VLM in a flash: I/O-Efficient Sparsification of Vision-Language Model via NEURON CHUNKING

Kichang Yang[*]
Seoul National University
kichang96@snu.ac.kr

Seonjun Kim[*]
Seoul National University
cyanide17@snu.ac.kr

Minjae Kim
Seoul National University
aingo03304@snu.ac.kr

Nairan Zhang
Meta
nairanzhang@meta.com

Chi Zhang
Amazon
zhanbchi@amazon.com

Youngki Lee[†]
Seoul National University
youngkilee@snu.ac.kr

Abstract

Edge deployment of large Vision-Language Models (VLMs) increasingly relies on flash-based weight offloading, where activation sparsification is used to reduce I/O overhead. However, conventional sparsification remains model-centric, selecting neurons solely by activation magnitude and neglecting how access patterns influence flash performance. We present NEURON CHUNKING, an I/O-efficient sparsification strategy that operates on *chunks*—groups of contiguous neurons in memory—and couples neuron importance with storage access cost. The method models I/O latency through a lightweight abstraction of access contiguity and selects chunks with high utility, defined as neuron importance normalized by estimated latency. By aligning sparsification decisions with the underlying storage behavior, Neuron Chunking improves I/O efficiency by up to 4.65× and 5.76× on Jetson Orin Nano and Jetson AGX Orin, respectively.

1 Introduction

Recent vision–language models (VLMs) demonstrate strong multimodal reasoning and real-time language interaction with visual scenes. Deploying these models on edge devices is becoming essential for applications such as augmented reality (AR) and autonomous robotics that require on-device inference for robustness to limited connectivity and privacy [5, 53]. These systems must process video frames continuously without frame drops while maintaining interactive latency.

The scalability of on-device inference is fundamentally constrained by memory capacity. Edge platforms provide far less memory than what modern VLMs require. Jetson Orin Nano, for example, offers only 8 GB of memory, while LLaVA-OneVision-7B [18] requires 16 GB (fp16) for weights alone. Recent systems [2, 3, 10, 36, 49] and inference engines [1, 9] address this mismatch through weight offloading, which stores model parameters in external flash memory and loads them on demand during inference. This method allows large models to execute on small devices but introduces substantial I/O latency, which often dominates total inference time.

Activation sparsification has been widely explored to mitigate this latency [2, 44, 49]. The approach loads only the weights corresponding to neurons with high importance (e.g., magnitude of activation value), reducing total data transfer and improving input adaptivity. Despite its effectiveness, existing

[*]Equal contribution
[†]Corresponding author

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

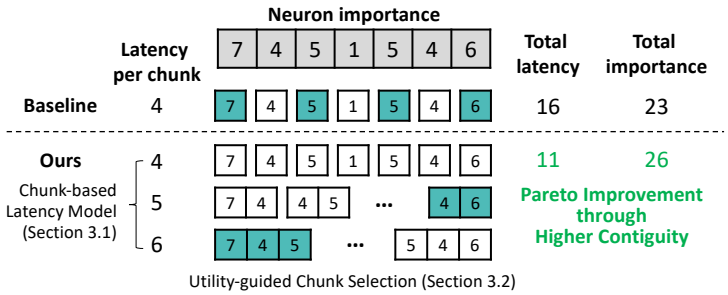


Figure 1: **Illustration of conventional sparsification *vs.* our approach**. Existing methods select neurons solely based on activation importance, which often leads to scattered, irregular access patterns with poor I/O efficiency. In contrast, our method explicitly accounts for actual I/O latency, favoring contiguous chunks that achieve better importance–latency trade-offs.

sparsification remains model-centric. It selects channels solely based on activation importance while assuming that I/O latency scales linearly with data size. Flash storage does not follow this assumption; its latency depends strongly on access contiguity, and scattered reads severely degrade throughput. Earlier methods were less affected because they targeted highly sparse LLMs or GPU memory transfers, where locality plays a smaller role. Modern VLMs exhibit smoother activation distributions and lower sparsity (Figure 2), leading to fragmented reads and high I/O overhead (Figure 4b).

We propose NEURON CHUNKING, a sparsification framework that improves the I/O efficiency of flash-offloaded inference by coupling activation sparsity with storage access behavior. The key idea is to jointly optimize neuron importance and flash I/O latency by selecting contiguous channel groups that provide a better trade-off between accuracy and latency. Contiguous reads provide higher flash throughput, allowing moderately important neighboring channels to be loaded more efficiently than distant but highly important ones (Figure 1). This design forms compact chunks that enhance access locality, leading to significant performance gains during VLM inference.

Efficient realization of this strategy requires capturing hardware I/O behavior in a form that the runtime can readily exploit. NEURON CHUNKING achieves this by reducing complex access patterns into a compact structural representation, termed the *contiguity distribution*. It summarizes how memory accesses cluster into contiguous groups (i.e., chunks) while omitting their exact spatial layout. This abstraction underlies two key components of our system. A *chunk-based latency model* profiles load latency for each chunk size and efficiently estimates the I/O latency of arbitrary access patterns from their contiguity distributions. A *utility-guided chunk selection algorithm* formulates neuron selection as a constrained optimization problem and iteratively selects chunks that maximize the importance–latency utility. Evaluation on Jetson Orin AGX and Nano using open-source VLMs and standard benchmarks shows consistent improvement in the accuracy–latency trade-off. At comparable accuracy, NEURON CHUNKING reduces I/O latency by an average of 2.19× on Nano and 2.89× on AGX, with maximum gains of 4.65× and 5.76×, respectively.

Our contributions are summarized as follows.

- We identify and characterize the hardware inefficiencies that arise when conventional activation sparsification techniques are applied to flash-offloaded VLM inference, revealing their mismatch with underlying storage access patterns.
- We propose NEURON CHUNKING, a sparsification framework that enhances flash I/O efficiency by coupling activation sparsity with storage access patterns. It jointly optimizes neuron importance and flash latency by selecting contiguous channel groups that balance accuracy and latency.
- We introduce the contiguity distribution as a compact representation of flash access behavior. Building on this abstraction, we develop a *chunk-based latency model* and a *utility-guided chunk selection algorithm* that together enable latency-aware sparsification by balancing model quality and I/O efficiency.
- We evaluate NEURON CHUNKING on Jetson Orin AGX and Nano with open-source VLMs and standard benchmarks. Results show consistent improvement in the accuracy–latency trade-off, reducing I/O latency by up to 5.76× on AGX and 4.65× on Nano while maintaining comparable accuracy.

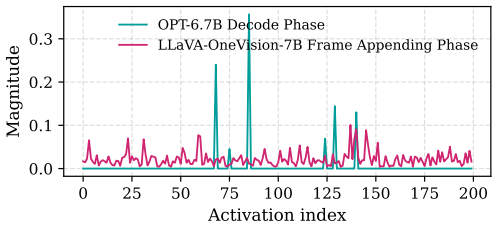


Figure 2: Activation-magnitude plot for two workloads: (teal) a ReLU-based LLM in the decode phase and (magenta) a gated-activation-based VLM in the frame appending phase. VLM exhibits a smoother distribution, with much less variation between high and low activation values.

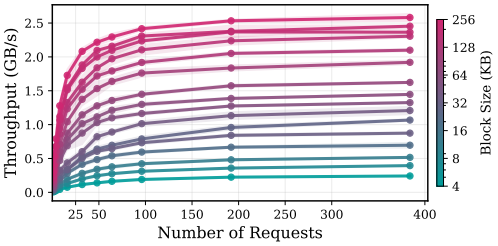


Figure 3: Read throughput as a function of block size and number of requests, profiled on Jetson AGX Orin with a Samsung 990 Pro SSD. Throughput quickly saturates and remains stable once the request count exceeds minimal thresholds.

2 Background and Motivation

2.1 Overview of LLM/VLM Inference Pipeline

An LLM inference pipeline consists of two stages: (i) the prefill stage, where the input prompt, consisting of multiple tokens, is processed to generate key-value (KV) cache, and (ii) the decoding stage, where the model generates tokens one at a time autoregressively using KV cache.

When processing an online video stream with VLMs, an additional *frame-appending* stage is introduced between the prefill and decoding stages. In this stage, incoming video frames are processed sequentially as they arrive. VLMs integrate a vision encoder alongside the backbone LLM. Each frame is divided into patches and passed through the vision encoder, which converts them into a sequence of visual tokens. These tokens are then fed into the language model, augmenting the existing KV cache generated from the language prompt (see Appendix B.1 for details).

2.2 Model-side Observation: Smooth Activation Profiles in VLMs

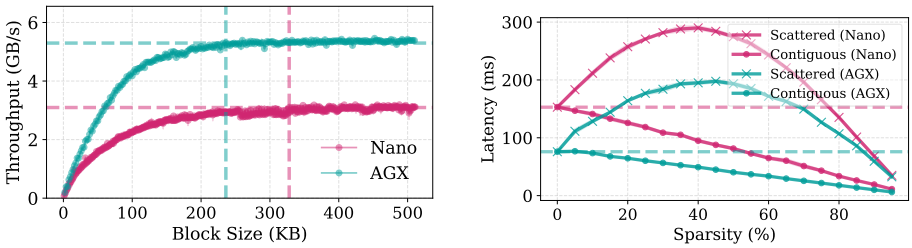
Modern VLMs exhibit smooth activation distributions as shown in Figure 2. This smoothness is a general property of the architecture rather than a model-specific artifact. It arises from two key factors: (i) gated activation functions such as SwiGLU [35] and GeLU [12] produce continuous rather than sparse activation values, and (ii) averaging these values over multiple visual tokens, as in LLaVA-OneVision with 14×14 tokens per frame, further reduces the variation in importance scores. As shown in Appendix C, this phenomenon consistently appears across diverse models.

This observation suggests that when activation values are less distinct, selection policies can balance model quality with system performance rather than focusing solely on the largest activations. Furthermore, the lack of sharply separated activations makes it difficult to depend on only a few dominant neurons, making moderate sparsity a more favorable operating point.

2.3 System-side Observation: Flash I/O Sensitivity to Access Contiguity

Flash read performance depends on memory access patterns, and access contiguity is the dominant factor. As shown in Figure 4a, larger contiguous reads improve throughput until reaching the bandwidth limit, marking a shift from overhead-bound to bandwidth-bound operation. This behavior reveals a counterintuitive aspect of sparsification: while higher sparsity reduces data transfer, it fragments memory accesses and can increase latency (Figure 4b).

Notably, throughput stabilizes once the request count exceeds a minimal threshold (Figure 3). This stability enables latency estimation from access contiguity using a one-time throughput profile across block sizes, making latency-aware sparsification practical during inference.



(a) Block Size vs. Flash Read Throughput

(b) Sparsity vs. Flash Read Latency

Figure 4: Flash read performance under varying access patterns. Left: Throughput vs. block size when reading 128 MB (MLP weight sizes in Qwen2-7B [50]). Right: Latency vs. sparsity across two access modes—*scattered* (random) and *contiguous* (sufficiently block-aligned to saturate throughput: 328 KB on AGX, 236 KB on Nano). Error bars show ± 1 std; dashed lines indicate saturate throughput and full-load latency. Experiments use Linux direct I/O [23] with 6-thread thread-pool in C++.

