FluxShard: Distributed Motion-Aware Cache Remapping for Real-Time Video Analytics

IEEE Publication Technology Department

***Abstract*—Edge-cloud video analytics caches intermediate features across consecutive frames to cut redundant computation and transmission. Existing methods operate at whole-scene granularity—reusing or invalidating entire feature maps—and assume a roughly stationary scene. Even moderate motion misaligns the cache with the current frame, triggering widespread invalidation although most content has merely shifted rather than truly changed. The root cause is a *granularity mismatch*: the cache is treated as an indivisible unit, yet real-world motion is local and heterogeneous.**

We present FluxShard, a motion-aware edge-cloud analytics system that elevates codec-level motion vectors (MVs) into a first-class control signal for feature caching. FluxShard decomposes the cached feature map into block-level shards that flow with observed motion, realigning reusable content and isolating genuinely changed regions as a minimal recomputation set. A *Receptive-Field Alignment Principle* guarantees bit-equivalent correctness of MV-guided reuse across convolutional layers, and a *profiling-driven truncation policy* sustains high sparsity under growing motion with negligible accuracy loss. At runtime, FluxShard adaptively routes the sparse residual workload between edge and cloud based on network conditions and device load. Evaluation on real-world dynamic video sequences spanning object detection and instance segmentation shows that FluxShard achieves up to 92% bandwidth reduction, 5.5× speedup, and ∼99% accuracy retention over state-of-the-art baselines.

***Index Terms*—Video analytics, robotics, edge-cloud collaborative system, CNN inference acceleration**

I. Introduction

Video analytics underpins a wide range of latency-critical applications such as embodied intelligence [27], [8], aerial drones [1], [20], augmented reality [28], and smart surveillance [12], where timely and accurate understanding of high-rate video streams is essential. Processing solely on edge devices eliminates transmission latency but is constrained by limited computational and energy resources; offloading to the cloud offers abundant compute but introduces significant network delay. A natural way to mitigate both costs is to *reuse* previously computed features across consecutive frames in a cache-like manner: since adjacent frames share substantial visual content, retaining unchanged results avoids redundant computation and transmission, reducing latency and bandwidth simultaneously.

However, existing reuse methods operate at the granularity of the *whole scene*, making them fundamentally brittle in dynamic environments. Pipeline-based methods such as SPINN [17] and COACH [6] cache entire intermediate feature maps and reuse them when frames are globally similar. Delta-based methods including CBinfer [2], Diffy [19], and DeltaCNN [23] maintain a scene-level reference cache and update it with pixel-wise differences. MotionDeltaCNN [22] extends this with a global homography to shift the entire cache. Because these approaches assume a roughly stationary scene, any camera ego-motion, object movement, or viewpoint change triggers widespread cache invalidation even when most content has merely *shifted*, leading to frequent cache misses that diminish the efficiency gains intended by feature reuse.

The root cause is a *granularity mismatch*: existing methods treat the cache as an indivisible unit, yet motion in real-world mobile video is inherently local and heterogeneous. We observe that motion vectors (MVs), long used in video coding, offer a finer-grained alternative: they capture block-level displacement between consecutive frames and thus can separate spatial shift from true content change. By re-indexing cached features according to these displacements, we can recover reusable content that whole-scene methods would discard, eliminating false misses. Meanwhile, MVs are inherently approximate—they cannot model non-rigid deformations, newly exposed regions, or complex occlusions—so recomputation in those areas remains necessary. Thus we exploit this complementarity: MVs serve as a lightweight, model-agnostic signal that maximizes reuse where displacement is the dominant factor, while focusing computation on regions where content has genuinely changed.

We present ***FluxShard***, a motion-aware edge-cloud video analytics system that achieves low-latency, high-accuracy inference under significant motion. Treating the feature cache as block-level shards that flow freely with motion vectors, FluxShard uses MVs as a first-class control signal to align cached features with the current frame and isolate the minimal regions requiring fresh computation. By quantifying this sparse residual workload against real-time network conditions and device load, FluxShard dynamically selects the most efficient execution path: local sparse updates on the edge, or selective offloading of only non-reusable regions to the cloud. This co-optimization of transmission and computation keeps both bandwidth and latency low, even when large portions of the scene are in motion.

Realizing this design raises two key challenges:

C1: How to guarantee that MV-guided reuse does not degrade accuracy. Even when a layer's input features are correctly aligned by warping cached shards with MVs, directly warping its *output* the same way is not always valid if the layer aggregates over a spatial neighborhood (e.g., convolution, pooling): motion can assemble a neighborhood in the current frame that differs from the one used to produce the cached output. Naïvely detecting such mismatches requires comparing

the assembled neighborhood against the cached one at every output position, incurring prohibitive overhead. We instead derive a **Receptive Field Alignment Principle** that translates the per-layer neighborhood check into a single, lightweight condition on the input-level MV field, enabling exact reuse decisions with negligible cost.

C2: How to keep sparsity high as motion intensity increases. Even after MV-guided alignment, the sparse set of regions marked for recomputation tends to expand—or *bleed*—through cascaded convolutional layers, progressively diminishing efficiency. Aggressive truncation preserves speed but risks accuracy; unchecked expansion negates the benefit of selective recomputation. We resolve this with a **profiling-driven truncation policy**: offline, we characterize the tightest accuracy-preserving truncation thresholds across a range of motion statistics; at runtime, the system indexes into this mapping based on observed motion, dynamically modulating truncation to sustain high sparsity with negligible accuracy loss.

We implement FluxShard by developing a high-performance suite of motion-aligned sparse CNN operators, composed of custom CUDA sparse-convolution kernels and Triton kernels jointly optimized for the memory access patterns of MV-guided cached feature reuse. We evaluate FluxShard on object detection and instance segmentation with YOLOv11 [16] at high resolution, using two real-world dynamic video benchmarks—DAVIS [24] and 3DPW [26] which exhibit substantial camera ego-motion, object movement, and occlusion. Baselines include Naive Offloading, COACH [6], DeltaCNN [23], and MotionDeltaCNN [22], tested on NVIDIA Jetson Xavier NX (edge) and RTX 3080 (cloud). Across all tasks and datasets, FluxShard achieves:

- **Bandwidth reduction** — up to **92%** savings over full-frame transmission.
- **Energy efficiency** — up to **xx%** reduction in edge energy consumption.
- **End-to-end acceleration** — up to **5.5**× speedup over the strongest baseline.
- **Accuracy retention** — preserves up to **99%** of peak accuracy.
- **Scalability** — maintains ∼xx% lower latency growth when scaling to multiple concurrent edge devices.

