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$\times$ speedup, and $\sim$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 latency-critical applications such as embodied intelligence [26], [10], aerial drones [1], [19], augmented reality [27], and smart surveillance [13]. Processing solely on edge devices avoids transmission delay but is constrained by limited compute and energy; offloading to the cloud provides ample resources but introduces network latency. A natural middle ground is to *reuse* previously computed features across consecutive frames: since adjacent frames share substantial visual content, caching and reusing unchanged results reduces both redundant computation and transmission.

Existing reuse methods, however, operate at the granularity of the *whole scene* and are therefore brittle under motion. Pipeline-level approaches [16], [8] cache entire intermediate feature maps and reuse them only when successive frames are globally similar. Delta-based methods [2], [18], [22] maintain a scene-level reference and update it via pixel-wise differences; MotionDeltaCNN [21] extends this with a single global homography. Because all these approaches treat the cache as an indivisible entity, any camera ego-motion or object movement triggers widespread cache invalidation—even when most content has merely *shifted* rather than changed. The root cause is a *granularity mismatch*: the cache is monolithic, yet real-world motion is local and heterogeneous.

Block-level motion vectors (MVs) from standard video codecs offer a natural remedy. Extracted as a free byproduct of decoding, MVs capture per-region displacement between consecutive frames and can therefore distinguish spatial shift from genuine content change. Re-indexing cached features according to these displacements recovers reusable content that whole-scene methods would discard, while regions where MVs are inherently insufficient—non-rigid deformation, disocclusion, newly appearing content—are isolated as a minimal *residual set* that truly requires recomputation.

Realizing this idea, however, introduces two challenges. (1) the **receptive field inconsistency** problem: even when cached features are correctly aligned via MVs, directly reusing the aligned output of a layer that aggregates over a spatial neighborhood of input (i.e., receptive field) with size larger than one is not always valid, because heterogeneous motion can assemble a neighborhood that differs from the one used to produce the cached output, silently corrupting results. Naïvely detecting such mismatches requires checking the full receptive field at every output position of every layer, incurring prohibitive cost rivaling recomputation itself. (2) the **sparsity decay** problem:the residual set tends to expand through cascaded layers as layers with receptive field size larger than one causes recomputed positions to bleed into their neighbors, progressively eroding the efficiency advantage of selective recomputation as motion intensity grows.

We propose **FluxShard**, a motion-aware edge-cloud video analytics system that treats the feature cache as block-level *shards* flowing freely with motion vectors. FluxShard uses MVs as a first-class control signal to align cached features with the current frame, isolate the minimal residual set, and dynamically route work—local sparse update on the edge or selective offloading of only residual regions to the cloud—based on residual size, network conditions, and device load.

To address receptive field inconsistency, FluxShard introduces the **Receptive Field Alignment Principle (RFAP)** (§IV-B), which folds the per-layer, per-position neighborhood check into a single lightweight pass over the input-level MV field. RFAP produces a conservative reuse mask at each layer in time linear in the input resolution, guaranteeing bit-equivalent correctness without per-layer verification overhead.

To counter sparsity decay, FluxShard employs a **profiling-driven truncation policy** (§IV-C). Offline, we characterize the tightest accuracy-preserving truncation thresholds across a range of motion intensities; at runtime, the system indexes

Grouped bars: edge-only latency vs. cloud RTT
at uplink = 5, 10, 20, 50 Mbps

Figure 1. Per-frame latency under varying uplink bandwidth. Neither pure-edge nor pure-cloud meets real-time across all conditions.

Line plot: reuse ratio (%) vs. avg motion (px/frame)
for a whole-scene cache baseline

Figure 2. Cache reuse ratio versus per-frame motion. Whole-scene granularity causes reuse to collapse under moderate motion.

Bars: residual set size w/o vs. w/ MV alignment
across low / medium / high motion bins

Figure 3. MV alignment eliminates displacement-induced false misses, cutting the residual set by **XX–XX**%.

Bars: mAP for oracle / whole-scene reuse / naïve MV
reuse

Figure 4. Naïve MV-aligned reuse destroys accuracy due to receptive field inconsistency, even when the residual set is small.

into this mapping based on observed motion, dynamically modulating truncation to sustain high sparsity with negligible accuracy loss.

In summary, this paper makes the following contributions:

- We identify granularity mismatch as the key limitation of whole-scene cache reuse and show that codec-level motion vectors provide the right abstraction to overcome it.
- We propose FluxShard, a system that decomposes the feature cache into motion-aligned shards and co-optimizes sparse recomputation with adaptive edge-cloud workload routing.
- We establish the Receptive Field Alignment Principle for bit-equivalent MV-guided feature reuse across layers, and a profiling-driven truncation policy that sustains high sparsity under varying motion with bounded accuracy loss.
- We evaluate FluxShard on object detection and instance segmentation with YOLOv11 [15] on two dynamic video benchmarks (DAVIS [23], 3DPW [25]). Compared to the strongest baseline, FluxShard achieves up to $5.5\times$ end-to-end speedup and 92% bandwidth reduction while retaining over 99% of full-model accuracy.

II. Challenges and Motivation

A. The Edge-Cloud Dilemma

Edge-cloud video analytics must navigate two competing bottlenecks. Running YOLOv11m [15] on an NVIDIA Jetson Xavier NX takes **XX** ms per 1080p frame, far exceeding a 33 ms real-time budget. Offloading every frame removes the compute bottleneck but introduces a transmission one: JPEG-compressed 1080p at 30 fps sustains **XX** Mbps, routinely exceeding edge uplink capacity. Figure 1 quantifies this tension.

