FluxShard: A Motion-Driven Framework for Distributed Real-Time Dense Video Analytics

Paper 1203, 12 pages

ABSTRACT

Real-time dense video analytics in dynamic applications like autonomous vehicles and smart cities demands both low latency and high accuracy, but existing edge-cloud solutions suffer from bandwidth bottlenecks, temporal staleness, and inefficient state management. To address these challenges, we propose **FluxShard**, a motion-driven framework that introduces *motion vector-wrapped blocks* — a novel representation for partitioning video frames into motion-aware regions — to unify computation, transmission, and state propagation across edge and cloud.

FluxShard achieves three key innovations: (1) asynchronous state updates that ensure temporal consistency with minimal communication overhead, (2) an optimization-based scheduling framework leveraging a new *Effective Accuracy (EA)* metric to balance latency and accuracy in real time, and (3) custom CUDA kernels that bridge motion-sensitive sparsity with dense GPU execution for high performance. Evaluations on state-of-the-art models show FluxShard reduces bandwidth by up to **92%**, improves latency by **67%**, and achieves up to **32%** higher EA compared to baselines like SPINN and DeltaCNN, all while preserving **99.8%** of the original model's accuracy. FluxShard establishes a scalable and generalizable foundation for edge-cloud video analytics, with its implementation available as open-source.

1 INTRODUCTION

The increasing deployment of dense video analytics in intelligent systems, such as autonomous vehicles [31–33], augmented reality (AR) [8, 39, 44], and smart cities [2, 3, 14], has driven the need for **real-time dense visual inference**. These applications require inference results to be both low-latency (**sub-100ms** [25]) and high-accuracy in rapidly evolving environments. Edge-cloud collaborative inference [10, 21, 36] offers a promising solution by distributing computational workloadbetween resource-constrained edge devices and powerful cloud servers. However, real-time dense visual inference in this paradigm is hampered by: (i) prolonged inference times and unpredictable latency spikes due to limited bandwidth and long tail latency due to unstable network conditions in wireless communication environments, and (ii) accuracy degradation manifests as temporal staleness when inference results become outdated and less relevant to the rapidly evolving real-world environment.s.

Existing approaches for edge-cloud inference have evolved into two main categories: pipeline-based and delta-based frameworks. Pipeline-based systems (e.g., SPINN [21] and COACH [10]) distribute model computation by deploying initial layers on edge devices and remaining layers on cloud servers, requiring full feature map transmission for each inference request. However, this approach introduces significant delays and network bandwidth consumption since it fails to leverage temporal sparsity, a common characteristic in video streams where consecutive frames exhibit changes in only small portions of their content.

Delta-based frameworks (e.g., DeltaCNN [36]) take a more efficient approach by selectively updating only the regions that show temporal differences between frames. While this method effectively reduces bandwidth usage, it relies heavily on state propagation, which attempts to precisely align spatial positions between cached previous states and new inputs. This selective computation strategy compromises inference accuracy and proves ineffective in real-world scenarios with dynamic camera movements or scene changes, resulting in redundant computations and degraded temporal consistency. These limitations underscore the need for a more flexible granularity scheme in selective updating mechanisms.

To overcome their limitations, our key insight is that motion vectors [24], which are widely used in video compression [9, 11, 26], can be repurposed as a lightweight mechanism to coordinate computation, transmission, and state propagation, even in the presence of motion. Motion vectors map how regions in subsequent frames shift due to object or camera movement, and by wrapping cached states with these motion vectors, we can delay physical updates to precomputed states without sacrificing their validity. This unification of motion vector-wrapped representations addresses challenges in adapting to continuous scene dynamics; it ensures that regions of interest can be selectively transmitted, scheduled, or queried in alignment with their motion. Furthermore, the abstraction of frames as independently wrapped blocks allows the system to operate with fine-grained granularity, enabling precise and flexible resource management across edge and cloud.

To build on this insight, we propose **FluxShard**, a motion-aware, block-centric framework that redefines video frames as *motion vector-wrapped blocks.* This representation partitions frames into independent spatial regions, each annotated with motion vectors that describe their displacement across frames. These motion vector-wrapped blocks serve as a unifying abstraction across three critical components of the edge-cloud pipeline: (1) *Transmission*, where only motion-sensitive blocks are communicated, significantly reducing bandwidth usage; (2) *Computation Scheduling*, where block-level workloads are dynamically distributed between the edge and cloud to optimize latency-accuracy trade-offs; and (3) *State Query and Propagation*, where cached computation states are aligned using motion vectors to maintain temporal consistency even under dynamic scenes.