In summary, this paper makes the following contributions. We identify the granularity mismatch as the key limitation of existing cache-based video analytics and show that motion vectors offer the right abstraction to bridge this gap. We propose FluxShard, a motion-aware edge-cloud video analytics system that leverages motion vectors for fine-grained cache alignment and adaptive workload routing between edge and cloud. We establish a Receptive-Field Alignment Principle that guarantees bit-equivalent correctness of MV-guided feature reuse across convolutional layers, and a profiling-driven truncation policy that sustains high sparsity under varying motion intensity with negligible accuracy loss. We implement these ideas in a unified operator library and demonstrate substantial reductions in bandwidth, energy, and latency while retaining near-peak accuracy on object detection and instance segmentation. More broadly, by enabling high-fidelity DNN perception at a fraction of the original cost, FluxShard lays a foundational building block for pervasive edge intelligence, making continuous, always-on visual understanding practical on the resource-constrained devices that will define the next generation of intelligent infrastructure.

II. Background

A. Edge-Cloud Video Analytics

Edge-cloud video analytics systems distribute DNN inference between a resource-constrained edge device and a cloud server with powerful GPUs. Neither extreme alone is satisfactory: executing a full model such as YOLOv11 [16] locally on an embedded GPU (e.g., NVIDIA Jetson Xavier NX) takes ∼xx ms per frame at 1080p, exceeding real-time budgets; offloading every frame to the cloud eliminates the compute bottleneck but introduces a transmission one—a single 1920×1080 RGB frame is ∼6 MB raw, and even JPEG-compressed streams at 30 fps sustain tens of Mbps, frequently exceeding typical edge uplink capacity. Practical systems therefore seek a middle ground, dynamically deciding whether to compute locally or offload based on current network conditions and workload.

Some prior edge-cloud video analytics systems place the scheduling boundary at an intermediate DNN layer, transmitting a mid-network feature map instead of the raw input [15], [17], [6]. This can be effective for classification models that possess a compact bottleneck, but detection and segmentation architectures (e.g., YOLO [16], Mask R-CNN [10]) maintain high-resolution, high-channel feature maps throughout, making mid-layer splitting impractical for these tasks. FluxShard, along with all baselines evaluated in this work instead schedule at the *input level*: either the edge or the cloud performs full-model inference. This input-level design offers these practical advantages: uint8 pixels of input are more compact than float32 intermediate and are easier to compress, and it makes the design of FluxShard *model-agnostic*.

B. Feature Cache Reuse in Video Inference

Because consecutive video frames share substantial visual content, a growing body of work caches previously computed features and reuses them for subsequent frames, reducing both computation and transmission. Existing approaches fall into two broad categories.

Pipeline-level caching. Methods such as SPINN [17] and COACH [6] cache intermediate feature maps or label-level predictions and reuse them when successive inputs are deemed globally similar. COACH, for example, maintains semantic cluster centers in the feature space and triggers an early exit—bypassing cloud inference entirely when a new frame's embedding falls within a similarity threshold. These methods make a *binary, whole-input* reuse decision: either the entire cached result is reused or it is fully recomputed. This coarse granularity suits image classification, where a single label summarizes the scene, but cannot approximate the spatially dense outputs required by segmentation or detection.

Delta-based sparse inference. A second line of work including RRM [21], CBinfer [2], Skip-Convolution [7], and DeltaCNN [23] maintains a pixel-level reference cache and propagates only the *difference* (delta) between the current frame and the reference through the network, computing only at affected output locations and reusing cached values elsewhere. Among these, DeltaCNN represents the most complete realization: it achieves end-to-end sparse propagation with truncation buffers that prevent error accumulation, enabling unbounded-length sequences without dense resets and pushing the efficiency of static-camera settings to its practical limit. MotionDeltaCNN [22] extends DeltaCNN to moving cameras by warping the cache with a single global homography before computing the delta. However, a global transformation cannot capture locally heterogeneous motion—independently moving objects, depth-induced parallax, or mixed ego-motion and object motion—leaving large residual deltas that erode sparsity.

Shared limitation. Across both categories, the cache is treated as an *indivisible, scene-level entity*: it is either globally valid, globally stale, or aligned uniformly using a single transformation. Real-world mobile video, however, exhibits motion that is local and heterogeneous: a camera pan shifts the background uniformly while foreground objects move independently, and depth discontinuities create parallax that no single warp can reconcile. MotionDeltaCNN [22] reported that their homography alignment succeeds on only a small fraction of the DAVIS [24] sequences (14 out of 80) evaluated. The result is a *granularity mismatch*: the cache granularity is the whole scene, but motion granularity is per-region. This mismatch forces existing methods to either waste computation re-deriving content that has merely shifted, or sacrifice correctness by reusing misaligned features.

Note that several other techniques also reduce the cost of video analytics, including ROI filtering [3], [18], input resolution adaptation [13], [5], frame sampling that skips inference on selected frames [3], [18], and model compression via quantization or pruning [11], [9]. These strategies are orthogonal to feature cache reuse: they govern *which* frames, regions, or model to run, whereas caching determines whether to recompute or reuse at each spatial location within a given inference. FluxShard can be composed with any of them; this work focuses exclusively on the cache reuse axis.

C. Motion Vectors as a Free Signal

Motion vectors (MVs), a staple of block-based video encoding, provide exactly the per-region displacement information that whole-scene methods lack. In H.264/H.265, the encoder partitions each frame into macroblocks (typically 16×16 or smaller) and, for each block, searches the reference frame for the best-matching patch; the resulting displacement (d_x, d_y) is the block's motion vector. Because MV estimation is an integral part of the encoding pipeline already running on the edge camera, these vectors are available at *zero additional computational cost*.

