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 [18], [5], aerial drones [1], [12], augmented reality [19], and smart surveillance [7], 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 [10] and COACH [4] cache entire intermediate feature maps and reuse them when frames are globally similar. Delta-based methods including CBinfer [2], Diffy [11], and DeltaCNN [14] maintain a scene-level reference cache and update it with pixel-wise differences. MotionDeltaCNN [13] 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. Spatially shifting cached features by motion vectors is valid for point-wise operations, but CNNs are dominated by convolutions whose output at each location depends on a spatial neighborhood. When parts of that neighborhood have undergone inconsistent motion, naive re-indexing produces incorrect features, and these errors compound through successive layers. We address this with a **Receptive-Field**

Alignment Principle that defines the necessary and sufficient conditions for valid reuse at each layer: it identifies exactly those regions where the aligned context is complete and marks the rest for recomputation, ensuring that the fused feature map is *bit-equivalent* to full-frame inference.

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 [9] at high resolution, using two real-world dynamic video benchmarks—DAVIS [15] and 3DPW [17] which exhibit substantial camera ego-motion, object movement, and occlusion. Baselines include Naive Offloading, COACH [4], DeltaCNN [14], and MotionDeltaCNN [13], 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 [9] 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 [**?**], [10], [4]. This can be effective for classification models that possess a compact bottleneck, but detection and segmentation architectures (e.g., YOLO [9], Mask R-CNN [**?**]) 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 [10] and COACH [4] 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 [**?**], CBinfer [2], Skip-Convolution [**?**],

and DeltaCNN [14]—maintains a pixel-level reference cache and propagates only the *difference* (delta) between the current frame and the reference through the network, computing convolution 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 [13] 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 [13] reported that their homography alignment succeeds on only a small fraction of the DAVIS [15] 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.

C. Motion Vectors as a Free Signal

Motion vectors (MVs), a staple of block-based video coding, 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 no amount of re-indexing can 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 coding. 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

We briefly formalize the interaction between convolution and MV-guided reuse to motivate the core correctness challenge addressed in §**??**.

Let $\mathbf{F}^l \in \mathbb{R}^{C_l\times H_l\times W_l}$ denote the feature map at layer l. A convolutional layer with kernel size k_l, stride s_l, and padding p_l computes

$$\mathbf{F}^{l+1}[c,i,j] \;=\; \sigma\Big(\sum_{c'}\sum_{u,v} W^l[c,c',u,v]\;\mathbf{F}^l\big[c',\; i s_l{+}u{-}p_l,\; j s_l{+}v{-}p_l\big]\Big), \tag{1}$$

where W^l is the kernel weight and σ an element-wise nonlinearity. The set of input positions that influence $\mathbf{F}^{l+1}[c,i,j]$ is its *receptive field*

$$\mathcal{R}^l(i,j) \;=\; \big\{(i\cdot s_l{+}u{-}p_l,\; j\cdot s_l{+}v{-}p_l)\;\big|\;0\le u,v<k_l\big\}. \tag{2}$$

Let $\mathbf{m}_B = (d_x, d_y)$ denote the motion vector of block B. After MV-guided alignment, a cached feature value at position (i,j) is spatially shifted to $(i+d_x,\, j+d_y)$, reflecting the displacement of its enclosing block. For a point-wise operation this shift is exact. For convolution, however, the output at (i,j) depends on all positions in $\mathcal{R}^l(i,j)$. If this receptive field spans multiple blocks whose MVs differ—i.e., the local motion is inconsistent—then the shifted cache entries were derived from *different* spatial contexts in the reference frame and cannot be combined to reproduce the correct convolution output. The resulting error is not confined to a single layer: it propagates and amplifies through successive convolutions, producing feature maps that silently diverge from the ground-truth full-frame result.

This observation establishes the central correctness requirement: MV-guided reuse is valid at a given layer and position *only when the entire receptive field undergoes consistent motion*. Formalizing this condition into a tractable, layer-wise reuse criterion—and efficiently enforcing it at runtime—is the subject of §**??**.

III. Overview

FluxShard is designed for real-time video analytics in resource-constrained edge-cloud settings, such as drones,

robots, or smart cameras, where high-resolution video must be processed under both computation limits and fluctuating wireless bandwidth. The key idea is to maintain *motion-induced cache coherence* across devices: coherence here does not mean keeping edge and cloud caches identical at every step, but rather ensuring that with motion warping and sparse updates, the features required for the current frame form a coherent view on at least one side. Intermediate features are opportunistically reused whenever motion allows, while true misses are selectively recomputed by either the edge or the cloud. Figure 1 illustrates the design.

