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

Paper 771, 12 pages

Abstract

In edge-cloud video analytics, limited edge computation power and bandwidth make feature reuse essential to sustain accuracy under strict latency constraints. Prior reuse strategies often fail in dynamic scenes where motion and viewpoint changes disrupt naive content matching and reusing. We formulate this challenge as a *motion-induced cache-coherence problem* and design a lightweight motion-vector-guided abstraction to align reusable features and identify true misses requiring recomputation. Around this abstraction, we propose **FluxShard**, a motion-aware collaborative video analytics system that prioritizes task-relevant updates, maintains feature freshness without stalling inference, and leverages a novel, hardware-efficient sparse computation layout. Evaluation shows that FluxShard achieves up to 92% bandwidth reduction, 3.25× speedup, and ~99% accuracy retention on various video analytics tasks across different datasets, outperforming state-of-the-art baselines. This formulation and system provide a principled foundation for motion-aware reuse in resource-constrained video analytics.

1 Introduction

Video analytics underpins a wide range of latency-critical applications such as autonomous driving, aerial drones, augmented reality, and smart surveillance, where timely and accurate understanding of high-rate video streams is essential. Processing on edge devices enables real-time responses but is hampered by limited computational and energy resources. Conversely, offloading to the cloud offers scalable, on-demand computing power but introduces significant latency from network transfer and remote execution, undermining real-time performance. A promising way to mitigate both transmission and computation costs is to *reuse* results across consecutive frames in a cache-like manner: if parts of the scene remain unchanged, their previously computed outputs can be retained instead of recomputed or re-sent, reducing both processing delay and bandwidth usage while keeping results consistent.

However, real-world video streams are rarely static. Camera motion, object dynamics, and changes in viewpoint cause scene content to shift within the frame, so that even visually unchanged regions may appear at different spatial locations across consecutive frames. Such displacements break naive pixel-wise or block-wise matching, leading to frequent cache misses and forcing unnecessary recomputation or retransmission. These inefficiencies undermine the potential latency and bandwidth savings of cache-like reuse, motivating the need for more robust mechanisms that can tolerate motion while preserving accuracy.

Prior work on avoiding redundant computation has primarily relied on cache-like reuse at various granularities, making these methods inherently brittle in dynamic environments where camera or object motion can rapidly invalidate cached results. Pipeline-based methods (e.g., SPINN [17], COACH [8]) cache intermediate feature maps along the model's execution path, reusing them when early-stage similarity checks exceed a threshold. Delta-based methods (e.g., DeltaCNN [29]) transmit only changes between consecutive frames, updating cached features in place. ROI-based systems focus computation on spatial regions likely to contain objects of interest, often using auxiliary detectors to identify those areas. While effective in relatively static scenarios, all three paradigms face substantial limitations in dynamic environments: global shifts from camera motion, object movement, or viewpoint changes can rapidly invalidate cached results or degrade similarity scores, forcing frequent recomputation or large data transfers. From a cache-design perspective, these approaches either fail to maintain temporal-spatial coherence at the right granularity or incur high overheads in tracking, updating, and validating cached content.

A promising direction for addressing these limitations is suggested by motion vectors (MVs), a concept long used in video encoding to capture block-level displacement between consecutive frames. Such information reflects appearance-level similarity over time and can help track the movement of reusable content, thereby reducing false cache misses that arise when small shifts or viewpoint changes are misinterpreted as content changes. However, motion vectors have inherent limits: they cannot fully capture non-rigid object deformation, scale variation, or occlusion, and in such regions feature misalignment persists and recomputation is necessary. Here, we take the concept of motion vectors out of the heavy encoding pipeline and use it efficiently as a lightweight, model-agnostic signal both to guide cache alignment for reuse and to identify regions that require recomputation in dynamic video analytics.

We present *FluxShard*, a motion-aware video analytics system for heterogeneous edge-server deployment. FluxShard models feature reuse across consecutive frames as a *motion-induced cache-coherence problem*, where coherence refers to the consistency between cached features and the current input frame, even under motion or viewpoint shifts where naive content matching fails. To maintain this consistency, FluxShard re-indexes cached features with motion vectors and selectively recomputes regions that cannot be aligned (true misses). Once the combined cached and recomputed parts form a coherent feature map for the latest frame, inference can proceed without error, while the system dynamically decides whether coherence resolution should occur on the edge or the server to achieve lowest latency while maintaining high accuracy.

The design of FluxShard has the following challenges. The first challenge is that while motion-vector (MV) based alignment effectively removes a large portion of *false* cache misses which can hit by correctly reindexing, in high-speed scenes a substantial number of *true* misses still remain, and updating all of them within the per-frame latency budget is infeasible. Moreover, true misses differ in their impact on the final prediction: regions more relevant to the task output deserve higher priority. Leveraging the fact that a vision model inherently focuses on output-relevant regions, we use *salience-guided miss prioritization*: we take the prediction from the previous frame and, via MV mapping, project its spatial salience onto the current frame; this yields a direct estimate of which miss blocks most influence the result, enabling us to update the most critical ones first to achieve a better accuracy-latency balance.

While prioritizing critical misses preserves immediate accuracy, non-critical regions left untouched will gradually drift and cause stale features. Refreshing all of them every frame is wasteful in both compute and bandwidth. FluxShard instead adopts an *opportunistic freshness* strategy: cache updates for less salient blocks are scheduled as background work, using idle cycles and bandwidth on the edge or server without interfering with latency-critical inference. This keeps the cache fresh over time while extending short-term prioritization into long-term efficiency.

We implement **FluxShard** in C++/CUDA with PyTorch bindings, supporting deployment across heterogeneous edge-cloud systems. The implementation incorporates optimized GPU kernels for the aforementioned motion-aware and irregular computation patterns of sparse workloads, ensuring high utilization and scalability. We evaluate FluxShard on three representative real-world video analytics tasks: YOLOv11 [16] for video object detection on the **DAVIS** dataset, YOLOv11 segmentation for high-resolution environments on the **SA-V** dataset, and SDPose [6] for multi-person keypoint detection on the **Panoptic** dataset. The baselines include Naive Offloading, SPINN [17], COACH [8], and DeltaCNN [29], tested on NVIDIA Jetson Xavier NX devices (edge) and an RTX 3080 PC (cloud). Across all tasks and datasets, FluxShard achieves:

- **Bandwidth reduction** - up to **92%** savings over full-frame transmission.
- **End-to-end acceleration** - up to **3.25**× faster inference; sparse computation efficiency approaches *proportional* scaling with active-region (miss) size.
- **Accuracy retention** - preserves approximately **99%** of peak accuracy.
- **Kernel efficiency** - optimized motion-aware functions sustain high GPU utilization and throughput even under high sparsity.
- **Scalability** - maintains xx% lower latency growth than baselines when scaling to multiple concurrent edge devices.

