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. Existing reuse strategies often fail in dynamic scenes where motion and viewpoint changes disrupt naive content matching. 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 cache misses requiring recomputation. Around this abstraction, we propose **FluxShard** a motion-aware collaborative video analytics system, employing a series of techniques that prioritize cache updates according to task relevance, maintain feature freshness without stalling critical inference, and new sparse computation layout efficient on modern hardware. Implemented in C++/CUDA, FluxShard achieves up to 92% bandwidth reduction, 3.25× speedup, and ~99% accuracy retention on detection, segmentation, and pose estimation across diverse 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. Edge devices process data close to the sensor for immediate responsiveness, but are constrained by limited compute, memory, and energy. Leveraging the cloud's powerful resources is a natural way to overcome these constraints, yet network transfer and remote processing introduce additional latency that can undermine real-time performance. A promising way to mitigate this cost 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 has explored several paradigms for avoiding redundant computation, all of which can be interpreted as cache-like reuse at different granularities. 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 [30]) 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, which models feature reuse across consecutive frames as a *motion'induced cache'coherence problem*. The system's objective is to reach cache coherence for the latest frame in the shortest possible time—whether the fully aligned feature state is established on the edge or on the server—so that high accuracy inference can proceed without stalling. Motion between frames acts as a shifting index on the feature cache: spatially aligned regions are quick hits, while misaligned regions manifest as motion'induced cache misses that must be fully resolved on at least one side before inference continues. For each frame, this entails deciding which side should handle the entire miss resolution, thereby balancing heterogeneous compute and network conditions to achieve the best latency'accuracy trade'off.

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, 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.

The second challenge is to keep the cache fresh over time. Prioritizing only critical misses preserves immediate accuracy, but unrefreshed blocks gradually degrade quality. Yet refreshing competes with the latency-critical path — foreground critical inference on either the edge or the server — for compute and bandwidth. We adopt an *opportunistic, role-adaptive freshness maintenance strategy*: when one side runs foreground critical inference, the other uses its idle resources to upload cache blocks or locally compute updates in the background, ensuring freshness without ever stalling the active critical path.

The third challenge is that the irregular computation patterns of sparse workloads are unfriendly to the throughput-oriented design of modern hardware. A common "vanilla" workaround structures the sparsity by stacking small blocks along the batch dimension before convolution, making the computation strictly proportional to the amount of active data. However, this still leaves considerable performance untapped due to suboptimal hardware utilization. We propose a new *H-W stacked sparse-computation layout* that further stacks along the spatial (H-W) dimensions, together with a customized convolution operator co-designed for this layout. This structured design improves kernel-level efficiency by maximizing data locality and parallel utilization, achieving substantial acceleration.

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 sparse-computation functions, 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 on the **SA-V** dataset for high-resolution environments, and SDPose [6] for multi-person keypoint detection on the **Panoptic** dataset. The baselines include Naive Offloading, SPINN [17], COACH [8], and DeltaCNN [30], 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 makes the following contributions. We cast feature reuse in dynamic video analytics as a **motion-induced cache-coherence problem**, and propose a lightweight, MV-guided abstraction to align cached features and isolate regions requiring true recomputation. Building on this, we design **FluxShard**, a motion-aware edge-cloud collaborative system to resolve motion-induced cache misses with minimal latency. FluxShard incorporates three key techniques: (i) *salience-guided miss prioritization* from projected prior predictions; (ii) *opportunistic role-adaptive freshness maintenance* overlapping background updates with active critical inference; and (iii) an *H-W stacked sparse layout* with custom GPU kernels for efficient high-sparsity execution. 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 [25–27], augmented reality [7, 33, 38], video surveillance [3, 11, 41], 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 [19, 29, 45], semantic segmentation [21, 35, 36], optical flow estimation [12, 39, 40], and depth estimation [15, 23, 24], 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 [42] 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 [47] and SegFormer [46], 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 [43]. 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 [20]. However, cloud offloading introduces 50~200 ms round-trip delays [22], 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 [30], 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 [44], H.265/HEVC [31], VP9 [28], 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 environments, such as robots, drones, or other autonomous systems continuously conducting tasks like surveillance or navigation. These edge devices stream high-definition video to a GPU server over bandwidth-limited and variable wireless uplinks (e.g., 4G, 5G, or Wi-Fi) while adhering to stringent latency constraints (e.g., under 100 ms). FluxShard addresses the challenges posed by limited edge computation, fluctuating wireless bandwidth, and the need for scalable, high-accuracy video processing.