A. Key Components

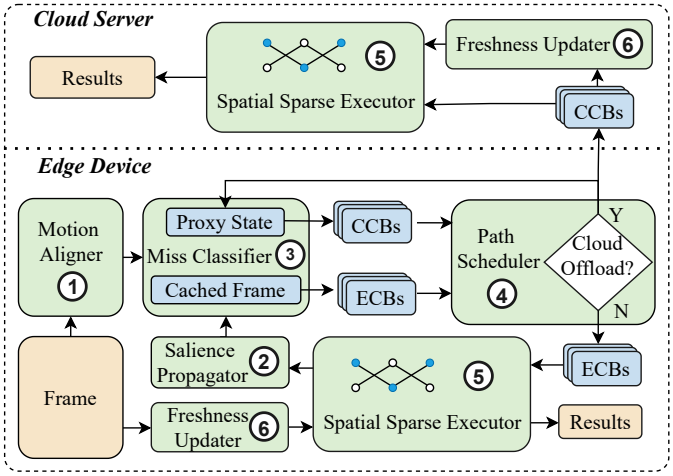

Figure 1. Workflow of FluxShard.

FluxShard coordinates several lightweight components to realize this design:

1. Motion Aligner. Estimates block-level motion between frames and warps cached features on edge and cloud so that large regions can be reused instead of recomputed.

2. Salience Propagator. Projects activation salience from the previous frame into the current one using the motion field, highlighting accuracy-sensitive regions.

3. Miss Classifier. Consolidates motion and salience signals into a set of *critical blocks* that must be refreshed to sustain coherence with the current input frame. As part of this classifier, we embed a lightweight proxy state that tracks the server's current reference image by bending the sent critical blocks into the reference image. The classifier ensures that, after motion warping and selective updates, the cached features on each side form a representation consistent with the frame. Since edge and cloud differ in compute capacity and cache freshness, their motion vectors and respective critical sets may not be identical and thus we have edge critical blocks (ECBs) and cloud critical blocks (CCBs).

4. Path Scheduler. With the salience-guided critical blocks identified and the current estimated network bandwidth, the scheduler decides whether edge or cloud should execute them for the current frame. The decision reflects available compute and bandwidth so that inference can be completed in a timely manner while maintaining coherence with the input. The chosen side produces the result, while the other may carry on background updates that improve cache freshness for future frames.

5. Spatial Sparse Executor. Executes the identified critical blocks with motion vector-guided indexing. In video analytics models where convolutions dominate, sparse execution is complicated by the need for neighborhood context (due to kernel padding) and by cache misalignment under frame-level motion. The executor leverages motion vectors to guide indexing, so that each block is augmented with only the boundary elements it depends on, while cached features are still correctly referenced even when global shifts occur. This motion-aware indexing ensures that sparse execution remains efficient and cache hits remain valid under motion, complementing the upstream coherence mechanisms.

6. Freshness Updater. Runs dense computation opportunistically when slack is available, refreshing stale cache regions to increase the chance of coherence in future frames.

B. System Workflow

At system startup, the edge and cloud share a synchronized reference frame along with a consistent set of cached features. This establishes a common starting point for subsequent collaborative inference.

Given a new frame, the *Motion Aligner* and *Salience Propagator* highlight blocks likely to miss, which the *Miss Classifier* consolidates into a coherent set of critical blocks for both ends. The *Path Scheduler* then evaluates both edge and cloud execution feasibility using startup compute profiles and real-time bandwidth measurements and selects one side to process this set. The chosen side invokes the *Spatial Sparse Executor* to compute the blocks efficiently and produces the final inference result. If any CCBs are transmitted, *Proxy State* for the server is incrementally synchronized to closely track the server's current reference image.

In parallel, the system opportunistically maintains cross-end consistency: while one side is occupied with critical block execution, bandwidth gaps are leveraged to incrementally synchronize *Proxy State*, and idle compute slots are utilized by the *Freshness Updater* to refresh cached features. These overlapping background actions ensure that both edge and cloud remain closely aligned for future frames at minimal extra cost.

Through this workflow, FluxShard enforces motion-induced coherence at the level of a full inference while still exploiting fine-grained sparsity. This enables accurate, low-latency video analytics under tight edge-cloud resource and bandwidth constraints.