Feature cache reuse bridges this gap: when consecutive frames share content, cached results substitute for fresh computation. However, existing mechanisms [16], [8], [22], [21] treat the cache as an indivisible whole-scene entity—whether the decision is binary (reuse or recompute the entire frame) or pixel-level (propagate a dense difference map). This granularity breaks down under motion. Figure 2 shows that even modest displacement triggers widespread invalidation: the reuse ratio drops from **XX**% on near-static sequences to below **XX**% once motion exceeds **XX** px/frame, despite most content being merely shifted rather than truly changed.

Takeaway. Cache reuse is essential, but whole-scene granularity cannot tolerate motion. A finer-grained, motion-aware mechanism is needed.

B. Motion Vectors: Opportunity and Correctness Failure

H.264/H.265 codecs estimate a per-block displacement (d_x, d_y) as part of normal encoding, providing per-region motion information at *zero additional cost*. While recent studies [7], [6], [4] have shown that the codec's full reconstruction pipeline—transform, quantization, etc—distorts feature statistics in ways that degrade downstream inference accuracy, the motion vectors themselves are a pure geometric signal untouched by these lossy stages. By shifting cached feature blocks according to these MVs, displaced content is realigned before differencing, and only genuinely changed regions—disocclusion, deformation, new objects—remain in the *residual set*. Figure 3 confirms MV alignment reduces the residual set by **XX–XX**% across motion intensities: most whole-scene cache misses stem from displacement, not true change.

Despite the smaller residual set, directly reusing MV-aligned features produces catastrophic accuracy loss. Figure 4 shows mAP drops from **XX**% to **XX**% under naïve MV reuse. The root cause is *receptive field inconsistency*: layers with receptive field size > 1 aggregate spatial neighborhoods. When adjacent blocks carry different MVs, re-indexing assembles patches that were never contiguous in the original frame; the cached output was computed from a different neighborhood. The resulting error compounds through cascaded layers.

Takeaway. MV alignment eliminates most false cache misses, but exploiting it requires a correctness guarantee under heterogeneous per-block motion. This is the *first challenge* our design must address.

C. Sparsity Decays Through Cascaded Layers

Even with correct reuse, a second challenge erodes efficiency. Each convolutional layer with kernel radius r reads r positions beyond every active position, expanding the residual set by a margin of r per side per layer. Over L layers the expansion compounds: a single active input position inflates into a region of radius $\sum_l r_l$ at the final layer.

Figure 5 traces per-layer residual ratio through YOLOv11m. A modest input residual ratio of **XX**% grows past **XX**% by mid-network and saturates near 100% before the last layer.

Lines: per-layer residual ratio (%) vs. layer index curves for input residual ratio = 5, 10, 20, 30%

Figure 5. Receptive field bleed causes the residual set to expand at each layer, approaching full recomputation before the final layer.

At that point selective recomputation offers no advantage over full inference.

Takeaway. Without an explicit mechanism to arrest receptive field bleed, selective recomputation degenerates into full recomputation under real-world motion. A sparsity-preserving strategy with bounded accuracy loss is the *second challenge* our design must address.

III. System Overview

This section presents the FluxShard system. We first define the system model and optimization objective (§III-A), then describe the end-to-end pipeline that addresses the two challenges identified above (§III-B).

A. System Model

We consider an edge-cloud video analytics system in which an edge device (e.g., an NVIDIA Jetson) receives a live video stream from a co-located camera and a remote cloud server provides supplementary compute capacity. The two endpoints communicate over a bandwidth-limited uplink with round-trip latency that varies over time. Each endpoint hosts an identical copy of the inference model and maintains a per-layer feature cache that stores the output produced during its most recent inference. For each incoming frame, the system makes a *dispatch decision*: whether to run inference locally at the edge or offload it to the cloud.

a) Notation.: Let $\{I_t\}$ denote the incoming frame sequence. We write $\mathbf{m}_t$ for the block-level motion vector (MV) field extracted from the codec when decoding frame I_t; each entry maps a block position in I_t to its corresponding position in the reference frame.

The DNN $\mathcal{N}$ comprises L layers indexed by $l \in \{1, \ldots, L\}$. Layer l takes an input feature map $\mathbf{F}^l \in \mathbb{R}^{H_l' \times W_l' \times C_l'}$ and produces an output feature map $\mathbf{O}^l \in \mathbb{R}^{H_l \times W_l \times C_l}$; by convention $\mathbf{F}^1 = I_t$. Since our analysis concerns only spatial positions, we omit the channel dimension hereafter. The output of a layer at spatial position (i,j) depends on a set of input positions called its *receptive field*, denoted $\mathcal{R}^l(i,j) \subseteq \{1,\ldots,H_l'\} \times \{1,\ldots,W_l'\}$, with radius r_l. For convolutional and linear layers, $\mathbf{w}^l$ denotes the weight tensor applied over this receptive field.

Each endpoint caches the output of every layer from its most recent inference; we write $\hat{\mathbf{O}}^l$ for the cached output at layer l and $\hat{\mathbf{F}}^l$ for the corresponding cached input. Given the MV field, position (i,j) in the current frame maps to a cached position $(\hat{\imath},\hat{\jmath})$; likewise, each input position $(p,q) \in \mathcal{R}^l(i,j)$ maps to its cached counterpart $(\hat{p},\hat{q})$.