3.1 Core Components

FluxShard consists of the following core components (also depicted in Fig. 1):

1. Motion Block Extractor: A component deployed on the edge that analyzes incoming video frames to identify motion-aware regions (motion blocks). The motion block extractor operates by comparing the video frame against two states: the *Local State*, representing intermediate features from the edge's most recent background dense inference, and the *Proxy Server State*, a loosely synchronized approximation of the server's state. Based on this comparison, the motion block extractor generates motion-triggered blocks, which include:

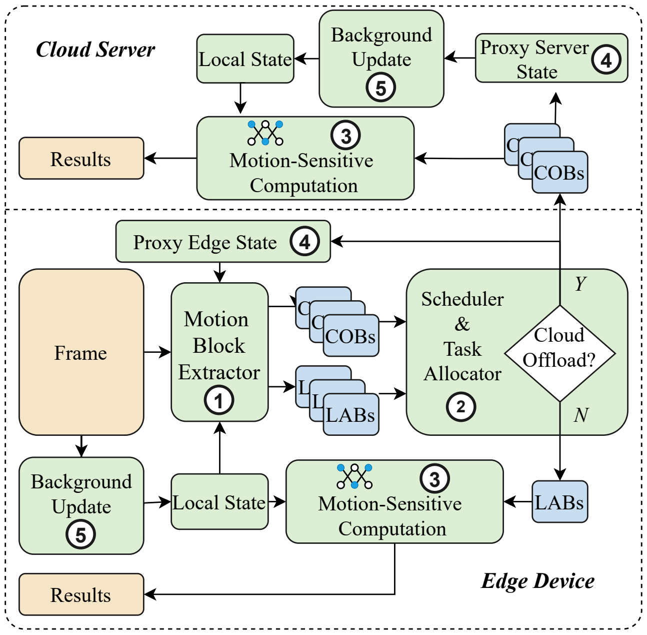

Figure 1: Overview of the workflow of FluxShard.

- *Locally Active Blocks (LABs):* Blocks requiring local sparse inference.
- *Cloud Offload Blocks (COBs):* Blocks to be transmitted to the server for inference.

2. Scheduler and Task Allocator: A collaborative module that operates primarily on the edge to determine task allocation between the edge and the cloud. Using profiled latency, bandwidth data, and the detected local active blocks and cloud offload blocks, the scheduler selects a fraction of these two blocks which optimize the achievable Effective Accuracy (EA) at both the edge (affected by local computation resources) and the cloud (affected by transmission latency and computation resources). Note EA is a metric balancing inference accuracy, resource usage, and communication delay. At the end, the one strategy of the two achieving higher EA is selected to achieve the optimal overall accuracy and latency tradeoff.

3. Motion-Sensitive Computation Engine: A lightweight backend on both the edge and the server that operates on the motion-triggered regions to execute motion sensitive computation (i.e., motion-vector-guided sparse inference) efficiently. The execution leverages the locally active blocks or cloud offload blocks, depending on task allocation, and uses optimized CUDA kernels on both the edge and server. Motion-sensitive computation works in concert with the local state (e.g., reusable computation intermediates aligned with motion vectors) to deliver up-to-date analytics without the need of up-to-date local state.

4. Loosely Synchronized Proxy States: Two proxy states are maintained to handle asynchrony between edge and cloud: 1. *Proxy Server State:* maintained on the edge, representing the server's inference state. It is updated by incrementally bending motion vectors and sparse updates transmitted from the edge into the initial state; 2. *Proxy Edge State:* maintained on the server, representing the edge's inference state. It is updated by applying the incoming edge transmissions sequentially to approximate the edge's computation state. Note that both state updates are transmission-driven and starting from a shared starting input. In this way, the two proxy states will remain synchronized.