In summary, this paper's primary contribution is to reframe feature reuse in dynamic video analytics as a **motion-induced cache-coherence problem**. We address this fundamental challenge by first introducing a lightweight, motion-vector-guided abstraction that precisely aligns cached features and isolates the minimal regions requiring recomputation. Building on this core idea, we design and implement **FluxShard**, a motion-aware edge-cloud system engineered to resolve these motion-induced cache misses with minimal latency. To make FluxShard practical and efficient, we introduce three key technical innovations: (i) *salience-guided miss prioritization* from projected prior predictions to overcome the infeasibility of updating all true cache misses in dynamic scenes; (ii) *opportunistic freshness* overlapping background updates with active critical inference to keep the cache fresh over time. Experiments on object detection, segmentation, and pose estimation show substantial bandwidth reduction, multi-fold speedup, and near-peak accuracy retention over state-of-the-art baselines. We expect our formulation and design to inspire future motion-aware reuse strategies in resource-constrained, collaborative video analytics.

2 Background

Real-time video analytics has become a cornerstone of modern intelligent systems, powering applications such as autonomous vehicles [24–26], augmented reality [7, 32, 37], video surveillance [3, 11, 40], and smart city infrastructure [1, 2, 10]. These applications often require immediate responses based on the continuous analysis of high-resolution video streams, which generate immense computational and bandwidth demands. While cloud computing offers powerful resources for processing such workloads, the latency and reliability factors of remote data transmission can introduce significant delays. Conversely, edge devices, although closer to the data source, are typically resource-constrained, with limited computational power, memory, and energy efficiency.

2.1 Main Models in Visual Analytics

Visual analytics tasks, such as object detection [18, 28, 44], semantic segmentation [20, 34, 35], optical flow estimation [12, 38, 39], and depth estimation [15, 22, 23], rely on advanced deep learning models to extract fine-grained spatial information from video data. These models predominantly fall into two families: convolutional neural networks (CNNs) and vision transformers.

2.1.1 Spatial Computation in CNNs and Vision Transformers.

CNNs are inherently spatially structured, with convolutional operations preserving the spatial relationships within feature maps. For example, models like Faster R-CNN [41] and DeepLab [4] retain pixel-to-pixel alignment between input frames and feature maps during each convolution and pooling operation, enabling precise spatial reasoning in tasks like segmentation or detection.

Vision transformers, such as DETR [46] and SegFormer [45], divide input frames into spatial patches before applying self-attention mechanisms. While transformers lack the strict locality bias of CNNs, they still maintain patch-wise spatial alignment throughout their computation pipeline, which we can exploit to track regions of interest.

2.1.2 Motion Vector-Wrapped Blocks in CNNs and Transformers.

FluxShard leverages the spatial alignment preserved by both CNNs and vision transformers to enable its motion vector-wrapped block abstraction. For CNNs, the blocked structure seamlessly integrates with the convolutional operations, as each block intrinsically corresponds to a subset of the spatially aligned feature map. Motion-sensitive updates can be computed efficiently by isolating block-level regions and propagating changes through the subsequent layers.

For vision transformers, the motion vector-wrapped block abstraction aligns naturally with the patch-based input representation. Dynamic patches are tracked and updated using motion vectors, enabling FluxShard to focus attention computations and self-attention mechanisms on only the changing regions of the frame. This enables efficient sparse processing without disrupting the global-context modeling characteristic of transformers.

2.2 Challenges in Edge-Cloud Video Analytics

Edge-cloud architectures must balance computation, bandwidth, and latency, but stringent resource constraints hinder real-time video analytics.

Bandwidth Bottlenecks: High-definition video streaming (e.g., 1080p @ 30 FPS) demands substantial uplink bandwidth, even with H.264 compression—ranging from 8~20 Mbps [42]. Multi-camera setups exacerbate this issue, with 10-camera systems requiring up to 200 Mbps, far exceeding practical limits in mobile networks, where uplink bandwidth often falls below 10 Mbps.

Edge Computation Limits: Resource-constrained edge devices struggle with real-time neural inference. For instance, YOLOv11 achieves only 5 FPS on a etson Xavier NX in our evaluation, operating near peak GPU utilization. Energy constraints in battery-powered devices further limit their capacity to sustain continuous analytics.

Latency Sensitivity: Autonomous driving and surveillance demand perception latencies below 50~100 ms [19]. However, cloud offloading introduces 50~200 ms round-trip delays [21], making such deadlines difficult to achieve.

To address these constraints, FluxShard introduces **motion vector-wrapped blocks**, reducing transmission overhead by prioritizing motion-sensitive updates while dynamically balancing edge-cloud computing to minimize latency.

2.3 Existing Systems and Their Limitations

Several frameworks have been proposed to tackle edge-cloud collaboration for video analytics, but they exhibit fundamental shortcomings in harnessing the spatiotemporal redundancy of video streams and addressing dynamic scene changes.

Cache-pipeline-based systems, such as SPINN [17] and COACH [8], decompose video analytics into stages like frame sampling, feature extraction, and inference, distributing computation between the edge and cloud. During inference, they attempt to reuse computation by caching early-exit results. While effective for simpler recognition tasks (e.g., classification), they struggle with dense workloads like segmentation or keypoint detection, which require precise spatial details.

As a result, inference results cannot be directly reused or approximated, necessitating the transmission of full or compressed intermediate data (e.g., feature maps or video frames) to the cloud, leading to:

- *High Transmission Overhead:* Sending entire feature maps or frames strains bandwidth, limiting scalability in multi-camera setups.
- *Redundant Processing:* These methods lack motion-awareness and recompute both static and dynamic regions, increasing computational overhead.
- *Accuracy Degradation:* Even when frame-to-frame similarity exceeds 95%, the critical changes often occur within the **5% non-similar regions**, which are essential for dense tasks. Cached results fail to capture these fine-grained updates, producing coarse segment boundaries in object segmentation and unstable keypoint estimates in pose detection, degrading accuracy.