A *shard* is a block-sized unit of a feature map whose spatial extent corresponds to one entry in the MV field. At deeper layers with reduced spatial resolution, neighboring MVs that map to the same shard are aggregated and their magnitudes divided by the stride factor, so that each shard retains a one-to-one correspondence with a single displacement measured in feature-map coordinates.

b) Assumptions.: The inference result (e.g., detections, segmentation masks) is consumed at the executing endpoint or forwarded to a co-located downstream service. Downstream processing is typically lightweight (e.g., issuing an alert, logging an action label), and the processed outcome is negligible in size; we therefore treat result delivery as outside the latency-critical path and exclude it from our optimization scope.

c) Objective.: Let $d_t \in \{\text{edge}, \text{cloud}\}$ denote the dispatch decision for frame I_t. The per-frame end-to-end latency $T(d_t)$ comprises local computation, data transmission, and remote computation as applicable. Let $A(d_t)$ denote the task accuracy achieved under decision d_t and A^* the accuracy of full recomputation without caching. In general, latency reduction techniques trade off accuracy for speed, so $A(d_t)$ depends on the specific optimizations applied at the selected endpoint. Our objective is:

$$\min_{d_t} \; T(d_t) \quad \text{s.t.} \quad A(d_t) \;\geq\; \alpha\, A^*. \tag{1}$$

B. Pipeline

Figure 6 illustrates the end-to-end FluxShard pipeline. Each endpoint maintains a *dispatch layer*: a lightweight bookkeeping structure that stores (i) an *input cache*—the input used in the endpoint's most recent inference—and (ii) an *accumulated MV field* from that cached input to the present frame. The dispatch layer enables per-endpoint reusability estimation without running the model.

When an encoded frame arrives at the edge, FluxShard proceeds in four stages.

a) Stage 1: MV extraction.: Block-level motion vectors $\mathbf{m}_t$ are extracted from the video codec as a free byproduct of decoding, capturing per-region motion at no additional cost. Both dispatch layers incorporate $\mathbf{m}_t$ into their respective accumulated MV fields, denoted $\mathbf{d}_e$ and $\mathbf{d}_c$ for the edge and cloud endpoints. When an endpoint performs inference, its accumulated field is reset; the other endpoint's field continues to accumulate, spanning a longer interval for future reuse.

b) Stage 2: Reusability estimation.: Each dispatch layer uses its accumulated MV field to shift its cached input, aligning displaced content with the current frame. A per-block comparison between the aligned cache and the decoded frame identifies positions with non-negligible difference; these positions form the *residual set*. From the residual set size, the edge dispatch layer estimates local sparse inference latency, while the cloud dispatch layer estimates the cost of transmitting the MV field and residual pixels plus remote inference under current network conditions.

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

FluxShard end-to-end pipeline.

Left: edge device (camera $\rightarrow$ codec $\rightarrow$ MV extraction $\rightarrow$ dispatch layer).
Center: dispatch decision (edge local vs. cloud offload).
Right: selected endpoint (shard alignment $\rightarrow$ RFAP $\rightarrow$ truncation $\rightarrow$ sparse inference $\rightarrow$ cache update).

Arrows show MV field and residual pixels transmitted to cloud when offloading.

Figure 6. Overview of the FluxShard pipeline. Numbered stages correspond to the description in §III-B.

d) Stage 4: Inference and cache update.: The selected endpoint applies its accumulated MV field to rearrange cached shards across all layers and fills in residual pixels at the input, producing the updated input; its input cache is then replaced and its accumulated MV field reset. The residual set is propagated layer by layer through the model: at each layer, the Receptive Field Alignment Principle (§IV-B) identifies positions that can be safely reused despite heterogeneous motion, and a truncation policy (§IV-C) suppresses positions whose residual magnitude is sufficiently small. Only the remaining positions are recomputed; their outputs are merged into the layer's feature cache. The final result is produced by fusing reused and freshly computed features at the last layer.

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

IV. System Design

The pipeline in §III-B described the four stages that every frame traverses; this section develops the mechanisms deferred in that overview. We first formalize when a cached output of a layer can be directly reused after MV-guided shard alignment without accuracy loss (§IV-A); this criterion underpins all transmission and computation savings in FluxShard. We then derive the Receptive Field Alignment Principle (§IV-B), which makes the criterion efficiently evaluable under heterogeneous per-block motion by folding multi-layer consistency checks into a single pass over the input-level MV field.

Next, we introduce a profiling-driven truncation policy (§IV-C) that compensates for the sparsity decay caused by receptive field expansion across cascaded layers, maintaining high computational savings with bounded accuracy loss. Finally, we present the dispatch scheduler (§IV-D), which combines the reuse map, current network conditions, and endpoint load into a per-frame decision of whether to update residual regions locally at the edge or offload them to the cloud, minimizing end-to-end latency while satisfying the accuracy constraint in Eq. (1).

A. Layer-Local Reuse Criterion

Stage 4 of the pipeline (§III-B) propagates the residual set layer by layer, reusing cached outputs wherever possible. The quality of the final result hinges on a per-position decision: after MV-guided shard alignment, is the cached output close enough to the true output to be reused? This subsection formalizes that decision as a layer-local criterion and identifies the structural obstacle that heterogeneous MVs pose for its efficient evaluation.

a) Setup and notation.: Three quantities coexist at every output position (i, j) of layer l:

- $\mathbf{O}^l(i,j)$: the *ground-truth* output, obtained by applying layer l to the current input $\mathbf{F}^l$;
- $\hat{\mathbf{O}}^l(\hat{\imath},\hat{\jmath})$: the *cached* output from a previous frame, where $(\hat{\imath},\hat{\jmath}) = (i,j) - \mathbf{d}^l_{\mathrm{out}}(i,j)$ is the MV-aligned source position;
- $\tilde{\mathbf{O}}^l(i,j)$: the *assembled* output actually consumed by layer $l+1$.

b) Reuse validity.: Reuse at position (i, j) is *valid* when substituting the cached value for the ground truth introduces bounded error:

$$\left|\,\mathbf{O}^l(i,j) - \hat{\mathbf{O}}^l(\hat{\imath},\hat{\jmath})\right| \;\le\; \tau, \tag{2}$$

where τ is a task-driven tolerance; $\tau = 0$ recovers exact reuse. This condition is uniform across layer types and serves as the invariant maintained throughout the layer stack.