5. Background Update: A preemptible task (background update in Fig. 1) on both the edge and the server that conduct dense inference on the latest available input (the latest input frame for the edge device and the latest proxy edge state for the server) to update the local state on the edge or the cloud. These updates ensure correctness by refining the state on dense input over time, avoiding dense processing overhead in the inference-critical path.

3.2 Overall Workflow

With all these components, as show in Fig. 1 the overall system works as: upon receiving a new video frame, the edge applies the motion block extractor to identify locally active blocks and cloud offload blocks by analyzing motion relative to the local state and the proxy server state. The scheduler determines the optimal computation placement, deciding the fraction of blocks to be processed locally or offloaded to the cloud based on effective accuracy, while accounting for latency and bandwidth constraints. Motion-sensitive computation engines at the edge and the cloud process the assigned blocks and produce inference results. Meanwhile, the background update thread incrementally updates the local state to ensure correctness over time without disrupting real-time processing. Proxy states on the edge and server are updated asynchronously during communication, providing efficient synchronization across the system.

4 Design

FluxShard introduces an efficient framework for real-time, distributed video analytics in edge-cloud environments. This section presents the detailed design of its core components and their interactions, including motion block extraction, scheduling, motion-sensitive computation, state management, and the integrated system workflow. The design aims to achieve low-latency and high-accuracy video processing, addressing key challenges such as constrained edge computational resources, fluctuating bandwidth, and the stringent latency requirements of modern autonomous systems.

4.1 Motion Block Extraction

The motion block extractor is responsible for identifying regions in incoming video frames that require computation either on the edge or on the cloud. The extractor utilizes a hierarchical block matching process to identify regions where pre-computed features from prior states can be reused based on motion vector alignment and dissimilarity thresholds. Blocks that do not align well with motion vectors or exhibit significant differences are designated as requiring updates through computation.

Hierarchical Block Matching Let F_t denote the incoming video frame at time t, and let S_{local} and S_{proxy} represent the Local State and Proxy Server State, respectively. Each video frame F_t is divided into non-overlapping blocks of size $h \times w$, denoted as $B^t_{i,j}$, where (i, j) identifies the block's location in the grid.

For a block $B^t_{i,j}$, hierarchical block matching determines whether features from a reference state S (either S_{local} or S_{proxy}) can be reused by minimizing a dissimilarity metric D while considering motion vector alignment. Formally:

$$B^t_{i,j} \rightarrow \arg \min_{B' \in \mathcal{N}(B^t_{i,j})} D\big(B^t_{i,j}, B'\big),$$

where $\mathcal{N}(B^t_{i,j})$ represents the search neighborhood for block $B^t_{i,j}$ in state S, guided by its predicted motion vector. The dissimilarity metric D is computed as:

$$D(B_1, B_2) = \frac{1}{hw} \sum_{x=1}^{h} \sum_{y=1}^{w} \big|B_1(x, y) - B_2(x, y)\big|,$$

where $B_1(x, y)$ and $B_2(x, y)$ represent pixel intensities in blocks B_1 and B_2, respectively.

To minimize computational cost, hierarchical block matching employs:

- **Coarse Matching:** Conducted at lower resolution, allowing efficient estimation of initial alignment.
- **Fine Matching:** Refinement at higher resolution for precise determination of alignment.

Integrating Motion Vectors for Reuse Motion vectors, derived from consecutive frames, are used to project block positions and guide matching within $\mathcal{N}(B^t_{i,j})$. A block $B^t_{i,j}$ in F_t is considered "motion-aligned" if its best match in S satisfies:

$$D(B^t_{i,j}, B^{'}) < \tau,$$

where τ is a dissimilarity threshold. In such cases, the block is deemed reusable, and its pre-computed features from S are directly propagated to the current frame, avoiding redundant computation or communication.

LAB and COB Classification For motion-triggered blocks exhibiting high dissimilarity ($D \geq \tau$) and thus misaligned with motion vectors, updates are required. These blocks are classified as follows:

- **Locally Active Blocks (LABs):** Blocks that will be computed on the edge due to computational feasibility and latency constraints.
- **Cloud Offload Blocks (COBs):** Blocks requiring offloading to the server for computation, often due to complexity or limited edge resources.