While motion vector-wrapped blocks enable efficient block-level processing, FluxShard presents several key implementation challenges that must be addressed rigorously:

Challenge 1: Maintaining State Consistency Under Motion-sensitive Updates. Motion-sensitive computation in FluxShard relies on treating computation states as "virtually wrapped" entities, aligned with motion vectors rather than physically updated during each frame. Over time, such reliance on virtual transformations may lead to state desynchronization, as inaccuracies accumulate across successive frames. To address this, we design an **asynchronous state update mechanism** in which motion vector-wrapped block updates are incorporated into the global state in the background, off the critical path of inference. This approach ensures temporal coherence without sacrificing the latency benefits of motion-sensitive computation.

Challenge 2: Scheduling for Latency-Accuracy Trade-offs. Effective scheduling in FluxShard must strike a system-level balance between accuracy and latency. This involves choosing whether to process motion-sensitive blocks on the cloud, which offers higher accuracy on the ability to process more blocks at the cost of transmission delays, or on the edge, which eliminates the transmission delay but incurs less accurate computation. To unify this decision-making process, FluxShard formulates the problem as an optimization objective, maximizing an **Effective Accuracy (EA)** metric. EA combines accuracy gains with temporal penalties, incorporating system parameters such as bandwidth, compute latency, and shard-level characteristics. By explicitly modeling compute latency using quadratic terms (e.g., accounting for memory contention and parallelization effects), the trade-off is mathematically optimized to determine the number of blocks processed on the cloud vs. the edge. A closed-form solution allows for real-time decision-making, ensuring that system resources are utilized efficiently while adapting dynamically to operating conditions.

Challenge 3: Achieving High-Performance motion-sensitive Computation. While the representation of frames as motion vector-wrapped blocks enables sparsity, direct implementations of motion-sensitive computation often underperform on modern hardware due to inefficient utilization of GPU memory and compute resources. To mitigate this inefficiency, FluxShard introduces **custom CUDA kernels** that transform motion-sensitive blocks into dense tensors. Specifically, these kernels enable *gathering* content relevant to computation and *recovering* motion vector-wrapped block results into a dense output format, ensuring compatibility with high-performance CUDA libraries such as cuDNN. This hybrid approach bridges the sparsity of motion-sensitive computation with the performance benefits of dense computation pipelines, significantly reducing latency.

We rigorously evaluate FluxShard using two state-of-the-art models representative of real-world video analytics tasks: *YOLOv11* [20], a CNN-based dense segmentation framework assessed on the SAM dataset for high-resolution environments, and *SDPose* [7], a keypoint detection model for structured annotations, tested on the Panoptic dataset. The evaluation framework compares FluxShard against baselines, including Naive Offloading, SPINN [21], COACH [10], and DeltaCNN [36], focusing on four key metrics: *Accuracy* and *Effective Accuracy (EA)*, 99th percentile latency, bandwidth usage, and compute load distribution across the edge (Jetson Xavier NX) and cloud (RTX 3080 PC). Results demonstrate that FluxShard outperforms baselines, reducing bandwidth usage (**up to 92%**), improving latency (**67% faster**), and maintaining competitive or superior accuracy (**maintaining 99.8% of accuracy of the original model and up to 32% higher EA**). The implementation incorporates scalability optimizations, including efficient CUDA kernels for motion vector-wrapped computation.

The main contribution of this paper is the **motion vector-wrapped block abstraction**, which represents video frames as independently updatable blocks annotated with motion vectors to efficiently manage computation, transmission, and state propagation across edge-cloud systems under dynamic scene conditions. Building on this abstraction, we propose **FluxShard**, a motion-aware framework that unifies edge-cloud collaboration through three key innovations: (1) an asynchronous state update mechanism that maintains temporal consistency while leveraging motion-sensitive updates to reduce communication overhead, (2) an optimization-driven scheduling framework guided by a novel Effective Accuracy (EA) metric, which mathematically balances latency and accuracy through real-time resource allocation, and (3) custom CUDA kernels that bridge sparsity and dense computation for efficient GPU execution. We validate FluxShard through extensive experimentation demonstrating significant improvements in bandwidth usage, latency, and Effective Accuracy compared to state-of-the-art baselines such as SPINN, COACH, and DeltaCNN. This work establishes motion vector-wrapped blocks as a scalable and generalizable abstraction for real-time video analytics in edge-cloud systems. FluxShard's code is released on https://github.com/sigcomm25paper1203/FluxShard.