c) Layer output assembly.: Based on this criterion, FluxShard assembles the output of layer l as

$$\tilde{\mathbf{O}}^l(i,j) = \begin{cases} \hat{\mathbf{O}}^l(\hat{\imath},\hat{\jmath}), & \text{if Eq. (2) holds,} \\ \mathbf{O}^l(i,j), & \text{otherwise.} \end{cases} \tag{3}$$

The assembled output is thus a mosaic of *MV-aligned cached values* at reusable positions and *freshly computed values* at residual positions, and serves as the input to layer $l+1$. Maximizing the reusable set while respecting Eq. (2) is the objective of the remainder of this section.

d) Input-side sufficient condition.: Eq. (2) cannot be checked directly without computing $\mathbf{O}^l(i,j)$, which defeats the purpose of reuse. For a linear layer, linearity yields

$$\left|\,\mathbf{O}^l(i,j) - \hat{\mathbf{O}}^l(\hat{\imath},\hat{\jmath})\right| \;\le\; \|\mathbf{w}^l\|_1 \cdot \max_{(p,q)\,\in\,\mathcal{R}^l(i,j)} \left|\,\mathbf{F}^l(p,q) - \hat{\mathbf{F}}^l(\hat{p},\hat{q})\right|, \tag{4}$$

so Eq. (2) is guaranteed whenever the input patch satisfies

$$\max_{(p,q)\,\in\,\mathcal{R}^l(i,j)} \left|\,\mathbf{F}^l(p,q) - \hat{\mathbf{F}}^l(\hat{p},\hat{q})\right| \;\le\; \frac{\tau}{\|\mathbf{w}^l\|_1}. \tag{5}$$

For nonlinear activations that are 1-Lipschitz (ReLU, sigmoid), the input-side bound directly implies the output-side one, so no extra margin is needed. Crucially, Eq. (5) uses only layer l's own input and cached output; it never recurses to the model input.

This formulation avoids computing $\mathbf{O}^l$ itself, yet the check still inspects every position in the receptive field $\mathcal{R}^l(i,j)$, whose cost is comparable to executing the layer. To make the scheme practical, we next show how to propagate the reuse decision from a single pass over the *model input*, eliminating per-layer inspection entirely.

e) Residual set propagation.: We denote by $\mathcal{S}_l$ the *residual set* at layer l: the set of output positions assigned to the "otherwise" branch of Eq. (3), where fresh computation is required. The initial residual set $\mathcal{S}_0$ is determined at the model input by comparing each MV-aligned position against its cached counterpart; subsequent layers inherit and potentially enlarge this set depending on their structure.

For a pointwise layer ($|\mathcal{R}^l| = 1$), each output depends on exactly one input position. Reused positions carry zero input difference and satisfy Eq. (5) for any $\tau \geq 0$; residual positions carry nonzero difference that cannot be absorbed by the tighter per-layer bound $\tau/\|\mathbf{w}^l\|_1$. The residual set therefore passes through unchanged: $\mathcal{S}_l = \mathcal{S}_{l-1}$.

Once $|\mathcal{R}^l| > 1$ (e.g., 3×3 convolution, pooling), the residual set can grow through two distinct mechanisms:

(a) MV inconsistency. Even when every input position in the receptive field is individually reusable (zero difference), heterogeneous MVs can relocate those positions from *disjoint* cached regions, assembling a spatial composition that never existed in the cached frame. The resulting mismatch violates Eq. (5) silently: the input-side check sees zero per-position difference yet the layer output diverges from its cached counterpart. As shown in Fig. 4, ignoring this inconsistency causes accuracy to degrade sharply with motion magnitude, dropping over XX% under moderate motion.

(b) Receptive field bleed. Each output position reads a spatial neighborhood. If any input position within that neighborhood belongs to $\mathcal{S}_{l-1}$, the output position may violate Eq. (5) and must join $\mathcal{S}_l$. Repeated application across cascaded spatial layers causes $\mathcal{S}_l$ to expand progressively, eroding the sparsity that makes selective recomputation worthwhile. This expansion is the structural cause of the sparsity decay characterized in §II-C and occurs even under a uniform MV field.

Mechanism (a) is unique to heterogeneous per-block MVs. Delta methods [22], [9], [21] avoid it by restricting all blocks to a single shared MV, under which the spatial composition within every receptive field is guaranteed consistent; however, as shown in Fig. 3, this uniform-MV assumption collapses under heterogeneous motion. No prior work addresses the combination of per-block heterogeneous MVs with layer-wise feature cache reuse, where mechanism (a) is unavoidable. Naively, it can be resolved by checking MV consistency within every receptive field at every spatial aggregation layer, but the per-layer cost is comparable to recomputation itself, negating the benefit of reuse. §IV-B derives conditions that identify MV-inconsistent positions from a single pass over the input-level MV field, without restricting the MV field or inspecting every layer.

Mechanism (b) is inherent to sparse reuse in any convolutional pipeline. §IV-C controls it by filtering low-magnitude discrepancies via a profiling-driven threshold; the same threshold simultaneously absorbs residual discrepancies from mechanism (a) that fall below a task-dependent tolerance, keeping the overall residual set compact.

B. Receptive Field Alignment Principle

The preceding subsection showed that spatial aggregation layers invalidate free propagation of the reuse map: each output position aggregates a neighborhood whose elements may originate from disjoint cached regions under heterogeneous motion. Checking the reuse criterion at every such layer costs as much as recomputation. This subsection derives the *Receptive Field Alignment Principle* (RFAP), which identifies two structural conditions on the input-level MV field that, when jointly satisfied, guarantee exact reuse at all downstream layers without per-layer inspection.