What MVs can do. A per-block MV field captures spatially heterogeneous motion: background blocks share a common displacement reflecting camera ego-motion, while foreground object blocks carry independent vectors. By re-indexing cached features according to per-block MVs, we can recover content that has merely shifted, thereby eliminating false cache misses.

What MVs cannot do. MVs model translational displacement only. They cannot represent non-rigid deformation, newly exposed regions due to dis-occlusion, or content appearing for the first time. These phenomena produce genuine residuals that re-indexing cannot resolve; they must be recomputed from fresh input. Crucially, however, these *true residuals* are typically a small fraction of the frame, concentrated at motion boundaries and newly revealed areas.

Distinction from video encoding. Standard video codecs also exploit MVs, but with a fundamentally different objective. After motion compensation, the codec encodes the *entire* residual frame via DCT, quantization, and entropy coding, targeting smaller volume and higher perceptual quality metrics (PSNR, SSIM). Every pixel of every frame enters the bitstream, albeit at varying precision. FluxShard, by contrast, does not aim to compress and transmit a complete frame. Its goal is to maximize *feature cache reuse* and minimize the volume of data that must be transmitted or recomputed for AI inference accuracy. Regions whose MV-aligned residual falls below a task-driven threshold are reused outright and never enter the transmission path; only the compact set of genuinely changed regions is processed. The trade-off axis is therefore not rate versus distortion, but *reuse coverage versus inference accuracy*—an objective for which standard codec pipelines are neither designed nor optimized.

D. Receptive Fields and the Correctness Challenge (C1)

The preceding sections establish that per-block MVs can, in principle, re-index cached features to recover shifted content. As a preliminary for our design, this section formalizes the layer-level reuse criterion and identifies the obstacle that per-block heterogeneous MVs pose to correct feature cache reuse.

Layer-local reuse criterion. Consider a layer l that computes an output feature map $\mathbf{O}^l \in \mathbb{R}^{H_l \times W_l \times C_l}$ from an input feature map $\mathbf{F}^l \in \mathbb{R}^{H_l' \times W_l' \times C_l'}$. Since our analysis concerns only spatial positions, we omit the channel dimension hereafter. The output at position (i, j) depends on a set of input positions called its *receptive field* $\mathcal{R}^l(i, j) \subseteq \mathbf{F}^l$. The feature cache stores the previous frame's output $\hat{\mathbf{O}}^l$; a motion vector maps (i, j) to a cached position $(\hat{\imath}, \hat{\jmath})$ whose value was produced from a corresponding receptive field in the previous input $\hat{\mathbf{F}}^l$. Reuse at (i, j) is valid when the output discrepancy stays within a task-driven tolerance:

$$\left|\mathbf{O}^l(i,j) - \hat{\mathbf{O}}^l(\hat{\imath},\hat{\jmath})\right| \;\le\; \tau. \tag{1}$$

This condition is universal across layer types.

For linear layers (convolution, addition, etc.), let $\mathbf{w}^l$ collect the coefficients applied over $\mathcal{R}^l(i, j)$; by linearity the output error satisfies

$$\left|\mathbf{O}^l(i,j) - \hat{\mathbf{O}}^l(\hat{\imath},\hat{\jmath})\right| \;\le\; \|\mathbf{w}^l\|_1 \cdot \max_{(p,q)\,\in\,\mathcal{R}^l(i,j)} \left|\mathbf{F}^l(p,q) - \hat{\mathbf{F}}^l(\hat{p},\hat{q})\right|, \tag{2}$$

where $(\hat{p}, \hat{q})$ is the MV-mapped counterpart of (p, q). Thus (1) is guaranteed whenever the input patch satisfies

$$\max_{(p,q) \in \mathcal{R}^l(i,j)} \left| \mathbf{F}^l(p,q) - \hat{\mathbf{F}}^l(\hat{p},\hat{q}) \right| \;\leq\; \frac{\tau}{\|\mathbf{w}^l\|_1}. \tag{3}$$

Layers with larger aggregate weight magnitudes demand tighter input agreement; element-wise additions, whose effective weights are unity, preserve the tolerance unchanged. For nonlinear activations such as ReLU or sigmoid that are 1-Lipschitz, Eq. (1) at the input side directly implies the same bound at the output, so no additional margin is needed. In practice, the check in Eq. (3) is the one executed at runtime for each linear layer, while Eq. (1) serves as the invariant maintained across the entire layer stack. Crucially, every evaluation uses only layer l's own input and cached output; it never recurses back to the model input.

When can reuse decisions be shared across layers? Amortizing the reuse decision across layers eliminates the per-layer input check in Eq. (3), which can itself be costly. When $|\mathcal{R}^l| = 1$ (e.g., 1×1 convolution, pointwise activation), such sharing is automatic: reused positions carry zero difference into the next layer and satisfy any threshold trivially, while recomputed positions carry nonzero difference and cannot pass the tighter threshold $\tau/\|\mathbf{w}^{l+1}\|_1 \leq \tau$. The reuse map is therefore inherited by all subsequent $|\mathcal{R}|{=}1$ layers at the same resolution with same MVs without re-evaluation.

Once $|\mathcal{R}^l| > 1$ (e.g., 3×3 convolution), sharing breaks down. Each output now aggregates a spatial neighborhood whose elements may have been relocated from arbitrary, disjoint positions by heterogeneous motion vectors. Even if every individual input is reused with zero difference, the spatial composition within the receptive field differs from the one that produced the cached reference output; the difference in Eq. (3) therefore no longer measures true content change, invalidating the reuse criterion. Moreover, any layer that alters spatial resolution (e.g., downsampling) changes the MV-to-position mapping, invalidating reuse decisions from earlier layers. A straightforward approach to maintaining correctness therefore requires an independent check at every layer with $|\mathcal{R}^l| > 1$ and every output position, a cost that can rival the computation it seeks to avoid.