Delta-based systems, such as DeltaCNN [29], improve upon pipeline-based methods by exploiting the temporal redundancy of video streams. These methods transmit and process only the changed regions (*deltas*) between consecutive frames. While this reduces data transmission and computation, delta-based approaches face critical limitations:

- *State Inconsistency:* Without explicit consideration of motion vectors, deltas fail to account for global scene changes, such as shifts caused by camera motion, leading to redundant updates and loss of context.

- *Inefficient Sparse Computation:* Sparse updates introduce irregular memory access patterns that degrade the performance of GPU-based dense computation kernels, limiting efficiency gains.

2.4 Towards a Motion-Aware Abstraction

Despite progress in cache-pipeline-based and delta-based current systems fail to address the unique challenges posed by highly dynamic, resource-constrained scenarios of dense visual analytics. This motivates a new abstraction that:

- **Encodes Spatiotemporal Redundancy:** Explicitly identifies and processes only localized, dynamic regions of video streams over time.
- **Scales Across Motion Dynamics:** Maintains state consistency while adapting to global scene shifts using motion-aware information.
- **Enables Unified Optimization:** Integrates computation, communication, and state propagation to holistically optimize bandwidth, accuracy, and latency.

In this work, we introduce the **motion vector-wrapped block abstraction** as the foundational design principle for **FluxShard**, a framework that addresses these challenges and sets a new paradigm for edge-cloud collaborative video analytics.

2.5 Relationship With Motion-Vector-Based Video Compression Standards

Modern video compression standards such as H.264 [43], H.265/HEVC [30], VP9 [27], and AV1 [9] reduce bandwidth requirements by exploiting redundancies in video streams through motion-vector-based interframe compression. These methods analyze temporal changes between consecutive frames, using motion vectors to represent the displacement of regions, thereby minimizing redundant data transmission.

FluxShard builds upon this principle by reusing motion vectors from these video codecs to guide its inference optimization pipeline. Specifically, the motion vector-wrapped block abstraction in FluxShard leverages these motion vectors to identify and update dynamically changing regions, avoiding unnecessary computation on unchanged spatial regions. This enables FluxShard to minimize the computational and communication overheads for video analytics tasks.

While motion-vector-based codecs optimize video encoding for efficient storage or transmission, FluxShard complements this by targeting computation and bandwidth efficiency at the inference level, where feature extraction and model processing dominate. By unifying these layers in the video analytics pipeline, FluxShard ensures system-wide efficiency in both video delivery and AI-driven analysis workflows.

3 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.

3.1 Key Components

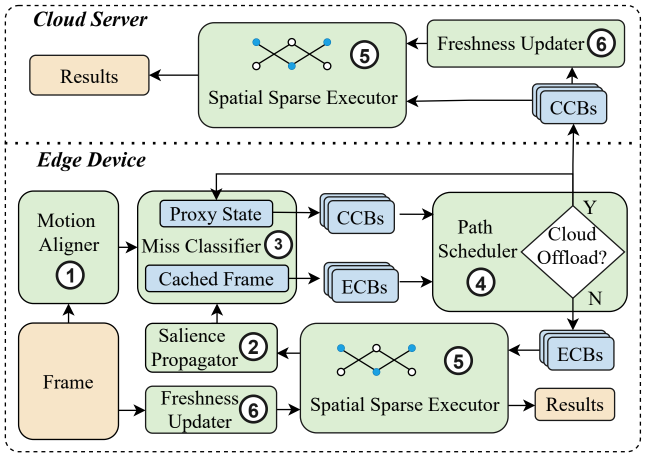

Figure 1. Workflow of FluxShard.

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.

3.2 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.

4 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.

4.1 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):

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

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.

4.2 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 C denote the set of recomputation candidates identified by the motion aligner. Each $(x, y) \in C$ 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.

4.3 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:

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

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.

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.

4.4 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,

$$\underset{p \in \{\mathrm{edge}, \mathrm{cloud}\}}{\arg\min} \; T_p,$$

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

4.5 Spatial Sparse Executor

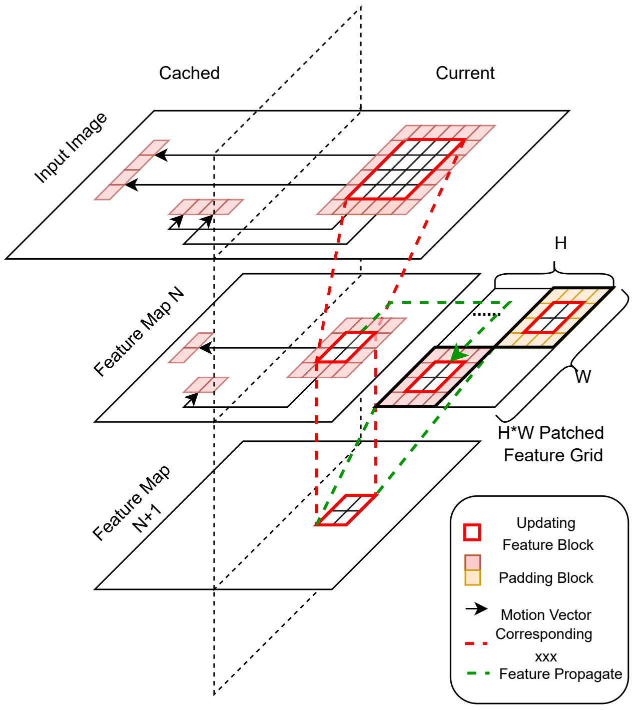

Figure 2. Demonstration of the intermediate propagation of the spatial sparse executor. The red line shows the coordinate correpondance relationship and the green line shows the actual data flow.

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 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.

4.6 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.