3 NEURON CHUNKING

Motivated by the aforementioned considerations, we introduce NEURON CHUNKING, an I/O efficiency-aware activation sparsification method. Unlike traditional sparsification methods, NEURON CHUNKING jointly considers neuron importance and the memory access costs associated with retrieving selected neurons from flash memory.

To enable such a joint optimization, two key requirements must be satisfied: (i) latency estimation for a given memory access pattern, and (ii) neuron selection that balances costs and benefits. Both tasks must be executed frequently at runtime, once per weight matrix (e.g., approximately 200 times per frame for LLaVA-OneVision-Qwen2-7B [18]). Each must therefore complete within a very short time frame (i.e., about a few milliseconds). Previous approaches avoided this complexity by assuming that latency scales monotonically with the number of selected neurons—an assumption that does not hold in our target workloads.

Our key idea is to abstract a memory access pattern into a concise representation called the *contiguity distribution*, defined as the frequency distribution of the contiguity of the selected neurons. Concretely, we group consecutively selected neurons into *chunks*, each representing a maximal contiguous range of neuron indices within the selected set. For example, selecting neurons with indices $\{1, 2, 4, 6, 7\}$ yields three chunks: $\{1, 2\}$, $\{4\}$, and $\{6, 7\}$. We then model the total read latency as a function of this contiguity distribution—e.g., in the above case, one chunk of size 1 and two of size 2—while discarding global structural cues such as the exact spatial arrangement of chunks. This abstraction both simplifies hardware modeling and substantially reduces the search space of the neuron selection.

- In Section 4.1, we present a *chunk-based latency model*. It builds a lookup table of per-chunk-size latencies via offline profiling and estimates total latency directly from the contiguity distribution.
- In Section 4.2, we introduce a *utility-guided chunk selection* algorithm. Given a list of activation importance, this algorithm generates candidate neuron chunks of varying sizes and greedily selects the chunks with high utility (i.e., importance-per-latency ratio).
- In Section 4.3, we apply a lightweight offline reordering step that groups neurons based on activation statistics to improve I/O contiguity, and find that this simple approach suffices without relying on co-activation information.

3.1 Chunk-based Latency Model

We first build a lightweight latency estimation model that predicts the I/O latency of arbitrary neuron-access patterns on flash storage. Even after abstracting an access pattern into a contiguity distribution, the space of possible distributions remains combinatorially large (the number of combinations grows exponentially with the number of channels), making exhaustive profiling infeasible. Meanwhile, prior SSD modeling frameworks [13, 43] provide device-level fidelity but require low-level hardware configurations or full-system simulations, which are impractical during inference.

To address this, we propose a simple and scalable latency model tailored for inference-time sparsification. We approximate the total read latency of the access pattern as the sum of the latencies of its constituent chunks. Let the selected chunks be C_i $(i = 1, \ldots, n)$, each of size s_i. The total latency is estimated as $L_{\text{total}} = \sum_{i=1}^{n} T[s_i]$ where $T[s]$ denotes the profiled read latency for a chunk of size s.

Profiling $T[s]$. We build a lookup table for $T[s]$ via offline microbenchmarks: for each chunk size s, we place a throughput-saturating number of chunks of size s at fixed strides and measure steady-state read latency (See Appendix D for details). Fixed overheads such as command setup or metadata access during flash read initiation are amortized and become negligible in $T[s]$. This procedure yields stable per-size latencies with low measurement variance.

Empirical Validation. We evaluate the effectiveness of our latency model across different devices and models, as shown in Figure 5. Each plot shows actual and estimated latency values of loading selected chunks from our selection algorithm (Section 3.2).

We observe a near-linear relation between estimated and measured latency, indicating a consistent proportional bias.[3] This bias arises from our contiguity-distribution abstraction and profile setup. The lookup table is built under idealized conditions, where each chunk size is measured in isolation with uniform strides. In contrast, real access patterns interleave diverse chunk sizes and strides, invoking pattern-dependent controller and queue behaviors. These effects accumulate and average out to a proportional lift in actual latency. The near-linear correlation weakens for smaller models or lower-end devices, where I/O concurrency is lower and controller dynamics amplify tail latency, reducing the averaging effect. Importantly, the error remains near-linear, leaving the greedy chunk selection algorithm unaffected (see Section 3.2).

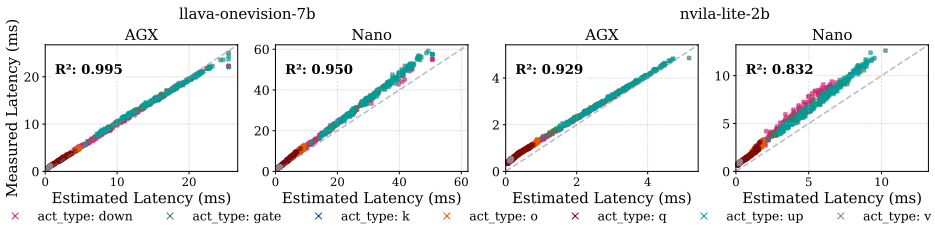


Figure 5: Comparison between real and estimated flash access latency across models and devices.

3.2 Utility-Guided Chunk Selection

Given the latency model, our goal is to jointly consider latency and neuron importance when selecting neurons. The selection problem is intractable, as the number of possible neuron combinations grows exponentially with the model size. To make the problem tractable, we leverage the contiguity-distribution abstraction together with the additive latency assumption, where total latency is approximated as the sum of per-chunk latencies. Under this scheme, latency becomes decomposable across chunks, which naturally motivates a greedy, chunk-level selection strategy.

3.2.1 Problem Formulation

Given activation magnitudes $V \in \mathbb{R}^N$ (where N is the number of neurons) and a selection budget R (the number of channels to load), we aim to output a binary selection mask $M \in \{0,1\}^N$ that maximizes importance per latency:

$$\max_{M \in \{0,1\}^N} \frac{\sum_{i=0}^{N-1} V_i \cdot M_i}{\text{Latency}(M)} \quad \text{s.t.} \quad \sum_{i=0}^{N-1} M_i \leq R,$$

where $\text{Latency}(M)$ is the estimated cost of loading the selected channels, modeled via the contiguity distribution of M.

[3]The latency gap between two points that differ by one additional chunk remains roughly proportional across the entire plot.

3.2.2 Algorithmic Procedure

Our algorithm consists of three main stages:

1. Candidate chunk generation. We construct candidate chunks by sliding windows of multiple sizes over the linear space of neuron indices, and each window position yields one candidate chunk. The maximum chunk size is set to the hardware-specific point where throughput saturates (Section 2.3). The minimum chunk size and the stride between windows are tunable hyperparameters that control the trade-off between search granularity and computational overhead of the algorithm.

2. Chunk evaluation. Each chunk is assigned a utility score, defined as the sum of its neuron importance values divided by its estimated latency. Latency is obtained efficiently from a pre-profiled lookup table based on our latency model.

3. Greedy selection. Chunks are sorted by utility, and the algorithm iteratively selects the highest-ranked ones while excluding those that overlap with previously selected chunks, continuing until the selection budget is met.

Because the latency model error is approximately linear, it merely scales all utility scores by a constant factor without changing their relative order. Therefore, the output of the greedy selection remains unaffected.

One limitation of this approach is that it does not account for potential synergies among adjacent low-scoring chunks, which could collectively form a more latency-efficient region. Even so, we observe that the algorithm performs robustly in practice and effectively identifies high-importance subsets within its runtime constraints (see Section 4.2).

A complete implementation, including mask updates, stride scheduling, and latency lookup logic, is provided in Appendix E.

3.3 Additional Optimization: Hot-cold Reordering

Previous works [2, 44] have observed neuron co-activation patterns and improved I/O efficiency in ReLU-based LLMs through offline reordering based on these statistics. Motivated by these findings, we explore a simpler reordering scheme that leverages hot–cold activation patterns observed in prior studies [38, 49]. Specifically, we reorder neurons according to their activation frequency, which yields comparable I/O efficiency improvements to co-activation-based methods without the need for complex optimization (See Appendix F, G for details). Thus, we adopt this hot–cold reordering as a preprocessing step during the offline profiling stage.

The procedure is as follows. We first count how frequently each neuron is activated (designating the top 50% by importance as active) using a calibration dataset. Then we sort the neurons in decreasing order of activation frequency. Based on this ordering, we permute the corresponding rows of the weight matrix so that frequently activated neurons are placed together. At runtime, the same permutation is applied to the activation vector, aligning it with the reordered weights. The runtime permutation operation incurs negligible overhead. For example, profiling the down-projection layer of LLaVA-OneVision-7B on Jetson Orin Nano over 100 trials showed a mean overhead of 1.5 ms with a 95% confidence bound of 1.8 ms, representing less than 0.02% of total inference latency.

4 Evaluation

4.1 Experimental Setup

Hardware. All experiments were performed on two different embedded device setups, representing low-end and high-end hardware environments:

- Jetson Orin Nano (8 GB memory) with SK Hynix Gold P31 SSD (peak sequential read: 3500 MB/s)
- Jetson Orin AGX (32 GB memory) with Samsung 990 Pro SSD (peak equential read: 7450 MB/s)

We cache the vision encoder and KV cache in memory. All weights of the backbone LLM are loaded from flash on demand. Unless specified, we use Jetson Orin Nano as our default device, reflecting the memory-constrained setting.

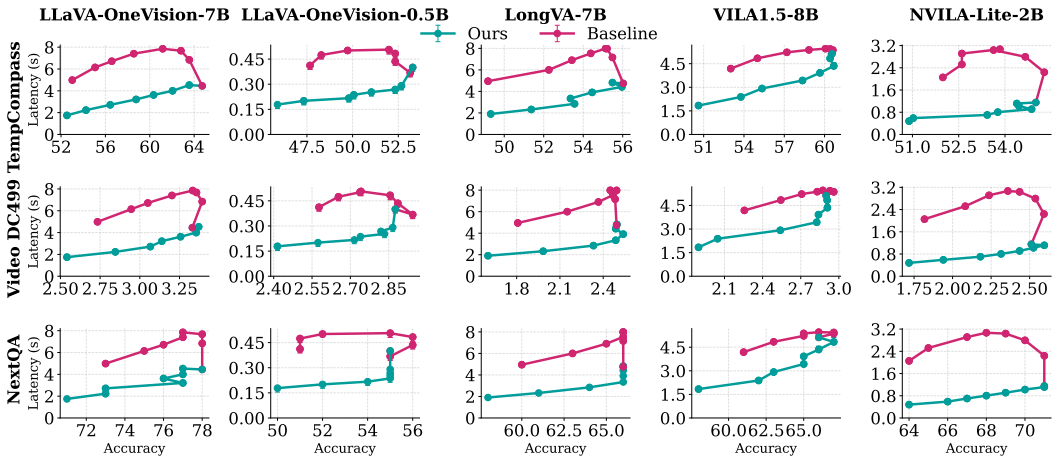


Figure 6: End-to-end performance on Jetson Orin Nano. Latency reported with error bar ± 1 std.