Delta methods as a degenerate but tractable special case. Delta methods [23], [22] sidestep this limitation by restricting every block to a single shared MV: zero for a static camera, or a global constant after homography alignment. Under this uniformity, the MV-mapped correspondence between the current and cached receptive field patches reduces to the same fixed displacement at every output position. If every position within a layer-l receptive field was itself reusable at layer $l{-}1$, the two patches in Eq. (3) are identical and the output position is reusable as well. The reuse map therefore only needs to be seeded at the input, where it is simply the set of unchanged pixels, and propagated through subsequent layers via receptive field inclusion.

This simplicity comes at a cost: a single global MV cannot accommodate heterogeneous motion, so positions whose receptive fields span regions with different true displacements inevitably violate Eq. (3), enlarging the residual set and eroding the sparsity that delta methods rely on.

The design challenge (C1). FluxShard operates in the general regime of per-block heterogeneous MVs, where the tractability shortcut of delta methods does not apply. §**??** derives the *Receptive Field Alignment Principle*: a set of sufficient conditions, evaluated entirely on the input-level MV field, under which a layer-l output position can be certified as reusable without inspecting the neighborhoods actually assembled by warping. These conditions collapse the per-layer, per-position neighborhood check into a single resolution-aware pass over the MV field, restoring tractability while preserving correctness under arbitrary per-block motion.

III. System Design

A. System Overview

Figure **??** illustrates the FluxShard architecture. The system comprises an edge node that receives the video stream and a cloud server that provides additional compute capacity. Both endpoints maintain an inference engine and a synchronized feature cache; the edge node additionally hosts the dispatch logic that determines, per frame, which endpoint should execute inference.

a) Per-frame pipeline.: When an encoded frame arrives at the edge, FluxShard proceeds in four stages.

(1) MV extraction. Block-level motion vectors are extracted from the video codec as a free byproduct of decoding (§II-C), capturing per-region heterogeneous motion at no additional cost.

(2) Reusability evaluation. FluxShard maintains two *dispatch layers*, one per endpoint. Each dispatch layer keeps (i) an *input cache*: the input used in the last inference at its endpoint, and (ii) an *accumulated MV field* from that cached frame to the present. Upon receiving the extracted MVs, both dispatch layers incorporate them into their respective accumulated fields.

Each dispatch layer is an identity operator ($|\mathcal{R}|{=}1$, $\|\mathbf{w}\|_1{=}1$), so the reuse condition (Eq. 3) reduces to a per-block pixel comparison against the MV-aligned input cache. Blocks whose difference exceeds a motion-adaptive threshold τ (§**??**) form the *residual set*. From the residual set size, each dispatch layer estimates end-to-end latency: the edge layer accounts for local sparse inference; the server layer additionally includes transmission of the MV field and residual pixels under current network conditions.

(3) Dispatch decision. The frame is assigned to the endpoint with the lower estimated latency, implicitly adapting to motion complexity, network conditions, and endpoint load.

(4) Execution and cache update. The selected endpoint applies its accumulated MV field to rearrange cached blocks and fills in the residual pixels, producing the updated input; its input cache is replaced accordingly and its accumulated MV field reset. The residual set is then propagated through the network using the Receptive Field Alignment Principle (§**??**) and the truncation policy (§**??**): at each layer, positions identified for recomputation are evaluated and merged back into that layer's feature cache, while the remaining positions

are directly reused. The final output is reconstructed by fusing newly computed and cached feature positions, and all feature caches are updated in place.

The non-selected endpoint retains its accumulated field, letting its input cache span a longer interval for future reuse. When the server is selected, the edge transmits only the MV field and residual pixels; a lightweight *receive layer* on the server mirrors the dispatch layer logic to reconstruct the full input before inference.

Practical relaxation. The MV-inconsistent region—positions whose receptive field spans blocks with different motion vectors—is determined entirely by the spatial distribution of the MV field and does not grow across layers. However, its *relative* footprint increases sharply at deeper layers because feature map resolution decreases while the inconsistent region remains spatially fixed: a single block boundary that occupies a small fraction of the input can cover a substantial share of a low-resolution feature map. Under strict per-layer enforcement of the Receptive Field Alignment Principle, these positions are unconditionally marked for recomputation and fall outside the scope of the truncation policy, which can only reclaim positions whose delta falls below τ^l. The result is a large, non-reclaimable active set in deeper layers that dominates the computation budget.

We therefore enforce the alignment principle only at the first convolutional layer. At this layer, all inconsistent positions are recomputed exactly, eliminating the primary error from MV misalignment. In subsequent layers, these positions are treated identically to all others and governed solely by the truncation policy. This is justified by two observations. If a formerly inconsistent position produces a delta exceeding τ^l at a later layer, truncation recomputes it regardless—the relaxation has no effect. If its delta falls below τ^l, the position would have been recomputed unnecessarily under strict enforcement; skipping it introduces at most τ^l error at that layer, which is within the per-layer budget already allocated by the truncation policy. Formally, this breaks the bit-exact guarantee of Eq. **??**, but the total additional error is absorbed into the existing $\sum_l \tau^l$ budget rather than constituting an uncontrolled source. We verify empirically in §**??** that the relaxation degrades mAP by less than **XX**% while recovering **XX**% of the sparsity lost to strict enforcement.

IV. Implementation

We prototype *FluxShard* on Ubuntu 20.04 with Python and C++/CUDA. The core computational modules—including motion alignment, salience scoring, and miss classification—are packaged as a custom *PyTorch extension*. This fused design avoids unnecessary host-device transfers: motion search, cost aggregation, and block selection execute directly as CUDA kernels with shared buffers, and results are exposed as PyTorch tensors for seamless integration with subsequent model layers.

To maintain long-term coherence, we implement the *freshness updater* as a Python generator. All model operators are profiled into a callable sequence; the updater iterates via a yielded `for`-loop, allowing background refresh computations to pause and resume at operator boundaries. This ensures that opportunistic updates exploit only idle cycles without interfering with latency-critical inference. Complementarily, the updater also leverages residual bandwidth to gradually transmit non-critical blocks, aligning server-side caches at negligible cost.