2 BACKGROUND

Real-time video analytics has become a cornerstone of modern intelligent systems, powering applications such as autonomous vehicles [31–33], augmented reality [8, 39, 44], video surveillance [4, 15, 47], and smart city infrastructure [2, 3, 14]. 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 Dense Visual Analytics

Dense visual analytics tasks, such as object detection [23, 35, 51], semantic segmentation [27, 41, 42], optical flow estimation [16, 45, 46], and depth estimation [19, 29, 30], 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 [48] and DeepLab [5] 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 [53] and SegFormer [52], 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.

2.1.3 Broad Applicability and Evaluation. The spatial consistency retained in both CNNs and vision transformers makes them highly compatible with FluxShard's block-level processing approach. In this work, we demonstrate these capabilities through evaluation on representative CNN models (e.g., DeepLab [5], Faster R-CNN [48]) and transformer models (e.g., DETR [53], SegFormer [52]). Results show that FluxShard effectively optimizes computation and minimizes bandwidth usage while preserving the accuracy of these state-of-the-art models, showcasing its broad applicability across dense video analytics tasks.

2.2 Impact of Resource Constraints on Video Analytics

The distributed nature of edge-cloud architectures imposes stringent trade-offs between computation, bandwidth, and latency. Below, we outline key constraints with supporting quantitative evidence:

- **Bandwidth Bottlenecks:** High-definition video streams, such as 1080p at 30 FPS, demand significant bandwidth for transmission. Even with H.264 compression, bitrate requirements typically range from 8˜10 Mbps for standard-quality settings and up to 15˜20 Mbps for high-quality settings [49]. These requirements quickly scale unsustainably for multi-camera systems (e.g., 10 cameras require 80˜200 Mbps), especially in bandwidth-limited environments like rural areas or mobile networks, where uplink capacities often fall below 10 Mbps.
- **Edge Device Limitations:** Resource-constrained edge devices are ill-suited for real-time inference of compute-intensive models. For instance, YOLOv4 achieves only 5 FPS for 1080p input on a Jetson Nano [1], with near-maximal GPU utilization. Additionally, energy constraints in battery-powered devices further restrict their ability to sustain continuous high-resolution video processing.
- **Latency Sensitivity:** Real-time applications like autonomous driving require perception latencies below 50 ms, and surveillance systems demand sub-100 ms turnaround [25]. Meanwhile, cloud offloading introduces round-trip latencies of 50˜100 ms in urban settings and up to 200 ms in rural areas [28], making it challenging to meet these tight deadlines.

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.

2.3.1 Pipeline-Based Frameworks. Pipeline-based systems, such as SPINN [21] and COACH [10], decompose video analytics into distinct stages including frame sampling, feature extraction, and inference, which are split between the edge and the cloud. However, these frameworks rely on transmitting full or compressed intermediate data (e.g., feature maps or video frames) to the cloud for further processing, resulting in:

- *High Transmission Overhead:* Sending complete feature maps or full-frame data is costly in terms of bandwidth, introducing scalability challenges, particularly in multi-camera systems.
- *Redundant Processing:* They lack motion-awareness to selectively emphasize dynamic regions, leading to repeated processing of static parts of video frames.

2.3.2 Delta-Based Approaches. Delta-based systems, such as DeltaCNN [36], 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.

2.3.3 Cache-Based Methods. Cache-based frameworks, such as *FoggyCache* [12], attempt to reuse computation across similar inputs by caching and sharing results across edge-cloud devices. While these methods are effective for low-accuracy, recognition-based tasks (e.g., classification), they are poorly suited for dense, pixel-level visual workloads such as segmentation or keypoint detection. Dense video analytics demand precise spatial and contextual information, and thus their inference results cannot be directly reused or approximated. This renders cache-based approaches irrelevant for the dense visual tasks targeted in this paper.

2.4 Towards a Motion-Aware Abstraction

Despite progress in pipeline-based, delta-based, and cache-based methods, 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 [50], H.265/HEVC [37], VP9 [34], and AV1 [13] 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 dense 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

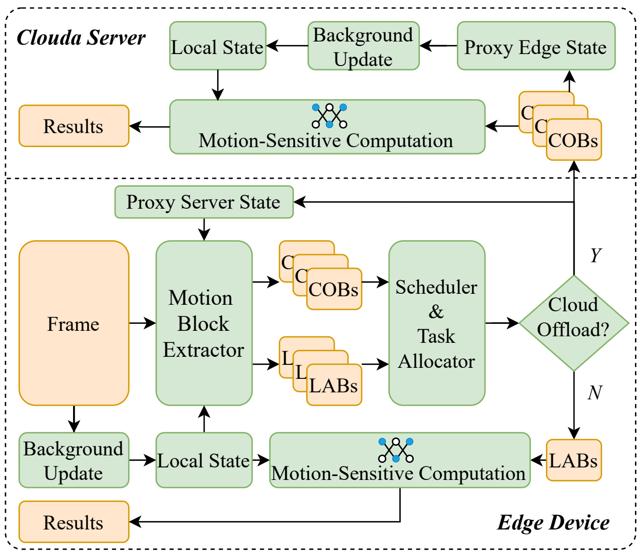

Figure 1: Overview of the workflow of FluxShard.