IV. Design

Building on the aforementioned components, this section details their concrete design and interactions, including how motion fields align cached features, how salience guides block selection, how misses are consolidated and scheduled, how sparse execution preserves efficiency under motion, and how background updates refresh cache freshness over time, thereby enabling FluxShard to achieve motion-induced coherence for accurate and low-latency video analytics.

A. Motion Aligner

To enable temporal reuse of features, we first compensate for local displacements caused by object or camera motion. We employ a block-based motion alignment that estimates motion vectors between consecutive frames via block matching. For each block $\Omega_{x,y}$ in frame t, the algorithm searches candidate displacements $(\Delta x, \Delta y)$ within a local window $\mathcal{W}$ in the previous frame $t-1$. Each candidate is scored by the sum of absolute differences (SAD):

$$\mathrm{SAD}_{x,y}(\Delta x, \Delta y) = \sum_{(u,v)\in\Omega_{x,y}} \big|\, I_t(u,v) - I_{t-1}(u+\Delta x,\, v+\Delta y) \,\big|.$$

The displacement with the lowest score is selected as the motion vector:

$$(\Delta x^*, \Delta y^*) = \arg\min_{(\Delta x,\Delta y)\in\mathcal{W}} \mathrm{SAD}_{x,y}(\Delta x, \Delta y).$$

By construction, this procedure always provides the best-matching offset within the search window, and when the minimum cost is sufficiently small, the corresponding aligned features are reliable for reuse. However, in regions subject to strong motion, occlusion, or lack of texture, the minimal SAD may still exceed a predefined threshold τ, indicating that even the "best" candidate is in fact poorly aligned. We treat such cases as *true misses*: although a motion vector can always be produced, it should not be trusted for propagation. Instead, blocks with matching cost $\mathrm{SAD}_{x,y}(\Delta x^*, \Delta y^*) > \tau$ are marked as *recomputation candidates*, to be revisited in the subsequent salience analysis.

B. Salience Propagator

Although all high-cost blocks from motion alignment are marked as recomputation candidates, not every candidate influences the final task prediction equally. To prioritize compute for the most relevant regions, we introduce a *salience propagator* that exploits task-level cues to estimate importance.

Instead of relying on heavyweight attention mechanisms, the salience score $s_{x,y}$ for each candidate block (x, y) is computed in a lightweight, task-aware manner by measuring its overlap with task-driven priors (e.g., segmentation masks, detected objects, or other confidence maps) derived from previous outputs. This provides a direct proxy of which regions are semantically crucial for the downstream task. Formally, let $\mathcal{C}$ denote the set of recomputation candidates identified by the motion aligner. Each $(x, y) \in \mathcal{C}$ is assigned a salience score $s_{x,y} \in [0, 1]$. A block is regarded as *salient* if

$$s_{x,y} \;\geq\; \sigma,$$

where σ is a task-driven threshold. Otherwise, the block is classified as *non-salient*. This filtering step ensures that recomputation is reserved only for regions likely to impact final predictions, while less critical areas can be approximated by aligned features from cached frames.

C. Miss Classifier

Given the motion cost $\mathrm{SAD}_{x,y}(\Delta x^*, \Delta y^*)$ from the aligner and the salience score $s_{x,y}$ from the propagator, the miss classifier decides whether block (x, y) should be recomputed or can be approximated by motion-aligned reuse.

The decision rule is summarized as:

$$F'_{x,y} = \begin{cases} \text{Aligned}(F_{t-1}, M_{x,y}), & \text{if } \mathrm{SAD}_{x,y} \leq \tau \;\text{ or }\; s_{x,y} < \sigma, \\ \text{Recompute}(I_t), & \text{if } \mathrm{SAD}_{x,y} > \tau \;\text{ and }\; s_{x,y} \geq \sigma, \end{cases}$$

where $M_{x,y}$ is the estimated motion vector, I_t is the current frame, τ controls alignment reliability, and σ is the salience threshold. We empirically set $\tau = 0.025$ and σ is task dependent.

Two groups of critical blocks. Since the reference features available at the edge and at the server differ, applying the same rule produces two corresponding sets of recomputation requirements: 1. *Edge Critical Blocks (ECBs)*, determined using edge-side references and their motion vectors; 2. *Cloud Critical Blocks (CCBs)*, determined using server-side references (also maintained by the edge as the proxy state) and their motion vectors.

Each set ensures that inference at its respective endpoint maintains temporal coherence to the current frame. With both critical sets established, the next step is to decide which side should carry out the necessary recomputation in order to achieve the best end-to-end efficiency.