The extractor outputs LABs and COBs (sorted in descend order based on their dissimilarity metric) alongside metadata such as block coordinates and dissimilarity scores. These classifications feed directly into downstream components for scheduling and computation.

4.2 Scheduling and Task Allocation

The scheduler in FluxShard is designed to allocate motion-triggered blocks to either the edge or cloud, optimizing for effective accuracy (EA) while adhering to system latency and resource constraints. The decision process is grounded in a mathematical optimization framework that depends on several assumptions for latency modeling and system behavior.

Effective Accuracy (EA) Objective. We aim to maximize EA, a metric incorporating both accuracy and latency trade-offs:

$$\text{EA} = \left(\frac{A \cdot k}{N}\right) \cdot e^{-\lambda T},$$

where A is the per-block inference accuracy (empirically set to the overall model accuracy), k is the number of allocated blocks, N is the maximum processable block (LABs or COBs) count, T is the latency of either local computation or transmission latency and cloud computation, and λ is the decay coefficient quantifying time sensitivity.

Assumptions for Latency Modeling. To simplify analysis and tractably model the behavior of the edge-cloud system, we make the following assumptions:

- **Quadratic scaling of compute time:** The per-shard compute time grows quadratically with the number of shards, incorporating effects of resource contention, memory bandwidth saturation, and task parallelization limits.
- **Independent processing pipelines:** The edge and cloud process shards independently, without interdependencies that might introduce task synchronization delays.
- **Stable bandwidth and shard sizes:** Shard transmission delays are modeled as linear with respect to their size (S) and available bandwidth (B), assuming temporal stability in network conditions.
- **Negligible queuing effects:** Processing overhead (τ_{wait}) including the motion block extraction and scheduling process is treated as a fixed parameter and does not significantly fluctuate based on system load.

Latency Modeling. Under these assumptions, the total latency for edge and cloud computation is modeled as follows:

- **Cloud Latency:**

$$T_{\text{cloud}} = \tau_{\text{wait}} + \frac{S \cdot k_{\text{cloud}}}{B} + \underbrace{a_c k_{\text{cloud}}^2 + b_c k_{\text{cloud}} + c_c}_{\text{Cloud Compute Time}},$$

where a_c, b_c, c_c are coefficients characterizing the cloud's quadratic compute scaling.

- **Edge Latency:**

$$T_{\text{edge}} = \tau_{\text{wait}} + \underbrace{a_e k_{\text{edge}}^2 + b_e k_{\text{edge}} + c_e}_{\text{Edge Compute Time}},$$

where a_e, b_e, c_e represent the edge's resource-constrained compute dynamics.

Optimization Problem. The scheduler seeks to allocate motion-triggered blocks (k) optimally between the cloud and the edge by maximizing the *Effective Accuracy* (EA), while considering latency constraints and system dynamics. The unified optimization problem for the edge and the cloud is defined as:

$$\max_k \text{EA} = \left(\frac{A \cdot k}{N}\right) \cdot e^{-\lambda T},$$

Closed-form solutions for optimal allocations are obtained by solving $\frac{d(\ln \text{EA})}{dk} = 0$, yielding:

$$k_{\text{cloud}}^* = \frac{-\left(b_c + \frac{S}{B}\right) + \sqrt{\left(b_c + \frac{S}{B}\right)^2 + \frac{8a_c}{\lambda}}}{4a_c}. \tag{1}$$

$$k_{\text{edge}}^* = \frac{-b_e + \sqrt{b_e^2 + \frac{8a_e}{\lambda}}}{4a_e}. \tag{2}$$

Decision Process. With the optimal shard allocations derived, the scheduler computes the corresponding effective accuracy for edge and cloud processing:

$$\text{EA}_{\text{cloud}} = \left(\frac{A_c \cdot k_{\text{cloud}}^*}{N_c}\right) \cdot e^{-\lambda T_{\text{cloud}}},$$

$$\text{EA}_{\text{edge}} = \left(\frac{A_e \cdot k_{\text{edge}}^*}{N_e}\right) \cdot e^{-\lambda T_{\text{edge}}}.$$

