

EvGNN: An Event-driven Graph Neural Network Accelerator for Edge Vision

Yufeng Yang, Adrian Kneip, *Member, IEEE*, and Charlotte Frenkel, *Member, IEEE*

Abstract—Edge vision systems combining sensing and embedded processing promise low-latency, decentralized, and energy-efficient solutions that forgo reliance on the cloud. As opposed to conventional frame-based vision sensors, event-based cameras deliver a microsecond-scale temporal resolution with sparse information encoding, thereby outlining new opportunities for edge vision systems. However, mainstream algorithms for frame-based vision, which mostly rely on convolutional neural networks (CNNs), can hardly exploit the advantages of event-based vision as they are typically optimized for dense matrix-vector multiplications. While event-driven graph neural networks (GNNs) have recently emerged as a promising solution for sparse event-based vision, their irregular structure is a challenge that currently hinders the design of efficient hardware accelerators. In this paper, we propose EvGNN, the first event-driven GNN accelerator for low-footprint, ultra-low-latency, and high-accuracy edge vision with event-based cameras. It relies on three central ideas: (i) directed dynamic graphs exploiting single-hop nodes with edge-free storage, (ii) event queues for the efficient identification of local neighbors within a spatiotemporally decoupled search range, and (iii) a novel layer-parallel processing scheme enabling the low-latency execution of multi-layer GNNs. We deployed EvGNN on a Xilinx KV260 Ultrascale+ MPSoC platform and benchmarked it on the N-CARS dataset for car recognition, demonstrating a classification accuracy of 87.8% and an average latency per event of 16 μ s, thereby enabling real-time, microsecond-resolution event-based vision at the edge.

Index Terms—Event-based cameras, edge computing, graph neural networks (GNNs), neural network accelerators, field-programmable gate arrays (FPGAs).

I. INTRODUCTION

EDGE vision systems combine digital cameras and embedded computing to capture and process visual information within a limited power budget, typically ranging from milliwatts to a few watts [1]–[4]. While successful in a wide range of applications, from the Internet-of-Things to robotics [5]–[7], edge vision systems relying on conventional frame-based cameras are ill-suited for latency-critical scenarios requiring decisions within microseconds, such as autonomous drone navigation [8]. Indeed, standard frame-based cameras fail to meet these latency requirements as they typically capture 30–60 frames per second (FPS), while the power dissipation of 1000-FPS high-speed cameras can reach tens of watts [9], far exceeding the power budget of common edge vision systems.

Event-based cameras, also called silicon retinas [10] or dynamic vision sensors (DVSes) [11], have attracted growing interest in low-power high-speed edge vision applications. As opposed to their frame-based counterparts, which record absolute light intensity for every pixel in the field of view at a fixed sampling rate, event-based cameras are locally sensitive: their pixels are individually activated only when the received light intensity changes beyond a preset threshold [12]. When this threshold is exceeded, event packets containing the pixel address are generated. This event-driven scheme operating at the pixel level leads to (i) a temporal resolution on the order of microseconds, which helps alleviate motion blur, and (ii) a power consumption on the order of milliwatts thanks to sparsity [12]. Indeed, as no event is generated in the absence of light intensity changes, static background information is filtered out: only the moving objects are contained in the event stream.

As the standard neural network model for computer vision tasks, convolutional neural networks (CNNs) have demonstrated excellent performance for scenarios such as object recognition or detection [13]–[18]. However, since they are optimized for the mainstream frame-based vision paradigm, techniques for sparse event-based data are still at an early stage. Indeed, current event-based vision systems usually rely on a straightforward conversion of event streams into images by accumulating events from each pixel within a given time window, which is also known as the *dense-frame approach* [19]–[21]. While this conversion allows leveraging mature frame-based algorithms to process event-based data, accumulation windows on the order of milliseconds are necessary to obtain reasonable performance [20], thus discarding the original microsecond-level resolution of the event-based data.

As solving this challenge requires departing from frame-based vision, graph representations of event-based data have recently started being explored [22]–[24]. By regarding individual events as *nodes* and their spatiotemporal relationship as *edges*, a stream of events can be converted into a fully equivalent graph (*i.e. event graph*) without degrading the temporal resolution nor losing the inherent sparsity of the data. While this sparse event graph is well suited for processing by graph neural networks (GNNs) [25], [26], the standard approach consisting of first constructing the full event graph, and then processing it with a GNN, fails to exploit the event-driven nature of the data and is bound to millisecond-level latencies in dedicated hardware [27], [28]. To solve this challenge, techniques to construct and process the event graph in a *dynamic* fashion (*i.e.* on a per-event basis) have recently been proposed [23], [24], thereby enabling both sparsity-aware and event-driven processing. However, designing efficient hardware accelerators for these sparse, irregular, and fully data-driven workloads, is currently an open challenge.

Yufeng Yang was with the Microelectronics Department (EEMCS Faculty), Delft University of Technology, 2628 CD Delft, Netherlands. Adrian Kneip is with the Microelectronics Department (EEMCS Faculty), Delft University of Technology, 2628 CD Delft, Netherlands, and with the Department of Electrical Engineering, KU Leuven, 3000 Leuven, Belgium. Charlotte Frenkel is with the Microelectronics Department (EEMCS Faculty), Delft University of Technology, 2628 CD Delft, Netherlands. (Corresponding author: c.frenkel@tudelft.nl).

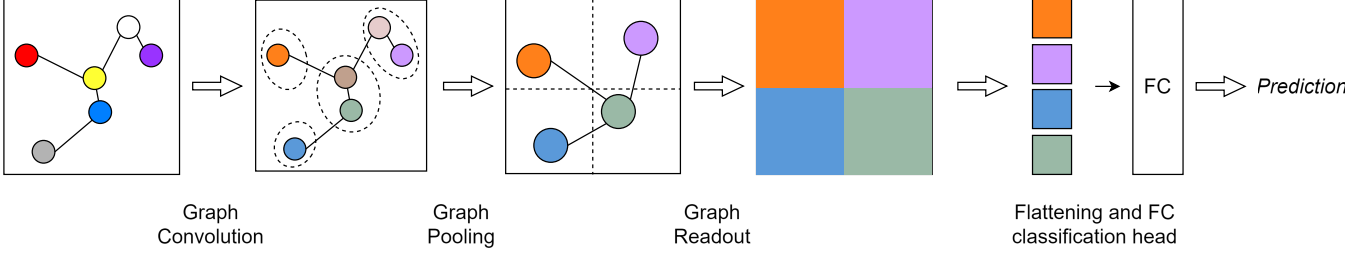


Fig. 1. Illustration of a typical event GNN pipeline. The colors of nodes represent their features. After the graph convolution, the features are updated (color changed). The graph pooling layer, where certain nodes are selected and edges are re-arranged, simplifies the graph structure through node down-sampling. The graph readout layer divides the graph into several grids, whose cells are assigned with node features. Finally, after flattening, the cell features are processed by the FC layer to derive the graph-level prediction results.

In this work, we propose EvGNN, the first event-driven GNN accelerator for edge vision at microsecond-level latencies that supports the end-to-end hardware acceleration of an event graph, from event-based input acquisition to dynamic event graph construction and real-time GNN inference. Our main contributions are as follows:

- We exploit the causality of event graphs through directed edges to achieve ultra-low-latency decisions by only processing the local subgraph of direct neighbors around a new event, preserving accuracy while enabling an edge-free storage that drastically reduces memory footprint.
- We adopt a hardware-friendly spatiotemporally decoupled prism neighbor search scheme during the event-graph construction. A dedicated search engine based on cascaded event queues is implemented in hardware to quickly identify valid neighbors in the spatial then temporal dimension, ensuring the efficient construction of event-wise local subgraphs.
- We introduce the concept of layer parallelism to speed up the processing of event-based GNNs with directed edges, reusing past information on a new event's neighborhood to parallelize the computation of every layer's new features, thereby reducing the end-to-end latency per event update.
- We finally deploy EvGNN on a Xilinx KV260 Ultrascale+ MPSoC platform and evaluate our design with the real-world dataset N-CARS [29] for car recognition. Our design achieves a prediction accuracy of 87.8%, and reaches an average 16- μ s latency per event, while limiting the overall on-chip memory footprint to 1.76MB, thereby enabling low-footprint low-latency edge vision.

This paper is organized as follows. Section II introduces key concepts and algorithms of event-based data, event graphs, and GNNs. Our hardware-algorithm co-design approach is described in Section III, covering the two key steps of event graph construction and GNN processing, with the aim to reduce the computational and memory footprints toward deployment on edge hardware. Section IV covers the proposed EvGNN architecture and implementation. Finally, benchmarking results are provided in Section V, while Section VI provides concluding remarks.

II. BACKGROUND – EVENT GRAPH CONSTRUCTION AND PROCESSING ALGORITHMS

In this section, we first introduce background information for event graphs (Section II-A), GNNs (Section II-B), and finally the methods allowing for an event-based construction and processing of event graphs (Section II-C).

A. Event Graphs

A graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$ is a data structure where objects are abstracted as nodes (vertices) $\mathcal{V}$ and their relationships are captured as edges $\mathcal{E}$. Each node in a graph can be assigned additional information, referred to as *features*. Each edge, which connects two nodes, can be either undirected or have a fixed direction pointing from one node to another, thus forming an *undirected* or a *directed* graph, respectively.

