79	54.58	100.00
DUAL-BLADE	**21.18**	**21.26**	**11.17**	**12.67**	**0.06**	**0.06**	**100.00**	**98.45**	100.00
SSD B									
Baseline	41.15	148.95	32.29	85.38	4.47	8.98	89.15	86.99	100.00
DUAL-BLADE	**22.85**	**94.00**	**19.03**	**23.64**	**0.06**	**0.15**	**100.00**	**94.58**	100.00

Note. NVMe busy reports the device-level utilization over the duration of the corresponding tensor I/O (per-tensor, not job-wide average). Per-tensor I/O sizes include a 128 MB (prefill write), 128–135 MB (decode read), and 256 KB (decode write).



(a) Prefill Write (SSD A) (b) Decode Read (SSD A)

(c) Prefill Write (SSD B) (d) Decode Read (SSD B)

Fig. 12: Disk throughput (GB/s) over time on SSD A/B under a 2 GB memory limit (higher is better).

NVMe utilization, which consequently translates into improved overall disk throughput.

E. Sequential LBA Access, Tighter Submit–Complete Latency

The NVMe-direct (§IV-B) further aims to sequentialize tensor access at the LBA level and reduce microsecond-scale latency inside the NVMe controller. To analyze this effect, we instrument NVMe submit/complete with `bpftrace` and group requests into QD bins by the observed queue depth. For a fair comparison, we cap DUAL-BLADE at a maximum QD of 32, and we normalize latency to µs/KB to control for request-size differences. Figure 14 reports per-QD-bin latency for both write and read NVMe commands on the SSD A/B.

On both SSDs, DUAL-BLADE lowers mean μs/KB and tightens the p5–p95 spread across most write/read QD bins. For writes, even though the Baseline shows sequential LBAs, it injects small non-sequential I/Os (e.g., journaling, metadata), which the NVMe-direct path removes. Similarly for reads, this design eliminates the interleaving of blk-mq streams in the Baseline. Thus, NVMe controller processes a clean sequential stream with minimal LBA discontinuities between tensors, leading to reduced internal latency, especially at the tail.

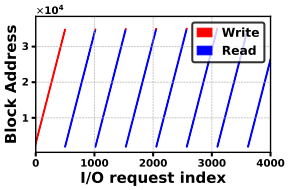
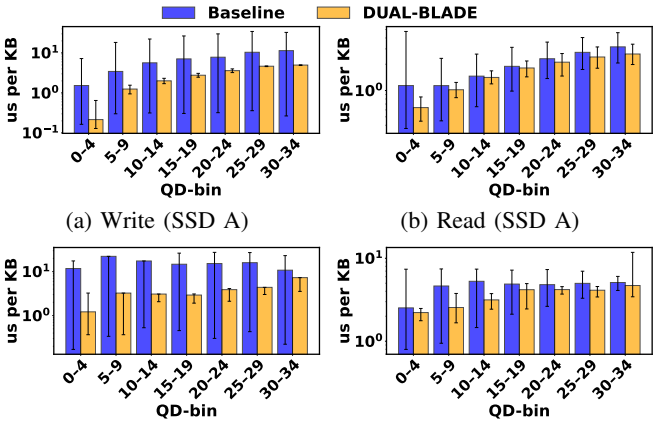


Fig. 13: LBA pattern.

Figure 13 plots SSD A's LBA access under DUAL-BLADE in prefill and decode, ordered by submission time, per request. It closely matches the ideal tensor-access pattern in Figure 6(a) and, unlike the interleaved read stream of Baseline in Figure 6(b), is fully sequential, end to end. Thus the design carries the application's sequential pattern down to device LBAs without distortion.

In a multi copy-thread environment, depending on our pipeline parallelism strategy, the I/O pattern can manifest as a single pure sequential stream or inevitably introduce the interleaving of two pure sequential streams. However, these two pure sequential streams represent the optimal access pattern under concurrency, which modern NVMe controllers can efficiently manage without performance degradation.



(a) Write (SSD A) (b) Read (SSD A)

(c) Write (SSD B) (d) Read (SSD B)

Fig. 14: Per-QD-bin I/O latency (submit→complete), normalized by request size (µs/KB, log scale). Bars show the mean. Black whiskers denote p5–p95.

F. Adaptive Pipeline Parallelism

We evaluate the impact of adaptive pipeline parallelism described in §IV-C for the decode phase. We refer to this optimization as P·P. We define α as the DRAM–SSD tiering ratio (available page-cache capacity relative to total KV size). A higher α indicates that DUAL-BLADE serves a larger fraction of KV tensors via the page-cache path.