Models and Datasets. We use VLM models that operate frame-by-frame, as models that process entire videos at once do not fit our real-time streaming scenario. We evaluate five models with various sizes: LLaVA-OneVision-Qwen2-7B, LLaVA-OneVision-Qwen2-0.5B [18], Llama-3-VILA1.5-8B [22], NVILA-Lite-2B [27], and LongVA-7B [54]. For datasets, we use two multiple-choice video QA benchmarks—TempCompass [25] and NExT-QA [47] (randomly sampled 3000 examples)—and a video description dataset, VideoDetailCaption [29]. Unless otherwise specified, we use LLaVA-OneVision-Qwen2-7B and the TempCompass dataset by default.

Comparison Setup. As a baseline, we implement top-k activation sparsification that selects the most important channels based on activation magnitude, following prior works [2, 16, 24]. We apply TEAL's [24] profiling-based method to determine layer-wise sparsity levels for both the baseline and our method, using 25 out of 410 videos from the TempCompass dataset (excluded from the main evaluation). See Appendix H for hyperparameter details.

Metric. We report accuracy and I/O latency. Accuracy is defined as the ratio of correct answers on multiple-choice QA datasets, and as a 0–5 score from ChatGPT-based evaluation on video description dataset, following prior works.[4] Due to the large dataset size, accuracy is measured using a Supermicro A+ Server 4124GS-TNR with 8 RTX A6000 GPUs. We measure accuracy under a sparsity level from 0% to 70% in 10% increments. Each latency experiment was repeated 30 times under identical conditions. We report the median latency together with 95% confidence intervals computed by a 10000-sample non-parametric bootstrap (bias-corrected and accelerated, BCa). We enable `jetson_clocks` and disable swap to ensure stable measurement.

4.2 Experimental Results

Accuracy-Latency Trade-off. Figure 6 shows the accuracy–latency curves for the baseline and our method across all models and datasets. Our method consistently achieves a better accuracy–latency trade-off, with an average I/O speedup of 2.19× and up to 4.65× at comparable accuracy levels based on linear interpolation. The baseline often suffers from poor latency, especially at low to medium sparsity levels, occasionally increasing total latency—consistent with our observations in Figure 4. This issue is pronounced in smaller models, where channels are smaller, leading to more fragmented I/O. In contrast, our method decisively addresses these inefficiencies by tailoring selection to storage behavior, enabling consistently faster inference. Note that the slight accuracy gain at higher sparsity can appear when weak or noisy activations are removed. Similar regularization effects have been reported in pruning-based model compression work [11, 15].

Cross-Device Evaluation. Figure 7 presents results on Jetson Orin AGX (the full results provided in Appendix I show consistent trends). Our method delivers similar relative improvements, achieving

[4]For VideoDetailCaption, we queried the OpenAI endpoint `gpt-4o-mini-2024-07-18`, a more recent and stronger version than used in earlier work; thus, results are not directly comparable.

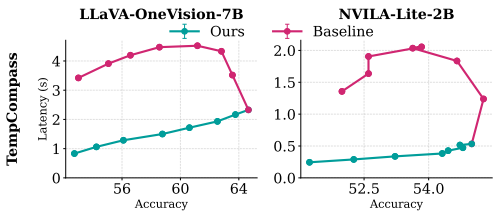


Figure 7: End-to-end performance on AGX.

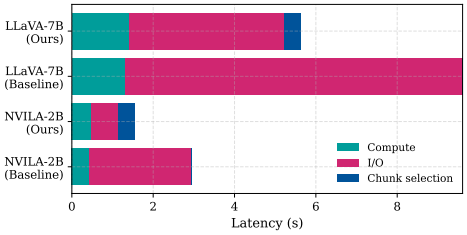


Figure 8: Latency breakdown.

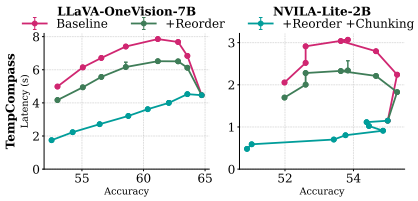


Figure 9: Ablation of two components: Reordering and Chunk selection.

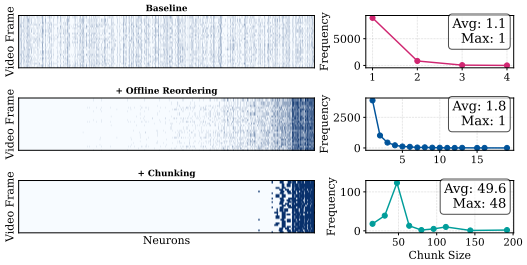


Figure 10: Contiguity distribution before/after our techniques are applied.

an average 2.89× speedup and up to 5.76× I/O latency reduction, computed over the full set of models and datasets. The larger speedup on AGX reflects its wider throughput gap between contiguous and scattered access.

Latency Breakdown. Figure 8 shows the latency breakdown at a 5% accuracy drop. The end-to-end speedup is smaller than the I/O-only gain because compute time remains nearly constant, so its relative share increases as I/O time decreases. This gap could narrow with optimized kernels or I/O–compute overlap [6, 10], which we do not apply in our evaluation. Our method substantially reduces I/O latency while incurring a slight compute increase, as maintaining the same accuracy requires loading marginally more channels. Nevertheless, the overall latency still decreases because data is fetched contiguously rather than through scattered accesses. The chunk selection overhead is modest—about 2 ms per weight matrix, totaling roughly 400 ms for the full model—which is small relative to total inference time.

Ablation Study. Figure 9 shows the accuracy–latency trade-off as each component is incrementally added: baseline, with hot–cold reordering, and with both reordering and chunk selection. For the LLaVA-7B model, hot–cold reordering yields up to a 1.23× speedup, which increases to up to 2.55× when chunk selection is additionally applied. Notably, online chunk selection plays a critical role, as the optimal subset of neurons is input-dependent and cannot be determined offline.

Visualization of Utility-Guided Chunk Selection Figure 10 visualizes selected channels for three variants—baseline, baseline with reordering, and baseline with both reordering and chunk selection—along with the contiguity distributions at matched accuracy. Reordering yields only modest gains by loosely clustering frequently activated channels. In contrast, chunk-based selection drives the dominant improvement: it targets high-utility contiguous regions, raising the average chunk size from roughly 1–2 to nearly 50. See Appendix J for results across a broader range of settings.

5 Discussion and Future Work

Generalization to other models and workloads. The proposed framework extends beyond vision-language models to a broader class of architectures and inference settings. The same principle of hardware-aware structured sparsification applies naturally to multi-token LLM inference scenarios such as speculative decoding, parallel sampling, and batched interactive serving, where activations aggregated across tokens yield smoother neuron-importance distributions. This property enables

latency-aware sparsification to maintain responsiveness in real-time, user-facing applications including chat assistants and copilots that operate under tight latency constraints.

The approach also generalizes to plain LLMs and ViT-based models that exhibit smooth activation magnitudes and operate under I/O-bound conditions. Recent LLMs increasingly employ non-ReLU activations such as SwiGLU or GeLU, making them amenable to our chunking formulation. Similarly, ViT-based models on edge devices benefit from reduced access fragmentation across smaller channel dimensions. Overall, these characteristics indicate that the proposed framework provides a general foundation for coupling structured sparsity with hardware-aware optimization across diverse model families. Additional details and preliminary results on LLMs and ViTs are presented in Appendix N.

Impact of Emerging I/O Mechanisms. Emerging I/O frameworks such as io_uring [4] offer improved support for asynchronous and scattered reads, which may reduce the performance gap between random and contiguous access. However, advances in storage hardware (e.g., internal prefetching, read coalescing) will likely continue to favor contiguity, suggesting that structured access optimization will remain beneficial.

Leveraging Additional Memory Budget for Caching. While our method assumes minimal memory availability, additional latency reduction is possible when the device has sufficient memory to cache frequently accessed weights. Caching strategies (e.g., hot-neuron caching) proposed in prior works [2, 38, 49] can be applied in a complementary manner by simply assigning zero importance to cached neurons. Once hot weights are cached, the remaining uncached accesses become more scattered (even after reordering), making our chunk-based selection more critical for sustaining I/O efficiency. However, if the device has sufficient memory to cache a large portion of the model weights, the overall flash I/O volume becomes negligible, reducing the benefit of our method.

6 Related Work

6.1 Activation Sparsification

Activation sparsification, which selectively loads weight channels corresponding to large activations, has been widely studied (See Appendix B.2 for details). Deja Vu [28] observed that MLP layers in LLMs exhibit significant dynamic sparsity, while CATS [16] extended this insight to modern LLMs, which utilize gated MLPs with non-ReLU activations. TEAL [24] further explored sparsification by applying it to attention layers. However, these methods are model-centric—they sparsify solely based on activation magnitude without considering hardware-level access patterns. This design choice was reasonable in their settings, where all weights reside in GPU VRAM and the bandwidth between device and shared memory saturates quickly even with limited access contiguity. In contrast, in flash-offloaded settings, such model-centric sparsification leads to significant performance degradation. Other approaches [31, 37, 39] seek to apply ReLU-ification to non-ReLU-based LLMs, fine-tuning them to enhance sparsity. Although effective, this method demands extensive retraining on a minimum of 50 billion tokens.

6.2 LLM Weight Offloading

LLM weights often exceed GPU VRAM capacity, prompting various offloading strategies. Some approaches [3, 36, 38] offload weights to CPU memory. This is impractical on edge SoCs with unified memory, where the CPU and GPU draw from the same DRAM pool and no additional capacity is gained [32].

Several recent works [2, 44, 49] adopt flash-based offloading and propose techniques to reduce I/O latency. LLM in a Flash [2] and PowerInfer-2 [49] improve I/O efficiency by bundling channels across projection layers, but the resulting gains in access contiguity are limited and rely on large memory budgets for caching (see Appendix L). Ripple [44] enhances access locality through offline neuron reordering, yet its reliance on ReLU-based sparsity and lack of hardware-aware runtime mechanisms limit its effectiveness to modern VLMs (see Section 4.2). These approaches were sufficiently effective for ReLU-based LLMs with high activation sparsity, where the total I/O volume was low enough to offset efficiency degradation. In contrast, VLMs exhibit smoother activation distributions and higher I/O demand, where such techniques fail to sustain efficiency under realistic I/O constraints.

6.3 LLM Compression

Various LLM compression techniques, including quantization [7, 21, 46], weight pruning [14, 26, 30, 48], and distillation [33, 40, 41], have been proposed to reduce computational and memory overhead. In contrast to these static compression approaches, activation sparsification is inherently input-adaptive, providing a distinct advantage in exploiting runtime activation dynamics. These methods are orthogonal to activation sparsification and can be combined to effectively mitigate I/O latency from storage devices [24].

7 Conclusion

We presented NEURON CHUNKING, a latency-aware activation sparsification approach tailored for flash-offloaded VLM inference. Unlike prior methods that treat I/O latency as a function of volume alone, our method models the performance implications of access contiguity and aligns neuron selection with storage behavior. We show that our contiguity-based latency model and utility-guided chunk selection algorithm consistently improve the accuracy–latency trade-off. These results underscore the importance of co-designing sparsification with hardware characteristics for efficient edge inference.

Acknowledgments and Disclosure of Funding

We sincerely thank our anonymous reviewers for their valuable comments. This work was supported by National Research Foundation (NRF) funded by the Korean government (MSIT) (No. RS-2024-00463802).

References

[1] llama.cpp. `https://github.com/ggml-org/llama.cpp`. URL `https://github.com/ggml-org/llama.cpp`. Accessed: 2025-02-27.