The scheduler dynamically selects the allocation strategy (cloud or edge) with the higher EA_{cloud} or EA_{edge}, enabling adaptive task offloading and computation based on workload conditions, network performance, and system resource availability, which best trade of between inference accuracy and latency. Note that while the modeling framework treats Locally Active Blocks (LABs) and Cloud Offload Blocks (COBs) equivalently, in practice, these blocks are sorted based on their *dissimilarity metric*. This metric estimates the potential accuracy improvement when a block is updated, ensuring that the k_{edge} and k_{cloud} blocks selected for processing contribute maximally to the overall effective accuracy (EA).

4.3 Motion-Sensitive Computation Engine

Traditional sparse computation methods [18, 30] rely on a combination of gather and scatter operations to process unstructured sparse regions efficiently. The *gather* process extracts active regions from the input tensors based on predefined sparsity patterns. Additionally, it retrieves portions of the surrounding feature maps or patches to provide local context, ensuring that the extracted regions carry sufficient spatial information for downstream computation. This gathered data is then compactly organized into the batch dimension to streamline processing.

Following inference, the processed outputs are incorporated back into the dense tensor through the *scatter* operation, which maps results to their corresponding original spatial locations.

Building on this framework, our Motion-Sensitive Computation Engine introduces motion vector alignment to enhance the fidelity of sparse processing. Motion-triggered blocks are tiled along the batch dimension as computation propagates. To emulate correct dense inference behavior and provide global context, motion vectors of surrounding feature maps or patches are employed to bilinearly sample the actually displaced surrounding feature maps or patches. This is actually treating the precomputed intermediates as virtually wrapped by the motion vectors, which preserves temporal consistency and compensates for motion-induced spatial displacements while maintaining the computational efficiency of sparse processing. We realize such design on torch extension integrating highly optimized CUDA kernels.

4.4 State Management

Efficient state management in FluxShard is critical to ensuring accurate and temporally consistent real-time inference while addressing the latency and bandwidth constraints of edge-cloud environments. FluxShard employs a two-layer state management mechanism: 1. Local State Updates: to maintain and refine the edge or cloud device's inference state. 2. Proxy State Synchronization: to loosely synchronize the approximated inference states between the edge and server.

1. Local State Updates. The *local state* represents the edge or cloud device's most recent inference context, updated periodically or according to heuristics through background update (dense inference) in a preemptible way. These updates are triggered outside the critical path of real-time processing. For every update, the background update thread processes the latest available input frame and refines the local state accordingly. The lazy scheduling of these updates ensures that real-time tasks, such as processing Locally Active Blocks and motion-sensitive computation, are not interrupted. This mechanism maintains accuracy over time by eventually reconciling the intermediate sparse processing results with dense computations.

2. Proxy State Synchronization. To handle edge-cloud collaboration efficiently, FluxShard maintains two proxy states:

- **Proxy Server State:** This state, hosted on the edge, approximates the server's inference state. It is updated locally by integrating sparse updates transmitted for cloud offloaded computations (e.g., Cloud Offload Blocks) and motion-vector-driven interpolations. This mechanism enables the edge to make motion-informed predictions without requiring frequent updates from the server.
- **Proxy Edge State:** This state, hosted on the server, approximates the edge device's inference state. It is updated based on incoming edge transmissions, such as LAB inference results. This global view allows the server to complement its computations with spatio-temporal contexts observed at the edge.

Both proxy states are derived from a shared initialization point (e.g., the initial dense inference) and remain synchronized through this lightweight, transmission-driven updates. The asynchronous nature of these updates mitigates communication overhead while prioritizing motion-sensitive regions to maximize Effective Accuracy (EA). This approach enables the edge and server to maintain complementary but decoupled inferences, efficient for real-time video analytics.

Note that when local computation is selected, FluxShard opportunistically initiates the transmission of Cloud Offload Blocks (COBs) and associated motion vectors to the cloud in parallel with the local computation process, provided it can meet the transmission deadline. This parallel transmission not only leverages idle network bandwidth but also allows the cloud server to prefetch critical context for subsequent processing, thereby improving system responsiveness and enabling proactive resource utilization.