Table V compares performance with and without P·P under $\alpha \in \{0.3, 0.5, 0.7\}$ and shows that enabling P·P further consistently reduces decode latency across all configurations. Across all α, SSD A achieves a 7–9% latency reduction, while SSD B shows an additional 3–5% improvement. This is because SSD A's lower latency increases the relative portion of GPU DMA time, making the latency hiding achieved by overlap more effective than on the slower SSD B.

To analyze DUAL-BLADE's runtime strategy selection, we examine the throughput dynamics for SSD A with $\alpha = 0.5$, as shown in Figure 15. This metric captures the per-layer throughput of KV-tensor transfers from their respective residency tiers to the GPU across decode iterations. After a warm-up (Iteration 1), the system profiles *Overlap-Intra* (Iteration 2) and *Overlap-Cross* (Iteration 3). The profiling reveals that *Overlap-Cross* yields higher throughput for both groups (e.g., boosting Group 1 from 11.93 GB/s to 13.13 GB/s), leading DUAL-BLADE to converge on this strategy in Iteration 4.

This outcome indicates that the concurrent access in *Overlap-Intra* saturates both DRAM (Group 1) and SSD (Group 2) bandwidth, resulting in device resource contention. Conversely, *Overlap-Cross* effectively mitigates this by hiding

TABLE V: DUAL-BLADE decode latency with and without pipeline parallelism (P·P) under varying DRAM–SSD tiering ratios α.

Device	Config	Decode latency (s)		
		$\alpha = 0.3$	$\alpha = 0.5$	$\alpha = 0.7$
SSD A	DUAL-BLADE w/o P·P	31.13	28.74	26.59
	DUAL-BLADE w/ P·P	**28.29** (×0.91)	**26.80** (×0.93)	**24.69** (×0.93)
SSD B	DUAL-BLADE w/o P·P	41.52	36.39	30.27
	DUAL-BLADE w/ P·P	**40.09** (×0.97)	**34.91** (×0.96)	**28.82** (×0.95)

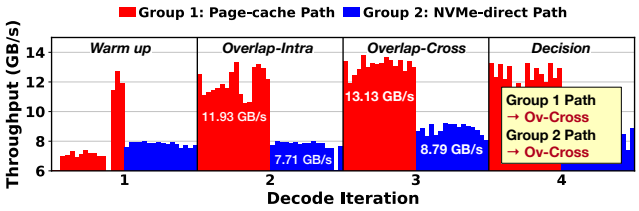


Fig. 15: Throughput dynamics during the adaptive pipeline strategy selection on SSD A ($\alpha = 0.5$).

the secondary thread's I/O latency behind the primary thread's GPU DMA, maximizing end-to-end throughput.

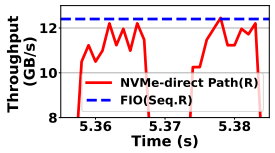


Fig. 16: Millisecond-level throughput analysis on SSD A. The dashed line marks the sequential read limit (FIO).

To investigate the root cause of this contention, we analyzed the instantaneous throughput of a *single* copy-thread (Figure 16). Unlike the per-second average in Figure 12, this fine-grained analysis at millisecond resolution captures the instantaneous throughput spikes, revealing that a single copy-thread alone is sufficient to saturate the SSD's peak sequential read bandwidth. Thus, the concurrent reads from multi copy-threads in *Overlap-Intra* coincided within these short, saturated intervals, resulting in device resource contention.

G. Evaluation on Practical Edge Scenarios

Beyond interactive chat applications, Edge AI systems utilize LLMs for *back-of-house* data wrangling to continuously clean and structure raw operational data [4], [12]. Specifically, they execute Entity Matching (EM), Data Imputation (DI), and Error Detection (ED) at the source.

As shown in Table VI, we evaluate four representative data wrangling tasks [39] with OPT-6.7B. These tasks are characterized by long input contexts (200–744 tokens) and short outputs (3–10 tokens). We conducted the experiments with a batch size of 32 under a strict 4 GB host memory limit. This leaves much less than 4 GB for the KV-cache.

DUAL-BLADE outperforms the Baseline in most cases (e.g., reducing latency to 0.85× in DI:Buy). An exception is ED:Hospital, where performance remains comparable (1.00×) because the relatively small KV-cache (1.58 GB) fits entirely within the available page-cache.