1) Two Conditions for Correct Reuse: The conflict above arises because each layer l carries two independent MV fields—$\mathbf{d}^l_{\text{in}}$ on its input grid and $\mathbf{d}^l_{\text{out}}$ on its output grid—and heterogeneous motion can make them geometrically incompatible. RFAP imposes two conditions that are jointly sufficient for reuse at any output position (i,j); violating either one can produce silent errors that accumulate across layers.

Condition 1: Intra-receptive-field uniformity. All input positions within the receptive field must share a single displacement:

$$\mathbf{d}^l_{\text{in}}(p,q) = \mathbf{d}^l_{\text{in}}(p',q'), \quad \forall\,(p,q),\,(p',q') \in \mathcal{R}^l(i,j). \tag{6}$$

A uniform shift preserves the relative arrangement of values consumed by the layer. If violated, the aligned patch stitches together fragments from non-contiguous cached regions—an input composition that never existed in the cached frame.

Condition 2: Input-output geometric coherence. Let $\pi^l_{p,q}(i,j) = s^l \cdot (i,j) + (p,q)$ map output position (i,j) to the input coordinate of receptive field entry (p,q), where s^l is the stride of layer l. When projected into input coordinates, the output-level displacement must land on the same cached position as the input-level displacement at every receptive field entry:

$$\pi^l_{p,q}\big((i,j) + \mathbf{d}^l_{\text{out}}(i,j)\big) \;=\; \pi^l_{p,q}(i,j) \;+\; \mathbf{d}^l_{\text{in}}\big(\pi^l_{p,q}(i,j)\big), \qquad \forall\,(p,q) \in \mathcal{R}^l(i,j). \tag{7}$$

For the affine $\pi^l_{p,q}$ above, this simplifies to $\mathbf{d}^l_{\text{out}} = \mathbf{d}^l_{\text{in}}/s^l$, linking the two fields through the layer stride. If violated, the cached output and the aligned input patch refer to inconsistent cached content, as the two alignment paths diverge.

When both conditions hold, every input to the layer is identical between the current and cached evaluations. By determinism of the layer, $\mathbf{O}^l(i,j) = \hat{\mathbf{O}}^l(\hat{\imath},\hat{\jmath})$, satisfying Eq. (2) with $\tau = 0$, i.e., exact reuse. Conversely, violating either condition can cause silent corruption that accumulates across layers. The set of output positions that fail either condition at layer l forms the *MV inconsistency mask* $\mathcal{M}^l$; these positions must be recomputed.

Recall that the reuse criterion in §IV-A already identifies positions that must be recomputed due to content change (the "otherwise" branch in Eq. (3)). $\mathcal{M}^l$ captures an orthogonal source of invalidity: positions where content may be unchanged yet the spatial arrangement within the receptive field differs from the cached evaluation. The union of both sets constitutes the complete per-layer residual set that is recomputed in Stage 4 of the pipeline; all remaining positions satisfy Eq. (2) and are safely assembled from cache. RFAP's role is therefore to make the reuse criterion of §IV-A enforceable under heterogeneous MV fields without per-layer recomputation: it identifies, from the input-level MV field alone, the minimal additional invalidations needed to keep the assembled output correct.

a) Practical scope.: Enforcing both conditions at every spatial-aggregation layer would negate the sparsity gains that motivate reuse. The two conditions, however, differ sharply in their cost to sparsity. Condition 1 must be checked at every layer whose receptive field exceeds one; such layers typically constitute a large share of the backbone (e.g., XX% to YY% in ResNet-50, EfficientDet, and YOLOv11). Under strong motion, neighboring positions frequently carry inconsistent MVs, so strict enforcement at every such layer marks a large fraction of positions as inconsistent, collapsing sparsity toward full recomputation. Condition 2, by contrast, applies only at layers that change spatial resolution; these often number several in a typical backbone (e.g., 3–4 in ResNet-50, EfficientDet, and YOLOv11), so its impact on overall sparsity remains limited.

We therefore enforce Condition 1 only at the first layer whose receptive field exceeds one, recomputing all inconsistent positions so that their outputs reflect the true assembled input. Condition 2 is enforced at every resolution-changing layer. Beyond the first spatial-aggregation layer, unchecked Condition 1 violations may weaken the per-layer bound in Eq. (2), but the relationship between per-layer feature error and end-task accuracy is inherently indirect: the bound is an operational proxy, not a formal accuracy guarantee. The actual accuracy guarantee comes from the profiling-driven truncation policy (§IV-C), which selects τ by directly measuring end-task accuracy under all approximations—including relaxed Condition 1 enforcement—and ensures $A(\tau) \geq \alpha \cdot A^*$. We verify in §VI that this strategy keeps accuracy loss below **XX**% while recovering **XX**% of the sparsity lost to strict enforcement.

C. Profiling-Driven Truncation Policy

After the first spatial-aggregation layer, positions in the residual set arise from two distinct sources: those violating RFAP and those whose underlying content has genuinely changed. Both sources produce a spectrum of discrepancy magnitudes: high-magnitude entries that matter for accuracy and low-magnitude entries that can be safely discarded.

With $\tau = 0$, every nonzero left-hand side of Eq. (5) triggers recomputation, preserving bit-exact correctness but collapsing sparsity toward the full-recomputation regime observed in §II-C. Raising τ allows positions whose input-patch discrepancy is small to be retained from cache, trading a controlled approximation for substantially higher sparsity.

Because the check in Eq. (5) operates on discrepancy magnitude regardless of origin, a single τ filters both sources uniformly: positions with large discrepancy are recomputed while those with small discrepancy are reused. The per-layer bound in Eq. (2) may not hold strictly at layers where Condition 1 is relaxed, but the relationship between per-layer feature error and end-task accuracy is inherently indirect. Rather than relying on this bound, the truncation policy selects τ by directly measuring end-task accuracy under the full pipeline—including RFAP relaxation, truncation, and cache assembly—so that all sources of approximation are absorbed into a single, empirically validated knob.