4.5 Overall Workflow

To ensure tight coordination between edge and cloud for real-time video analytics, FluxShard follows a structured workflow that integrates its key components: motion block extraction, task allocation, motion-sensitive inference, state management, and background updates. This workflow is summarized in Algorithm 1 and 2.

4.5.1 Edge-Side Workflow.

4.5.1 Edge-Side Workflow. Algorithm 1 presents the edge-side operations of FluxShard, responsible for real-time video frame processing. The edge maintains a local state $S_{\text{local}}^{\text{edge}}$ for inference and a proxy state $S_{\text{proxy}}^{\text{server}}$ (a synchronized view of the server's state) to coordinate with the server. Each incoming frame $\mathbf{F}_t$ triggers five main steps.

Algorithm 1: Edge-Side Workflow of FluxShard

```
Input: Incoming video frames $\{\mathbf{F}_t\}$ at each time $t$
Output: Inference results $\{\mathbf{R}_t\}$ for each frame
Initialization:
Initialize local state $S_{\text{local}}^{\text{edge}}$ and proxy states $S_{\text{proxy}}^{\text{server}}$;
foreach incoming frame $\mathbf{F}_t$ do
    Pause background update task;
    // Step 1: Motion Block Extraction
    Detect motion blocks using $S_{\text{local}}^{\text{edge}}$ and $S_{\text{proxy}}^{\text{server}}$;
    Classify blocks into Locally Active Blocks (LABs)
      and Cloud Offload Blocks (COBs);
    // Step 2: Task Allocation
    Compute predicted Effective Accuracy (EA) for
      two strategies:;
        $\mathcal{S}_1$: Process $k_1$ LABs locally, opportunistically
          transmit COBs within deadlines;
        $\mathcal{S}_2$: Transmit $k_2$ COBs to the cloud, defer LAB
          processing;
    // Select optimal strategy
    $\mathcal{S}^* = \arg\max\{\text{EA}(\mathcal{S}_1), \text{EA}(\mathcal{S}_2)\}$;
    // Step 3: Motion-Sensitive Computation
    if $\mathcal{S}^* = \mathcal{S}_1$ then
        Motion-sensitive computation on $k_1$ LABs to
          get results;
        Opportunistically Transmit $k_2$ COBs and
          motion vectors in parallel;
    else
        Transmit $k_2$ COBs to the server;
    // Step 4: State Management
    Update $S_{\text{proxy}}^{\text{server}}$ with transmitted COBs and motion
      vectors;
    // Step 5: Background Update
    Resume or trigger background update task on $\mathbf{F}_t$
      to update $S_{\text{local}}^{\text{edge}}$, if resources permit;
```

First, *motion block extraction* identifies Locally Active Blocks (LABs) and Cloud Offload Blocks (COBs) based on motion evaluation. Then, *task allocation* compares two strategies ($\mathcal{S}_1$ local computation vs. $\mathcal{S}_2$ offloading) using predicted Effective Accuracy (EA) and selects the optimal one. Depending on the strategy, the edge runs *motion-sensitive computation*, either processing LABs locally while opportunistically transmitting COBs or offloading computation (COBs) to the server. In *state management*, proxy updates ensure synchronization with the server. Finally, a background task refines $S_{\text{local}}^{\text{edge}}$ using idle resources, improving future frame processing. This

Algorithm 2: Server-Side Workflow of FluxShard

```
Input: Cloud Offload Blocks (COBs), Sparse Inference
       Requests, Motion Vectors from Edge
Output: Inference results, updated proxy edge state
       $S_{\text{proxy}}^{\text{edge}}$
Initialization:
Initialize local state $S_{\text{local}}^{\text{server}}$ and proxy edge state $S_{\text{proxy}}^{\text{edge}}$;
while FluxShard system is active do
    Pause background update task;
    // Step 1: Receive Data from Edge
    Receive incoming COBs, motion vectors, and
      sparse inference request flag;
    // Step 2: Update Proxy Edge State
    Merge COBs and motion vector into $S_{\text{proxy}}^{\text{edge}}$;
    // Step 3: Sparse Inference
    if sparse inference is requested then
        Execute motion-sensitive computation on
          COBs to get results;
    // Step 4: Background Update
    Resume or trigger background update task on
      $S_{\text{proxy}}^{\text{edge}}$ to update $S_{\text{local}}^{\text{server}}$, if resources permit;
```

workflow optimizes latency-sensitive task distribution while adapting to resource constraints.