Event stream data is a series of events generated from the activated pixels of event-based cameras, in which an event $ev = (x, y, t, p)$ is represented as a combination of the pixel spatial position (x, y) , the timestamp t , and the binary polarity $p = \{0, 1\}$ to indicate a positive or negative change in light intensity, respectively. An event stream can be viewed as a graph, denoted as an *event graph*, where each event is represented as a node with two key features: the spatiotemporal position (x, y, t) and the polarity p [30], [31]. Nodes connected by an edge are referred to as *neighbors* for each other. For a given node i , a subgraph $\mathcal{N}(i)$ consisting of all neighbors and connecting edges is denoted as the *(1-hop) neighborhood* of the node. Similarly, the *2-hop* neighborhood subgraph will include the neighbors' neighbors and their connecting edges, while the *n-hop* neighborhood subgraph will expand to the n^{th} -order neighbors of the node [32].

B. Graph Neural Networks (GNNs)

For inference on graph data, GNNs gather information from nodes and edges of a graph to generate a new representation of the graph. Similar to CNNs, convolutional GNNs (simply referred to as GNNs hereafter) are composed of graph convolutional layers, graph pooling layers, graph readout layers, and standard fully-connected (FC) layers for the final prediction head (Fig. 1):

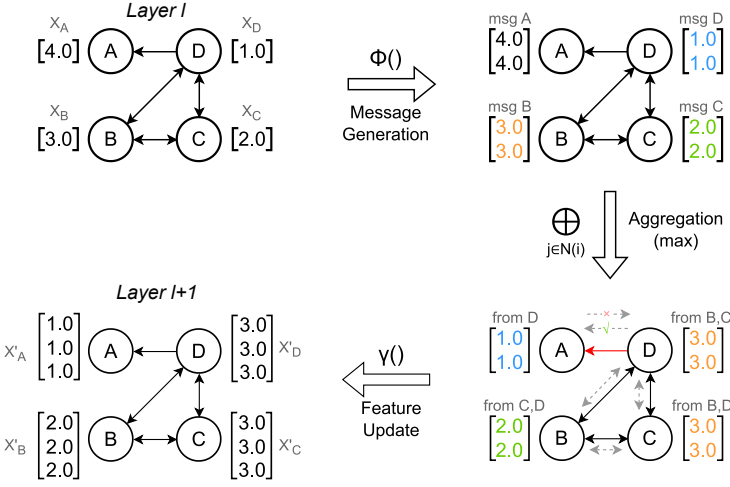


Fig. 2. Illustration of graph convolution, consisting of three steps: message generation, aggregation, and feature update. In the graph, each node i has a feature vector (numbers in brackets). First, every node generates its message (colored numbers) by the differentiable function $\phi()$ (*message generation*, illustrated with a replication operation as a toy linear transformation). Next, messages are exchanged through the edges and nodes aggregate the messages they receive (*aggregation*, illustrated with a max-value operation). Note that the message from A cannot be aggregated by D due to the directed edge. Finally, the aggregated features are transformed by $\gamma()$, generating the new layer's feature vectors (*feature update*, illustrated with a replication operation).

1) *Graph convolution*: The general description of various graph convolution types, typically expressed as a *message passing algorithm*, consists of three major steps: message generation, aggregation, and feature update [33]. As illustrated in Fig. 2, within a graph convolutional layer l , these steps are carried out for each node i as per Eq. (1) [34], ignoring any edge feature aside from connectivity out of simplicity:

$$\mathbf{x}_i^{l+1} = \gamma \left(\mathbf{x}_i^l, \bigoplus_{j \in \mathcal{N}(i)} \phi(\mathbf{x}_i^l, \mathbf{x}_j^l) \right), \quad (1)$$

where $\mathbf{x}_i^l, \mathbf{x}_j^l \in \mathbb{R}^{C_{in}}$ are the input feature vectors respectively associated with the current node i and each of its neighbors $j \in \mathcal{N}(i)$, with C_{in} the number of input feature channels, and $\mathbf{x}_i^{l+1} \in \mathbb{R}^{C_{out}}$ is the resulting output feature vector with C_{out} feature channels.

The convolution steps are thus as follows:

- i) *Message generation*: for each neighbor j , the features of node i and that of neighbor j , $(\mathbf{x}_i^l, \mathbf{x}_j^l)$, go through a learnable differentiable function ϕ (e.g., a linear transformation [25] or a multi-layer perceptron (MLP) [35]).
- ii) *Aggregation*: as each neighbor j of node i generates one message, all messages are aggregated by a chosen permutation-invariant function $\bigoplus$ (e.g., summation, average, or maximum). In the case of directed edges, a message can only be aggregated if it points from j to i .
- iii) *Feature update*: the aggregated message is transformed by another learnable differentiable function γ to generate the $\mathbf{x}_i^{l+1}$ output feature vector of node i , which is transmitted to the next GNN layer.

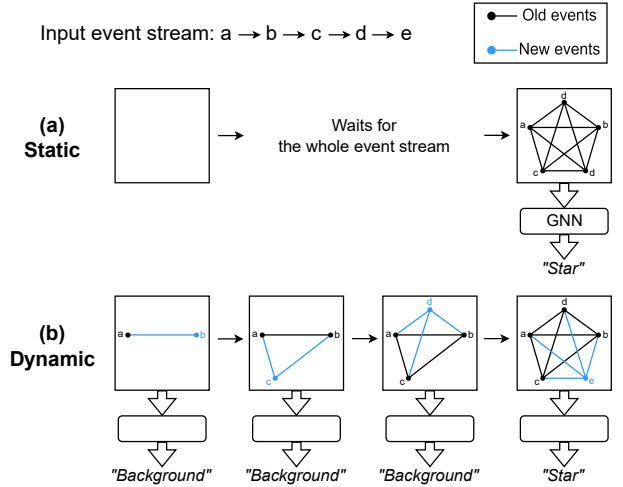


Fig. 3. Static and dynamic event graphs generated from the same event stream. (a) The whole event stream is first transformed into a static event graph, then uses a GNN to provide a prediction result. (b) Whenever a new event is generated (shown in blue), the dynamic event graph is updated and then processed by a GNN, thereby generating an updated prediction result on a low-latency, per-event basis.

2) *Graph pooling*: Graph pooling layers down-sample the nodes of the input graph to generate a coarser representation [36]. One typical method, cluster-based graph pooling (e.g., DiffPool [37] and EigenPool [38]), maps a group of original nodes into a new node (*cluster*) and gathers their features into the cluster node. The number of nodes decreases, while the connections between original node groups are kept and rebuilt as the edges between new cluster nodes.

3) *Graph readout*: While graphs have a data-dependent number of nodes, the final FC classification head requires inputs with fixed dimensions. To solve this mismatch, graph readout layers transform event graphs into Euclidean, image-like data by globally pooling features of all nodes according to a permutation-invariant function (e.g., summation, average or max-value) [36], [39], [40]. Several variants exist, such as the grid-based graph readout used in the work of Schaefer *et al.* [23], which operates by first defining a 2D grid based on the spatial position (x, y) of each node, and then executing the pooling operation for nodes within each cell of the grid.

4) *FC prediction head*: With the regular feature data provided by the graph readout layer, a set of one or multiple FC layer(s) allows to generate the final prediction results.

C. Event Graph Construction and Processing Strategies

The *graph construction* step populates a graph with nodes and connects them with edges if their spatiotemporal distance is within a certain threshold. This step can take place either in a *static* or a *dynamic* fashion [23], [24]. In the former, the entire event sequence is pre-processed and edges are generated according to the spatiotemporal positions of all events. In the latter, the graph is constructed in an event-driven manner: each new event adds a node and its associated edges to the current dynamic graph. In other words, static graph construction is carried out once the full event stream is available, while dynamic graph construction is distributed on a per-event basis.

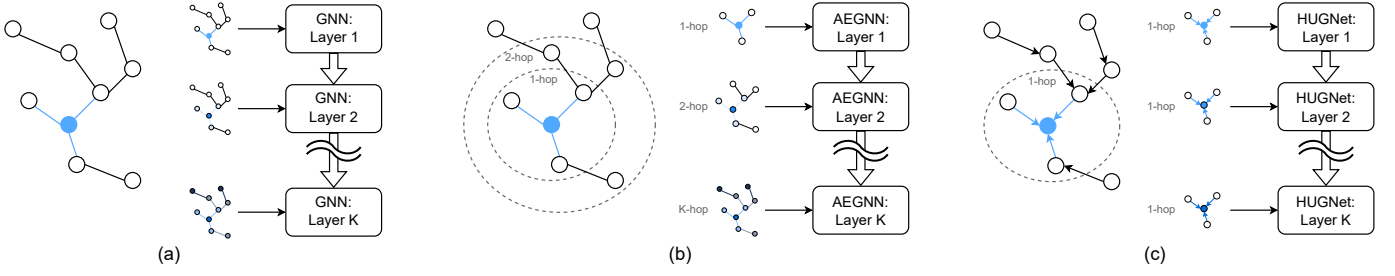


Fig. 4. Event-driven K-layer GNN processing schemes. (a) Naive scheme – A dynamic event graph, with the blue dot representing the new event node, is processed entirely for each layer of a conventional GNN. The colored nodes represent the updated features along the GNN, while uncolored ones depict unchanged features. (b) AEGNN scheme [23] – The same event graph is processed within a k^{th} subgraph by the k^{th} layer of the event-driven GNN, only involving nodes whose features will be effectively updated. (c) HUGNet scheme [24] – A directed dynamic event graph implies a causal flow of information, hence only the 1-hop subgraph is needed as the input of each layer as the features of neighboring nodes remain untouched.