Edge-cloud communication is realized via a minimal TCP runtime with block-structured serialization. Despite its simplicity, this lightweight design sustains real-time throughput over wireless links while keeping the codebase portable across heterogeneous devices.

V. Evaluation

A. Evaluation Setup

a) Testbed.: We evaluate FluxShard on a hybrid edge-cloud testbed comprising several NVIDIA Jetson Xavier NX devices (384 CUDA cores, 8 GB LPDDR4) and a cloud server with an NVIDIA RTX 3080 GPU (10 GB GDDR6X). Both run Ubuntu 20.04 with CUDA 11. Edge devices handle preprocessing and local inference, while the cloud provides additional compute resources for offloaded tasks. All devices connect via a 1000 Mbps Ethernet switch.

To simulate real-world LTE/5G networks, we apply bandwidth-limited traces from the Madrid LTE Dataset [25] via the Linux `tc` utility, enforcing bandwidth and latency constraints:

- High (130 Mbps): Optimal LTE/5G conditions.
- Medium (56 Mbps): Moderately loaded networks.
- Low (25 Mbps): Congested or degraded conditions.

b) Models and Datasets.: We evaluate FluxShard mainly on two most common video analytics deep tasks including object detection using YOLO11m [14] (referred to as Detect) and dense segmentation using YOLO11m-seg [14] (referred to as Segment) on two real-world datasets: 3DPW [26] and DAVIS [24]. We use the medium version of their family of models to balance between the accuracy and the latency on the edge. DAVIS is a video dataset featuring a large variety of moving object categories in the wild and diverse camera motion, while 3DPW hosts video sequences of human activities in cities captured from a mildly moving hand-held camera. We also include two extra tasks to demonstrate the further impact of the application of FluxShard on the other tasks: image classification using Vision Transformer [4] (referred to as Classify) and multi-person keypoint detection using Mask R-CNN [**?**] (referred to as Keypoint).

The statistics for Detect and Segment and their target datasets are shown in Table I.

Table I
STATISTICS OF THE EVALUATED TASKS AND MODELS.

Model	Detect	Segment
Dataset	3DPW	DAVIS
Resolution	1024×1024	1024×1024
Parameter (M)	20.1	22.4
Server Latency (ms)	17.56	22.23
Edge Latency (ms)	386.26	619.82
Edge Power (mW)	12696.7	15736.5

c) Baselines: We compare FluxShard against four baselines:

- Full Offload: Executes all inference on the server, incurring high transmission costs.
- COACH [6]: Pipeline-based method with early exit for computation reduction (by comparing the similarity of the current frame with the cached history frames) and quantization for bandwidth reduction.
- DeltaCNN [23]: Processes only pixel-level frame deltas.
- ROI: Vanilla ROI-based method. Processes only the regions of interest (ROI) of the frame, which is the bounding boxes of the detected objects by a small yolo11n-detect model [14] with only 2.6M parameters on the edge.

d) Metrics.: FluxShard's evaluation considers:

- Accuracy: Intersection over Union (IoU) for Detect and Segment, top-1 accuracy for Classify and mean Average Precision (mAP) for Keypoint, normalized by the accuracy of the larger version of the model (for the YOLO series of models) or the full model (for the others).
- Latency: Average end-to-end frame processing time on the critical path.
- Bandwidth Usage: Average transmission bandwidth (MBps).
- Cache Hit Rate: Cache reuse ratio under different bandwidth conditions and different scene motions; for COACH, it measures early-exit computation reuse and for ROI, it measures the transmission saved by only processing the regions of interest.
- Compute Load Distribution: Fraction of total computations on the critical path handled on edge and cloud.

B. End-to-End Results with a Single Edge Device

a) End-to-End Latency: Figure 1 plots the end-to-end latency across bandwidth levels, showing FluxShard as consistently the fastest method. In high-bandwidth Detect, FluxShard shortens execution by about $5.5\times$ compared to Full Offload, $2.4\times$ against COACH, $3.2\times$ against DeltaCNN, and nearly $5\times$ relative to ROI. For segmentation, the gains remain strong with $3.7\times$ over Full Offload and 1.8–$3.2\times$ over other baselines. At medium bandwidth, FluxShard maintains substantial advantages: $4.5\times$ faster than Full Offload in Detect, 2.9–$3.0\times$ over COACH and DeltaCNN, and $4.4\times$ relative to ROI. For segmentation, the margins are $3.1\times$ over Full Offload and 2.4–$2.8\times$ against COACH and DeltaCNN, with $2.6\times$ speedup over ROI. Even under low bandwidth—the most challenging regime—the improvements persist, with 3.4–$3.7\times$ advantages over Full Offload, roughly twofold over COACH and DeltaCNN, and up to $3.6\times$ relative to ROI. Note that the DAVIS dataset used in segmentation is more dynamic than 3DPW in Detect, limiting feature reuse and causing all reuse-based methods in Segment to show less acceleration over Full Offload than in Detect.

b) Accuracy: Table II further shows the accuracy trade-offs when using different schemes, with the output of the larger version of the model (YOLO11x and YOLO11x-seg [14]) as the ground truth reference. FluxShard maintains accuracy at a high level: for Segment it achieves 95.5–96.8%, corresponding to about 3–4.5% drop, while for Detect it stays between 98.5–99.0%, i.e., within 1–1.5% of the full model. DeltaCNN shows a nearly identical profile, retaining 96.9–97.1% for Segment and 99.0–99.2% for Detect. By contrast, COACH incurs a much larger degradation of around 11–15%, and ROI consistently remains about 6–7% lower than the full model. Overall, FluxShard sustains multi-fold latency gains while bounding accuracy loss within 1–4.5%, a clear improvement over prior baselines.