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 $Y_t$
```

5 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.

6 Implementation

Prototype Development. The FluxShard prototype is implemented using Python and C++ with CUDA on Ubuntu

20.04. The motion block extraction module and the motion-sensitive computation module are implemented as a custom *PyTorch extension*. This design leverages the extensibility of PyTorch [31] while achieving high-performance GPU acceleration through optimized low-level CUDA kernels. By directly integrating CUDA kernels with PyTorch, the motion-sensitive computation achieves low-latency execution while seamlessly integrating into the system pipeline.

Edge-cloud communication is implemented using basic TCP sockets. Using a simple TCP-based design minimizes system dependencies and maintains compatibility across edge and cloud platforms. Together, the custom computation module and the lightweight communication mechanism enable a high-performance, real-time video-analytic system that efficiently operates under edge-cloud constraints.

7 Evaluation

7.1 Evaluation Setup

Testbed. We evaluate FluxShard on a hybrid edge-cloud testbed comprising up to three 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 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 [33] 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.

Models and Datasets. We evaluate FluxShard using two deep learning models on real-world datasets:

- **YOLOv11 [13] (SA-V dataset [36])** A CNN for dense segmentation (640×640 input). SA-V provides high-quality spatio-temporal segmentation masks.
- **SDPose [5] (Panoptic dataset [14])** A Transformer-based model for keypoint detection (192×256 input). The Panoptic dataset captures human activities for keypoint tracking.

7.2 Baselines for Evaluation

We compare FluxShard against four baselines:

- **Full Offload** Executes all inference on the server, incurring high transmission costs.
- **SPINN** [17] A split DNN system with early-exit caching.
- **COACH** [8] SPINN with quantization for bandwidth reduction but no motion-awareness.
- **DeltaCNN** [29] Processes only pixel-level frame deltas, struggling with high-motion workloads.

Table 1. Dense Inference Performance and Latency Model Parameters

Parameter	YOLOv11	SDPose
Edge Latency (s)	0.209	0.132
Edge Power (mW)	10196	9514
Server Latency (s)	0.012	0.016
Quadratic Coeff. (edge)	3.60×10^{-8}	4.96×10^{-7}
Linear Coeff. (edge)	2.13×10^{-4}	6.64×10^{-4}
Constant (edge)	0.0429	0.00772
Quadratic Coeff. (cloud)	2.63×10^{-9}	7.33×10^{-8}
Linear Coeff. (cloud)	1.52×10^{-5}	6.23×10^{-5}
Constant (cloud)	0.00714	0.00412

These baselines represent a spectrum of edge-cloud collaboration strategies, enabling a comprehensive evaluation of FluxShard's adaptability.

Metrics. FluxShard's evaluation considers:

- **Accuracy:** IoU for YOLOv11 and Keypoint mAP for SDPose, normalized to full-ground-truth accuracy.
- **Latency:** Average end-to-end frame processing time.
- **Bandwidth Usage:** Average transmission bandwidth (MBps).
- **Cache Hit Rate:** Reuse ratio; for FluxShard and DeltaCNN, this reflects shard reuse, while for SPINN and COACH, it measures early-exit computation reuse.
- **Compute Load Distribution:** Fraction of total computations handled on edge vs. offloaded to the cloud.
- **Energy Consumption:** Average energy consumption on the edge device (milli-watt).

7.3 End-to-End Results with a Single Edge Device

We evaluate FluxShard under different network conditions (high, medium, and low bandwidth) on a single Jetson Xavier NX device, comparing it with Full Offload, SPINN, COACH, and DeltaCNN.

Latency. Figure 3 presents the frame processing latency across different bandwidth conditions. Full Offload exhibits consistently high latency due to excessive data transmission overhead and is omitted from detailed comparisons. Among the other methods, FluxShard achieves the lowest latency across all scenarios by efficiently managing both computation and transmission.

Across all tested conditions, FluxShard provides substantial speedup—ranging from **1.76X to 3.25X** for YOLOv11 and **1.04X to 1.68X** for SDPose in high-bandwidth conditions; **2.26X to 2.61X** for YOLOv11 and **1.49X to 2.01X** for SDPose in medium-bandwidth conditions; and **1.98X to 2.16X** for YOLOv11 and **1.83X to 1.87X** for SDPose in low-bandwidth conditions. This efficiency stems from FluxShard's ability to **minimize redundant transmissions** by selectively offloading motion-triggered regions rather than entire frames.

Additionally, through **adaptive task allocation**, FluxShard effectively balances computation between the edge and the cloud based on available resources, ensuring low-latency processing even under varying bandwidth constraints.

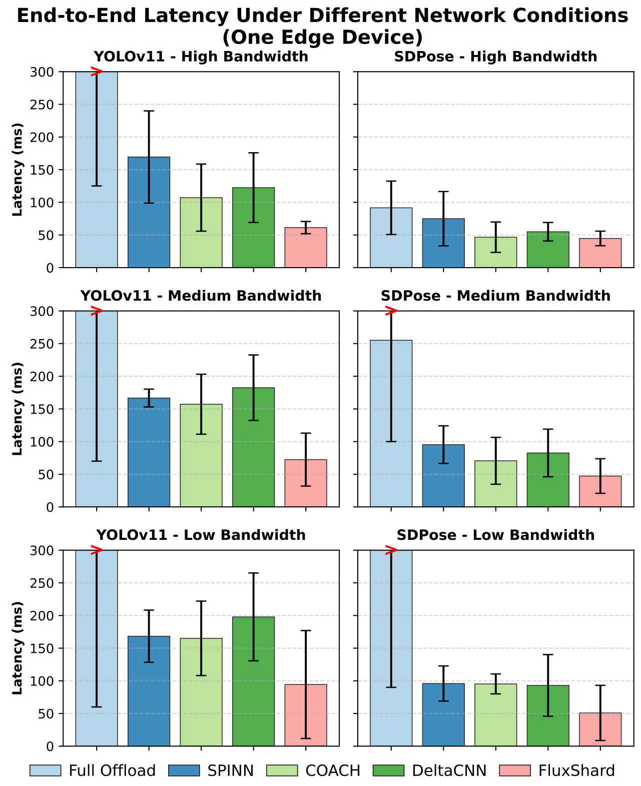

Figure 3. Latency comparison under different bandwidth conditions. FluxShard consistently outperforms the other non-vanilla baselines, particularly in constrained network environments.

Power Consumption. Figure 4 compares the energy consumption of different methods under varying bandwidth conditions. Full Offload has the lowest power consumption, as it shifts computation entirely to the cloud, but its high latency makes it impractical. Among the other methods, FluxShard maintains competitive energy efficiency while achieving significantly lower latency.