[2] Keivan Alizadeh, Seyed Iman Mirzadeh, Dmitry Belenko, S Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. Llm in a flash: Efficient large language model inference with limited memory. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12562–12584, 2024.

[3] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff 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*, pages 1–15. IEEE, 2022.

[4] Jens Axboe. Efficient i/o with io_uring. `https://kernel.dk/io_uring.pdf`, 2019. Accessed: 2025-05-15.

[5] Joya Chen, Zhaoyang Lv, Shiwei Wu, Kevin Qinghong Lin, Chenan Song, Difei Gao, Jia-Wei Liu, Ziteng Gao, Dongxing Mao, and Mike Zheng Shou. Videollm-online: Online video large language model for streaming video. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 18407–18418, 2024.

[6] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

[7] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in neural information processing systems*, 35: 30318–30332, 2022.

[8] NVIDIA Developer Forum. Gen 3 PCIe NVMe SSD with x4 Lanes Gets Higher IOPS on Nano Compared to the Xavier NX. `https://forums.developer.nvidia.com/t/`

gen-3-pcie-nvme-ssd-with-x4-lanes-gets-higher-iops-on-nano-compared-to-the-xavier-nx/228818, 2022. [Accessed 21-10-2025].

[9] Sylvain Gugger, Lysandre Debut, Thomas Wolf, Philipp Schmid, Zachary Mueller, Sourab Mangrulkar, Marc Sun, and Benjamin Bossan. Accelerate: Training and inference at scale made simple, efficient and adaptable. https://github.com/huggingface/accelerate, 2022. Accessed: 2025-02-27.

[10] Liwei Guo, Wonkyo Choe, and Felix Xiaozhu Lin. Sti: Turbocharge nlp inference at the edge via elastic pipelining. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 791–803, 2023.

[11] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *NeurIPS*, 2015.

[12] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

[13] Joo Kim, Donghyuk Lee, Sam Choi, and Myoungsoo Jung. Simplessd: Modeling high-fidelity modern ssds for holistic system simulation. *IEEE Computer Architecture Letters*, 15(1):31–34, 2016.

[14] Eldar Kurtic, Daniel Campos, Tuan Nguyen, Elias Frantar, Mark Kurtz, Benjamin Fineran, Michael Goin, and Dan Alistarh. The optimal bert surgeon: Scalable and accurate second-order pruning for large language models. *arXiv preprint arXiv:2203.07259*, 2022.

[15] Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. In *NeurIPS*, 1990.

[16] Donghyun Lee, Je-Yong Lee, Genghan Zhang, Mo Tiwari, and Azalia Mirhoseini. Cats: Contextually-aware thresholding for sparsity in large language models. *arXiv preprint arXiv:2404.08763*, 2024.

[17] Jingyu Lee, Hyunsoo Kim, Minjae Kim, Byung-Gon Chun, and Youngki Lee. Maestro: The analysis-simulation integrated framework for mixed reality. In *Proceedings of the 22nd Annual International Conference on Mobile Systems, Applications and Services*, pages 99–112, 2024.

[18] Bo Li, Yuanhan Zhang, Dong Guo, Renrui Zhang, Feng Li, Hao Zhang, Kaichen Zhang, Peiyuan Zhang, Yanwei Li, Ziwei Liu, et al. Llava-onevision: Easy visual task transfer. *arXiv preprint arXiv:2408.03326*, 2024.

[19] Mengtian Li, Yu-Xiong Wang, and Deva Ramanan. Towards streaming perception. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part II 16*, pages 473–488. Springer, 2020.

[20] Zonglin Li, Chong You, Srinadh Bhojanapalli, Daliang Li, Ankit Singh Rawat, Sashank J Reddi, Ke Ye, Felix Chern, Felix Yu, Ruiqi Guo, et al. The lazy neuron phenomenon: On emergence of activation sparsity in transformers. *arXiv preprint arXiv:2210.06313*, 2022.

[21] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of Machine Learning and Systems*, 6:87–100, 2024.

[22] Ji Lin, Hongxu Yin, Wei Ping, Pavlo Molchanov, Mohammad Shoeybi, and Song Han. Vila: On pre-training for visual language models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 26689–26699, 2024.

[23] Linux. llama.cpp. https://docs.kernel.org/next/filesystems/iomap/operations.html. URL https://docs.kernel.org/next/filesystems/iomap/operations.html. Accessed: 2025-03-05.

[24] James Liu, Pragaash Ponnusamy, Tianle Cai, Han Guo, Yoon Kim, and Ben Athiwaratkun. Training-free activation sparsity in large language models. *arXiv preprint arXiv:2408.14690*, 2024.

[25] Yuanxin Liu, Shicheng Li, Yi Liu, Yuxiang Wang, Shuhuai Ren, Lei Li, Sishuo Chen, Xu Sun, and Lu Hou. Tempcompass: Do video llms really understand videos? *arXiv preprint arXiv:2403.00476*, 2024.

[26] Zejian Liu, Fanrong Li, Gang Li, and Jian Cheng. Ebert: Efficient bert inference with dynamic structured pruning. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 4814–4823, 2021.

[27] Zhijian Liu, Ligeng Zhu, Baifeng Shi, Zhuoyang Zhang, Yuming Lou, Shang Yang, Haocheng Xi, Shiyi Cao, Yuxian Gu, Dacheng Li, et al. Nvila: Efficient frontier visual language models. *arXiv preprint arXiv:2412.04468*, 2024.

[28] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, et al. Deja vu: Contextual sparsity for efficient llms at inference time. In *International Conference on Machine Learning*, pages 22137–22176. PMLR, 2023.

[29] LMMs-Lab. Videodetailcaption dataset. `https://huggingface.co/datasets/lmms-lab/VideoDetailCaption`, 2024.

[30] Xinyin Ma, Gongfan Fang, and Xinchao Wang. Llm-pruner: On the structural pruning of large language models. *Advances in neural information processing systems*, 36:21702–21720, 2023.

[31] Iman Mirzadeh, Keivan Alizadeh, Sachin Mehta, Carlo C Del Mundo, Oncel Tuzel, Golnoosh Samei, Mohammad Rastegari, and Mehrdad Farajtabar. Relu strikes back: Exploiting activation sparsity in large language models. *arXiv preprint arXiv:2310.04564*, 2023.

[32] *Jetson AGX Orin Series Technical Reference Manual*. NVIDIA Corporation, 2022. URL `https://developer.nvidia.com/embedded/downloads`. Unified memory architecture specification.

[33] Haojie Pan, Chengyu Wang, Minghui Qiu, Yichang Zhang, Yaliang Li, and Jun Huang. Meta-kd: A meta knowledge distillation framework for language model compression across domains. *arXiv preprint arXiv:2012.01266*, 2020.

[34] PyTorch Core Team. Sort.cu: Cuda sorting kernels in pytorch. `https://github.com/pytorch/pytorch/blob/main/aten/src/ATen/native/cuda/Sort.cu`, 2025. Accessed: 2025-05-23.

[35] Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

[36] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR, 2023.

[37] Chenyang Song, Xu Han, Zhengyan Zhang, Shengding Hu, Xiyu Shi, Kuai Li, Chen Chen, Zhiyuan Liu, Guangli Li, Tao Yang, et al. Prosparse: Introducing and enhancing intrinsic activation sparsity within large language models. *arXiv preprint arXiv:2402.13516*, 2024.

[38] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. Powerinfer: Fast large language model serving with a consumer-grade gpu. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 590–606, 2024.

[39] Yixin Song, Haotong Xie, Zhengyan Zhang, Bo Wen, Li Ma, Zeyu Mi, and Haibo Chen. Turbo sparse: Achieving llm sota performance with minimal activated parameters. *arXiv preprint arXiv:2406.05955*, 2024.

[40] Siqi Sun, Yu Cheng, Zhe Gan, and Jingjing Liu. Patient knowledge distillation for bert model compression. *arXiv preprint arXiv:1908.09355*, 2019.

[41] Siqi Sun, Zhe Gan, Yu Cheng, Yuwei Fang, Shuohang Wang, and Jingjing Liu. Contrastive distillation on intermediate representations for language model compression. *arXiv preprint arXiv:2009.14167*, 2020.

[42] Amy Tai, Igor Smolyar, Michael Wei, and Dan Tsafrir. Optimizing storage performance with calibrated interrupts. *ACM Transactions on Storage (TOS)*, 18(1):1–32, 2022.

[43] Arash Tavakkol, Amirali Boroumand, and Onur Mutlu. Mqsim: A framework for enabling realistic studies of modern multi-queue ssd devices. In *FAST*, 2018.

[44] Tuowei Wang, Ruwen Fan, Minxing Huang, Zixu Hao, Kun Li, Ting Cao, Youyou Lu, Yaoxue Zhang, and Ju Ren. Ripple: Accelerating llm inference on smartphones with correlation-aware neuron management. *arXiv preprint arXiv:2410.19274*, 2024.

[45] Yuetian Weng, Mingfei Han, Haoyu He, Xiaojun Chang, and Bohan Zhuang. Longvlm: Efficient long video understanding via large language models. In *European Conference on Computer Vision*, pages 453–470. Springer, 2024.

[46] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pages 38087–38099. PMLR, 2023.

[47] Junbin Xiao, Xindi Shang, Angela Yao, and Tat-Seng Chua. Next-qa: Next phase of question-answering to explaining temporal actions. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9777–9786, 2021.

[48] Dongkuan Xu, Ian EH Yen, Jinxi Zhao, and Zhibin Xiao. Rethinking network pruning–under the pre-train and fine-tune paradigm. *arXiv preprint arXiv:2104.08682*, 2021.

[49] Zhenliang Xue, Yixin Song, Zeyu Mi, Xinrui Zheng, Yubin Xia, and Haibo Chen. Powerinfer-2: Fast large language model inference on a smartphone. *arXiv preprint arXiv:2406.06282*, 2024.

[50] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

[51] Kichang Yang, Juheon Yi, Kyungjin Lee, and Youngki Lee. Flexpatch: Fast and accurate object detection for on-device high-resolution live video analytics. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pages 1898–1907. IEEE, 2022.

[52] Kichang Yang, Minkyung Jeong, Juheon Yi, Jingyu Lee, KyoungSoo Park, and Youngki Lee. Logan: Loss-tolerant live video analytics system. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, pages 1314–1329, 2024.

[53] Shengyuan Ye, Jiangsu Du, Liekang Zeng, Wenzhong Ou, Xiaowen Chu, Yutong Lu, and Xu Chen. Galaxy: A resource-efficient collaborative edge ai system for in-situ transformer inference. In *IEEE INFOCOM 2024-IEEE Conference on Computer Communications*, pages 1001–1010. IEEE, 2024.

[54] Peiyuan Zhang, Kaichen Zhang, Bo Li, Guangtao Zeng, Jingkang Yang, Yuanhan Zhang, Ziyue Wang, Haoran Tan, Chunyuan Li, and Ziwei Liu. Long context transfer from language to vision. *arXiv preprint arXiv:2406.16852*, 2024. URL `https://arxiv.org/abs/2406.16852`.