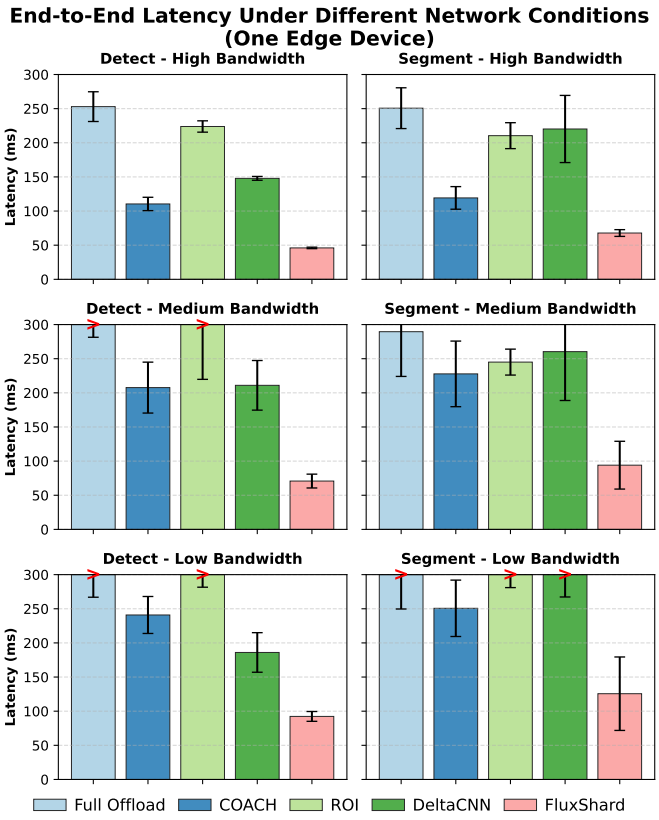

Figure 1. Latency comparison of different systems. FluxShard consistently outperforms the other baselines under different bandwidth conditions.

Table II
ACCURACY COMPARISON OF DIFFERENT SYSTEMS UNDER DIFFERENT BANDWIDTH CONDITIONS (%).

Bandwidth	Model	ROI	COACH	DeltaCNN	FluxShard
High	Segment	93.0	84.5	97.1	96.8
	Detect	94.2	87.2	99.2	99.0
Medium	Segment	93.0	85.2	97.0	96.3
	Detect	94.2	88.1	99.1	98.8
Low	Segment	93.0	86.0	96.9	95.5
	Detect	94.2	89.0	99.0	98.5

c) Cache Hit Ratio: Table III presents the cache hit ratios of different methods under varying bandwidth conditions for Segment and Detect. Higher cache hit ratio reflects that more false misses are excluded and more cached features are being reused by each system. COACH rely on whole-frame similarity for early exit, leading to low feature reuse. On dynamic datasets (e.g., DAVIS), their cache hit ratio remains as low as $\sim$1.4% for Segment, but improves to $\sim$25.6% for Detect when

Table III
CACHE HIT RATIO (%) COMPARISON AND AVERAGE BANDWIDTH CONSUMPTION (MB/S) UNDER DIFFERENT BANDWIDTH CONDITIONS. AVERAGE BANDWIDTH CONSUMPTION IS SHOWN IN PARENTHESES.

Bandwidth	Model	ROI	COACH	DeltaCNN	FluxShard
High	Segment	64.5 (10.05)	1.4 (10.85)	22.4 (6.57)	75.3 (2.05)
	Detect	53.5 (7.42)	25.6 (3.17)	52.8 (4.64)	91.1 (0.26)
Medium	Segment	64.5 (5.58)	1.4 (5.05)	23.1 (3.08)	73.8 (1.04)
	Detect	53.5 (3.38)	25.6 (1.26)	53.4 (2.34)	91.2 (0.24)
Low	Segment	64.5 (0.113)	1.4 (0.67)	23.7 (0.89)	72.5 (0.32)
	Detect	53.5 (0.020)	25.6 (0.092)	53.1 (0.26)	90.7 (0.23)

dealing with more stable camera scenarios. However, both methods maintain high bandwidth consumption, with COACH requiring up to 10.85 MB/s for Segment in high-bandwidth conditions.

DeltaCNN improves reuse efficiency by applying partial feature caching, achieving hit ratios of ∼23% (Segment) and ∼73% (Detect). Its bandwidth usage varies significantly across settings, consuming up to 16.57 MB/s in high-bandwidth scenarios but dropping to 4.89 MB/s in low-bandwidth conditions. ROI method only foucuses on the ROI of the frame, which varies accross different scenes and average to a hit ratio of ∼64.5% for Segment and ∼53.5% for Detect.

FluxShard achieves the highest hit ratio 75.3% for Segment and 91.1% for Detect by leveraging motion-vector-guided cache remapping that maintains high cache locality even under dynamic environments. Importantly, FluxShard maintains the lowest bandwidth consumption across all settings, requiring only 2.05 MB/s at high bandwidth and 0.32 MB/s at low bandwidth while sustaining peak cache efficiency. This demonstrates its ability to adaptively manage feature transmission for efficient bandwidth utilization and improved performance consistency.

The dominant reason for these consistent speedups is that using cache remapping, FluxShard minimizes redundant computation and transmission of costly *false misses* that other methods often treat as cache failures. Instead, only *true misses* are forwarded and fully recomputed, while the majority of aligned regions are served directly from cache. Moreover, activation-guided salience back-projection highlights only the blocks that truly matter for final predictions, ensuring that updates prioritize accuracy-critical regions without wasting bandwidth on irrelevant areas. Together, these mechanisms greatly reduce both compute and communication overhead, explaining why FluxShard sustains several-fold acceleration across network conditions while maintaining high accuracy.