D. Path Scheduler

We assume that execution time under different environments can be profiled as functions of the number of recomputed blocks: $f_e(\cdot)$ on the edge and $f_c(\cdot)$ on the cloud. For transmission overhead, each block has average size s, so sending N_{CCB} critical blocks takes $\frac{N_{\mathrm{CCB}} \cdot s}{B}$ time under bandwidth B. Hence

$$T_{\mathrm{edge}} = f_e(N_{\mathrm{ECB}}), \qquad T_{\mathrm{cloud}} = f_c(N_{\mathrm{CCB}}) + \frac{N_{\mathrm{CCB}} \cdot s}{B}.$$

The scheduler then selects the path with the smaller cost,

$$\arg\min_{p\in\{\text{edge, cloud}\}} T_p,$$

thereby guaranteeing minimal inference latency without any compromise to the accuracy of the baseline model

E. Spatial Sparse Executor

The spatial sparse executor bridges block scheduling and actual computation (Fig. 2). In the previous stage, the scheduler marks *critical blocks* that require recomputation. These blocks are always extracted directly from the current input image, and thus their central regions do not rely on motion alignment. This differs from propagated blocks, whose contents are carried from past frames via motion-guided warping.

To guarantee consistency with dense inference, however, each critical block must be expanded with surrounding padding to recover the full receptive field. Here motion offsets become indispensable: while the block centers come from the current image, their border context would be mismatched

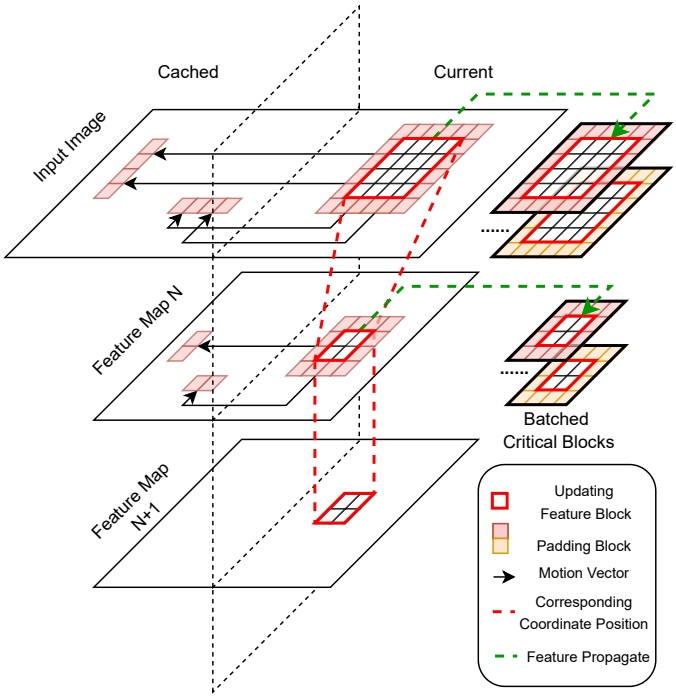

Figure 2. Demonstration of the intermediate propagation of the spatial sparse executor. The red line shows the coordinate correpondance relationship and the green line shows the actual data flow.

without motion alignment. We therefore employ *offset-aware padding*: the main block region is taken verbatim from the current frame, while the padding is sampled according to motion vectors. Offsets are discretized to the nearest neighbor, which stabilizes alignment and avoids the boundary erosion often caused by bilinear schemes.

During intermediate propagation, all reused and recomputed blocks remain in this block-wise format with offset-aware padding. We intentionally avoid forcing a full-frame alignment at every layer, since such global overwriting would convert sparsity back into dense computation and undermine efficiency. Instead, blocks are concatenated along the batch dimension so that irregular, motion-guided updates can be executed as uniform batched workloads.

Finally, at the *model output stage*, the executor performs one global alignment and update. All blocks are projected back to their designated global coordinates, and recomputed critical blocks overwrite outdated content. This final merging step restores dense-level consistency only at the prediction boundary, balancing efficiency during intermediate propagation with accuracy at the output.

F. Freshness Updater

Block-sparse execution ensures that each frame can be processed within a tight latency budget, but over time the cache may drift away from what would have been produced by dense inference. This gradually weakens the coherence between cached representations and the visual content of the current input. To counteract such drift without sacrificing responsiveness, we exploit opportunities in both computation and communication, allowing accuracy to recover opportunistically while the sparse critical path remains intact.