Formulation. The truncation operator retains the cached value at position p in layer l whenever the left-hand side of Eq. (5) falls below $\tau/\|\mathbf{w}^l\|_1$, removing p from the residual set $\mathcal{S}_l$. Because Eq. (5) normalizes the raw input-patch discrepancy by the layer-specific factor $\|\mathbf{w}^l\|_1$, the threshold τ is a single global scalar shared across all layers. The optimization objective is:

$$\max_{\tau \geq 0} \ \tau \qquad \text{s.t.} \quad A(\tau) \ \geq \ \alpha \cdot A^*, \tag{8}$$

where $A(\tau)$ is end-task accuracy under threshold τ and A^* is the accuracy of full inference without any reuse.

Offline profiling. We solve (8) offline on a representative calibration set. We characterize each frame by its *motion intensity*, defined as the spatial standard deviation of the accumulated MV field:

$$m \ = \ \sqrt{\frac{1}{N}\sum_p \left\|\mathbf{d}(p) - \bar{\mathbf{d}}\right\|^2}, \tag{9}$$

where $\bar{\mathbf{d}}$ is the spatial mean of the field. This metric captures the *heterogeneity* rather than the absolute magnitude of motion: a uniform camera pan yields $m \approx 0$ despite large displacements, correctly reflecting that shard alignment can recover nearly all content; conversely, a mix of independently moving objects produces high m even if individual displacements are moderate, signaling that RFAP violations will be frequent.

For a discrete set of motion intensity levels, we sweep τ upward from zero and record the accuracy at each value. The output is a lookup table mapping each motion intensity level to the maximum feasible τ satisfying the accuracy constraint.

Empirically, the feasible τ increases with motion intensity: when motion is heterogeneous, discrepancy magnitudes are collectively elevated, so a higher threshold still retains sufficient signal. This trend is favorable because aggressive truncation becomes available precisely in the high-heterogeneity regime where the residual set is largest and sparsity recovery is most needed.

Runtime selection. At each frame, the dispatch layer computes m from the current accumulated MV field via Eq. (9) and looks up τ from the profiled table. At each layer l, the left-hand side of Eq. (5) is evaluated per position: any position whose value falls below $\tau/\|\mathbf{w}^l\|_1$ is removed from $\mathcal{R}_l$. The cost is one scalar lookup plus one element-wise comparison per layer, adding negligible overhead.

Following DeltaCNN [22], we restrict truncation to activation layers only. Because activations are the natural sparsification points in a DNN pipeline, applying the threshold there is sufficient to shrink the residual set; all other layers simply operate on whichever residual set they inherit. Furthermore, the threshold check is evaluated only over positions currently *in* the residual set: positions already marked reusable by RFAP or by alignment have zero discrepancy by construction and satisfy any $\tau > 0$ automatically.

Dispatch layer integration. The dispatch layer is an identity mapping prepended to the model, so Eq. (5) applies with $\|\mathbf{w}\|_1 = 1$: each MV-aligned input position is compared against its cached counterpart, and positions whose discrepancy falls below τ are excluded from the residual set. When inference is offloaded to the cloud, only the residual positions and the MV field are transmitted; the cloud reconstructs reusable positions from its own cache via the same alignment procedure and merges them with the received residual to assemble the complete input for subsequent layers.

D. Dispatch Scheduler

The preceding subsections determine *what* to reuse and *what* to recompute; this subsection decides *where* the recomputation takes place. For each frame, the scheduler makes a binary decision—edge local inference or cloud offload—to minimize end-to-end latency; the accuracy constraint is already enforced by the truncation policy.

a) Cost estimation.: The edge maintains two dispatch layers, one tracking the local inference state and one mirroring the cloud state (§III). For each incoming frame, both dispatch layers independently perform shard alignment and residual set computation using their respective cached inputs and accumulated MV fields. Each dispatch layer derives its own motion intensity m_e and m_c via Eq. (9) and looks up the corresponding truncation threshold τ_e and τ_c from the profiled table, yielding residual sets $\mathcal{S}_0^e$ and $\mathcal{S}_0^c$.

Estimating end-to-end sparse inference latency from $|\mathcal{S}_0|$ alone is unreliable: the input-layer residual count does not capture how sparsity evolves across layers, which depends on the magnitude distribution of residual activations and the truncation threshold. We observe empirically that motion intensity subsumes these factors—higher heterogeneity produces both larger and higher-magnitude residual sets—making it a reliable proxy for end-to-end latency. Accordingly, we profile sparse inference latency offline on each endpoint as a function of motion intensity (Eq. (9)) and store the result as a lookup table mapping each discretized motion intensity level to the measured latency. At runtime, edge latency is estimated as $T_{\text{edge}} \leftarrow \mathcal{T}_{\text{edge}}[m_e]$. Cloud latency combines the profiled inference cost with a transmission term: $T_{\text{cloud}} \leftarrow \mathcal{T}_{\text{cloud}}[m_c] + |\mathcal{S}_0^c|/\hat{B}$, where $\hat{B}$ is an exponentially weighted moving average of recent bandwidth measurements. The scheduler selects the endpoint with lower total latency; when the two estimates are within a margin ϵ, it prefers cloud offload to conserve edge energy.

Algorithm 1 summarizes the complete per-frame pipeline, integrating MV extraction, reusability estimation, dispatch, and cache maintenance.