FluxShard reduces energy consumption by **1.0% to 34.9%** for YOLOv11 and **0.2% to 5.1%** for SDPose in high-bandwidth conditions where computation is primarily handled in the cloud. As bandwidth decreases and more computation shifts to the edge, FluxShard achieves **17.0% to 26.2%** savings for YOLOv11 and **29.2% to 43.9%** for SDPose in medium-bandwidth conditions. In low-bandwidth settings, where traditional methods become more reliant on local computation, FluxShard continues to provide energy savings of **13.7% to 22.9%** for YOLOv11 and **41.1% to 42.1%** for SDPose.

This efficiency comes from FluxShard's ability to **dynamically distribute workloads**, reducing redundant computation on the edge and effectively leveraging cloud resources when beneficial. By selectively determining which computations to perform locally and which to offload, FluxShard prevents unnecessary energy consumption while maintaining real-time performance.

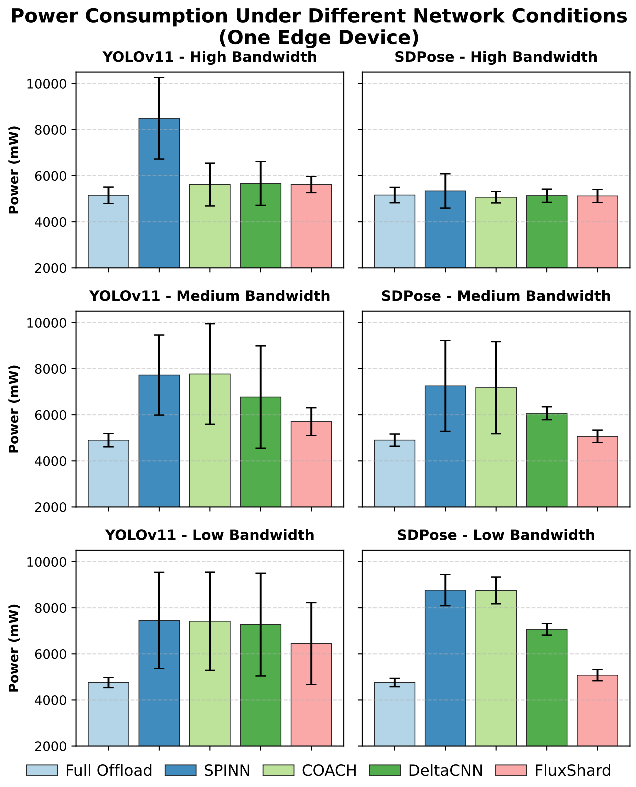

Figure 4. Power consumption comparison across different bandwidth conditions. FluxShard reduces power usage through efficient offloading and adaptive workload distribution.

Accuracy. Table 2 shows the accuracy of different methods under various bandwidth conditions. While Full Offload achieves 100% accuracy (excluded from the table), SPINN and COACH exhibit the largest accuracy drop, as their **early exit strategies** rely on whole-image similarity matching, leading to accuracy reductions of up to **86.0% for YOLOv11** and **89.0% for SDPose**.

DeltaCNN maintains high accuracy (~97% for YOLOv11, ~99% for SDPose) by preserving feature integrity through delta encoding. FluxShard achieves a comparable level of accuracy but sees a slight reduction under low bandwidth conditions, reaching **95.5% for YOLOv11** and **98.5% for SDPose**.

Table 2. Accuracy comparison under different bandwidth conditions (%).

Bandwidth	Model	SPINN	COACH	DeltaCNN	FluxShard
High	YOLOv11	86.0	84.5	97.1	96.8
	SDPose	89.0	87.2	99.2	99.0
Medium	YOLOv11	86.0	85.2	97.0	96.3
	SDPose	89.0	88.1	99.1	98.8
Low	YOLOv11	86.0	86.0	96.9	95.5
	SDPose	89.0	89.0	99.0	98.5

The minor drop in FluxShard's accuracy is attributed to its **motion-aware adaptive scheduling**, which dynamically determines the optimal balance between transmission and local computation. Under tighter bandwidth constraints, FluxShard selectively limits the number of transmitted / locally processed motion blocks to prioritize efficiency, ensuring a strong tradeoff between accuracy, latency, and energy consumption.

Scalability. Scalability analysis is conducted under the high-bandwidth scenario with multiple edge devices and a single cloud server (tc constraining ingress bandwidth), as it provides the most feasible environment for scaling in real-world deployments, and since the effect of bandwidth on the other metrics have been analyzed earlier, we focus on how latency evolves as the number of edge devices increases under optimal transmission conditions.

As shown in Figure 5, SPINN, COACH, and DeltaCNN experience significant latency growth due to queuing bottlenecks at both the cloud and edge. When increasing devices from one to two, SPINN's latency rises by 1.9X, COACH by 1.7X, and DeltaCNN by 1.8X, showing early-stage scaling inefficiencies. With three devices, these values further increase to 2.8X, 2.3X, and 2.5X, respectively, highlighting their inability to balance computation across devices. FluxShard, by contrast, scales more efficiently, with only a 1.20X increase from one to two devices and 1.35X at three. This improvement stems from its adaptive workload distribution, which prevents redundant processing and minimizes computational congestion. These results indicate that while all methods degrade with more devices, FluxShard maintains significantly lower latency growth, making it well-suited for scalable edge deployments.

7.4 Micro-benchmark and Ablation Study

Table 3 presents the cache hit ratios of different methods under varying bandwidth conditions for YOLOv11 and SDPose. SPINN and 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 ~10% for YOLOv11, but improves to ~43% for SDPose when dealing with stationary-camera scenarios. However, both methods maintain high bandwidth consumption, with COACH requiring up to 10.85 MB/s for YOLOv11 in high-bandwidth conditions.

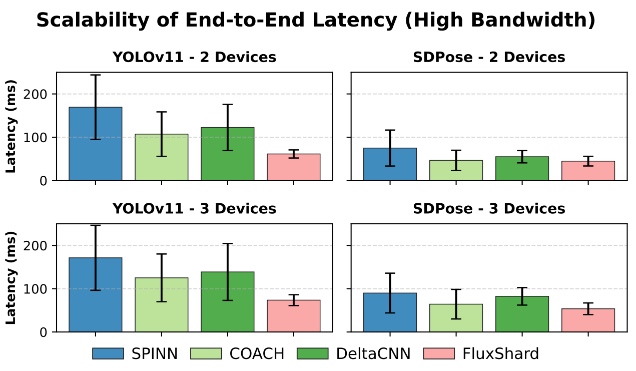

Figure 5. Scalability comparison of end-to-end latency in high-bandwidth conditions. FluxShard scales more efficiently, maintaining lower latency growth as devices increase.

Table 3. Cache hit ratio (%) comparison and average bandwidth consumption (MB/s) under different bandwidth conditions. Average bandwidth consumption is shown in parentheses.

Bandwidth	Model	SPINN	COACH	DeltaCNN	FluxShard
High	YOLOv11	10.5 (10.05)	9.8 (10.85)	22.4 (6.57)	75.3 (2.05)
	SDPose	42.7 (7.42)	43.5 (3.17)	72.8 (0.64)	91.1 (0.26)
Medium	YOLOv11	10.3 (0.058)	9.7 (5.05)	23.1 (3.08)	73.8 (1.04)
	SDPose	42.9 (3.38)	43.0 (1.26)	73.4 (0.34)	91.2 (0.24)
Low	YOLOv11	10.3 (0.113)	9.9 (0.67)	23.7 (0.89)	72.5 (0.32)
	SDPose	43.1 (0.020)	42.8 (0.092)	73.1 (0.26)	90.7 (0.23)