[55] Yuanhan Zhang, Bo Li, haotian Liu, Yong jae Lee, Liangke Gui, Di Fu, Jiashi Feng, Ziwei Liu, and Chunyuan Li. Llava-next: A strong zero-shot video understanding model, April 2024. URL `https://llava-vl.github.io/blog/2024-04-30-llava-next-video/`.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: Yes, the main claims in the abstract and introduction are well-aligned with the paper's contributions.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Yes, the paper discusses its limitations, including the greedy algorithm's inability to capture interactions between adjacent low-scoring chunks, and the latency model's exclusion of overheads, particularly in low-volume or low-performance settings.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [NA]

Justification: N/A — the paper does not present formal theoretical results or proofs, as the focus is on empirical modeling and algorithm design for practical sparsification under I/O constraints.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: Yes, the paper provides sufficient detail to reproduce the main experimental results, including latency modeling procedures, benchmark setup, hardware configurations, and algorithmic design. While full code and data availability may vary, all key components affecting the main claims are clearly described.

Guidelines:

- The answer NA means that the paper does not include experiments.
- If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some

way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: We plan to open-source the code and provide it with sufficient instructions to reproduce the main experimental results. However, at the time of review, the code and data are not yet publicly available.

   Guidelines:

   - The answer NA means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: Yes, although the paper does not involve training models from scratch, it provides detailed information on the experimental setup used for evaluation, including sparsity levels, chunk size configurations, latency measurement procedures, and hardware settings. These details are sufficient to understand and interpret the results.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: Yes, the paper reports medians with 95% bias-corrected and accelerated (BCa) bootstrap confidence intervals across 30 independent runs for key latency experiments, providing an appropriate indication of variability and robustness in the results.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: The paper provides hardware details for all experiments, including device models (Jetson AGX Orin, Jetson Orin Nano), SSD types, and use of Linux direct I/O with multi-threaded C++ implementations. While exact memory usage and execution time are not reported for each experiment, the provided information is sufficient to approximate the computational setup required for reproduction.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: Yes, the research conforms to the NeurIPS Code of Ethics. It does not involve human subjects, sensitive data, or deceptive practices, and it promotes efficient computation by aligning sparsification with hardware constraints—contributing positively to sustainability and resource-aware machine learning.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.

- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [NA]

    Justification: N/A — the paper focuses on algorithmic and system-level contributions and does not discuss societal impacts.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: N/A — the paper does not release new pretrained models or datasets that pose a high risk of misuse.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: Yes, the paper properly credits the creators of all models and datasets used, such as Qwen2-7B and LLaVA-OneVision, and uses them in accordance with their respective licenses and terms of use.

Guidelines:

- The answer NA means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [Yes]

Justification: The paper introduces a new algorithm and latency modeling framework. While these are clearly described in the main text and appendix, the associated code and assets are not yet released. We are planning to open-source the implementation, which is expected to include accompanying documentation.

Guidelines:

- The answer NA means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [NA]

Justification: N/A — the paper does not involve crowdsourcing or research with human subjects.

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: N/A — the paper does not involve human subjects and therefore does not require IRB approval.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: Yes, the paper clearly describes the usage of LLMs (e.g., Qwen2-7B, LLaVA-OneVision) as part of the experimental setup for evaluating sparsification strategies during inference. Their role is central to the method and analysis, and they are properly credited.

    Guidelines:

    - The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.

A Appendix Overview

This appendix provides additional details supporting the main paper.

Default Setup. Unless otherwise specified, we adopt the same default configuration as used in the main paper's evaluation: Llava-OneVision-Qwen2-7B [18] as the model and the TempCompass [25] dataset as the input source. For end-to-end evaluation on Jetson Orin AGX (Appendix I), we use the full set of evaluation datasets following the setup in the main paper. For analysis-oriented experiments—including visualization, reordering, and activation statistics—we use a subset of 25 videos from TempCompass that were excluded from the main evaluation (as described in Section 5.1). In the reordering experiment, 20 videos are used for calibration and 5 for validation (i.e., result visualization). When analyzing attention and MLP modules, we focus on the `q`, `o`, `gate`, and `down` projections, and omit `k`, `v`, and `up` since they share input activations with `q` and `gate`, respectively. Additionally, we include three representative layers—early (0), middle (13), and late (27)—to capture variation across layer depths (LLaVA-OneVision-Qwen2-7B has 28 layers). We restate this setup where relevant throughout the appendix.

B Extended Background on VLM Sparsification

B.1 Vision–Language Model Inference Process

A vision–language model consists of a vision encoder f_{vision}, a projector f_{proj}, and a backbone LLM f_{llm}.

Given a language prompt $p = \{t_1, t_2, \ldots, t_m\}$ of m tokens, the model first performs a *prefill* step that processes all tokens at once to generate key–value (KV) caches for each transformer layer:

$$t_{m+1},\ \mathrm{KV}_{<m+1} = f_{llm}(p)$$

In the *frame appending* stage, each video frame F_i ($i = 1, \ldots, N$) is processed upon arrival. Each frame is encoded into n visual tokens via the vision encoder and projector:

$$v^{(i)} = \{v_1^{(i)}, v_2^{(i)}, \ldots, v_n^{(i)}\} = f_{proj}(f_{vision}(F_i))$$

These tokens are then fed into the LLM to produce additional KV pairs, which are appended to the existing cache:

$$t_{m+ni+1},\ \mathrm{KV}_{<m+ni+1} = f_{llm}(v^{(i)}, \mathrm{KV}_{<m+n(i-1)+1})$$

During the prefill and frame appending stage, the token output of f_{llm} is ignored; only the KV cache is used in subsequent decoding.

In the *decoding* stage, a new token is generated one at a time autoregressively. At decoding step j, the model takes the previous token t_{m+nN+j} and the current KV cache to generate the next token:

$$t_{m+nN+j+1},\ \mathrm{KV}_{<m+nN+j+1} = f_{llm}(t_{m+nN+j}, \mathrm{KV}_{<m+nN+j})$$

The decoding stage may begin in one of two ways: either via an explicit query provided by the user (e.g., a natural language instruction, which is appended in a similar way to visual tokens), or by designing the model to emit a special control token at important frames, signaling that decoding should commence. In the former case, decoding starts after appending the query tokens; in the latter, the final generated token from the last frame is preserved and used as the first input of the decoding stage.

B.2 Activation Sparsification

Notation. Let the activation vector (also referred to as hidden states) for a single layer be $a \in \mathbb{R}^m$ and the corresponding weight matrix be $W \in \mathbb{R}^{m \times n}$, where m is the number of neurons (also referred to as channels) and n is the output dimension. Each row $W_i \in \mathbb{R}^n$ of W corresponds to a single neuron, contributing to the output through the dot product $a_i W_i$. Thus, the output $y \in \mathbb{R}^n$ is computed as:

$$y = a^\top W = \sum_{i=1}^{m} a_i W_i,$$

which can be interpreted as a weighted sum over neurons, where the activation values a_i act as per-sample dynamic weights.

Saliency via Magnitude. In modern LLMs where non-ReLU activation functions (e.g., SwiGLU, GeGLU) are standard, activation values are not exactly zero—as opposed to ReLU-based activation functions, which produce exact zeros for inactive neurons. As a result, identifying salient neurons—those most critical to output quality—is nontrivial. Prior works such as TEAL [24] and CATS [16] propose using the magnitude of activations as a proxy for saliency. This approach assumes that neurons with higher $|a_i|$ contribute more significantly to the output.

Magnitude-Based Sparsification. Given a sparsity target $s \in [0, 1)$, the goal is to retain only the top-$(1 - s)m$ neurons per input based on their importance. The process is as follows:

1. Compute importance scores $v_i = |a_i|$ for $i = 1, \ldots, m$.
2. Select a binary mask $M \in \{0, 1\}^m$ such that $M_i = 1$ if $|a_i|$ is among the top-$(1 - s)m$ entries of v, and $M_i = 0$ otherwise.
3. Construct the sparsified output:

$$\tilde{y} = \sum_{i=1}^{m} M_i a_i W_i.$$

This technique is input-dependent and requires re-evaluation of M at each inference step. While simple and effective, it does not consider the memory access cost of retrieving weight rows from flash storage, which becomes critical in flash-offloaded inference settings. An alternative to top-$(1 - s)m$ selection is to use a fixed activation threshold to filter out low-importance neurons.

In vision-language models (VLMs), where a single input (e.g., image) corresponds to multiple tokens, we extend this method by computing the importance of each neuron as the average absolute activation magnitude across tokens. This yields a single importance vector per input, allowing sparsification to proceed as in the single-token case.

C Additional Evidence of Activation Smoothness Across VLMs

To further validate that the smoothing effect is a general architectural property of VLMs rather than a model-specific behavior, we measured the coefficient of variation (CV) of neuron importance before the down-projection layer—where conventional sparsification is typically applied in ReLU-based LLMs—across multiple VLM architectures and a ReLU-based baseline (OPT-6.7B).

Table 1: Coefficient of variation (CV) of neuron importance before the down-projection layer across multiple models.

Layer	LLaVA-7B	LLaVA-0.5B	VILA-8B	NVILA-2B	LongVA	OPT-6.7B
First	1.44	1.31	1.25	1.07	1.20	11.65
Mid	1.25	1.33	1.38	1.32	1.34	8.63
Last	3.30	3.58	2.48	4.55	3.01	9.19

Across all VLM models, the CV values (1.07–4.55) are dramatically lower than those of the ReLU-based baseline (8.63–11.65), demonstrating that smooth activation distributions are a consistent property of modern VLM architectures. This smoothing effect makes contiguity-aware selection particularly beneficial for VLMs: when importance differences between neurons are small, I/O efficiency becomes the decisive factor for overall performance.

D Benchmark Details

We profiled read throughput as a function of chunk size on two devices: Jetson AGX Orin (Samsung 990 Pro SSD) and Jetson Orin Nano (SK Hynix Gold P31 SSD). Each device reaches 99% of its peak

throughput at approximately 236 KB (AGX) and 348 KB (Nano). Measurements were taken in 1 KB increments up to the saturation point, with all runs completing within 20 minutes per device.

Profiling Setup.

- Prepare a large dummy file (e.g. 128MB) on flash-backed storage.
- Issue sequential reads of size $s \in \{1\,\text{KB}, 2\,\text{KB}, \ldots, S_{\max}\}$, where $S_{\max}$ is the smallest size reaching 99% of peak throughput.
- Record average throughput over multiple trials.

Throughput variance was negligible (standard deviation $<1\%$ of the mean) across all sizes.

E Algorithm Implementation Details

Algorithm 1 provides a pseudocode of our multi-scale chunk selection method. Below, we describe the corresponding implementation in detail, which is designed for runtime efficiency and integrates both CPU and GPU components.

Inputs. The inputs to Algorithm 1 are:

- $V \in \mathbb{R}^N$: Activation magnitudes.
- R: Total number of rows to select.
- `row_size_KB`: Size of each row in kilobytes, used to convert kilobyte-based parameters to row units.
- $[s_{\text{min}}, s_{\text{max}}]$ and Δs (in KB): Define the chunk size range and the granularity of sizes considered.
- `jump_cap` (in KB): Limits the maximum stride between starting indices of candidate chunks for efficiency.
  - By default, stride equals the chunk size (i.e., non-overlapping).
  - If the chunk size exceeds `jump_cap`, stride is clipped to the cap, allowing overlapping candidates.
- $L(\cdot)$: Device-specific latency lookup function mapping chunk size (in rows) to access cost.
- In the actual implementation, a device flag selects the appropriate lookup table (AGX or Nano). In the pseudocode, this is simplified by directly passing $L(\cdot)$.