The *graph processing* step, based on the execution of a GNN, is strongly dependent on the selected graph construction strategy. Static graph construction is the most common strategy as it allows for the use of vanilla GNNs [22], [30], [31]. However, it comes at the expense of latency, as the full event stream needs to be available before a prediction can be generated (Fig. 3(a)). To solve this issue, dynamic graph construction has recently been investigated to allow for event-based GNN execution [23], [24]: each new event not only updates the graph, it is also immediately processed by the GNN to update node features and generate a new prediction, a system that we refer to as an *event-driven GNN* (Fig. 3(b)).

Naive event-driven approaches, which combine dynamic event graphs directly with conventional GNNs, imply that each layer of the GNN has to process the entire updated graph each time a new event is received (Fig. 4(a)), thereby leading to high computational overhead [23]. However, as each layer of a GNN only needs information from the 1-hop neighborhood of a node for message passing (Eq. (1)), the final output features of a node in a K-layer GNN are governed exclusively by its K-hop neighborhood (*i.e.* the *receptive field* of the GNN) [41], [42]. This implies that, for dynamic event graphs, event-driven GNNs only need to process 1-to-K-hop subgraphs for each layer upon the reception of a new event, rather than the whole graph. Introduced in AEGNN [23] and illustrated in Fig. 4(b), this method significantly reduces computation while maintaining mathematical equivalence.

In HUGNet [24], Dalgaty *et al.* exploit *directed* dynamic event graphs, where the edges can only point from past nodes to newly added ones (Fig. 4(c)). This makes message passing in GNN layers a *causal* process, as opposed to AEGNN where undirected graphs can lead to information being shared from new nodes to past ones, leading to three key advantages. First, HUGNet solves one of the main issues of AEGNN for event graph construction, namely that a new event node needs to wait for future potential neighbors. Second, once a new event is received, only the features of the corresponding new node need to be updated: features of its neighbors within the K-hop range, constituted of past nodes, are fixed as information cannot flow in an anti-causal direction. This implies that the processing range of a K-layer event-driven GNN can be decreased from 1-to-K-hop subgraphs to only the 1-hop subgraph, as illustrated in Fig. 4(c). Finally, as the range of message passing is

restricted to the 1-hop neighborhood of each new event, edges can be computed on-the-fly for the 1-hop range and do not need to be computed nor stored beyond this range.

III. HARDWARE-AWARE GRAPH CONSTRUCTION AND PROCESSING

Event-driven GNN algorithms such as AEGNN and HUGNet have been optimized for accuracy on high-performance CPU and GPU backends. Hence, their computational complexity, as well as their memory usage and access patterns, does not allow for straightforward porting to edge platforms and, to the best of our knowledge, no hardware platform for event-driven GNNs has been proposed to date. In this section, we revisit the key steps of graph construction and processing for hardware efficiency, while minimizing the penalty on accuracy (see Fig. 5 for an illustration of the overall processing pipeline). We select the real-world task of car recognition with event-based vision, benchmarked on the N-CARS dataset [29]. This dataset contains several 100ms samples of event-based camera recording, representative of a multi-event real edge-vision use case. As of today, AEGNN is currently the state-of-the-art event-driven GNN approach on N-CARS [23]. AEGNN will thus be adopted as a baseline, while our co-design developments will be carried out on a validation set from the N-CARS dataset consisting of 15% of the training set, following a bootstrap strategy [43]. Moreover, the accuracy obtained for each experiment below is averaged over five trials.

A. Graph Construction

During the first step of graph construction (see Section II-C), readily executed upon the reception of each new event, spatiotemporal neighbors have to be identified and selected within a certain spatiotemporal distance. This process, known as *radius search*, is however computationally expensive: it usually relies on a *k-d tree* search with frequent insertion and deletion of node data for space partitioning, or on a structural modification of the downstream GNN [22]. To solve this challenge, we propose to leverage directed edges and simplified neighborhood search ranges.

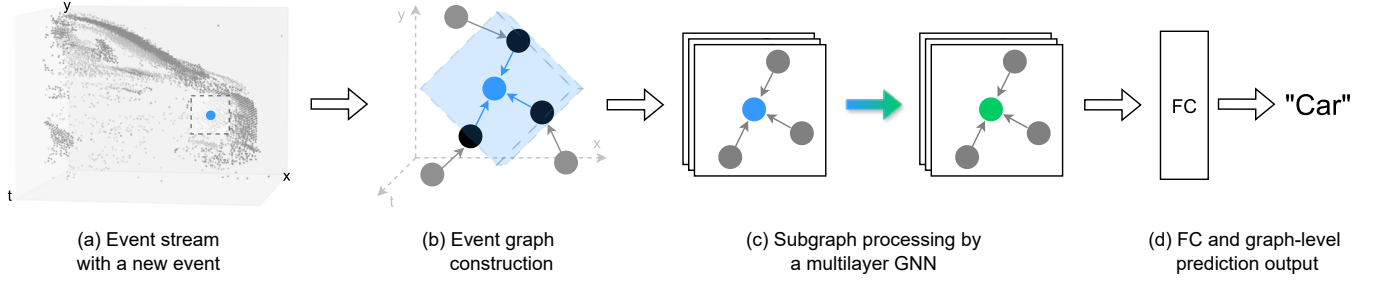


Fig. 5. Proposed event-driven GNN processing pipeline. (a) A new input event (blue) in an event stream, and its neighboring past events, are processed together in an event-driven fashion. (b) The new event searches potential neighbors in the blue causal prism region, and connects with them by directed edges, thus constructing the event graph (Section III-A). (c) The updated event graph is processed by a multilayer GNN (Section III-B), where only (i) the 1-hop subgraph containing the new event is processed, and (ii) the features of the new node are updated. (d) The GNN updates the graph-level prediction.

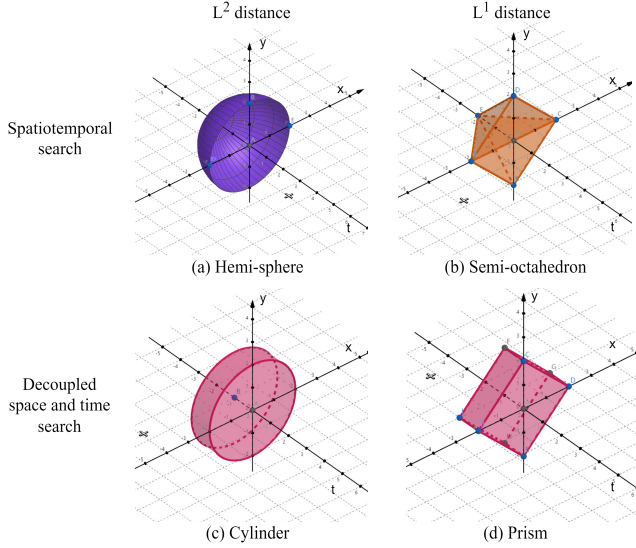


Fig. 6. Illustration of four spatiotemporal neighbor search ranges: (a) hemi-sphere of radius r , (b) semi-octahedron, (c) cylinder with base radius r_s and height r_t , and (d) prism, obtained by combining two different search schemes with two different L^p distance metrics. Note that the range parameters of all search ranges can be tuned according to the selected application scenario.

TABLE I
PREDICTION ACCURACY FOR AEGNN WITH DIRECTED EDGES COMPARED TO THE ORIGINAL AEGNN BASELINE

Edge type	Accuracy $\pm$ Standard Deviation
Baseline (Undirected)	95.4% $\pm$ 0.49%*
Directed	95.7% $\pm$ 0.26%

* Based on AEGNN open-source codes on the validation set.

1) *Directed graph adaptation*: Inspired by HUGNet [24], which was proposed for optical flow estimation tasks (Section II-C), we adopt the directed graph idea for our classification task scenario. Doing so reduces the spatiotemporal neighborhood search space from a full sphere to a hemisphere (Fig. 6(a)), as relationships are now causal and no information flows from future nodes to past nodes. Beyond simplifying the search space, we show in Table I that, compared to the AEGNN baseline, adopting directed graphs does not adversely affect accuracy performance.

2) *Prism Neighbor Search Range*: The *neighbor search range* defines within which spatiotemporal region a previous event can be regarded as a neighbor to a new one. The distance defined by the L^p norm of a difference vector of two position vectors is known as the L^p distance [44]. For example, for nodes in the $\mathcal{R}^3$ spatiotemporal space with position $\mathbf{pos}_i = (x_i, y_i, t_i)$, the L^1 and L^2 distances are

$$\begin{aligned} L^1(\mathbf{pos}_1, \mathbf{pos}_2) &= \|\mathbf{pos}_1 - \mathbf{pos}_2\|_1 \\ &= |x_1 - x_2| + |y_1 - y_2| + |t_1 - t_2| \\ &= |dx| + |dy| + |dt|, \end{aligned}$$

$$\begin{aligned} L^2(\mathbf{pos}_1, \mathbf{pos}_2) &= \|\mathbf{pos}_1 - \mathbf{pos}_2\|_2 \\ &= \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (t_1 - t_2)^2} \\ &= \sqrt{dx^2 + dy^2 + dt^2}. \end{aligned}$$

While the L^2 distance is the most prevalent distance definition used in graph-based event vision algorithms [22]–[24], [30], [31], it involves the computationally costly square root function to calculate the spatiotemporal neighborhood range of a node. On the contrary, the L^1 distance involves only additions and absolute values, both of which are hardware-friendly functions. We illustrate the influence of L^p distances on the shape of the spatiotemporal neighbor search range in Fig. 6: with directed graphs, switching from an L^2 to an L^1 distance metric leads the search range to become a semi-octahedron instead of a hemi-sphere (see Figs. 6(b) and 6(a), respectively).