DeltaCNN improves reuse efficiency by applying partial feature caching, achieving hit ratios of ~23% (YOLOv11) and ~73% (SDPose). However, its bandwidth usage varies significantly across settings, consuming up to 6.57 MB/s in high-bandwidth scenarios but dropping to 0.89 MB/s in low-bandwidth conditions.

FluxShard achieves the highest hit ratio **75.3% for YOLOv11 and 91.1% for SDPose** by leveraging **motion-aware caching** that maintains high cache locality even under dynamic environments. Importantly, FluxShard maintains the lowest bandwidth consumption across all settings, requiring only 2.05 MB/s at high bandwidth and 0.32 MB/s at low bandwidth while sustaining peak cache efficiency. This demonstrates its ability to adaptively manage feature transmission for efficient bandwidth utilization and improved performance consistency.

Figure 6 shows the execution time breakdown of SPINN, COACH, DeltaCNN, and FluxShard under medium bandwidth conditions for YOLOv11 and SDPose. FluxShard effectively reduces both transmission and computation time through its **motion-aware feature caching** and **adaptive**

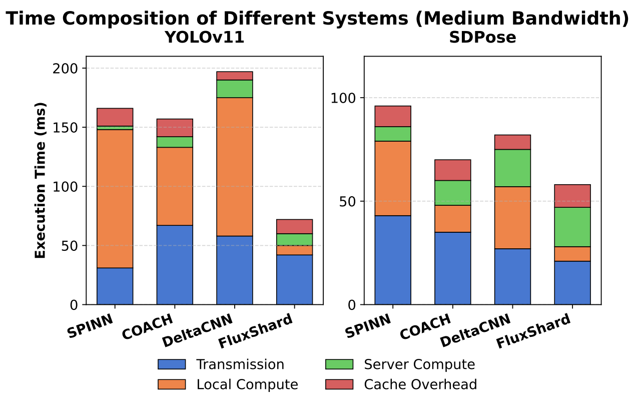

Figure 6. Execution time composition of SPINN, COACH, DeltaCNN, and FluxShard under medium bandwidth conditions for YOLOv11 and SDPose workloads.

Table 4. Ablation study on FluxShard's optimizations in medium bandwidth (YOLOv11). We test the impact of removing Adaptive Task Allocation (No-ATA) and motion-sensitive computation (No-MS-Comp).

Method	YOLOv11 Lat. (ms)	Cache Hit (%)	Accuracy (%)
Full-Scale FluxShard	72.40	73.8	96.3
No-ATA	85.74	78.4	96.5
No-MS-Comp	94.65	73.8	96.3

task allocation. Instead of offloading entire frames like SPINN, 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.

Table 4 presents an ablation study analyzing the influence of **Adaptive Task Allocation (ATA)** and **motion-sensitive computation** on FluxShard's performance under medium bandwidth (YOLOv11). Removing **Adaptive Task Allocation (No-ATA)** slightly increases cache hit ratio from 73.8% to 78.4% and accuracy from 96.3% to 96.5%, as all recoginized blocks that cannot be aligned with the motion vector are offloaded or being recomputed. However, this results in a latency increase of 18.4% due to reduced computational flexibility. Deactivating **motion-sensitive computation (No-MS-Comp)** disables feature reuse during inference. Latency worsens to 94.65 ms (+30.7%), as the inference is always conducted at full scale which is especially imapctful for the edge device. However, cache hit ratio and accuracy remain unchanged.

Overall, the full-scale FluxShard design ensures optimal efficiency, balancing latency reduction and high cache utilization. These findings confirm that combining **motion-aware caching** with **adaptive task allocation** significantly enhances performance for real-time processing under constrained bandwidth scenarios.

7.5 Limitations and Future Work

FluxShard is specifically designed for continuous video streams, leveraging temporal redundancy to optimize computation; however, it is **not applicable to general-purpose vision tasks** without continuous input. Additionally, its performance depends on **high cache hit ratios**, meaning its efficiency deteriorates with fast-moving camera perspectives, where rapid scene changes reduce feature reuse and increase transmission overhead. FluxShard also incurs **memory and computation overhead**, as maintaining and referencing cached features demand additional resources, which may impact deployment on constrained edge devices. To address these limitations, future work will explore **adaptive memory management** to optimize cache usage on edge devices and **overhead optimizations** to further enhance computational and memory efficiency.

8 Conclusion

FluxShard introduces a motion-aware, block-centric framework for edge-cloud collaborative video analytics, leveraging motion vector-wrapped blocks to optimize computation, transmission, and state propagation. By dynamically aligning cached states with motion vectors, FluxShard effectively reduces redundant computation and bandwidth consumption while preserving temporal consistency under scene motion. Key innovations include an **asynchronous state update mechanism** for motion-sensitive computation, an **optimization-driven scheduling framework** guided by Effective Accuracy (EA), and **custom CUDA kernels** that efficiently bridge sparse motion-sensitive updates with dense GPU computation. By enabling low-latency, resource-efficient inference, FluxShard paves the way for scalable, real-time video analytics on resource-constrained mobile edge devices.

References

[1] Shahab S Band, Sina Ardabili, Mehdi Sookhak, Anthony Theodore Chronopoulos, Said Elnaffar, Massoud Moslehpour, Mako Csaba, Bernat Torok, Hao-Ting Pai, and Amir Mosavi. 2022. When smart cities get smarter via machine learning: An in-depth literature review. *IEEE Access* 10 (2022), 60985–61015.

[2] Sweta Bhattacharya, Siva Rama Krishnan Somayaji, Thippa Reddy Gadekallu, Mamoun Alazab, and Praveen Kumar Reddy Maddikunta. 2022. A review on deep learning for future smart cities. *Internet Technology Letters* 5, 1 (2022), e187.