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:

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

2. 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.

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. 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.

5. 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.

3.2 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:

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

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:**

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

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

- **Edge Latency:**

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

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:

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

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

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

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

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

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

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

The scheduler dynamically selects the allocation strategy (cloud or edge) with the higher $\mathrm{EA}_{\mathrm{cloud}}$ or $\mathrm{EA}_{\mathrm{edge}}$, 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 [22, 36] 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. Algorithm 1 presents the edge-side operations of FluxShard, responsible for real-time video frame processing. The edge maintains a local state $\mathbf{S}^{\text{edge}}_{\text{local}}$ for inference and a proxy state $\mathbf{S}^{\text{server}}_{\text{proxy}}$ (a synchronized view of the server's state) to coordinate with the server. Each incoming frame $\mathbf{F}_t$ triggers five main steps.

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 $\mathbf{S}^{\text{edge}}_{\text{local}}$ using idle resources, improving future frame processing. This 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 $\mathbf{S}^{\text{edge}}_{\text{proxy}}$ (a synchronized view of the edge's state) and $\mathbf{S}^{\text{server}}_{\text{local}}$.

Upon receiving COBs and motion vectors from the edge, the server updates $\mathbf{S}^{\text{edge}}_{\text{proxy}}$ to maintain consistency. If sparse

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;

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;

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 [38] 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 Setup

Testbed. We evaluate FluxShard on a hybrid edge-cloud testbed that consists of multiple edge devices and a central cloud server. The edge devices include up to three NVIDIA Jetson Xavier NX systems, each equipped with a Volta GPU containing 384 CUDA cores, Tensor Cores, and 8 GB of

LPDDR4 memory. All devices run Ubuntu 20.04 with CUDA 11.4 and are capable of performing local inference and edge-side computations of FluxShard. The cloud server is equipped with an NVIDIA RTX 3080 GPU with 10 GB of GDDR6X memory, running Ubuntu 20.04 with CUDA 11.3. The server provides additional computational capacity to offload and execute more complex tasks that exceed the Jetson's resources. Devices and the server are connected to a gigabit Ethernet switch via 1000 Mbps Ethernet links for reliable local communication.

To emulate wireless wide-area network conditions, we use bandwidth-limited traces derived from the Madrid LTE Dataset [40]. These traces model typical conditions in real-world LTE/5G systems and are categorized based on their average bandwidth:

- **High Bandwidth:** 130 Mbps, representing optimal LTE/5G conditions without congestion.
- **Medium Bandwidth:** 56 Mbps, simulating moderately loaded cellular networks.
- **Low Bandwidth:** 25 Mbps, approximating heavily congested or degraded network states.

These traces are applied during evaluation using the Linux `tc` utility, which enforces both bandwidth restrictions and latency profiles corresponding to the trace conditions.

Models and Datasets. To evaluate the performance and robustness of FluxShard, we use two diverse models and datasets:

- **YOLOv11 [17] (Segment Anything Video (SA-V) dataset [43]):** A state-of-the-art convolutional neural network (CNN) with 22.4M parameters for dense segmentation. This model operates on 640×640-resolution input images, and we use the SA-V dataset, which includes real-world captured videos of various scenarios with high-quality spatio-temporal segmentation masks.
- **SDPose [6] (Panoptic dataset [18]):** A lightweight Transformer-based model with 6M parameters designed for human keypoint detection at 192×256-resolution input images. The Panoptic dataset features continuous videos of human activities suitable for tasks of human keypoint detection.

We list their average inference performance profile results on the edge and the server as the following table 1.

6.2 Baselines for Evaluation

We compare the proposed system against four baselines: (1) **Full Offload**, a naive approach that performs all computation on the server, incurring high communication costs; (2) **SPINN** [21], another split DNN framework with static partitioning but no compression, making it less adaptable to workload variability; (3) **COACH** [10], a split DNN framework that partitions computation between edge and server while applying quantization to reduce transmission overhead, but without motion-specific optimizations; and (4) **DeltaCNN** [36], which processes only pixel-level deltas between frames, optimizing for low-motion scenarios but struggling with high-motion workloads. These baselines cover diverse paradigms, from server-centric processing to motion-agnostic split DNN designs, allowing a holistic evaluation of the proposed system's adaptability and performance.