4.5.2 Server-Side Workflow. Algorithm 2 outlines the server's role as a collaborative backend, reacting to edge-provided offloaded data. The server manages $S_{\text{proxy}}^{\text{edge}}$ (a synchronized view of the edge's state) and $S_{\text{local}}^{\text{server}}$.

Upon receiving COBs and motion vectors from the edge, the server updates $S_{\text{proxy}}^{\text{edge}}$ to maintain consistency. If sparse inference is requested, the server performs motion-sensitive computation on the specified COBs to get inference results. A periodic *background update* refines the server's local state $S_{\text{local}}^{\text{server}}$ using the latest $S_{\text{proxy}}^{\text{edge}}$. This workflow focuses on proxy synchronization, minimal offload latency, and resource-aware refinement, enabling efficient edge-cloud collaboration. Together, these workflows ensure real-time processing, low-latency interactions, and adaptive resource utilization.

5 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 [32] 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. This mechanism ensures efficient data transfers, including Cloud Offload Blocks (COBs), motion vectors, and metadata required for server-side inference. 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.

6 Evaluation

6.1 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 [34] 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 [37])** 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.

6.2 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** [30] 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.

6.3 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 2 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.

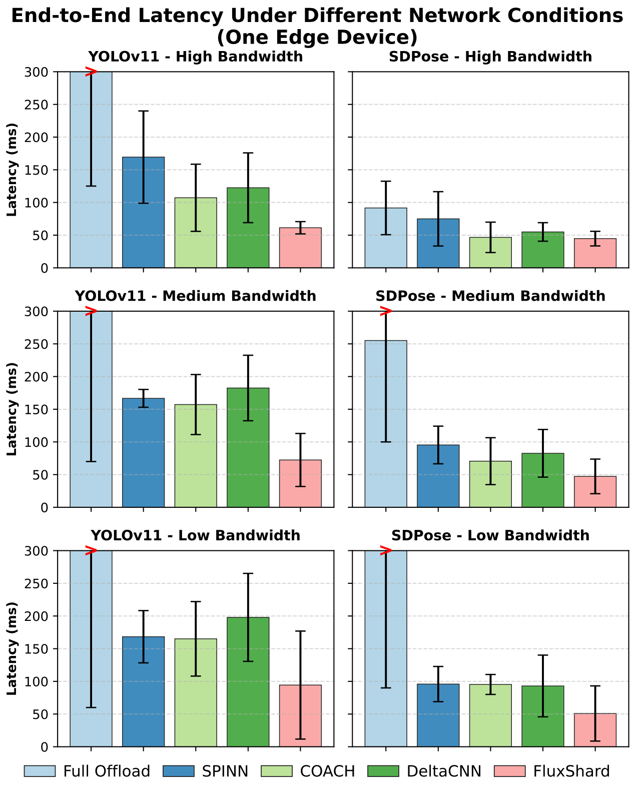

Figure 2: Latency comparison under different bandwidth conditions. FluxShard consistently outperforms the other non-vanilla baselines, particularly in constrained network environments.

Power Consumption. Figure 3 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.