C. Micro-benchmark

a) Cache Hit Ratio under Different scene motions:

b) Computation Time against Cache Hit Ratio:

c) Time Composition of Different Systems: Figure 2 shows the execution time breakdown of ROI, COACH, DeltaCNN, and FluxShard under medium bandwidth conditions for Segment and Detect. FluxShard effectively reduces both transmission and computation time through its motion-vector guided cache remapping and adaptive execution path

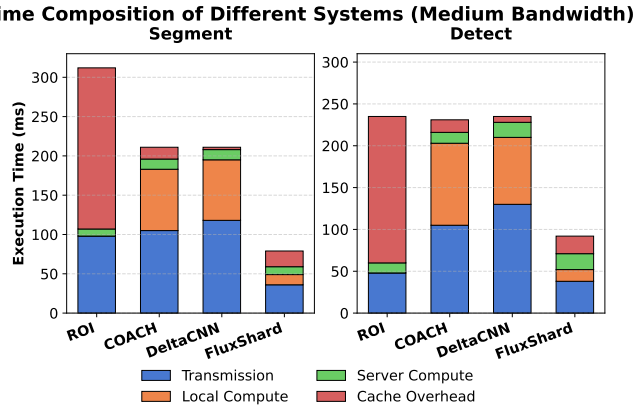

Figure 2. Execution time composition of COACH, ROI, DeltaCNN, and FluxShard under medium bandwidth conditions for Segment and Detect workloads.

scheduling. Note that while ROI method uses a tiny yolo11n-detect model to detect the region of interest, it still incurs high overhead since it would need to handle multiple regions of interest when the testing scene is complex. Instead of offloading region of interest like ROI, quantized frames like COACH, or frame deltas like DeltaCNN, FluxShard transmits only motion-triggered regions in a selective manner, significantly reducing transmission overhead. Additionally, by dynamically distributing computation between the edge and the cloud, it minimizes redundant local processing and reduces server-side inference cost. These optimizations allow FluxShard to achieve a lower overall execution time while maintaining high inference accuracy, making it well-suited for real-time video analytics in resource-constrained edge-cloud settings.

D. Scalability

We study scalability by varying the number of workers under the *medium* bandwidth setting, focusing on the Segment model as the heavier workload. Medium bandwidth is chosen because high bandwidth masks contention effects, while low bandwidth is already network-bound. Fig. 3 shows that latency increases with more workers due to contention on the shared link, but the growth rate differs sharply across methods. Full Offload grows from 290 ms (1 worker) to 798 ms (4 workers), a 2.8× rise. COACH increases more moderately, 1.9× (228 → 427 ms), while DeltaCNN rises 2.2× (260 → 582 ms). ROI also grows slowly (1.8×, 245 → 434 ms). FluxShard, by contrast, only increases from 94 ms to 159 ms, a 1.7× rise—the flattest curve, and starting from the lowest latency. At 4 workers, FluxShard remains 5.0× faster than Full Offload, 3.7× faster than DeltaCNN, and 2.7× faster than both COACH and ROI. These results confirm that FluxShard sustains both the lowest absolute latency and the slowest relative growth under bandwidth contention.

E. Sensitivity Analysis and Ablation Study

Table IV presents an ablation study under medium bandwidth with the Segment model. Removing salience-guided propagation notably hurts efficiency: end-to-end latency increases from 94 ms to 149 ms while cache hit rate drops from

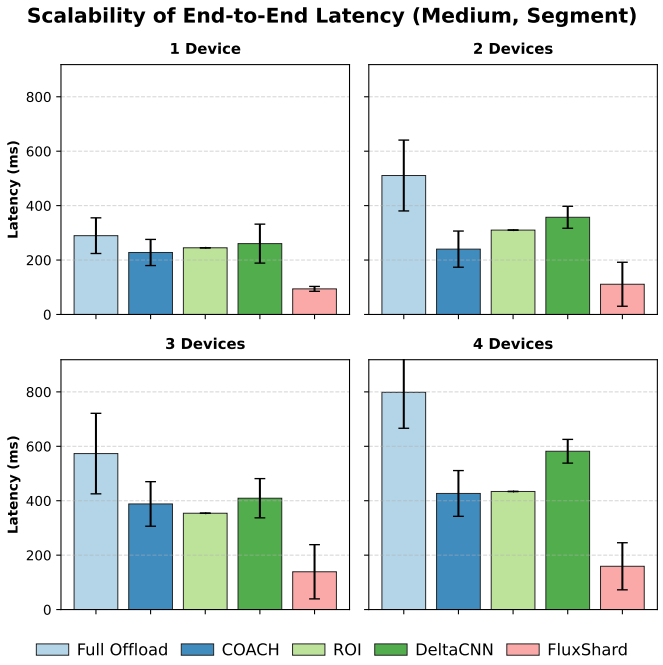

Figure 3. Scalability comparison of end-to-end latency in medium-bandwidth conditions for Segment. FluxShard scales more efficiently, maintaining lower latency growth as devices increase.

Table IV
ABLATION STUDY ON FLUXSHARD'S OPTIMIZATIONS IN MEDIUM BANDWIDTH (SEGMENT). WE TEST THE IMPACT OF REMOVING SALIENCE PROPAGATION (NO-SALIENCE) AND SPATIO SPARSE EXECUTING (DENSE RECOMPUTATION).

Method	E2E Latency (ms)	Cache Hit (%)	Accuracy (%)
Full-Scale FluxShard	94.40	73.8	96.3
No Salience	148.74	53.4	96.5
No-MS-Comp	120.65	73.8	96.3

74% to 53%, showing that accurate salience projection is essential for reusing the right regions. Disabling the motion-sensitive sparse execution yields a more moderate degradation (121 ms), as recomputation reduces efficiency but leaves hit rate and accuracy unaffected. Overall, both components contribute, with salience propagation being the dominant factor in sustaining high reuse and low latency.

F. Limitations and Future Work

FluxShard is mainly designed around continuous video inputs, where temporal redundancy enables effective reuse. Its efficiency may decrease in cases of rapid scene changes that lower cache hit ratios, and the use of cached features introduces some memory and computation overhead on resource-constrained devices. Future work will consider adaptive cache management and lightweight optimizations to improve robustness and efficiency across a wider range of scenarios.

VI. CONCLUSION

We have presented *FluxShard*, a motion-aware video analytics system that reframes feature reuse in dynamic environments as a *motion-induced cache-remapping* problem. By combining motion-vector-guided alignment with salience-driven prioritization and opportunistic freshness, FluxShard minimizes redundant recomputation while sustaining accuracy under camera and object motion. Our edge–cloud implementation demonstrates substantial bandwidth reduction, multi-fold acceleration, and strong scalability across diverse vision tasks, consistently outperforming state-of-the-art baselines. More broadly, FluxShard shows that lightweight motion signals, when paired with task-aware scheduling, provide an effective foundation for efficient and robust video analytics. We believe this perspective opens new opportunities for motion-aware reuse strategies, adaptive caching, and collaborative processing in future resource-constrained, real-time systems.