Complementary to the choice of the distance function, selecting the radius r that bounds the maximum spatiotemporal distance often involves the definition of a proper scaling factor between time and space dimensions, which can have widely different scales (*e.g.*, a few pixels vs. thousands of microseconds). Hence, previous works [23], [24] adopted a modified position $\mathbf{pos}_i^*$ with a preset time scaling factor β in the L^2 spatiotemporal distance:

$$L^2(\mathbf{pos}_1^*, \mathbf{pos}_2^*) = \sqrt{dx^2 + dy^2 + (\beta dt)^2} \leq r. \quad (2)$$

However, given that β can compensate for a scale difference of several orders of magnitude, this approach is only practical when high-precision arithmetic is available. Therefore, instead

TABLE II
PREDICTION ACCURACY FOR DIRECTED-EDGE AEGNN WITH DIFFERENT
NEIGHBOR SEARCH RANGES

Search Ranges	Search Scheme	L^p	Acc. $\pm$ S.D.
Hemi-sphere	Spatiotemporal	L^2	$95.7\% \pm 0.26\%$
Semi-octahedron		L^1	$95.9\% \pm 0.39\%$
Cylinder	Spatiotemporally-decoupled	L^2	$95.5\% \pm 0.41\%$
Prism		L^1	$95.1\% \pm 0.45\%$

of following a rescaling approach with a single spatiotemporal distance metric with radius r , we propose to separate it into two independent parts: the spatial distance range r_s and the temporal distance range r_t . In this way, the spatial and temporal metrics can be decoupled, and each of them can be processed in different scales and resolutions. The spatial and temporal conditions for L^2 and L^1 distance metrics are given in Eqs. (3) and (4), and correspond to the cylinder and prism search ranges illustrated in Figs. 6(c) and 6(d), respectively.

$$\sqrt{dx^2 + dy^2} \leq r_s \text{ and } dt \leq r_t \quad (3)$$

$$|dx| + |dy| \leq r_s \text{ and } dt \leq r_t \quad (4)$$

Table II provides the prediction accuracies of all four search ranges summarized in Fig. 6. The proposed prism search range offers a lower footprint for deployment on custom hardware without any significant impact on accuracy.

B. GNN Architecture Search and Optimization

Once the graph construction phase for a new event is over, graph features can be extracted by executing the GNN algorithm. Two key design choices influencing the tradeoff between performance and resource usage in GNNs are (i) the type of graph convolution operation, and (ii) the overall network architecture, which we investigate hereafter.

1) *Graph Convolution*: A standard baseline for graph convolutions is the one proposed for the *graph convolution network* (GCN) proposed in [25]. Implementing Eq. (1), its formulation of the node features update at layer l can be expressed from for unitary edge and linear operators:

$$\mathbf{x}_i^{l+1} = \Theta_l^T \cdot \left(\sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{\mathbf{x}_j^l}{\sqrt{(d_j + 1)(d_i + 1)}} \right), \quad (5)$$

where $\Theta_l \in \mathbb{R}^{C_{in} \times C_{out}}$ are layer l 's learnable weights and d_i, d_j are the *degree* (i.e., the number of neighbors) of nodes i and j , respectively. The latter serve for normalizing the features of each neighbor j (i.e. message generation), before summing them (i.e. aggregation) and applying a linear transformation with learnable parameters Θ (i.e. feature update). This operation is applied for each convolutional layer l of the GCN. While Eq. (5) only depends on the neighbor's features, and thus hardly exploits information related to the relative positions between two nodes, recent event-based GNNs such as AEGNN [23], [24] have aimed to explicitly exploit this

TABLE III
PERFORMANCE METRICS OF DIFFERENT TYPES OF GRAPH CONVOLUTION

Convolution Types	Acc. $\pm$ S.D.	#Parameters
GCNC baseline	$92.5\% \pm 1.09\%$	4.7k
SplineConv	$95.4\% \pm 0.49\%$	30.4k
This work	$95.1\% \pm 0.43\%$	4.8k

TABLE IV
PERFORMANCE METRICS FOR DIFFERENT NETWORK STRUCTURE
SIMPLIFICATIONS ON N-CARS VALIDATION DATASET

Structure	Acc. $\pm$ S.D.	Param. Memory Footprint
SplineConv	$95.4\% \pm 0.49\%$	121.6kB
This work (FP32)	$95.5\% \pm 0.50\%$	26.4kB
This work (INT8)	$95.4\% \pm 0.35\%$	6.6kB

information via *SplineConv*, a graph convolution method derived from [26] that encodes the positional information of neighboring nodes using B-spline kernels, on top of the nodes' features. The wider representation ability unlocked by this encoding was shown to help improve the classification accuracy of event-based GNNs on several tasks, as highlighted in Table III. Nonetheless, B-splines bring a significant computational burden as well as a vast parameter overhead, being hardly compatible with low-latency and low-footprint event-based vision at the edge.

Alternatively, Qi *et al.* introduced *PointNetConv* [35], a variant of graph convolution supporting positional encoding. In each of the GCN layers, PointNetConv convolutions concatenate the relative position of each neighbor together with its corresponding features, before applying a learnable transform and a max-value aggregation. The updated features of node i for layer $l + 1$ are thus obtained as

$$\mathbf{x}_i^{l+1} = \gamma_{\Theta_0}^l \left(\max_{j \in \mathcal{N}(i) \cup \{i\}} \phi_{\Theta_1}^l(\mathbf{x}_j^l, \mathbf{pos}_j - \mathbf{pos}_i) \right), \quad (6)$$

where $\gamma_{\Theta_0}^l, \phi_{\Theta_1}^l$ are multi-layer perceptrons (MLPs) with weights Θ_0 and Θ_1 , respectively. Such a graph convolution avoids the resource-intensive B-splines while properly behold-ing spatial information. Still, it requires to store large end-to-end MLP models. Therefore, we introduce here a simplified version of Eq. (6), in which ϕ is reduced to a single linear layer and γ performs the identity operation. Hence, the features update becomes

$$\mathbf{x}_i^{l+1} = \max_{j \in \mathcal{N}(i)} \left(\Theta_l^T \cdot (\mathbf{x}_j^l, |dx_{i,j}|, |dy_{i,j}|) \right). \quad (7)$$

Table III shows that, while using $6.3 \times$ less parameters than the SplineConv-based AEGNN baseline, our simple PointNetConv in Eq. (7) still achieves a comparable accuracy. This further underlines the importance of preserving spatial information along the GCN layers.

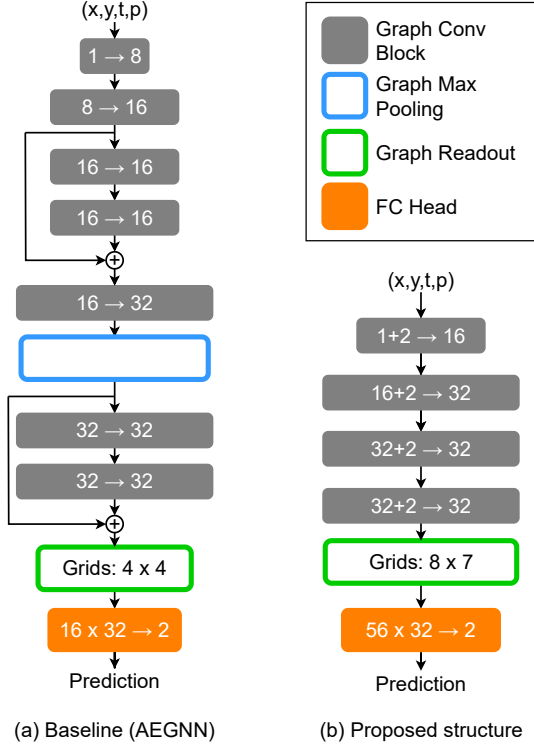


Fig. 7. GNN architecture of (a) the baseline (AEGNN), and (b) the proposed simplified network. A *Graph Conv Block* contains a graph convolution, an activation function, and a batch normalization layer, where batchnorm folding [45] can be applied. Numbers labeled on graph convolution blocks indicate the corresponding *input* $\rightarrow$ *output channels dimensions*. The +2 part in (b) corresponds to the concatenated position differences in Eq. (7).

2) *Network Architecture*: Toward a low-cost hardware implementation, we simplify the AEGNN architecture as presented in Fig. 7. We show that (i) using a shallower structure based on PointNetConv graph convolution layers, (ii) removing residual connections and graph pooling layers, and (iii) applying batchnorm folding [45] together with post-training quantization [46] to 8-bit integers, allows to reduce the required number of parameters by $5\times$ without adversely affecting the classification accuracy (Table IV).