Algorithm 1: FluxShard Per-Frame Pipeline

```
Input: Current frame $I_t$; edge dispatch layer ($\hat{I}_e$, $\mathbf{d}_e$);
       cloud dispatch layer ($\hat{I}_c$, $\mathbf{d}_c$); profiled tables
       $\mathcal{T}_\tau$, $\mathcal{T}_{\text{edge}}$, $\mathcal{T}_{\text{cloud}}$
Output: Inference result $\mathbf{y}_t$
Extract MV field $\mathbf{d}_t$ from codec for frame $I_t$;
$\mathbf{d}_e \leftarrow$ Accumulate($\mathbf{d}_e$, $\mathbf{d}_t$);
$\mathbf{d}_c \leftarrow$ Accumulate($\mathbf{d}_c$, $\mathbf{d}_t$);
$m_e \leftarrow$ MotionIntensity($\mathbf{d}_e$) ;        // Eq. (9)
$m_c \leftarrow$ MotionIntensity($\mathbf{d}_c$);
$\tau_e \leftarrow \mathcal{T}_\tau[m_e]$;   $\tau_c \leftarrow \mathcal{T}_\tau[m_c]$;
$\mathcal{S}_0^e \leftarrow$ ResidualSet($I_t$, $\hat{I}_e$, $\mathbf{d}_e$, $\tau_e$);
$\mathcal{S}_0^c \leftarrow$ ResidualSet($I_t$, $\hat{I}_c$, $\mathbf{d}_c$, $\tau_c$);
$T_{\text{edge}} \leftarrow \mathcal{T}_{\text{edge}}[m_e]$;
$T_{\text{cloud}} \leftarrow \mathcal{T}_{\text{cloud}}[m_c] + |\mathcal{S}_0^c| \,/\, \hat{B}$;
if $T_{\text{edge}} < T_{\text{cloud}} - \epsilon$ then
    $\mathbf{y}_t \leftarrow$ SparseInfer$_{\text{edge}}$($I_t$, $\mathcal{S}_0^e$, $\tau_e$, $\mathbf{d}_e$);
    Update $\hat{I}_e$;   reset $\mathbf{d}_e$;
else
    Transmit $\mathcal{S}_0^c$ and $\mathbf{d}_c$ to cloud;
    $\mathbf{y}_t \leftarrow$ SparseInfer$_{\text{cloud}}$($\mathcal{S}_0^c$, $\tau_c$, $\mathbf{d}_c$);
    Update $\hat{I}_c$;   reset $\mathbf{d}_c$;
return $\mathbf{y}_t$;
```

V. Implementation

We prototype FluxShard on NVIDIA Jetson Xavier NX (edge) and NVIDIA RTX 3080 GPU (cloud), connected over a TCP link whose bandwidth is shaped with `tc` to emulate cellular uplink conditions.

a) MV extraction.: In deployment, block-level MVs are available as a free byproduct of video codec decoding. Since the evaluation datasets used in this work provide individual decoded frames rather than encoded streams, we implement a lightweight block matching kernel in Triton that computes per-block (16×16) MVs by exhaustive search within a fixed search window. This kernel adds less than 0.5 ms per frame on the Jetson and produces MV fields equivalent in granularity to those exported by H.264/H.265 codecs; in a production pipeline, even this cost would be eliminated.

b) Sparse inference engine.: All feature maps are stored in channel-last (NHWC) layout, so that each spatial position occupies a contiguous memory segment. This choice has two benefits. First, MV-guided shard alignment reduces to pointer arithmetic along the spatial dimensions; the channel vector at each position remains contiguous regardless of the displacement, eliminating scattered memory accesses that a channel-first layout would incur. Second, skipping a spatial position avoids a contiguous block of computation rather than strided fragments, enabling efficient predication.

We fuse MV alignment with computation: each operator reads directly from the MV-redirected source position, so alignment requires no separate data movement pass. On this basis we implement two families of operators in Triton. Elementwise operators (ReLU, batch normalization, residual addition) simply bypass inactive positions via a mask check.

The convolution operator follows an implicit-GEMM structure with the same load/store pattern as its dense counterpart but skips arithmetic for inactive positions; cached values are preserved in place without memory movement.

On fully dense inputs, the convolution operator achieves approximately 80% of cuDNN throughput; as sparsity increases, execution time decreases near-proportionally, yielding wall-clock speedups that directly reflect the sparsity ratios reported in our evaluation.

c) Dispatch decision.: The dispatch layer maintains an input cache and an accumulated MV field per endpoint, both stored as contiguous GPU tensors. Per-frame dispatch requires only several table lookup indexed by motion intensity and a scalar comparison of the estimated edge and cloud latencies, completing in under 0.1 ms; its overhead is excluded from the latency measurements reported in §VI.

VI. Evaluation

A. Experimental Setup

a) Testbed.: The edge device is an NVIDIA Jetson Xavier NX (384 CUDA cores, 8 GB LPDDR4x) and the cloud server hosts an NVIDIA RTX 3080 GPU (10 GB GDDR6X). Both run Ubuntu 20.04 with CUDA 11. Devices connect via a 1000 Mbps Ethernet switch. To emulate realistic wireless conditions, we use the Linux `tc` utility to enforce bandwidth limits drawn from the Madrid LTE trace dataset [24] and add a fixed 30 ms round-trip delay. We define three bandwidth regimes that capture the range of practical deployments: *High* (130 Mbps, optimal LTE/5G), *Medium* (56 Mbps, moderately loaded), and *Low* (25 Mbps, congested or degraded). Unless otherwise noted, results are reported under the Medium regime.

b) Models and tasks.: We evaluate on two dense prediction tasks using the YOLO11 family [?]: (i) *instance segmentation* with YOLO11m-seg, and (ii) *pose estimation* with YOLO11m-pose. Both models accept 1920×1088 input (1080p padded to a multiple of 32) and are deployed without architectural modification; FluxShard and all baselines share identical model weights. Since the evaluation datasets lack task-specific ground truth at the required granularity, we generate *pseudo ground truth* by running the largest model variant (YOLO11x-seg and YOLO11x-pose, respectively) on every frame with full inference and no caching. All methods are evaluated against this pseudo GT, whose accuracy serves as A^* in our formulation.