[3] Jianguo Chen, Kenli Li, Qingying Deng, Keqin Li, and S Yu Philip. 2019. Distributed deep learning model for intelligent video surveillance systems with edge computing. *IEEE Transactions on Industrial*

Informatics (2019).

[4] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L Yuille. 2017. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *IEEE transactions on pattern analysis and machine intelligence* 40, 4 (2017), 834–848.

[5] Sichen Chen, Yingyi Zhang, Siming Huang, Ran Yi, Ke Fan, Ruixin Zhang, Peixian Chen, Jun Wang, Shouhong Ding, and Lizhuang Ma. [n. d.]. SDPose: Tokenized Pose Estimation via Circulation-Guide Self-Distillation. In *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2024-06-16). IEEE, 1082–1090. https://doi.org/10.1109/CVPR52733.2024.00109

[6] Sichen Chen, Yingyi Zhang, Siming Huang, Ran Yi, Ke Fan, Ruixin Zhang, Peixian Chen, Jun Wang, Shouhong Ding, and Lizhuang Ma. 2024. SDPose: Tokenized Pose Estimation via Circulation-Guide Self-Distillation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 1082–1090.

[7] John Estrada, Sidike Paheding, Xiaoli Yang, and Quamar Niyaz. 2022. Deep-learning-incorporated augmented reality application for engineering lab training. *Applied Sciences* 12, 10 (2022), 5159.

[8] Luyao Gao, Jianchun Liu, Hongli Xu, Sun Xu, Qianpiao Ma, and Liusheng Huang. 2024. Accelerating End-Cloud Collaborative Inference via Near Bubble-free Pipeline Optimization. *arXiv preprint arXiv:2501.12388* (2024).

[9] Jingning Han, Bohan Li, Debargha Mukherjee, Ching-Han Chiang, Adrian Grange, Cheng Chen, Hui Su, Sarah Parker, Sai Deng, Urvang Joshi, Yue Chen, Yunqing Wang, Paul Wilkins, Yaowu Xu, and James Bankoski. 2021. A Technical Overview of AV1. *Proc. IEEE* 109, 9 (2021), 1435–1462. https://doi.org/10.1109/JPROC.2021.3058584

[10] Arash Heidari, Nima Jafari Navimipour, and Mehmet Unal. 2022. Applications of ML/DL in the management of smart cities and societies based on new trends in information technologies: A systematic literature review. *Sustainable Cities and Society* 85 (2022), 104089.

[11] Yassine Himeur, Somaya Al-Maadeed, Hamza Kheddar, Noor Al-Maadeed, Khalid Abualsaud, Amr Mohamed, and Tamer Khattab. 2023. Video surveillance using deep transfer learning and deep domain adaptation: Towards better generalization. *Engineering Applications of Artificial Intelligence* 119 (2023), 105698.

[12] Junhwa Hur and Stefan Roth. 2020. Optical flow estimation in the deep learning age. *Modelling human motion: from human perception to robot design* (2020), 119–140.

[13] Glenn Jocher, Jing Qiu, and Ayush Chaurasia. 2023. Ultralytics YOLO. (Jan. 2023). https://github.com/ultralytics/ultralytics

[14] Hanbyul Joo, Tomas Simon, Xulong Li, Hao Liu, Lei Tan, Lin Gui, Sean Banerjee, Timothy Scott Godisart, Bart Nabbe, Iain Matthews, Takeo Kanade, Shohei Nobuhara, and Yaser Sheikh. 2017. Panoptic Studio: A Massively Multiview System for Social Interaction Capture. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

[15] Faisal Khan, Saqib Salahuddin, and Hossein Javidnia. 2020. Deep learning-based monocular depth estimation methods—a state-of-the-art review. *Sensors* 20, 8 (2020), 2272.

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

[17] Stefanos Laskaridis, Stylianos I Venieris, Mario Almeida, Ilias Leontiadis, and Nicholas D Lane. 2020. SPINN: synergistic progressive inference of neural networks over device and cloud. In *Proceedings of the 26th annual international conference on mobile computing and networking*. 1–15.

[18] Zheng Li, Yongcheng Wang, Ning Zhang, Yuxi Zhang, Zhikang Zhao, Dongdong Xu, Guangli Ben, and Yunxiao Gao. 2022. Deep learning-based object detection techniques for remote sensing images: A survey. *Remote Sensing* 14, 10 (2022), 2385.

[19] Shaoshan Liu, Liangkai Liu, Jie Tang, Bo Yu, Yifan Wang, and Weisong Shi. 2019. Edge Computing for Autonomous Driving: Opportunities and Challenges. *Proc. IEEE* 107, 8 (2019), 1697–1716. https://doi.org/10.1109/JPROC.2019.2915983

[20] Jinna Lv, Qi Shen, Mingzheng Lv, Yiran Li, Lei Shi, and Peiying Zhang. 2023. Deep learning-based semantic segmentation of remote sensing images: a review. *Frontiers in Ecology and Evolution* 11 (2023), 1201125.

[21] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B. Letaief. 2017. A Survey on Mobile Edge Computing: The Communication Perspective. *IEEE Communications Surveys & Tutorials* 19, 4 (2017), 2322–2358. https://doi.org/10.1109/COMST.2017.2745201

[22] Armin Masoumian, Hatem A Rashwan, Julián Cristiano, M Salman Asif, and Domenec Puig. 2022. Monocular depth estimation using deep learning: A review. *Sensors* 22, 14 (2022), 5353.

[23] Alican Mertan, Damien Jade Duff, and Gozde Unal. 2022. Single image depth estimation: An overview. *Digital Signal Processing* 123 (2022), 103441.

[24] Arzoo Miglani and Neeraj Kumar. 2019. Deep learning models for traffic flow prediction in autonomous vehicles: A review, solutions, and challenges. *Vehicular Communications* 20 (2019), 100184.

[25] Fábio Eid Morooka, Adalberto Manoel Junior, Tiago FAC Sigahi, Jefferson de Souza Pinto, Izabela Simon Rampasso, and Rosley Anholon. 2023. Deep learning and autonomous vehicles: Strategic themes, applications, and research agenda using SciMAT and content-centric analysis, a systematic review. *Machine Learning and Knowledge Extraction* 5, 3 (2023), 763–781.