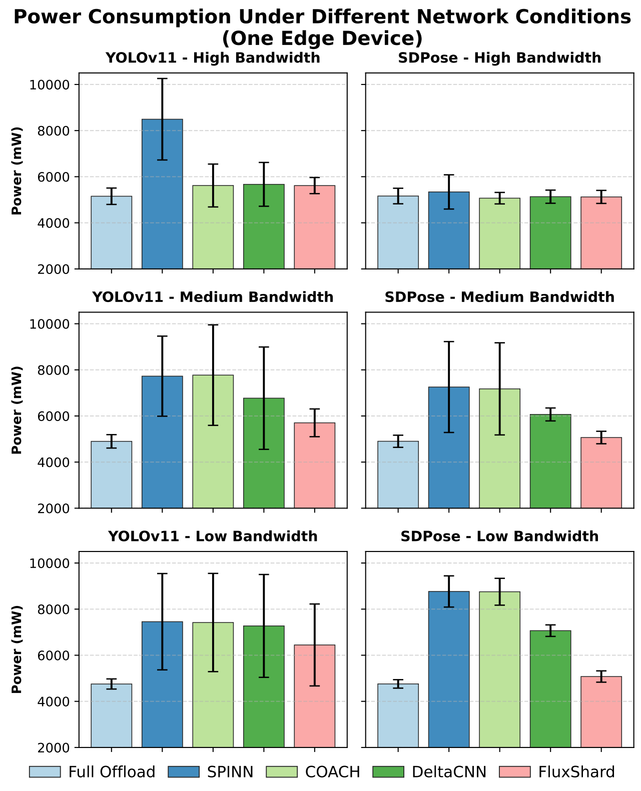

Figure 3: Power consumption comparison across different bandwidth conditions. FluxShard reduces power usage through efficient offloading and adaptive workload distribution.

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 4, 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.

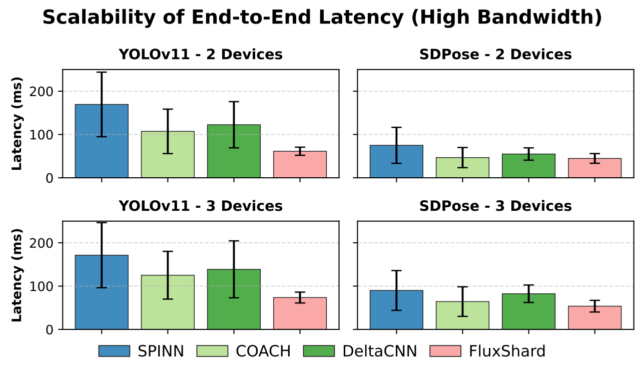

Figure 4: Scalability comparison of end-to-end latency in high-bandwidth conditions. FluxShard scales more efficiently, maintaining lower latency growth as devices increase.

6.4 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.

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 5 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 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 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)

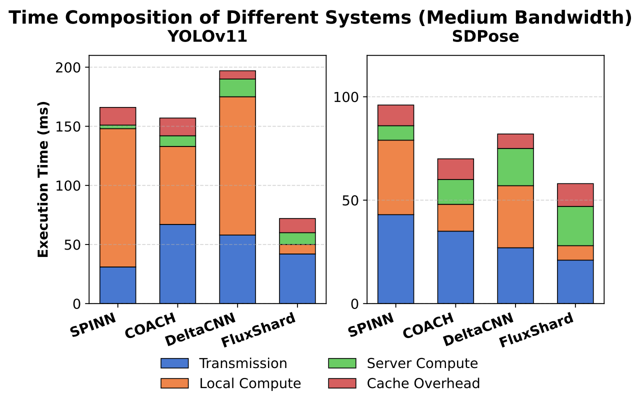

Figure 5: Execution time composition of SPINN, COACH, DeltaCNN, and FluxShard under medium bandwidth conditions for YOLOv11 and SDPose workloads.

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

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.

6.5 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.

7 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] Muyang Li, Ji Lin, Chenlin Meng, Stefano Ermon, Song Han, and Jun-Yan Zhu. [n. d.]. Efficient Spatially Sparse Inference for Conditional GANs and Diffusion Models. 45, 12 ([n. d.]), 14465–14480. https://doi.org/10.1109/TPAMI.2023.3316020 Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence.

[19] 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.

[20] 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

[21] 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.

[22] 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

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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

[29] 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.

[30] 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.

[31] 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

[32] 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.

[33] 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.

[34] 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

[35] 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.

[36] 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.

[37] 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

[38] 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.

[39] 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.

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

[41] 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.

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

[43] 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

[44] 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

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

[46] 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.

[47] 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).