c) Datasets.: We use three video benchmarks that span a range of camera and object motion characteristics. All experiments use the training split at 30 fps, as validation splits contain too few sequences for statistically meaningful evaluation; we employ official YOLO11 checkpoints trained on COCO and perform no fine-tuning on any of these datasets. Frames are resized to 1920×1088.
leftmargin=*,nosep

- *DAVIS 2017* [?]: 60 validation sequences captured with freely moving handheld cameras. Scenes contain a mixture of camera ego-motion, independently moving foreground objects, and frequent occlusion and dis-occlusion events.
- *3DPW* [25]: 24 outdoor validation sequences with GPS/IMU-annotated camera trajectories. Scenes feature walking pedestrians filmed from a moving camera, producing simultaneous ego-motion and multi-person motion with depth-induced parallax.
- *CMU Panoptic Studio* [?]: multi-view sequences captured by static dome cameras. Camera ego-motion is absent; all motion arises from human subjects. This dataset serves as a favorable baseline for delta-based methods and verifies that FluxShard does not regress under low-motion conditions.

d) MV extraction.: Because all three datasets are distributed as image sequences rather than encoded video, we cannot extract motion vectors from a codec bitstream. Instead, we implement a lightweight block matching kernel in Triton that estimates per-block motion vectors on 16×16 macroblocks between consecutive frames. The kernel runs on the edge GPU; its latency is included in all end-to-end measurements.

e) Baselines.: We compare against five alternatives spanning the design space:
leftmargin=*,nosep

- **Full Offload**: every frame is transmitted to the cloud at full resolution for complete inference.
- **Full Edge**: every frame is processed locally on the edge device with no cloud involvement.
- **COACH** [8]: the edge caches the 10 most recent input frames and their corresponding model outputs. For each new frame, SSIM is computed against all cached inputs; if the highest score exceeds 0.92, the matched cached output is reused directly.
- **DeltaCNN** [22]: pixel-level frame differencing identifies changed regions, and only those regions are processed through a sparse convolution pipeline.
- **MotionDeltaCNN** [21]: extends DeltaCNN with global homography compensation before computing pixel deltas.

To ensure a fair comparison, COACH, DeltaCNN, and MotionDeltaCNN are all augmented with the same dispatch logic described in §IV-D: at each frame the scheduler estimates edge-local latency (cache lookup for COACH, sparse inference for the other two) and cloud offload latency under the current bandwidth, then selects the faster endpoint. For brevity we refer to these baselines by their original names throughout the remainder of the paper.

f) Metrics.: We report the following metrics: leftmargin=*,nosep

- *End-to-end latency*: wall-clock time per frame, including MV extraction, reusability estimation, dispatch decision, data transmission (when offloading), model inference, and cache update.
- *Throughput*: frames per second sustained over each test sequence.
- *Accuracy*: mask mAP for segmentation, keypoint AP for pose estimation, both measured against pseudo ground truth.
- *Bandwidth*: bytes transmitted per frame (edge-to-cloud direction).

- *Sparsity ratio*: fraction of feature map positions skipped via reuse and truncation.
- *Energy*: per-frame energy consumption on the edge device, measured via the on-board power monitor accessed through `jtop` [**?**].

g) FluxShard configuration.: The accuracy preservation ratio is set to $\alpha = 0.98$. The offline profiling phase discretizes motion intensity m into 100 uniform bins over $[0, 15]$ and sweeps the truncation threshold τ over 100 uniform steps in $[0, 30]$, yielding a 100×100 lookup table $\mathcal{T}_\tau$. Edge and cloud sparse inference latency tables $\mathcal{T}_{\text{edge}}$ and $\mathcal{T}_{\text{cloud}}$ are profiled over the same motion intensity bins. Profiling uses 5% of the validation sequences selected by uniform sampling; total profiling time is under one hour on the cloud GPU per model. The dispatch tie-breaking margin is set to $\epsilon = 2\,\text{ms}$. The bandwidth estimator uses an EWMA with smoothing factor $\beta = 0.3$.

VII. Related Work

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 [16] and COACH [8] 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 [20], CBinfer [2], Skip-Convolution [9], and DeltaCNN [22] 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 [21] 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 [21] reported that their homography alignment succeeds on only a small fraction of the DAVIS [23] 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], [17], input resolution adaptation [14], [5], frame sampling that skips inference on selected frames [3], [17], and model compression via quantization or pruning [12], [11]. 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.

VIII. 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] Hyomin Choi and Ivan V. Bajić. Scalable image coding for humans and machines. 31:2739–2754.

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

[6] Lingyu Duan, Jiaying Liu, Wenhan Yang, Tiejun Huang, and Wen Gao. Video coding for machines: A paradigm of collaborative compression and intelligent analytics. 29:8680–8695.

[7] Leonardo Galteri, Lorenzo Seidenari, Marco Bertini, and Alberto Del Bimbo. Deep generative adversarial compression artifact removal. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 4836–4845. ISSN: 2380-7504.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

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

[14] 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.

[15] Rahima Khanam and Muhammad Hussain. Yolov11: An overview of the key architectural enhancements. *arXiv preprint arXiv:2410.17725*, 2024.

[16] 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.

[17] 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.

[18] 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.

[19] Kien Nguyen, Feng Liu, Clinton Fookes, Sridha Sridharan, Xiaoming Liu, and Arun Ross. Person recognition in aerial surveillance: A decade survey. pages 1–1.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] Qi Wu, Zipeng Fu, Xuxin Cheng, Xiaolong Wang, and Chelsea Finn. Helpful DoggyBot: Open-world object fetching using legged robots and vision-language models.

[27] 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.