VI. Related Work

SSD-Enabled LLM Inference Techniques. Studies have explored offloading techniques for LLM inference to NVMe SSDs to overcome memory limits. LLM-in-a-Flash [21] achieves weight-centric offloading by storing model parameters on flash for on-demand loading via flash-friendly transfers. KVSwap [20] uses a low-rank predictor to approximate attention scores, selectively fetching critical KV groups from flash storage. AttentionStore [40] enables multi-turn reuse by persisting KV states across a DRAM–SSD hierarchy for efficient reinstatement. A quantitative study [41] identifies NVMe-level I/O bottlenecks when offloading model weights or KV tensors across inference frameworks [13], [14]. InstInfer [42] and INF^2 [43] target Computational Storage Devices (CSDs) over commodity SSDs to offload attention/KV handling in-storage.

TABLE VI: Inference latency (s) of data wrangling tasks [39].

Dataset	Queries	KV-cache (GB)	*SSD A* Base	*SSD A* DUAL-BLADE	*SSD B* Base	*SSD B* DUAL-BLADE
EM:Fodors-Zagats	189	5.84	75.28	**67.17** (×0.89)	86.40	**79.06** (×0.92)
EM:Walmart-Amazon	200	5.87	80.92	**72.18** (×0.89)	93.17	**85.41** (×0.92)
DI:Buy	65	3.89	30.55	**27.90** (×0.91)	34.05	**29.27** (×0.86)
ED:Hospital	200	1.58	19.63	**19.60** (×1.00)	19.62	**19.66** (×1.00)

At the system layer, LLM serving frameworks like FlexLLMGen [14], llama.cpp [17], and vLLM [44] (with LMCache [18]) leverage heterogeneous memory tiers spanning GPU memory, host DRAM, and SSDs to manage model weights and KV tensors. DUAL-BLADE is orthogonal to FlexLLMGen and can work in conjunction with it to offload the KV-cache to SSD more efficiently. Its design principles are framework-agnostic and can be applied to KV offloading backends (e.g., LMCache) to accelerate storage I/O across different serving stacks.

Hardware-Oriented Data-Movement Mechanisms. Beyond LLM-specific offloading policies, several hardware-oriented primitives define the data path among GPU memory, host DRAM, and storage. UVM [5] provides a unified managed address space and on-demand page migration across CPU and GPU, enabling oversubscription beyond physical VRAM by leveraging host DRAM, which is useful on low-memory GPUs. GDS [45] enables direct storage-to-GPU DMA by bypassing the CPU staging path (host bounce buffers), supporting GPU-targeted I/O directly into device memory. BaM [46] enables GPU-initiated direct storage-to-GPU access via GPU-resident submission/completion queues (doorbells), supporting fine-grained transfers with minimal CPU involvement.

Although GDS and BaM can accelerate storage offloading via GPU–SSD P2P DMA, these capabilities typically assume datacenter-class GPUs (e.g., A100/H100) and supported server platforms with compatible PCIe topologies, making them best suited for cloud-scale inference deployments. In contrast, DUAL-BLADE targets edge AI platforms (e.g., on-premise and embedded deployments) where GPU–SSD P2P is unavailable. Furthermore, DUAL-BLADE's designs can be integrated on top of GDS/BaM-style GPU–SSD P2P primitives when available, further improving offloading performance.

VII. Conclusion

This paper presents DUAL-BLADE, a KV-cache offloading architecture that accelerates LLM inference in memory-constrained edge AI systems by integrating dual-path KV residency, NVMe-direct I/O, and adaptive pipeline parallelism. Evaluations show DUAL-BLADE reduces prefill and decode latency by up to 33.1% and 42.4%, boosting SSD utilization by up to 2.2× under diverse memory budgets.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (RS-2024-00453929) (RS-2024-00416666).

REFERENCES

[1] Y. Zheng, Y. Chen, B. Qian, X. Shi, Y. Shu, and J. Chen, "A review on edge large language models: Design, execution, and applications," *ACM Computing Surveys*, vol. 57, no. 8, pp. 1–35, 2025.

[2] G. Pan, V. Chodnekar, A. Roy, and H. Wang, "A cost-benefit analysis of on-premise large language model deployment: Breaking even with commercial llm services," *arXiv preprint arXiv:2509.18101*, 2025.

[3] Z. Liu, C. Zhao, F. Iandola, C. Lai, Y. Tian, I. Fedorov, Y. Xiong, E. Chang, Y. Shi, R. Krishnamoorthi, *et al.*, "Mobilellm: Optimizing sub-billion parameter language models for on-device use cases," in *Forty-first International Conference on Machine Learning*, 2024.