Table 1: Profiled Dense Inference Performance and Latency Modeling Parameters for Motion-Sensitive Computation (Block Size: 16×16**)**

Parameter	YOLOv11	SDPose
Edge Latency (s)	2.09e-01	1.32e-01
Server Latency (s)	1.19e-02	1.58e-02
Quadratic Coefficient (edge)	3.60e-08	4.96e-07
Linear Coefficient (edge)	2.13e-04	6.64e-04
Constant (edge)	8.29e-03	7.72e-03
Quadratic Coefficient (cloud)	2.63e-09	7.33e-08
Linear Coefficient (cloud)	1.52e-05	6.23e-05
Constant (cloud)	7.14e-05	4.12e-03

Metrics. FluxShard's performance is measured using the following key metrics:

- **Accuracy:** We use the pixel-level *Intersection over Union (IoU)* for YOLOv11 and *Keypoint Mean Average Precision (mAP)* for SDPose as the accuracy metrics. IoU models how the predicted segmentation overlaps with the groundtruth segmentation and mAP estimates the distances from predicted key points to the groundtruth key points. To ensure fair comparisons across different systems, both metrics are normalized by dividing the obtained accuracy by the ideal accuracy, which corresponds to the results achieved with complete groundtruth data and no system-imposed constraints.
- **Effective Accuracy (EA):** The whole scene level effective accuracy that captures the trade-off between task accuracy and real-time performance. Computed as $\text{EA} = \text{Accuracy} \cdot e^{-\lambda T}$.
- **Latency:** The 99th percentile end-to-end frame processing time, reflecting worst-case response times for real-time scenarios.
- **Bandwidth Usage:** The average transmission bandwidth (in Mbps) required to offload data between edge devices and the cloud server.
- **Compute Load Distribution:** Tracks the division of computational workloads between Jetson Xavier NX

devices and the cloud server, expressed as a fraction of total compute cycles handled at the edge.

6.3 End-to-End Results

motion block extraction yolov11 3.12ms; sdpose 1.23ms motion yolov11 3.12ms; sdpose 1.23ms

7 CONCLUSION

REFERENCES

[1] Ali Ahmadinia and Jaabaal Shah. 2022. An Edge-based Real-Time Object Detection. In *2022 21st IEEE International Conference on Machine Learning and Applications (ICMLA)*. 465–470. https://doi.org/10.1109/ICMLA55696.2022.00075

[2] 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.

[3] 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.

[4] 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).

[5] 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.

[6] 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

[7] 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.

[8] 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.

[9] Borko Furht, Joshua Greenberg, and Raymond Westwater. 2012. *Motion estimation algorithms for video compression*. Vol. 379. Springer Science & Business Media.

[10] 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).

[11] Stefan Grewatsch and Erika Miiller. 2004. Sharing of motion vectors in 3D video coding. In *2004 International Conference on Image Processing, 2004. ICIP'04.*, Vol. 5. IEEE, 3271–3274.

[12] Peizhen Guo, Bo Hu, Rui Li, and Wenjun Hu. 2018. FoggyCache: Cross-Device Approximate Computation Reuse. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking (MobiCom '18)*. Association for Computing Machinery, New York, NY, USA, 19–34. https://doi.org/10.1145/3241539.3241557

[13] 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

[14] 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.

[15] 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.

[16] 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.

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

[18] 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*.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] Bede Liu and Andre Zaccarin. 1993. New fast algorithms for the estimation of block motion vectors. *IEEE Transactions on Circuits and Systems for Video technology* 3, 2 (1993), 148–157.

[25] 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

[26] Jiancong Luo, Ishfaq Ahmad, Yongfang Liang, and Viswanathan Swaminathan. 2008. Motion estimation for content adaptive video compression. *IEEE Transactions on Circuits and Systems for video Technology* 18, 7 (2008), 900–909.

[27] 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.

[28] 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

[29] 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.

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

[31] 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.

[32] 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.

[33] 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.

[34] 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

[35] 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.

[36] 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.

[37] 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

[38] 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.

[39] 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.

[40] 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

[41] 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.

[42] 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.

[43] 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

[44] 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.

[45] 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.

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

[47] 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.

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

[49] 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

[50] 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

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

[52] 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.

[53] 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).