REFERENCES

[1] Hassan J. Al Dawasari, Muhammad Bilal, Muhammad Moinuddin, Kamran Arshad, and Khaled Assaleh. Deepvision: Enhanced drone detection and recognition in visible imagery through deep learning networks. *Sensors*, 23(21), 2023.

[2] Lukas Cavigelli, Philippe Degen, and Luca Benini. CBinfer: Change-based inference for convolutional neural networks on video data.

[3] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, SenSys '15, pages 155–168. Association for Computing Machinery.

[4] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. *CoRR*, abs/2010.11929, 2020.

[5] Kuntai Du, Qizheng Zhang, Anton Arapin, Haodong Wang, Zhengxu Xia, and Junchen Jiang. AccMPEG: Optimizing video encoding for video analytics.

[6] Luyao Gao, Jianchun Liu, Hongli Xu, Sun Xu, Qianpiao Ma, and Liusheng Huang. Accelerating end-cloud collaborative inference via near bubble-free pipeline optimization. *arXiv preprint arXiv:2501.12388*, 2024.

[7] Amirhossein Habibian, Davide Abati, Taco S. Cohen, and Babak Ehteshami Bejnordi. Skip-convolutions for efficient video processing. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2694–2703. ISSN: 2575-7075.

[8] Beining Han, Meenal Parakh, Derek Geng, Jack A. Defay, Gan Luyang, and Jia Deng. FetchBench: A simulation benchmark for robot fetching.

[9] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. In *Proceedings of the 29th International Conference on Neural Information Processing Systems - Volume 1*, volume 1 of *NIPS'15*, pages 1135–1143. MIT Press.

[10] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-CNN.

[11] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2704–2713. IEEE.

[12] Fatemeh Jalali, Morteza Khademi, Abbas Ebrahimi Moghadam, and Hadi Sadoghi Yazdi. Robust scene aware multi-object tracking for surveillance videos. 638:130114.

[13] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: scalable adaptation of video analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 253–266. Association for Computing Machinery.

[14] Glenn Jocher, Jing Qiu, and Ayush Chaurasia. Ultralytics YOLO, January 2023.

[15] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 615–629. Association for Computing Machinery.

[16] Rahima Khanam and Muhammad Hussain. Yolov11: An overview of the key architectural enhancements. *arXiv preprint arXiv:2410.17725*, 2024.
[17] Stefanos Laskaridis, Stylianos I Venieris, Mario Almeida, Ilias Leontiadis, and Nicholas D Lane. Spinn: synergistic progressive inference of neural networks over device and cloud. In *Proceedings of the 26th annual international conference on mobile computing and networking*, pages 1–15, 2020.
[18] Yuanqi Li, Arthi Padmanabhan, Pengzhan Zhao, Yufei Wang, Guoqing Harry Xu, and Ravi Netravali. Reducto: On-camera filtering for resource-efficient real-time video analytics. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '20, pages 359–376. Association for Computing Machinery.
[19] Mostafa Mahmoud, Kevin Siu, and Andreas Moshovos. Diffy: a déjà vu-free differential deep neural network accelerator. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, pages 134–147. IEEE Press.
[20] Kien Nguyen, Feng Liu, Clinton Fookes, Sridha Sridharan, Xiaoming Liu, and Arun Ross. Person recognition in aerial surveillance: A decade survey. pages 1–1.
[21] Bowen Pan, Wuwei Lin, Xiaolin Fang, Chaoqin Huang, Bolei Zhou, and Cewu Lu. Recurrent residual module for fast inference in videos. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1536–1545. ISSN: 2575-7075.
[22] Mathias Parger, Chengcheng Tang, Thomas Neff, Christopher D. Twigg, Cem Keskin, Robert Wang, and Markus Steinberger. MotionDeltaCNN: Sparse CNN inference of frame differences in moving camera videos with spherical buffers and padded convolutions. pages 17292–17301.
[23] Mathias Parger, Chengcheng Tang, Christopher D Twigg, Cem Keskin, Robert Wang, and Markus Steinberger. Deltacnn: End-to-end cnn inference of sparse frame differences in videos. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12497–12506, 2022.
[24] Jordi Pont-Tuset, Federico Perazzi, Sergi Caelles, Pablo Arbeláez, Alexander Sorkine-Hornung, and Luc Van Gool. The 2017 davis challenge on video object segmentation. *arXiv:1704.00675*, 2017.
[25] Pablo Fernández Pérez, Claudio Fiandrino, and Joerg Widmer. Characterizing and modeling mobile networks user traffic at millisecond level. In *Proceedings of the 17th ACM Workshop on Wireless Network Testbeds, Experimental evaluation & Characterization*, WiNTECH '23, pages 64–71. Association for Computing Machinery.
[26] Timo von Marcard, Roberto Henschel, Michael Black, Bodo Rosenhahn, and Gerard Pons-Moll. Recovering accurate 3d human pose in the wild using imus and a moving camera. In *European Conference on Computer Vision (ECCV)*, sep 2018.
[27] Qi Wu, Zipeng Fu, Xuxin Cheng, Xiaolong Wang, and Chelsea Finn. Helpful DoggyBot: Open-world object fetching using legged robots and vision-language models.
[28] Jun Zhang, Mina Henein, Robert Mahony, and Viorela Ila. VDO-SLAM: A visual dynamic object-aware SLAM system.

A. Biographies and Author Photos

```
\begin{IEEEbiographynophoto}{Jane Doe}
Biography text here without a photo.
\end{IEEEbiographynophoto}
```

or a biography with a photo

```
\begin{IEEEbiography}[{\includegraphics
[width=1in,height=1.25in,clip,
keepaspectratio]{fig1.png}}]
{IEEE Publications Technology Team}
In this paragraph you can place
your educational, professional background
and research and other interests.
\end{IEEEbiography}
```

Please see the end of this document to see the output of these coding examples.

Jane Doe Biography text here without a photo.

IEEE Publications Technology Team In this paragraph you can place your educational, professional background and research and other interests.