Prefix Sum. To enable constant-time computation of the sum of importance for any contiguous chunk, a CPU-side prefix sum of the activation magnitudes is first computed.

Chunk Candidate Generation. For each chunk size s (converted to row count), the algorithm slides a window across the activation vector in steps of $\min(s, \texttt{jump\_cap})$. Each candidate chunk is scored using the ratio of summed importance to estimated latency, with latency values retrieved from a pre-profiled lookup table $L(s)$ based on hardware throughput (see Appendix D).

GPU Sorting. The importance-to-cost scores of all candidate chunks are transferred to GPU memory and sorted in descending order using PyTorch's GPU-accelerated sort. This step enables scalable candidate prioritization with minimal overhead.

Greedy Selection. Candidates are selected greedily based on the sorted scores. Each selected chunk is added to the output mask if it does not overlap with already selected rows and does not exceed the remaining budget R. Overlaps are checked with early termination, and the mask is updated in-place.

The algorithm design reflects the trade-off between optimality and runtime feasibility: by limiting the chunk search space and leveraging GPU sorting, it enables input-dependent sparsification at inference time within few milliseconds latency.

Algorithm 1 Multi-scale Chunk Selection

Require: Activation magnitudes $V \in \mathbb{R}^N$, number of rows to select R, row size in KB, chunk size range $[s_{\text{min}}, s_{\text{max}}]$ in KB, step size Δs in KB, jump cap in KB, latency lookup function $L(\cdot)$
Ensure: Binary mask indicating selected rows

Convert chunk-related parameters to row units:
  $r_{\text{min}} \leftarrow \max(1, \lfloor s_{\text{min}}/\text{row\_size\_KB} \rfloor)$
  $r_{\text{max}} \leftarrow \max(1, \lfloor s_{\text{max}}/\text{row\_size\_KB} \rfloor)$
  $\Delta r \leftarrow \max(1, \lfloor \Delta s/\text{row\_size\_KB} \rfloor)$
  jump_cap_rows $\leftarrow \max(1, \lfloor \text{jump\_cap}/\text{row\_size\_KB} \rfloor)$
Compute prefix sum array: cumsum$[0:N] \leftarrow$ prefix_sum(V)
Initialize empty candidate list C
for r from r_{min} to r_{max} with step Δr **do**
  stride $\leftarrow \min(r, \text{jump\_cap\_rows})$
  for $i = 0$ to $N - r$ with step stride **do**
    benefit $\leftarrow$ cumsum$[i + r] -$ cumsum$[i]$
    cost $\leftarrow L(r)$
    Append candidate $(\text{benefit}/\text{cost}, i, r)$ to C
  end for
end for
Sort C by score descendingly (GPU-accelerated)
Initialize mask $[0:N] \leftarrow 0$, selected $\leftarrow 0$
for candidate $(\_, i, r)$ in sorted C **do**
  if chunk overlaps selected rows or $r >$ remaining **then**
    continue
  end if
  Set mask$[i : i + r] \leftarrow 1$
  selected $\leftarrow$ selected $+ r$
  if selected $\geq R$ **then**
    break
  end if
end for
return mask

F Neuron Activation Frequency Analysis

Figure 11 illustrates the distribution of neuron activation frequency across different layers when the effective sparsity is 40%. The plot is structured as a 3×4 grid, where each row corresponds to a layer and each column to an activation type. Many neurons are neither always-on nor always-off, confirming the presence of input-dependent sparsity in VLMs, consistent with prior findings in LLMs [20, 28]. This suggests that input-aware sparsification remains effective in our setting, although such dynamic sparsity inevitably leads to fragmented access patterns.

Additionally, TEAL profiling introduces sparsity variation across layers, resulting in some layers with very high or low sparsity (e.g., q projection of layer 0 has 94% sparsity). These layers exhibit a high proportion of hot or cold neurons, suggesting that simple offline hot–cold reordering can be effective for improving contiguity in these cases.

G Impact of Offline Reordering Schemes

Although offline reordering is not our primary focus—we target online policies—Figure 12 compares the contiguity of selected neurons before and after applying offline reordering, using either hot–cold reordering or Ripple's [44] coactivation-based method. Both methods yield modest improvements over the original ordering, with comparable gains across most layers. While Ripple performs better in one case (the o projection of layer 0), the overall difference is minor, suggesting that hot–cold reordering offers a lightweight and effective alternative.

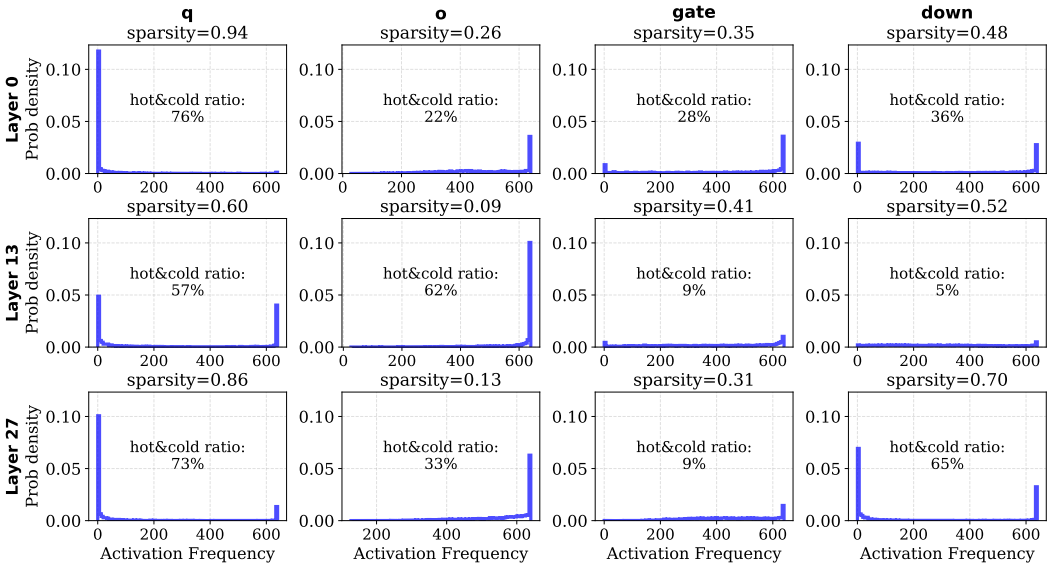


Figure 11: Activation frequency of neurons across layers, with effective sparsity set to 40% (layer-wise sparsity determined by TEAL [24] profiling). The text in the center of each plot indicates the proportion of hot neurons (activated >99% of the time) and cold neurons (<1%).

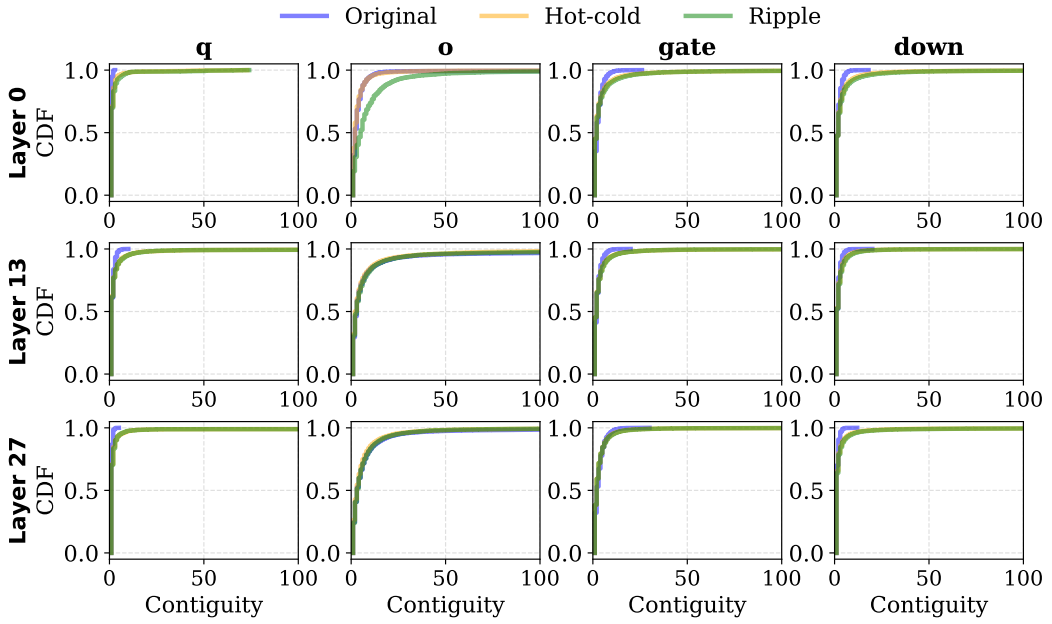
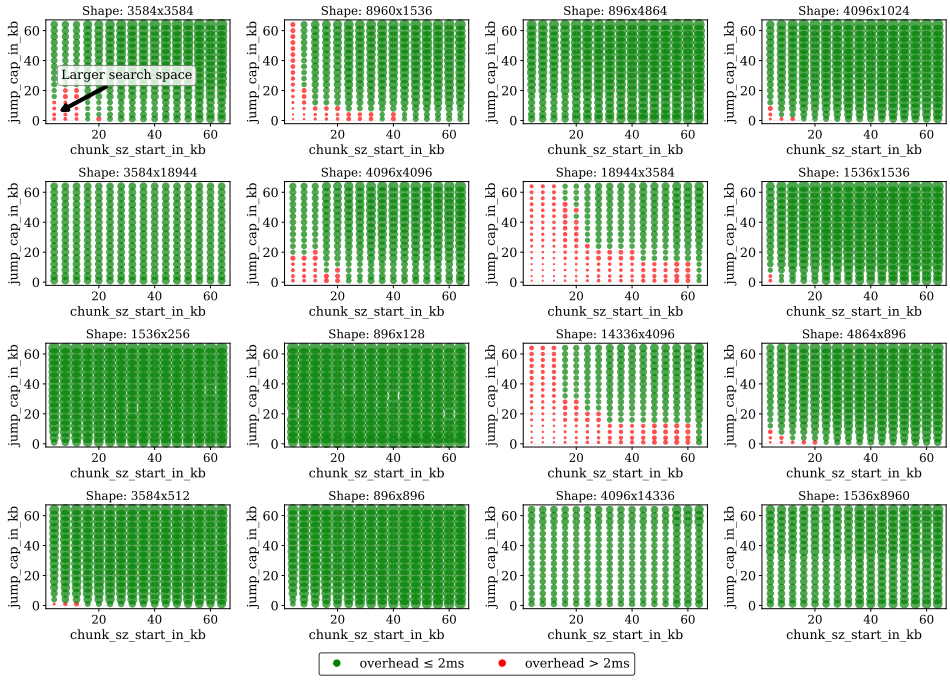
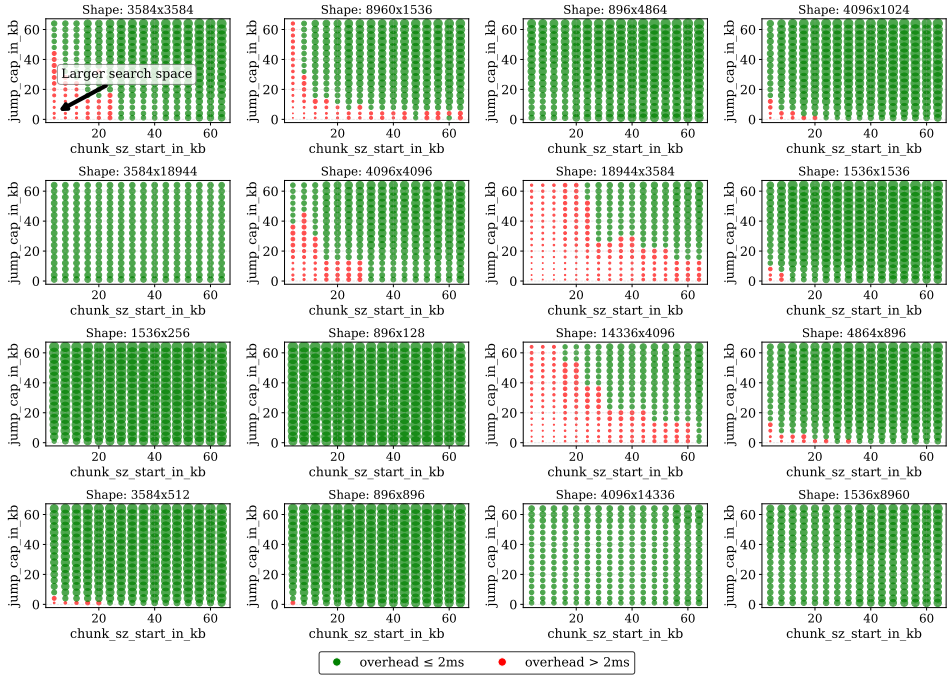