[4] X. Wang, Z. Xu, and X. Sui, "Intelligent data analysis in edge computing with large language models: applications, challenges, and future directions," *Frontiers in Computer Science*, vol. 7, p. 1538277, 2025.

[5] NVIDIA, "CUDA Programming Guide: Unified and system memory." https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/understanding-memory.html, Dec. 2025. Accessed: 2026-01-21.

[6] G. Team, T. Mesnard, C. Hardin, R. Dadashi, S. Bhupatiraju, S. Pathak, L. Sifre, M. Rivière, M. S. Kale, J. Love, *et al.*, "Gemma: Open models based on gemini research and technology," *arXiv preprint arXiv:2403.08295*, 2024.

[7] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, "Mistral 7b," *arXiv preprint arXiv:2310.06825*, vol. 3, 2023.

[8] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan, *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[9] J. Zhang, Z. Hou, X. Lv, S. Cao, Z. Hou, Y. Niu, L. Hou, Y. Dong, L. Feng, and J. Li, "Longreward: Improving long-context large language models with ai feedback," in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3718–3739, 2025.

[10] Y. Bai, X. Lv, J. Zhang, H. Lyu, J. Tang, Z. Huang, Z. Du, X. Liu, A. Zeng, L. Hou, *et al.*, "Longbench: A bilingual, multitask benchmark for long context understanding," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3119–3137, 2024.

[11] H. Liu, C. Li, Q. Wu, and Y. J. Lee, "Visual instruction tuning," *Advances in neural information processing systems*, vol. 36, pp. 34892–34916, 2023.

[12] A. Zhong, D. Mo, G. Liu, J. Liu, Q. Lu, Q. Zhou, J. Wu, Q. Li, and Q. Wen, "Logparser-llm: Advancing efficient log parsing with large language models," in *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 4559–4570, 2024.

[13] R. Y. Aminabadi, S. Rajbhandari, A. A. Awan, C. Li, D. Li, E. Zheng, O. Ruwase, S. Smith, M. Zhang, J. Rasley, *et al.*, "Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, IEEE, 2022.

[14] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica, and C. Zhang, "Flexgen: High-throughput generative inference of large language models with a single gpu," in *International Conference on Machine Learning (ICML '23)*, pp. 31094–31116, PMLR, 2023.

[15] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[16] W. Lee, J. Lee, J. Seo, and J. Sim, "Infinigen: Efficient generative inference of large language models with dynamic kv cache management," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 155–172, 2024.

[17] G. Gerganov and contributors, "llama.cpp: Llm inference in c/c++." https://github.com/ggml-org/llama.cpp, 2023. Accessed: 2026-01-09.

[18] Y. Liu, Y. Cheng, J. Yao, Y. An, X. Chen, S. Feng, Y. Huang, S. Shen, R. Zhang, K. Du, *et al.*, "Lmcache: An efficient kv cache layer for enterprise-scale llm inference," *arXiv preprint arXiv:2510.09665*, 2025.

[19] Y. Song, Z. Mi, H. Xie, and H. Chen, "Powerinfer: Fast large language model serving with a consumer-grade gpu," in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pp. 590–606, 2024.

[20] H. Zhang, C. Xia, and Z. Wang, "KVSwap: Disk-aware KV Cache Offloading for Long-Context On-device Inference," *arXiv preprint arXiv:2511.11907*, 2025.

[21] K. Alizadeh, S. I. Mirzadeh, D. Belenko, S. Khatamifard, M. Cho, C. C. Del Mundo, M. Rastegari, and M. Farajtabar, "Llm in a flash: Efficient large language model inference with limited memory," in *62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12562–12584, 2024.

[22] S. Kannan, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, Y. Wang, J. Xu, and G. Palani, "Designing a true direct-access file system with devfs," in *16th USENIX Conference on File and Storage Technologies (FAST '18)*, pp. 241–256, 2018.

[23] Intel, "Storage performance development kit (spdk)." https://spdk.io, 2025. Accessed: 2026-01-21.

[24] J. Zhang, M. Kwon, D. Gouk, S. Koh, C. Lee, M. Alian, M. Chun, M. T. Kandemir, N. S. Kim, J. Kim, *et al.*, "Flashshare: Punching through server storage stack from kernel to firmware for ultra-low latency ssds," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 477–492, 2018.

[25] K. Joshi, A. Gupta, J. González, A. Kumar, K. K. Reddy, A. George, S. Lund, and J. Axboe, "I/o passthru: Upstreaming a flexible and efficient i/o path in linux," in *22nd USENIX Conference on File and Storage Technologies (FAST '24)*, pp. 107–121, 2024.