IV. PROPOSED HARDWARE ARCHITECTURE FOR EVENT-DRIVEN GNNs

In this section, we design a hardware architecture that accelerates the graph construction and processing algorithms introduced in Section III for a low-power, low-latency deployment at the edge. The proposed hardware accelerator, EvGNN, targets modern Xilinx MPSoC platforms.

A. Overall System Architecture

Xilinx MPSoC platforms contain *Programmable Logic (PL)* together with a *Processing System (PS)*. A system overview is shown in Fig. 8: our hardware accelerator, EvGNN, is located in the PL, while an ARM core in the PS acts as a host CPU that is responsible for (i) loading dataset samples from the external DRAM, (ii) feeding data to and reading prediction results from

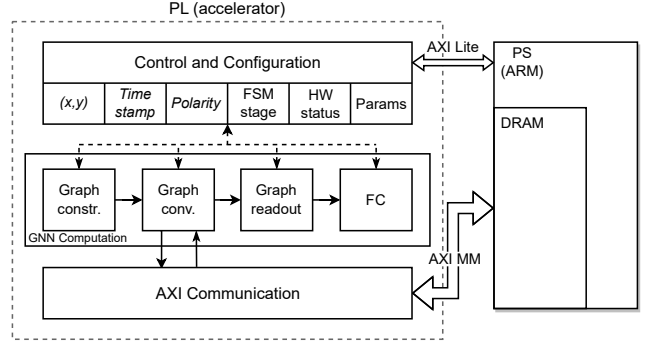


Fig. 8. Overall system architecture. The EvGNN accelerator is located in the PL, while the PS host CPU is mainly responsible for benchmarking and monitoring.

EvGNN through AXI system buses, and (iii) calculating the prediction accuracy and measuring the runtime.

The EvGNN accelerator consists of three major blocks. The *GNN Computation* block is the core implementation of our event-driven GNN algorithm in hardware. It mainly consists of a graph construction module, a graph convolution module, as well as graph readout and FC modules for classification. The *Control and Configuration* block receives event-based data and configurable parameters from the PS, broadcasting them to the GNN computation block, and then sending the prediction result back to the PS. Finally, the *AXI Communication* block implements AXI protocols, including data buffers. In the following, we will elaborate on the modules of the GNN computation block.

B. Graph Construction Module

As introduced in Section III-A, the first step is to build a local, directed dynamic event graph upon the reception of each new event. A flow diagram for the graph construction hardware is shown in Fig. 9, which is based on two core design decisions regarding (i) the storage of the event graph, and (ii) the selection of neighbors for each new event.

1) *Event queues for graph storage*: As the use of directed graphs allows us to forgo the storage of edges (Section II-C), only the nodes need to be stored and accessed, which can be done in a straightforward manner with *event queues* [22], [47], as depicted in Fig. 10. All event queues are stored in a *global event buffer*, where each event queue is associated with a pixel of an event-based camera and stores events whose spatial position is the same as this pixel's physical position (x, y) . Event information other than the spatial coordinates is stored in the queue entries, including the timestamp t , the polarity p , and the event index in the event stream n , numbered in chronological order. Pushing a new event pops the oldest one if the queue is full, and every entry can be independently read-accessed.

2) *Spatiotemporally-decoupled neighbor selection*: In order to implement the spatiotemporally decoupled prism search range introduced in Section III-A, we leverage the fact that the event queue storage scheme intrinsically decouples space and time: spatial information is found in the event queue index,

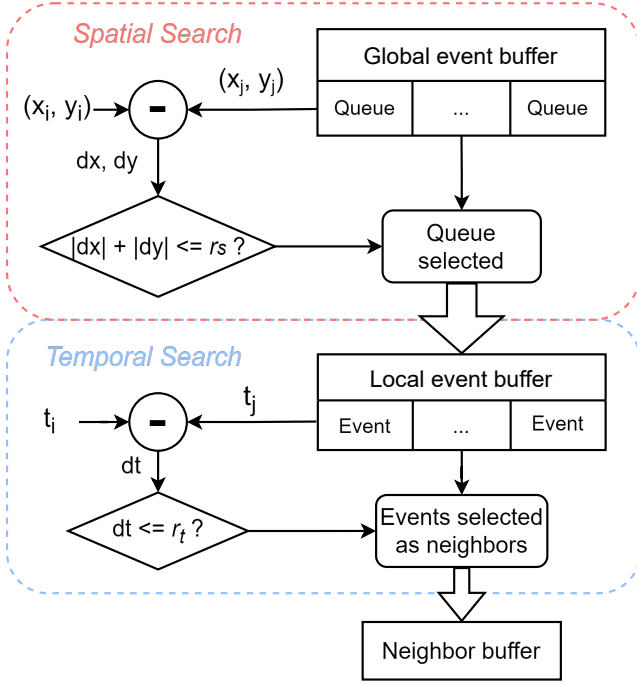


Fig. 9. Hardware flow diagram of the graph construction module. For each new event i with spatiotemporal position (x_i, y_i, t_i) , following decoupled spatial and temporal search processes, the graph construction module selects the neighbors of the new node and stores them in the neighbor buffer for subsequent processing by the graph convolution module.

while temporal information is found inside the entries of a queue. This allows the graph construction module to perform the spatial and temporal search steps of Eq. (4) independently.

According to the (x_i, y_i) location of the new event, *spatial search* (upper half of Fig. 9) is performed to pick all event queues within an L^1 spatial distance of r_s from the new event. They are accessed from the global event buffer and transferred to a *local event buffer*, skipping out-of-bound locations around the target node.

The *temporal search* process (bottom half of Fig. 9), which can be pipelined with the spatial one, first reads the timestamps t_j of each entry stored in the local event buffer, before subtracting them with the current timestamp t_i to derive the time difference dt . Each previous event within a temporal distance r_t of the new event will be selected as a neighbor and pushed to the *neighbor buffer*, whose depth D_{max} bounds the maximum number of neighbors that can be attributed to a given node. Eventually, the neighbor buffer can be processed by the graph convolution module for the event-driven GNN update.

C. Graph Convolution Module

In the graph convolution module, we first introduce the concept of *layer-parallel execution* in event-driven GNNs. Then, we propose a hardware architecture and computation flow for this module that exploits this layer-wise parallelism.

1) *Layer-parallel execution*: When using GNNs with static directed graphs (Fig. 11(a)), the features of all nodes change after each convolutional layer. As updating the node features of

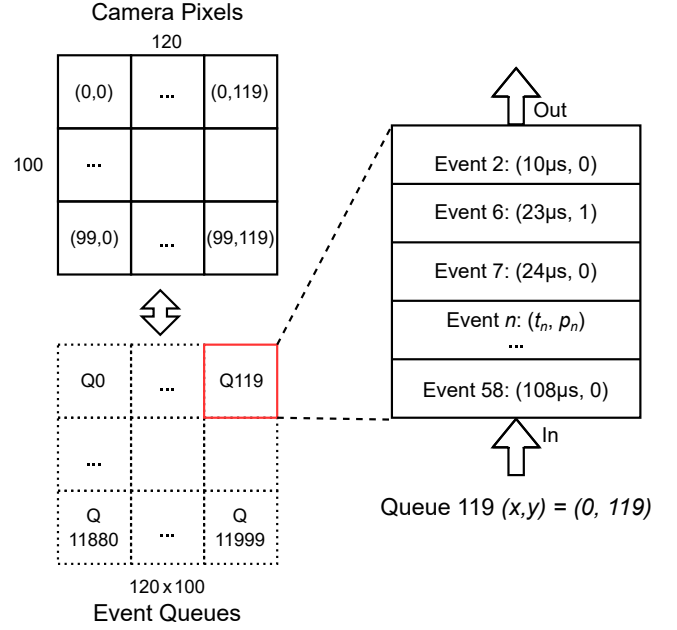


Fig. 10. Event-queue-based graph storage, where each queue corresponds to the spatial location of a given pixel and collects every event generated by that pixel. Inside a queue, the event index in the event stream n , the timestamp t , and the polarity p are stored. The event queue is sized as per the event-based camera in the selected application (*i.e.*, in this case, with 120×100 event queues for the camera used in the N-CARS dataset).

the current layer depends on the new features of the previous one, layers need to be executed sequentially.

When using dynamic directed event graphs, however, the causal nature of the graph prevents any feature update of the previous nodes, such that only the features of new incoming events have to be processed, based on their local neighborhood (Fig. 11(b)). Moreover, for convolution operations such as Eq. (7), the computation of output features related to a new event only depends on the node's past neighboring node features, thereby removing any data dependency between GNN layers to compute a new node's features. Hence, as shown in Fig. 11(c), each layer's features associated with the new node's neighbors can be first retrieved, before executing all graph convolutional layers in parallel to get their new node's features. This layer-parallel approach drastically shortens the execution time of multi-layer GNNs, and thus the overall per-event latency of the system.