On the client side, whenever cycles become available, the system performs small fragments of a dense update. These updates are preemptable: they run only when compute resources are idle, and yield immediately when the next sparse inference step is scheduled. Incrementally, they rewrite parts of the cache with dense features, tightening its coherence with the live input stream and reducing long-term degradation under reuse.

Communication offers a complementary opportunity. The server strictly requires only critical blocks, which keeps baseline bandwidth low. However, whenever residual capacity is available, the client can forward additional non-critical blocks. These opportunistic transmissions gradually align the server-side cache with the client's current frame, improving feature coherence across devices without imposing extra latency.

Through this dual mechanism, spare compute cycles and residual bandwidth are repurposed to continually reinforce the model state. The result is a system where caches remain coherent both locally and remotely, progressively approaching the fidelity of dense execution while retaining the efficiency of the sparse pipeline.

V. Overall Workflow

Putting the above components together, FluxShard processes each frame through the cooperative workflow summarized in Algorithm 1. At the edge, the system derives motion vectors, salience, and criticality from the cached previous frame (together with the proxy state of). Based on current bandwidth and compute capacities, it compares the estimated latency of local execution with that of offloading, and dynamically assigns the critical path to either edge or server. When the edge executes locally, it opportunistically transmits spare non-critical blocks to the server; when the server takes over, the edge instead refreshes stale cache regions. The server, in both cases, integrates any received non-critical blocks into its cache. Through this cooperative mechanism, FluxShard achieves low-latency inference while keeping cache states consistent across edge and server. The coherence maintained between motion alignment, salience propagation, and opportunistic updates ensures that both accuracy and efficiency are sustained throughout continuous video processing.

VI. 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

Algorithm 1: Cooperative execution of FluxShard at frame F_t

Input: Frame F_t, previous cached frame F_{t-1}, proxy state for server P_{t-1}, bandwidth B_t
Output: Inference result Y_t

Edge-side procedure: ;
$(\mathcal{M}_e, \mathcal{M}_s, SAD_e, SAD_s) \leftarrow$ MotionAligner(F_{t-1}, P_{t-1}, F_t) ;
$S_t \leftarrow$ SaliencePropagator$(F_{t-1}, \mathcal{M}_e, \mathcal{M}_s)$;
$(\mathcal{B}_e, \mathcal{B}_c) \leftarrow$ MissClassifier(S_t, SAD_e, SAD_s) ;
$T_e \leftarrow$ estimate_edge_time$(\mathcal{B}_e)$;
$T_c \leftarrow$ estimate_cloud_time$(\mathcal{B}_c, B_t)$;
if $T_e \leq T_c$ **then**
  /* Edge executes the critical path */
  $Y_t \leftarrow$ SparseExecutor$_{edge}(\mathcal{B}_e, \mathcal{M}_e)$;
  Opportunistically transmit extra non-critical blocks to server ;
  Update proxy state P_t with sent non-critical blocks ;
else
  /* Server executes the critical path */
  Send $\mathcal{B}_c$ and $\mathcal{M}_c$ to server ;
  Opportunistically refresh stale blocks via freshness updater ;
  Update proxy state P_t with $\mathcal{B}_c$ and $\mathcal{M}_c$;

Server-side procedure: ;
if *received* $\mathcal{B}_c$ *and* $\mathcal{M}_e$ **then**
  $Y_t \leftarrow$ SparseExecutor$_{server}(\mathcal{B}_c, \mathcal{M}_e)$;
Opportunistically integrate received non-critical blocks into cache ;

return Y_t

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.

VII. 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 [16] 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 [8] (referred to as Detect) and dense segmentation using YOLO11m-seg [8] (referred to as Segment) on two real-world datasets: 3DPW [17] and DAVIS [15]. 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 [3] (referred to as Classify) and multi-person keypoint detection using Mask R-CNN [6] (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 [4]: 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 [14]: 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 [8] 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