Figure 12: CDF of contiguity of selected neurons before and after reordering, with sparsity=0.4.

H Hyperparameter Selection



(a) Jetson Orin AGX



(b) Jetson Orin Nano

Figure 13: Runtime overhead of chunk selection across hyperparameter configurations on Jetson Orin AGX (top) and Jetson Orin Nano (bottom). Each point represents a configuration defined by starting chunk size (`chunk_sz_start_in_kb`, x-axis) and jump cap (`jump_cap_in_kb`, y-axis). Step size is set equal to the start size; the chunk size end is fixed from I/O profiling (236 KB for AGX, 348 KB for Nano). Circle size is inversely proportional to runtime, and color indicates whether the 2 ms latency threshold is exceeded.

The hyperparameters of our chunk selection algorithm are chosen with two objectives in mind: (i) the runtime overhead must remain within a practical latency threshold (under 2 ms), and (ii) the compromise in selection quality for computational efficiency should be minimal.

We adopt a two-stage selection strategy. First, we filter out configurations that exceed 2 ms of runtime overhead. Since the overhead depends on the shape of the weight matrix, we benchmark each configuration across representative matrix shapes drawn from the models used in our evaluation. Measurements are conducted at sparsity 0.1 to conservatively capture the worst-case overhead.

Among the remaining feasible configurations, we heuristically select those near the lower-left region of the search space—where chunk sizes grow in a fine-grained manner and the chunk stride is small. These settings allow for broader search coverage while maintaining overhead within budget.

Figure 13 shows the measured overhead for Jetson Orin AGX (top) and Jetson Orin Nano (bottom). For each device, we sweep over the starting chunk size (`chunk_sz_start_in_kb`, x-axis) and the jump cap (`jump_cap_in_kb`, y-axis), where the hyperparameter space spans from 0 to 64 KB in 4 KB increments. For simplicity, the step size is set equal to the start size, and the end size is fixed from I/O profiling—236 KB for AGX and 348 KB for Nano. Each configuration is evaluated 30 times using randomly generated activation magnitudes. This provides a reliable estimate, as over 80% of the total runtime is dominated by GPU sorting via a data-independent radix sort [34], allowing random inputs to be used for measuring overhead.

We observe two clear trends: (i) configurations involving large weight matrices (e.g., 18944×3584) tend to incur higher overhead, making some configurations infeasible; (ii) AGX supports more configurations due to its higher compute capacity compared to Nano.

Final hyperparameters are selected near the boundary between feasible (green) and infeasible (red) regions, with a small margin for safety. The selected settings are summarized in Table 2.

While we have not conducted a full sensitivity analysis, our empirical findings suggest that the method performs consistently well across a range of hyperparameter settings. A more thorough investigation into the effects of different configurations remains an interesting direction for future work.

Table 2: Selected hyperparameters per weight matrix shape on Jetson Orin AGX and Nano

Shape (Rows × Cols)	**AGX**		**Nano**	
	`chunk_sz`	`jump_cap`	`chunk_sz`	`jump_cap`
(3584, 3584)	20	20	24	36
(8960, 1536)	16	16	20	20
(896, 4864)	8	8	8	8
(4096, 1024)	12	12	16	16
(3584, 18944)	8	8	8	8
(4096, 4096)	20	20	24	24
(18944, 3584)	32	32	36	36
(1536, 1536)	16	12	16	12
(1536, 256)	8	8	8	8
(896, 128)	8	8	8	8
(14336, 4096)	32	32	40	36
(4864, 896)	12	16	20	16
(3584, 512)	8	12	8	12
(896, 896)	8	8	8	8
(4096, 14336)	8	8	8	8
(1536, 8960)	8	8	8	8

I Full Evaluation Results on Jetson Orin AGX

Figure 14 presents full results on Jetson Orin AGX. Due to its powerful SSD and compute capability, overall latency is lower compared to Jetson Orin Nano.

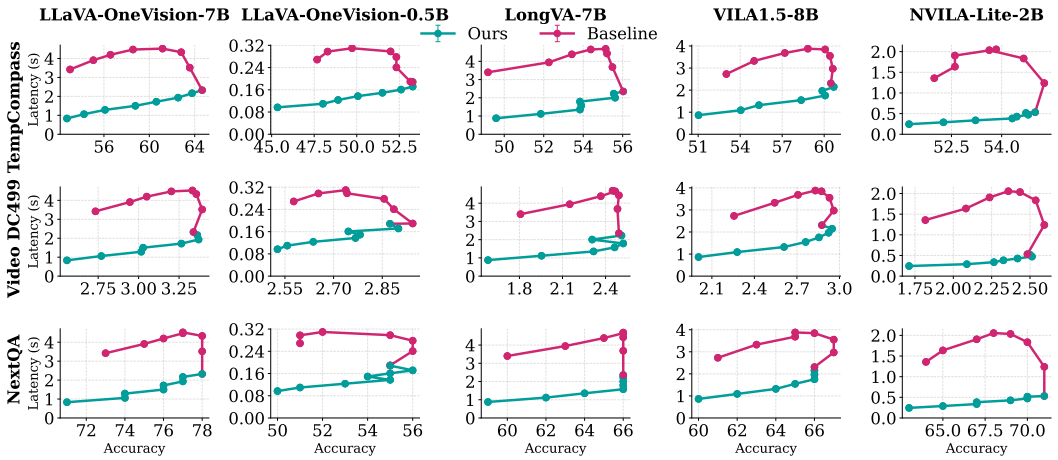


Figure 14: End-to-end performance on Jetson Orin AGX.

J Extended Visualization of Mask Patterns and Contiguity

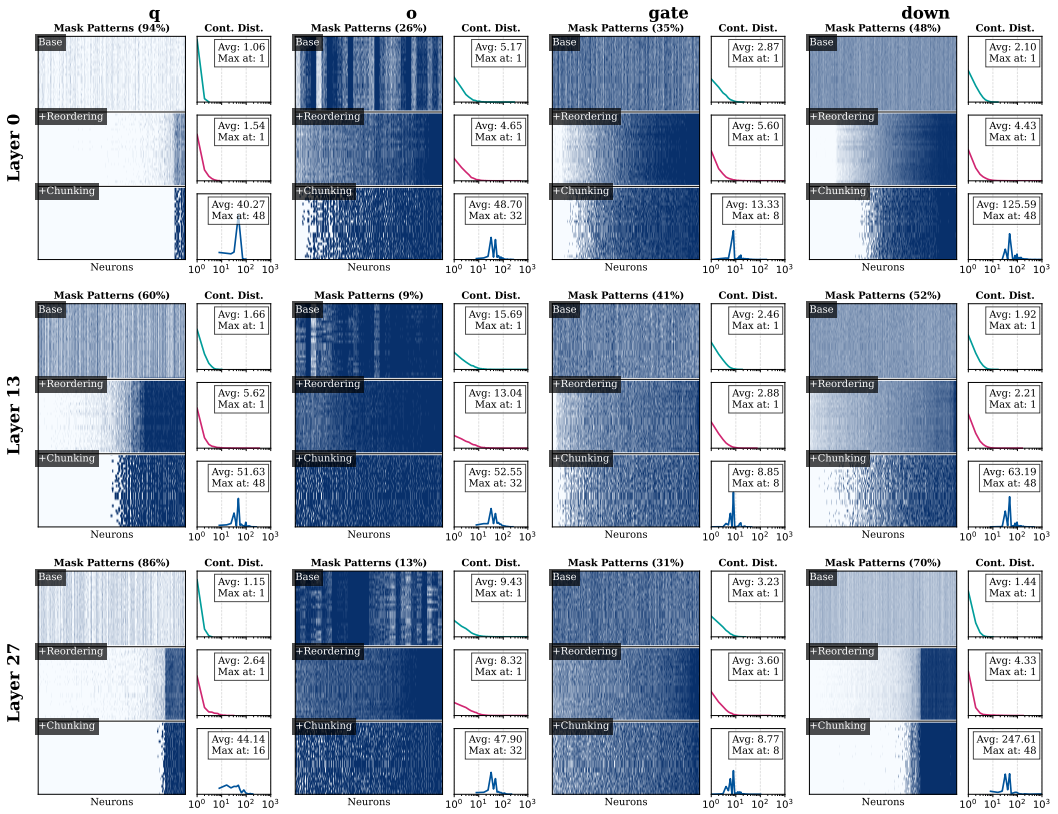


Figure 15: Mask patterns and corresponding contiguity distributions before and after applying our method, shown across different layers (0, 13, 27) and activation types (q, o, gate, down) at sparsity=0.4.

Figure 10 presented a case study on the effect of our method on mask patterns and contiguity distributions for layer 0, activation type q, under effective sparsity 0.3 and profiled sparsity 0.9.

Here, we provide an extended visualization across a broader range of settings in Figure 15. The visualization is structured as a 3×4 grid, where each row corresponds to a layer and each column

to an activation type. Each cell contains two subfigures: the left shows the binary mask patterns for three configurations—baseline, baseline with reordering, and baseline with reordering plus chunking—stacked vertically. The x-axis represents neuron index and the y-axis represents different input samples. The right subfigure presents the corresponding contiguity distribution, with the x-axis (log-scaled) denoting chunk size and the y-axis showing density. We annotate both the average and the mode (i.e., the most frequent chunk size) for each distribution.

These visualizations highlight that our method consistently promotes contiguity across layers and activation types. Qualitatively, the mask patterns become visibly less fragmented, particularly in high-sparsity regimes such as `q` and `down`. Quantitatively, both the average and mode of the contiguity distribution shift toward larger chunk sizes. While offline reordering provides marginal improvements, the majority of the contiguity gain arises from our online chunk selection policy, which adapts to input-dependent activation patterns (see Appendix F).