2) *Graph convolution hardware architecture*: The hardware diagram of our graph convolution module with layer-parallel execution is shown in Fig. 12. It involves three main stages, which can be pipelined:

- i) *Neighbors features retrieval*: Neighbors are sequentially popped from the neighbors buffer, one at a time (i). The features of the selected neighbor, previously computed for all graph convolutional layers, are fetched from the external DRAM, which is accessed through an AXI MM bus via the PS.
- ii) *Layer-Parallel Execution*: This step implements the graph convolution mechanism of Eq. (7), parallelized across all four GNN layers as per the selected network architecture (Fig. 7). The features vector $(\mathbf{x}_j, |dx_{i,j}|, |dy_{i,j}|)^t$ of each

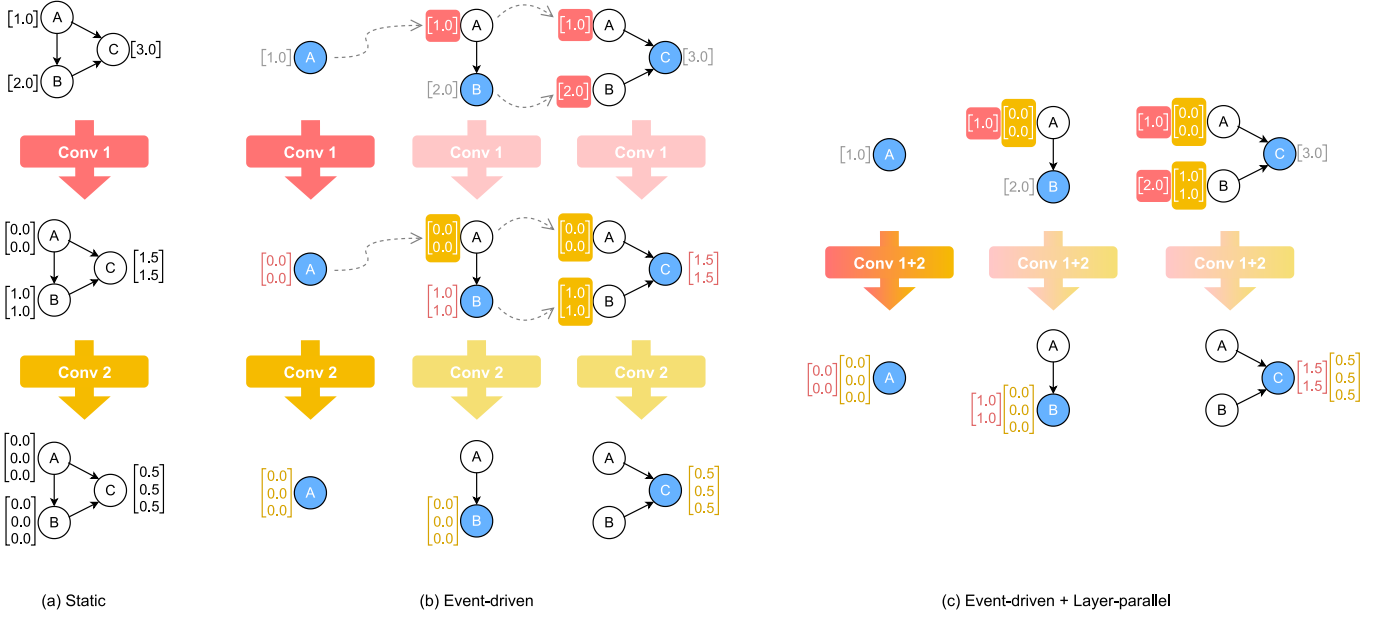


Fig. 11. Event graphs processed by an example GNN containing two graph convolutional layers, *Conv 1* and *Conv 2*, both of which consist of a simple average-and-replicate operation for illustration purposes. (a) A directed *static* event graph is processed by the GNN. Each graph convolutional layer is applied sequentially, computation of new features can be parallelized over nodes. (b) A directed *dynamic* event graph is processed by the GNN in an event-driven fashion, with new events colored in blue. The inputs of layers have colored backgrounds with white fonts, while the outputs of layers have white backgrounds with colored fonts. The outputs are identical to those computed with the static graph. Note that the output of a convolutional layer is only used by the *next* layer for *future* events, as shown with dotted gray arrows. (c) The proposed layer-parallel execution of the same directed processed by an event-driven GNN, which is mathematically equivalent to (b).

neighbor, including its relative position information, are sequentially broadcast to the message generation and aggregation sub-units for every layer l in parallel (ii). As detailed in Fig. 13, each message generation module performs the matrix-vector multiplication between a neighbor's feature vector and the weight matrix Θ^l of that layer, stored in a local BRAM. This operation takes place in $C_{in}^l + 2$ cycles (iii) and is parallelized across all C_{out}^l output channels by means of a *MatVec* unit, which embeds one digital multiply-and-accumulate (MAC) engine per output channel. Then, the aggregation module receives the newly generated C_{out} messages and performs a max-valued comparison, so as to sequentially keep track of the maximum element per output channel across all neighbors (ii). Finally, a *bias* term, ReLU *activation* function, and *quantization* to INT8 is applied to each selected output, which steps are abbreviated as *BAQ* in Fig. 12.

- iii) *New event features storage*: The outputs of the four *BAQ* units are the new event's output feature vectors, one for each layer. These features are eventually sent back to the external DRAM over the AXI MM bus (iv), while the last layer's output is sent to the graph readout module to serve in the graph prediction update (v).

D. Graph Readout and FC Modules

Once a new event has been received and processed by the *Graph Construction* and the *Graph Convolution* modules, the prediction of the overall graph can be updated, which is carried out by the *Graph Readout* and *FC* modules (Fig. 8).

The former implements a grid-based graph readout layer (Section II-B3), which divides the full 120×100 -pixel input range into an 8×7 grid and selects the maximal features within 16×16 -pixel patches. The latter then implements the FC prediction head that provides classification results. In order to carry out the matrix-vector multiplication between the FC weight matrix and the output of the readout layer, we reuse the *MatVec* unit presented in Fig. 13.

V. RESULTS

Finally, we evaluate the performance of the designed EvGNN accelerator on the N-CARS dataset after implementing it on a resource-constrained test platform. Hereafter, we first introduce the experimental setup before assessing the achieved performance and comparing it to the state-of-the-art.

A. Experimental Setup

On the software side, experiments are deployed on NVIDIA RTX A6000 GPUs while implemented, trained, and tested using the PyTorch Geometric (PyG) library [34]. Similarly to AEGNN [23], we first train our event-driven GNN using static event graphs pre-compiled from the N-CARS event streams, where the total N-CARS training data (15,422 event stream samples) is divided into 85% as a training set and 15% as a validation set, following the bootstrapping approach (Section III). We use a batch size of 64 and an initial learning rate of 0.002 on the Adam optimizer with 100 epochs. After training, we benchmark our event-driven GNN for inference using the N-CARS test set (8,607 event stream samples), where we represent data as dynamic event graphs.

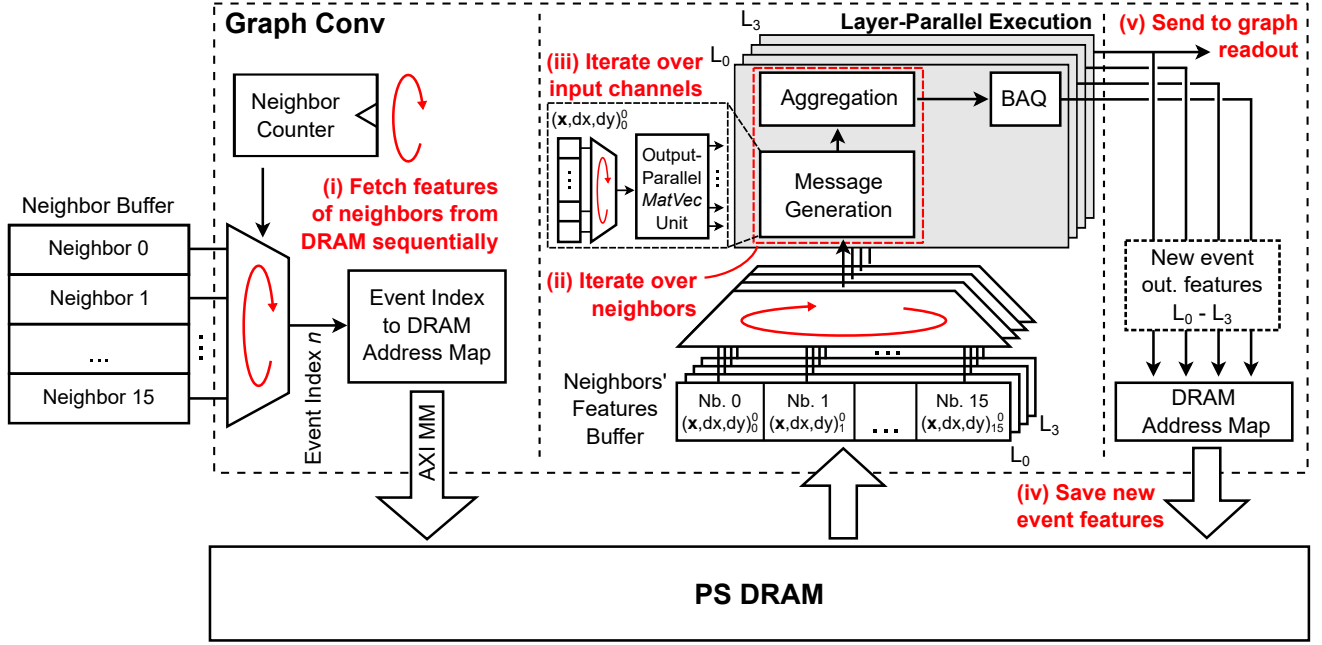


Fig. 12. Hardware diagram of the graph convolution module, corresponding to the four-layers network architecture in Fig. 7. Node features are first loaded from off-chip DRAM to the graph convolution module through AXI MM buses (i), according to the nodes in the neighbor buffer. Afterwards, for every neighbor (ii), the graph convolution engine performs message passing, iterating over every input channel (iii) before performing the message aggregation. The execution of these steps are parallelized in the *MatVec* across the output channel dimension as well as every layer, according to our layer-parallel execution scheme. Then, bias-activation-quantization (BAQ) is applied to generate features of the new event, which are stored back into the DRAM (iv). The last layer's updated feature is eventually broadcast to the readout unit (v).