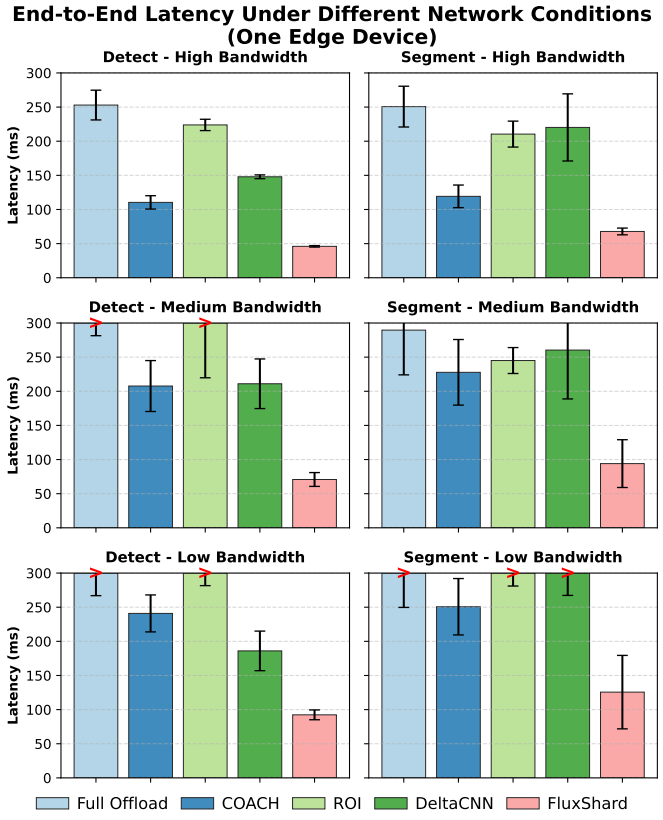

Figure 3. Latency comparison of different systems. FluxShard consistently outperforms the other baselines under different bandwidth conditions.

a) End-to-End Latency: Figure 3 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.

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

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 [8]) 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.

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 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 $\sim$23% (Segment) and $\sim$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 $\sim$64.5% for Segment and $\sim$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

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)

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 4 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 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

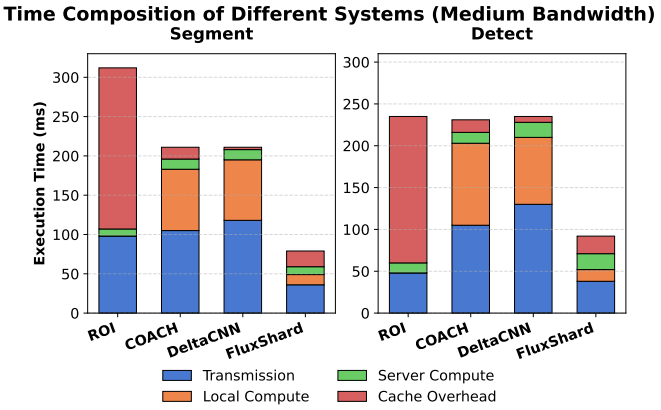

Figure 4. Execution time composition of COACH, ROI, DeltaCNN, and FluxShard under medium bandwidth conditions for Segment and Detect workloads.

bandwidth is already network-bound. Fig. 5 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.

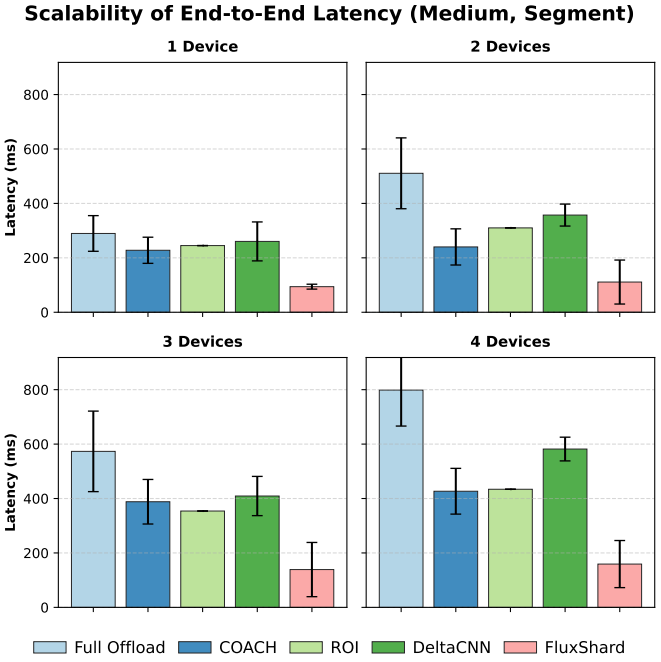

Figure 5. Scalability comparison of end-to-end latency in medium-bandwidth conditions for Segment. FluxShard scales more efficiently, maintaining lower latency growth as devices increase.

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

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 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.

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] 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.

[4] 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.

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

[6] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn, 2018.

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

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

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

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.