K Effect of Visual Token Density

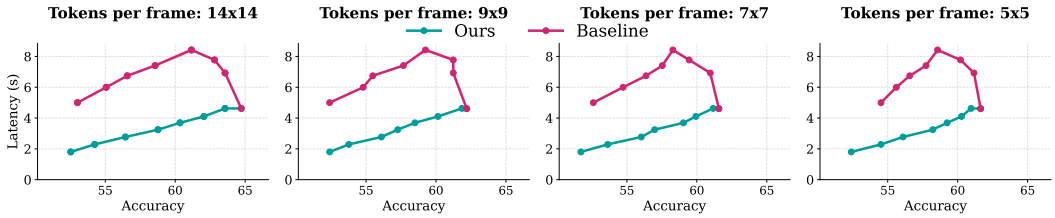


Figure 16: End-to-end performance on Jetson Orin Nano under different token counts per frame.

A large number of tokens per frame limits how many frames can fit within the model's context window. To address this, various token reduction techniques have been proposed, ranging from simple spatial pooling [55] to more advanced clustering [45]. We evaluate the impact of per-frame token count in Figure 16, using spatial pooling to control the number of tokens. As the token count decreases, we observe a modest drop in accuracy across all methods. Nonetheless, our method consistently outperforms the baseline, indicating that the I/O benefits of our approach are robust to changes in visual token density.

L Extended Comparison with Related Methods

We provide extended comparisons with three major classes of related methods: (i) *LLM in a Flash* [2], which also addresses flash-offloaded inference; (ii) *TEAL* [24], upon which our baseline implementation is built; and (iii) regularization-based pruning methods (e.g., L1 and group-Lasso), which promote sparsity through training-time penalties.

LLM in a Flash. *VLMs' smooth activation distributions require fundamentally different approaches.* Our work identifies critical inefficiencies in flash-offloaded VLM inference unaddressed by LLM in a Flash (hereafter LLMFlash). LLMFlash targets ReLU-based LLMs with >90% sparsity; we target non-ReLU VLMs with smoother activation distributions requiring 40-60% sparsity. This difference creates a counterintuitive phenomenon: in VLMs, higher sparsity can increase latency due to fragmentation-induced throughput degradation outweighing data savings. This workload difference motivates our methodological divergence. While LLMFlash simply reduces total I/O volume using sparsification, we explicitly model I/O latency through contiguity-aware cost modeling and jointly consider neuron importance and I/O efficiency during sparsification. Given these workload differences, we now analyze why LLMFlash's core techniques cannot effectively address our contiguity challenges.

Neuron bundling proves insufficient without explicit contiguity optimization. LLMFlash employs row–column bundling, grouping weights corresponding to the same activation (up-projection columns with down-projection rows). However, bundling alone is insufficient in our setting due to both hardware and methodological differences. Hardware-wise, LLMFlash was evaluated on MacBooks, where throughput saturates at chunk sizes $<$100 KB, whereas our Jetson devices require 236–348 KB for peak throughput (Figure 4a). This discrepancy likely arises from Jetson boards routing NVMe interrupts to a single CPU core, causing IOPS saturation, in contrast to MacBooks' multi-core

interrupt distribution [8, 42]. Methodologically, LLMFlash's up/down-projection bundling conflicts with our predictor-free approach, where each matrix is sparsified based on its own activations. Even when adapted to bundle matrices sharing input activations (e.g., Q/K/V or up/gate), the largest bundled weights ($\sim$74 KB) achieve only half the optimal bandwidth on our hardware. Thus, bundling alone cannot achieve peak I/O performance—explicitly targeting contiguity as a design objective is essential.

Furthermore, when techniques such as TEAL's sparsification are combined with bundling, they introduce preprocessing and postprocessing overheads and can yield paradoxical I/O behavior. For instance, bundling Q/K/V matrices based on overlapping neurons improves locality for bundled weights but scatters remaining unbundled neurons across matrices, leading to fragmented reads. The net performance depends on whether contiguity gains outweigh fragmentation penalties, making bundling's effectiveness highly pattern-dependent.

Table 3: Comparison between our method and bundling-based implementations across models and datasets. Each cell shows two average speedup ratios: (1) ours vs. baseline and (2) ours vs. baseline+bundling.

Dataset / Model	**LLaVA-7B**	**LLaVA-0.5B**	**VILA-8B**	**NVILA-2B**	**LongVA**
TempCompass	2.06/2.41	2.05/1.94	1.60/1.83	3.24/3.76	2.15/2.50
Video DC499	2.11/2.45	2.06/2.02	1.60/1.78	3.22/3.70	2.25/2.59
NextQA	1.76/1.98	2.12/1.99	1.50/1.70	3.44/3.96	2.04/2.34

Table 3 reports the experimental results comparing our method with bundling-based implementations. Our method achieves consistent speedups of 1.5–3.4$\times$ over the baseline and 1.7–4.0$\times$ over bundling-based implementations across all models and datasets. Bundling degrades performance in most cases except LLaVA-0.5B, confirming that its benefits are unpredictable and pattern-dependent, whereas our contiguity-aware approach consistently improves efficiency.

Sliding window caching vs. offline reordering trades memory for adaptability. LLMFlash's sliding-window caching maintains recently activated neurons' weights in memory, trading memory for reduced latency—an infeasible approach in our memory-constrained edge deployments. Both their caching and PowerInfer [38]'s hot neuron caching leverage additional memory to reduce flash accesses by exploiting statistical access patterns. Instead, we employ offline reordering for comparable benefits: both methods make frequently accessed weights cheaper to load, but caching is runtime-adaptive while consuming memory, whereas reordering imposes zero memory overhead while being less adaptive. In practice, we keep only essential weights—vision encoder, LM head, and KV cache—in device memory, representing the truly "hottest" components. Our chunk selection algorithm naturally accommodates any caching strategy by assigning zero importance to cached neurons, making it flexible and complementary to memory optimization approaches.

TEAL. Our method builds upon TEAL's fine-grained sparsity allocation across matrices rather than applying uniform sparsity. However, our focus differs fundamentally: TEAL determines *how much* to sparsify each layer under uniform access cost assumptions, whereas we determine *which* neurons to load at runtime by jointly considering activation importance and I/O efficiency in flash-based systems. This introduces latency-aware chunk selection that restructures selected neurons into contiguous memory layouts, aligning model-level sparsification with system-level latency behavior.

Regularization-based pruning. L1 regularization typically operates at the individual-weight level and rarely eliminates entire rows or columns of weights. As a result, it does not reduce the number of rows that must be loaded from flash, limiting its effect on activation sparsity and overall latency.

To meaningfully impact latency, sparsity must be *structured* (e.g., at the row or column level). This can be achieved by replacing L1 with group-Lasso regularization that applies L2-norm penalties to entire rows or columns. Column-wise regularization (e.g., applied to gate or up-projection matrices) encourages certain output activation channels to become zero, effectively deactivating the corresponding rows in the down-projection matrix and increasing activation sparsity. Row-wise regularization can also promote sparsity when the neuron-importance metric incorporates both activation magnitude and weight norm, lowering the importance of neurons associated with low-norm rows.

These regularization-based approaches are inherently input-agnostic: they prune the same weight regardless of the input context. This limits their achievable sparsity before severe accuracy degradation occurs—as shown in TEAL [30], where even 20% pruning causes noticeable performance drops.

M On Tradeoffs Between Accuracy and Latency

Our method is designed not to preserve accuracy at all costs, but to enable a more favorable tradeoff between accuracy and latency. In latency-sensitive deployments, it is often preferable to accept a modest accuracy degradation in exchange for significantly faster responses. The objective is not merely to match the performance of dense inference, but to shift the accuracy–latency Pareto frontier—achieving lower latency for comparable accuracy, or improved accuracy within a fixed latency budget. This tradeoff is particularly valuable in practical vision-language applications where the input is a video and the output is a natural language response. In many such scenarios, the user can quickly validate or refine the system's output, making responsiveness more critical than marginal gains in precision. Examples include:

- **Object or person retrieval.** When the user asks, e.g., "Where is the person in a red shirt?" or "Is the car still visible in this scene?", delivering fast candidate answers enables immediate visual verification and iteration.
- **Temporal localization.** In tasks like "When does the object fall?" or "At what time does the person enter the room?", coarse-grained temporal answers that arrive quickly are often more useful than delayed fine-grained ones.

In these streaming input scenarios, delayed responses can themselves degrade performance. This phenomenon—often referred to as *streaming accuracy*—has been observed in streaming perception literature, where the timeliness of model outputs directly influences their correctness [17, 19, 51, 52].

In such use cases, VLMs are part of interactive systems where user experience benefits more from fast responses than from exact answers. Our method supports this objective by enabling structured sparsification that reduces latency while maintaining actionable utility in response quality.

N Generalization to Other Use Cases

Extension to multi-token LLM inference. Although our method is evaluated in the context of vision-language models, the core idea—hardware-aware structured sparsification—extends beyond this domain. In particular, it is well-suited for multi-token LLM inference scenarios such as speculative decoding, parallel sampling for reasoning, and batched inference in interactive applications. In such settings, activations from multiple tokens are aggregated, leading to smoother neuron-importance distributions similar to those observed in VLMs. [5]

These multi-token inference workloads also underpin latency-critical user-facing systems such as chat assistants, copilots, and dialogue agents, which must maintain responsiveness under real-time constraints while often serving multiple concurrent requests. Minimizing end-to-end latency in these deployments is therefore essential, and extending our framework to such applications may offer broader benefits in practical LLM serving environments.

Extension to plain LLM / ViT inference. Our method can also be directly applied to plain LLMs and ViT-based models. The system relies on two key conditions: (i) the model exhibits smooth activation magnitude distributions (e.g., due to non-ReLU activations or multi-token inputs), and (ii) the hardware–model pair operates below I/O saturation when loading a single weight row. Recent LLMs increasingly employ smooth activation functions such as SwiGLU or GeLU, yielding moderate sparsity levels. While their activations are typically less smooth than those of multi-token scenarios, our method remains applicable, albeit with slightly smaller gains.

For ViT models, the transformer backbone remains largely compatible with our framework. As long as activation sparsity exists to a measurable degree, our approach can be applied without modification. Although ViTs are generally smaller (hundreds of millions of parameters), they still

[5] The sparsity mask generated from aggregated activations is shared across tokens, ensuring uniform inference latency across them (e.g., all samples in a batch finish simultaneously).

face I/O bottlenecks on resource-limited devices. In such cases, our chunking method becomes particularly valuable, as smaller weight channels make access fragmentation proportionally more severe.

To assess applicability to plain LLMs, we conducted a preliminary experiment on LLaMA3-8B and Qwen2-7B using the GSM8k dataset. We used the sum of selected neuron importance as a proxy for accuracy rather than full-dataset evaluation. We measured the importance–latency tradeoff in the first, middle, and last layers, observing average speedups of $1.22\times$ and $2.09\times$ for LLaMA3-8B and Qwen2-7B, respectively. These initial results suggest that our method generalizes to LLMs, though further work is needed to validate accuracy–latency tradeoffs at scale.