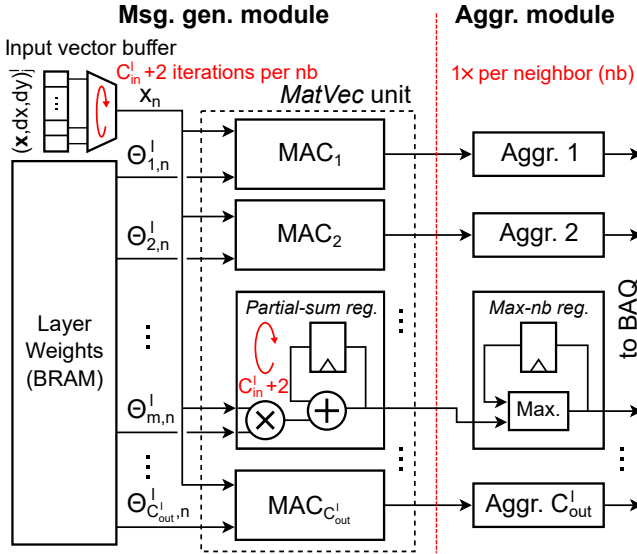


Fig. 13. Hardware diagram of the message generation (Msg. gen.) and aggregation (aggr.) sub-units, including a *MatVec* unit and BRAM weight storage. The input vector $(x_j, dx_{i,j}, dy_{i,j})$ of the selected neighbor j contains the C_{in} concatenated node's features of the current layer l and the node's position difference (see Eq. (7)). The *MatVec* unit performs $C_{in} + 2$ sequential accumulations, parallelized across the output feature dimension using C_{out} MAC units. The C_{out} output messages are then broadcast to aggregation units, which store for each output channel the max-valued message amongst the sequentially-addressed neighbors.

On the hardware side, the EvGNNaccelerator is deployed on a Xilinx KV260 development board, which contains a Zynq UltraScale+ MPSoC in a Kria K26 System-On-Module platform, targeting edge vision applications. As outlined in

TABLE V
RESOURCE USAGE OF EVGNN ON THE XILINX KV260 PLATFORM.

Resources		Used	Available	Utilization
Logic	LUT	30,908	117,120	26.4%
	FF	24,083	234,240	10.3%
	DSP	228	1,248	18.3%
On-chip memory	LUTRAM	0.45 kB	0.44 MB	0.1%
	BRAM	85.2 kB	0.63 MB	13.2%
	UltraRAM	1.68 MB	2.25 MB	75.0%

Section IV, EvGNN is implemented in the PL while an ARM core in the PS takes care of feeding data to EvGNN and collecting its performance metrics, such as the runtime per sample and the overall classification accuracy.

B. Implementation and Benchmarking Results

The FPGA resource usage of our EvGNN accelerator is reported in Table V. The logic usage amounts to 26.4% of look-up tables (LUTs) as logic, 10.3% of flip-flops (FFs), and 18.3% of digital signal processors (DSPs), which are mainly used for MAC units in the graph convolutional module. On-chip memory resource usage amounts to 0.1% of LUTRAM, 13.2% of BRAM, and 75.0% of UltraRAM, which are mainly consumed by the event queues in the graph construction module.

State-of-the-art techniques, together with EvGNN, are summarized and compared in terms of classification accuracy in Table VI. We qualify an entry as *event-driven* if it can (i) process the input data stream on an event-by-event basis, and (ii) update the prediction output upon each new event.

TABLE VI
SUMMARY AND PERFORMANCE COMPARISON OF STATE-OF-THE-ART
APPROACHES ON THE N-CARS TEST SET.

Networks	Representation	Event-driven	Local	Accuracy
H-First [48]	Spikes	✓	✗	56.1%
HOTS [49]	TS	✗	✗	62.4%
HATS [29]	TS	✓	✗	90.2%
YOLE [50]	EH	✓	✓	92.7%
AsyNet [51]	EH	✓	✓	94.4%
NVS-S [22]	Graph	✓	✓	91.5%
EvS-S [22]	Graph	✓	✓	93.1%
AEGNN [23]	Graph	✓	✓	86.7%*
Ours (software)	Graph	✓	✓	88.0%
Ours (hardware)	Graph	✓	✓	87.8%

* Based on the AEGNN open-source code on the N-CARS test set.

Similarly, *local* methods only process a neighboring region around the new event. HFirst [48] uses spiking neural networks (SNNs) to process event-based data, which support event-driven processing but are updated in a global fashion for each new event. HOTS [49] and HATS [29] exploit time-surface (TS) representations of event streams, which are processed by global classifiers. YOLE [50] and AsyNet [51] use event histograms (EH) to achieve both event-driven and local computation, but require $20\times$ more parameters than our graph-based approach, degrading the update latency and memory footprint. Finally, the graph-based NVS-S and EvS-S from [22], as well as AEGNN from [23] are both event-driven and local. On the one hand, NVS-S and EvS-S adopt a slide graph convolution method to achieve local computation. They first identify nodes and edges to be updated within the K-hop subgraph of the new event, using a sliding window, before applying graph convolution on these elements. On the other hand, AEGNN identifies the increasing neighborhood reached along the GNN layers, processing neighbors from the 1-hop subgraph to the K-hop one. Both approaches achieve accuracy levels similar to our work, yet their convolution operation involve a larger computational complexity. Note that the accuracy drop between validation and test results in our work, similar to AEGNN, comes from a statistical discrepancy between the validation and test sets. Moreover, NVS-S and EVS-S do not report their split between training, validation and test sets, such that the accuracy comparison in Table VI should be taken with a grain of salt. In contrast to these methods, our approach carries out computation locally within 1-hop subgraphs, in an event-driven fashion. Despite being optimized for low-footprint edge applications, we achieve classification accuracies of 88.0% (software) and 87.8% (hardware) on the N-CARS test set, which outperform the classification accuracy of AEGNN. The dynamic, event-by-event classification accuracy of EvGNN is reported in Fig. 14, which shows that, for low-latency applications, reliable classification can be obtained without having to process the full duration of N-CARS samples. Indeed, these results are obtained with an average latency per event of $16\mu s$, which demonstrates that the event-driven GNN hardware acceleration enabled by EvGNN allow to exploit the μs -level resolution of event-based cameras at the edge.

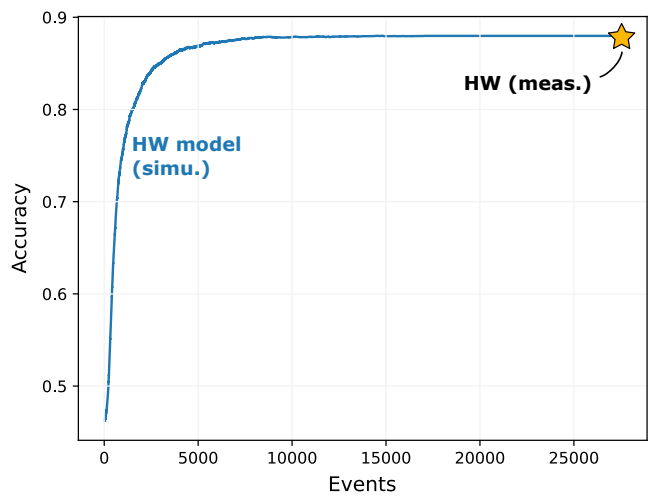


Fig. 14. Classification accuracy obtained with EvGNN as a function of the number of events on samples of the N-CARS test set.

VI. CONCLUSION

We proposed EvGNN, the first event-driven GNN accelerator, which enables low-latency, high-accuracy edge inference applications for event-based vision systems. It relies on three key ideas: (i) exploiting causality in directed graphs to enable local 1-hop subgraph processing, (ii) a lightweight spatiotemporally decoupled prism neighbor search implemented with flexible event queues, and (iii) a novel layer-parallel execution scheme to reduce the overall processing latency in multi-layer event-driven GNNs. The proposed accelerator was deployed on a Xilinx KV260 MPSoC platform with onboard benchmarking on the N-CARS dataset for car recognition. Our accelerator achieved a prediction accuracy of 87.8% and an average prediction latency of $16\mu s$ per event, demonstrating the capability to efficiently enable near real-time and microsecond-latency event-based vision intelligence at the edge.

ACKNOWLEDGMENTS

The authors would like to thank Prof. Marian Verhelst (KU Leuven) for fruitful discussions and feedback.