[26] Khan Muhammad, Amin Ullah, Jaime Lloret, Javier Del Ser, and Victor Hugo C de Albuquerque. 2020. Deep learning for safe autonomous driving: Current challenges and future directions. *IEEE Transactions on Intelligent Transportation Systems* 22, 7 (2020), 4316–4336.

[27] Debargha Mukherjee, Jim Bankoski, Adrian Grange, Jingning Han, John Koleszar, Paul Wilkins, Yaowu Xu, and Ronald Bultje. 2013. The latest open-source video codec VP9 - An overview and preliminary results. In *2013 Picture Coding Symposium (PCS)*. 390–393. https://doi.org/10.1109/PCS.2013.6737765

[28] Sankar K Pal, Anima Pramanik, Jhareswar Maiti, and Pabitra Mitra. 2021. Deep learning in multi-object detection and tracking: state of the art. *Applied Intelligence* 51 (2021), 6400–6429.

[29] Mathias Parger, Chengcheng Tang, Christopher D Twigg, Cem Keskin, Robert Wang, and Markus Steinberger. 2022. 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*. 12497–12506.

[30] Grzegorz Pastuszak and Andrzej Abramowski. 2016. Algorithm and Architecture Design of the H.265/HEVC Intra Encoder. *IEEE Transactions on Circuits and Systems for Video Technology* 26, 1 (2016), 210–222. https://doi.org/10.1109/TCSVT.2015.2428571

[31] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martín Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems (NeurIPS)*. 8024–8035.

[32] Roberto Pierdicca, Flavio Tonetto, Marina Paolanti, Marco Mameli, Riccardo Rosati, and Primo Zingaretti. 2024. DeepReality: An open source framework to develop AI-based augmented reality applications. *Expert Systems with Applications* 249 (2024), 123530.

[33] Pablo Fernández Pérez, Claudio Fiandrino, and Joerg Widmer. [n. d.]. Characterizing and Modeling Mobile Networks User Traffic at Millisecond Level. In *Proceedings of the 17th ACM Workshop on Wireless Network Testbeds, Experimental evaluation & Characterization* (2023-10-02) *(WiNTECH '23)*. Association for Computing Machinery, 64–71.

https://doi.org/10.1145/3615453.3616509

[34] Imran Qureshi, Junhua Yan, Qaisar Abbas, Kashif Shaheed, Awais Bin Riaz, Abdul Wahid, Muhammad Waseem Jan Khan, and Piotr Szczuko. 2023. Medical image segmentation using deep semantic-based methods: A review of techniques, applications and emerging trends. *Information Fusion* 90 (2023), 316–352.

[35] Niri Rania, Hassan Douzi, Lucas Yves, and Treuillet Sylvie. 2020. Semantic segmentation of diabetic foot ulcer images: dealing with small dataset in DL approaches. In *Image and Signal Processing: 9th International Conference, ICISP 2020, Marrakesh, Morocco, June 4–6, 2020, Proceedings 9*. Springer, 162–169.

[36] Nikhila Ravi, Valentin Gabeur, Yuan-Ting Hu, Ronghang Hu, Chaitanya Ryali, Tengyu Ma, Haitham Khedr, Roman Rädle, Chloe Rolland, Laura Gustafson, Eric Mintun, Junting Pan, Kalyan Vasudev Alwala, Nicolas Carion, Chao-Yuan Wu, Ross Girshick, Piotr Dollár, and Christoph Feichtenhofer. 2024. SAM 2: Segment Anything in Images and Videos. *arXiv preprint arXiv:2408.00714* (2024). https://arxiv.org/abs/2408.00714

[37] Yulan Ren, Yao Yang, Jiani Chen, Ying Zhou, Jiamei Li, Rui Xia, Yuan Yang, Qiao Wang, and Xi Su. 2022. A scoping review of deep learning in cancer nursing combined with augmented reality: The era of intelligent nursing is coming. *Asia-Pacific Journal of Oncology Nursing* 9, 12 (2022), 100135.

[38] Zhe Ren, Junchi Yan, Bingbing Ni, Bin Liu, Xiaokang Yang, and Hongyuan Zha. 2017. Unsupervised deep learning for optical flow estimation. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 31.

[39] Stefano Savian, Mehdi Elahi, and Tammam Tillo. 2020. Optical flow estimation with deep learning, a survey on recent advances. *Deep biometrics* (2020), 257–287.

[40] GSDMA Sreenu and Saleem Durai. 2019. Intelligent video surveillance: a review through deep learning techniques for crowd analysis. *Journal of Big Data* 6, 1 (2019), 1–27.

[41] Xudong Sun, Pengcheng Wu, and Steven CH Hoi. 2018. Face detection using deep learning: An improved faster RCNN approach. *Neurocomputing* 299 (2018), 42–50.

[42] Aditya Tandon, Rajveer Shastri, Mantripragada Yaswanth Bhanu Murthy, Parismita Sarma, P.N. Renjith, and Masina Venkata Rajesh. 2022. Video streaming in ultra high definition (4K and 8K) on a portable device employing a Versatile Video Coding standard. *Optik* 271 (2022), 170164. https://doi.org/10.1016/j.ijleo.2022.170164

[43] T. Wiegand, G.J. Sullivan, G. Bjontegaard, and A. Luthra. 2003. Overview of the H.264/AVC video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology* 13, 7 (2003), 560–576. https://doi.org/10.1109/TCSVT.2003.815165

[44] Xiongwei Wu, Doyen Sahoo, and Steven CH Hoi. 2020. Recent advances in deep learning for object detection. *Neurocomputing* 396 (2020), 39–64.

[45] Enze Xie, Wenhai Wang, Zhiding Yu, Anima Anandkumar, Jose M Alvarez, and Ping Luo. 2021. SegFormer: Simple and efficient design for semantic segmentation with transformers. *Advances in neural information processing systems* 34 (2021), 12077–12090.

[46] Xizhou Zhu, Weijie Su, Lewei Lu, Bin Li, Xiaogang Wang, and Jifeng Dai. 2020. Deformable detr: Deformable transformers for end-to-end object detection. *arXiv preprint arXiv:2010.04159* (2020).