[26] Facebook, "Opt-6.7b." https://huggingface.co/facebook/opt-6.7b, 2022. Accessed: 2026-01-21.

[27] T. Johnson and D. Shasha, "X3: A low overhead high performance buffer management replacement algorithm," in *20th VLDB Conference*, pp. 439–450, 1994.

[28] N. Megiddo and D. S. Modha, "Arc: A self-tuning, low overhead replacement cache," in *2nd USENIX Conference on File and Storage Technologies (FAST '03)*, 2003.

[29] Z. Li and G. Zhang, "Streamcache: Revisiting page cache for file scanning on fast storage devices," in *2024 USENIX Annual Technical Conference (ATC '24)*, pp. 1119–1134, 2024.

[30] B. Gregg, *BPF performance tools*. Addison-Wesley Professional, 2019.

[31] G. Lee, S. Shin, W. Song, T. J. Ham, J. W. Lee, and J. Jeong, "Asynchronous i/o stack: A low-latency kernel i/o stack for ultra-low latency ssds," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 603–616, 2019.

[32] J. Woo, M. Ahn, G. Lee, and J. Jeong, "D2FQ:Device-Direct Fair Queueing for NVMe SSDs," in *19th USENIX Conference on File and Storage Technologies (FAST '21)*, pp. 403–415, 2021.

[33] D. Park and D. Shin, "iJournaling:Fine-Grained journaling for improving the latency of fsync system call," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pp. 787–798, 2017.

[34] S. Bhattacharya, S. Pratt, B. Pulavarty, and J. Morgan, "Asynchronous i/o support in linux 2.5," in *Linux Symposium*, pp. 371–386, Citeseer, 2003.

[35] C. Whitaker, S. Sundar, B. Harris, and N. Altiparmak, "Do we still need io schedulers for low-latency disks?," in *15th ACM Workshop on Hot Topics in Storage and File Systems*, pp. 44–50, 2023.

[36] Z. Ren, K. Doekemeijer, N. Tehrany, and A. Trivedi, "Bfq, multiqueue-deadline, or kyber? performance characterization of linux storage schedulers in the nvme era," in *15th ACM/SPEC International Conference on Performance Engineering*, pp. 154–165, 2024.

[37] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, *et al.*, "Opt: Open pre-trained transformer language models," *arXiv preprint arXiv:2205.01068*, 2022.

[38] NVIDIA, "Cuda c++ programming guide." https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2024. Accessed: 2026-01-21.

[39] A. Narayan, I. Chami, L. Orr, S. Arora, and C. Ré, "Can foundation models wrangle your data?," *arXiv preprint arXiv:2205.09911*, 2022.

[40] B. Gao, Z. He, P. Sharma, Q. Kang, D. Jevdjic, J. Deng, X. Yang, Z. Yu, and P. Zuo, "Cost-efficient large language model serving for multi-turn conversations with cachedattention," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pp. 111–126, 2024.

[41] Z. Ren, K. Doekemeijer, T. De Matteis, C. Pinto, R. Stoica, and A. Trivedi, "An i/o characterizing study of offloading llm models and kv caches to nvme ssd," in *5th Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*, pp. 23–33, 2025.

[42] X. Pan, E. Li, Q. Li, S. Liang, Y. Shan, K. Zhou, Y. Luo, X. Wang, and J. Zhang, "Instinfer: In-storage attention offloading for cost-effective long-context llm inference," *arXiv preprint arXiv:2409.04992*, 2024.

[43] H. Jang, S. Noh, C. Shin, J. Jung, J. Song, and J. Lee, "INF2: High-throughput generative inference of large language models using near-storage processing," 2025.

[44] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Ma, J. Huang, Y. Ko, A. Anandkumar, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *29th ACM Symposium on Operating Systems Principles (SOSP '23)*, pp. 811–828, Association for Computing Machinery, 2023.

[45] NVIDIA, "GPUDirect Storage: A Direct Path Between Storage and GPU Memory." https://developer.nvidia.com/gpudirect-storage, 2025. Accessed: 2026-01-21.

[46] Z. Qureshi, V. S. Mailthody, I. Gelado, S. Min, A. Masood, J. Park, J. Xiong, C. J. Newburn, D. Vainbrand, I.-H. Chung, *et al.*, "Gpu-initiated on-demand high-throughput storage access in the bam system architecture," in *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 325–339, 2023.