REFERENCES

- [1] Z. Wang *et al.*, "Sparse-yolo: Hardware/software co-design of an fpga accelerator for yolov2," *IEEE Access*, vol. 8, pp. 116 569–116 585, 2020.
- [2] P. Bonazzi *et al.*, "Tinytracker: Ultra-fast and ultra-low-power edge vision in-sensor for gaze estimation," in *IEEE SENSORS*, 2023, pp. 1–4. DOI: 10.1109/SENSORS56945.2023.10325167.
- [3] F. Conti *et al.*, "Pulp: A ultra-low power parallel accelerator for energy-efficient and flexible embedded vision," *Journal of Signal Processing Systems*, vol. 84, pp. 339–354, 2016.
- [4] J. Tang *et al.*, "Lopecs: A low-power edge computing system for real-time autonomous driving services," *IEEE Access*, vol. 8, pp. 30 467–30 479, 2020. DOI: 10.1109/ACCESS.2020.2970728.
- [5] S. Benninger *et al.*, "Edgeeye: A long-range energy-efficient vision node for long-term edge computing," in *International Green and Sustainable Computing Conference (IGSC)*, 2019, pp. 1–8.
- [6] H. Genc *et al.*, "Flying iot: Toward low-power vision in the sky," *IEEE Micro*, vol. 37, no. 6, pp. 40–51, 2017. DOI: 10.1109/MM.2017.4241339.
- [7] L. Huang *et al.*, "Obstacle distance measurement under varying illumination conditions based on monocular vision using a cable inspection robot," *IEEE Access*, vol. 9, pp. 55 955–55 973, 2021. DOI: 10.1109/ACCESS.2021.3070877.

- [8] A. Suleiman *et al.*, “Navion: A fully integrated energy-efficient visual-inertial odometry accelerometer for autonomous navigation of nano drones,” in *IEEE Symposium on VLSI Circuits*, 2018, pp. 133–134. DOI: 10.1109/VLSIC.2018.8502279.
- [9] M. Hillebrand *et al.*, “High speed camera system using a cmos image sensor,” in *Proceedings of the IEEE Intelligent Vehicles Symposium (Cat. No. 00TH8511)*, 2000, pp. 656–661. DOI: 10.1109/IVS.2000.898423.
- [10] M. A. Mahowald and C. Mead, “The silicon retina,” *Scientific American*, vol. 264 5, pp. 76–82, 1991.
- [11] P. Lichtsteiner, C. Posch, and T. Delbruck, “A 128×128 120 db $15 \mu\text{s}$ latency asynchronous temporal contrast vision sensor,” *IEEE Journal of Solid-State Circuits*, vol. 43, no. 2, pp. 566–576, 2008. DOI: 10.1109/JSSC.2007.914337.
- [12] G. Gallego *et al.*, “Event-based vision: A survey,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 44, no. 1, pp. 154–180, 2020.
- [13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, May 2017, ISSN: 0001-0782. DOI: 10.1145/3065386. [Online]. Available: <https://doi.org/10.1145/3065386>.
- [14] K. He *et al.*, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
- [15] P. Adarsh, P. Rathi, and M. Kumar, “Yolo v3-tiny: Object detection and recognition using one stage improved model,” in *International Conference on Advanced Computing and Communication Systems (ICACCS)*, 2020, pp. 687–694. DOI: 10.1109/ICACCS48705.2020.9074315.
- [16] A. Howard *et al.*, “Searching for mobilenetv3,” in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019, pp. 1314–1324.
- [17] J. Zhong, J. Chen, and A. Mian, “Dualconv: Dual convolutional kernels for lightweight deep neural networks,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 34, no. 11, pp. 9528–9535, 2023. DOI: 10.1109/TNNLS.2022.3151138.
- [18] Z. Li *et al.*, “A survey of convolutional neural networks: Analysis, applications, and prospects,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 12, pp. 6999–7019, 2022. DOI: 10.1109/TNNLS.2021.3084827.
- [19] D. Gehrig *et al.*, “End-to-end learning of representations for asynchronous event-based data,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 5633–5643.
- [20] M. Liu and T. Delbruck, “Adaptive time-slice block-matching optical flow algorithm for dynamic vision sensors,” in *British Machine Vision Conference (BMVC)*, Sep. 2018. [Online]. Available: <https://doi.org/10.5167/uzh-168589>.
- [21] A. Nguyen *et al.*, “Real-time 6dof pose relocalization for event cameras with stacked spatial lstm networks,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, Jun. 2019.
- [22] Y. Li *et al.*, “Graph-based asynchronous event processing for rapid object recognition,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 934–943.
- [23] S. Schaefer, D. Gehrig, and D. Scaramuzza, “Aegnn: Asynchronous event-based graph neural networks,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2022, pp. 12 371–12 381.
- [24] T. Dalgaty *et al.*, “Hugnet: Hemi-spherical update graph neural network applied to low-latency event-based optical flow,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, Jun. 2023, pp. 3952–3961.
- [25] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” *arXiv preprint arXiv:1609.02907*, 2016.
- [26] M. Fey *et al.*, “Splinecnn: Fast geometric deep learning with continuous b-spline kernels,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 869–877.
- [27] R. Sarkar *et al.*, “Flowgnn: A dataflow architecture for real-time workload-agnostic graph neural network inference,” in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 1099–1112.
- [28] Z. Zhou *et al.*, “Blockgnn: Towards efficient gnn acceleration using block-circulant weight matrices,” in *ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 1009–1014.
- [29] A. Sironi *et al.*, “Hats: Histograms of averaged time surfaces for robust event-based object classification,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 1731–1740.
- [30] A. Mitrokhin *et al.*, “Learning visual motion segmentation using event surfaces,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2020, pp. 14 414–14 423.
- [31] Y. Bi *et al.*, “Graph-based spatio-temporal feature learning for neuromorphic vision sensing,” *IEEE Transactions on Image Processing*, vol. 29, pp. 9084–9098, 2020. DOI: 10.1109/TIP.2020.3023597.
- [32] W. Liu and Z. Li, “An efficient parallel algorithm of n-hop neighborhoods on graphs in distributed environment,” *Frontiers of Computer Science*, vol. 13, pp. 1309–1325, 2019.
- [33] M. M. Bronstein *et al.*, “Geometric deep learning: Grids, groups, graphs, geodesics, and gauges,” *arXiv preprint arXiv:2104.13478*, 2021.
- [34] M. Fey and J. E. Lenssen, “Fast graph representation learning with pytorch geometric,” *arXiv preprint arXiv:1903.02428*, 2019.
- [35] C. R. Qi *et al.*, “Pointnet: Deep learning on point sets for 3d classification and segmentation,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 652–660.
- [36] Z. Wu *et al.*, “A comprehensive survey on graph neural networks,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 1, pp. 4–24, 2021. DOI: 10.1109/TNNLS.2020.2978386.
- [37] Z. Ying *et al.*, “Hierarchical graph representation learning with differentiable pooling,” in *Advances in Neural Information Processing Systems*, vol. 31, 2018.
- [38] Y. Ma *et al.*, “Graph convolutional networks with eigenpooling,” in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, 2019, pp. 723–731.
- [39] R. Li *et al.*, “Adaptive graph convolutional neural networks,” in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, 2018.
- [40] J. Atwood and D. Towsley, “Diffusion-convolutional neural networks,” in *Advances in Neural Information Processing Systems*, vol. 29, 2016.
- [41] P. Quan *et al.*, “A brief review of receptive fields in graph convolutional networks,” in *IEEE/WIC/ACM International Conference on Web Intelligence-Companion Volume*, 2019, pp. 106–110.
- [42] Z. Zhang *et al.*, “Multireceptive field: An adaptive path aggregation graph neural framework for hyperspectral image classification,” *Expert Systems with Applications*, vol. 217, p. 119 508, 2023, ISSN: 0957-4174. DOI: <https://doi.org/10.1016/j.eswa.2023.119508>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S095741742300009X>.
- [43] A. Rinaldo, L. Wasserman, and M. G. Sell, “Bootstrapping and sample splitting for high-dimensional, assumption-lean inference,” *The Annals of Statistics*, vol. 47, no. 6, pp. 3438–3469, 2019. DOI: 10.1214/18-AOS1784. [Online]. Available: <https://doi.org/10.1214/18-AOS1784>.
- [44] J. Brimberg and R. F. Love, “General considerations on the use of the weighted lp norm as an empirical distance measure,” *Transportation Science*, vol. 27, no. 4, pp. 341–349, 1993.
- [45] B. Jacob *et al.*, “Quantization and training of neural networks for efficient integer-arithmetic-only inference,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 2704–2713.
- [46] R. Krishnamoorthi, “Quantizing deep convolutional networks for efficient inference: A whitepaper,” *arXiv preprint arXiv:1806.08342*, 2018.
- [47] S. Tulyakov *et al.*, “Learning an event sequence embedding for dense event-based deep stereo,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 1527–1537.
- [48] G. Orchard *et al.*, “Hirst: A temporal approach to object recognition,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 37, no. 10, pp. 2028–2040, 2015.
- [49] X. Lagorce *et al.*, “Hots: A hierarchy of event-based time-surfaces for pattern recognition,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 39, no. 7, pp. 1346–1359, 2016.
- [50] M. Cannici *et al.*, “Asynchronous convolutional networks for object detection in neuromorphic cameras,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, Jun. 2019.
- [51] N. Messikommer *et al.*, “Event-based asynchronous sparse convolutional networks,” in *Computer Vision – ECCV 2020*, A. Vedaldi *et al.*, Eds., 2020, pp. 415–431, ISBN: 978-3-030-58598-3.