Delft University of Technology

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

Yang, Y.; Kneip, A.; Frenkel, C.

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 allowing for a 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 $16\mu s$, 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], [2], [3], [4]. While successful in a wide range of applications, from the Internet-of-Things to robotics [5], [6], [7], edge vision systems relying on conventional frame-based cameras are ill-suited for latency-critical scenarios requiring decisions within microseconds, such as autonomous driving or drone navigation tasks [8], [9], [10]. Nonetheless, 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 [11], far exceeding the power budget of common edge vision systems.

Event-based cameras, also called silicon retinas [12] or dynamic vision sensors (DVSes) [13], 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 [14]. 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 [14]. 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 [15], [16], [17], [18], [19], [20]. However, since they are optimized for the mainstream frame-based vision paradigm, techniques for sparse event-based data are still at an early development 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* [21], [22], [23]. 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 [22], 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
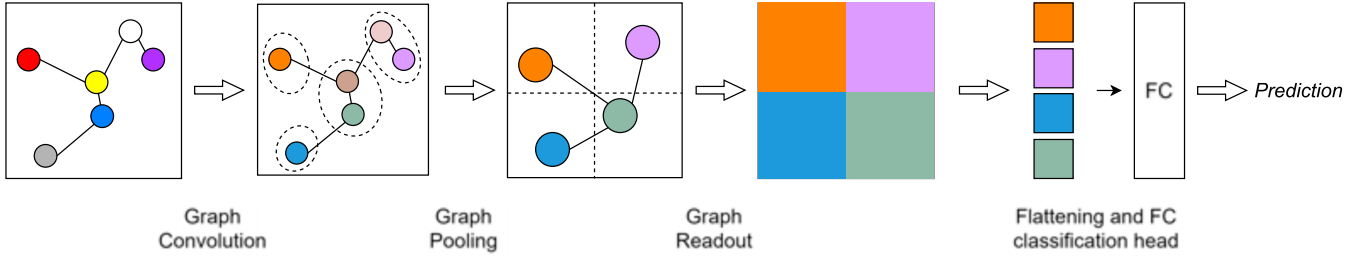
Fig. 1.   Illustration of a typical event GNN pipeline. The colors of nodes represent their features. After the graph convolution, the features are updated (color changed). The graph pooling layer, where certain nodes are selected and edges are re-arranged, simplifies the graph structure through node 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.

recently started being explored [24], [25], [26]. By regarding individual events as *nodes* and their spatiotemporal relationships as *edges*, a stream of events can be represented as 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) [27], [28], 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 [29], [30]. 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 [25], [26], 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.

In this work, we introduce EvGNN, the first <u>ev</u>ent-driven <u>GNN</u> 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. This preserves accuracy while forgoing the need to store edges, thereby drastically reducing the memory footprint.
- We adopt a hardware-friendly *spatiotemporally decoupled prism neighbor search* 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 dimensions, 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, reducing the end-to-end update latency per event.

- We finally *deploy* EvGNN on a Xilinx KV260 Ultrascale+ MPSoC platform and evaluate our design with the real-world dataset N-CARS [31]. We achieve a prediction accuracy of 87.8% with 8b weights and activations, with an average latency per event of 16μs and a total on-chip memory footprint of 1.76MB. We thereby demonstrate that μs-level edge vision can be achieved with low-cost hardware solutions, avoiding the need and cost of fabricating a dedicated accelerator on silicon.

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, which reduces the computational and memory footprints of graph construction and processing steps toward deployment on edge hardware. Section IV covers the proposed EvGNN accelerator, while benchmarking results are finally provided in Section V. Both the proposed algorithm and the hardware are available in open source at https://github.com/cogsys-tudelft/evgnn.

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's 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
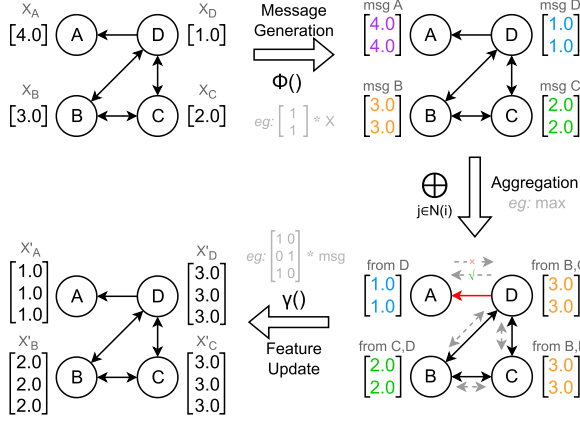
Fig. 2. Illustration of graph convolution, consisting of three steps: message generation, aggregation, and feature update. In the graph, each node i has a feature vector. First, every node generates its message (colored numbers) by the differentiable function $\phi()$ (*message generation*, here illustrated with a simple replication operation). 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).
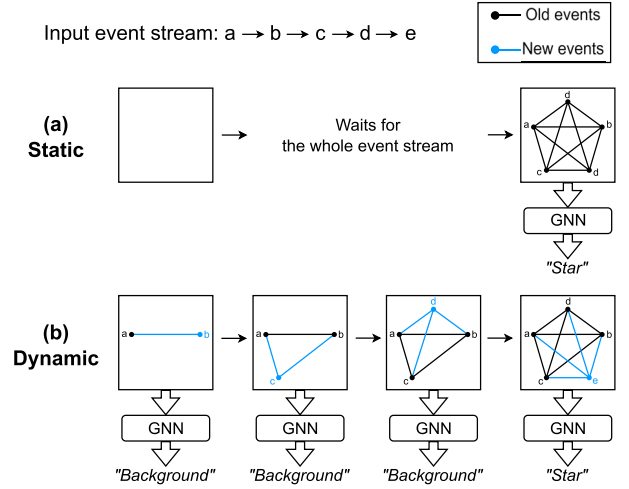

Fig. 3. Static and dynamic event graphs generated from the same event stream. (a) The whole event stream is first transformed into a static event graph, then uses a GNN to provide a prediction result. (b) Whenever a new event is generated (shown in blue), the dynamic event graph is updated and then processed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

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 [32], [33]. 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 nth-order neighbors of the node [34].

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):

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 [35]. As illustrated in Fig. 2, within a graph convolutional layer l, these steps are carried out for each node i as per Eq. (1) [36], ignoring by simplicity any edge feature aside from binary connectivity:

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

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 [37]:
  i) *Message generation*: for each neighbor j, the features of node i and those of neighbor j, $(\mathbf{x}_i^l, \mathbf{x}_j^l)$, go through a learnable differentiable function $\phi()$ (*e.g.,* a linear transformation [27] or a multi-layer perceptron (MLP) [38]).
  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 $\gamma()$ to generate the $\mathbf{x}_i^{l+1}$ output feature vector of node i for the next GNN layer.

2) Graph Pooling: Graph pooling layers down-sample the nodes of the input graph to generate a coarser representation [37]. One typical method, cluster-based graph pooling (*e.g.,* DiffPool [39] and EigenPool [40]), 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 a frame-based representation by globally pooling features of all nodes according to a permutation-invariant function [37], [41], [42]. Several variants exist, such as the grid-based graph readout used in the work of Schaefer et al. [25], which operates by first defining a 2D grid based on the spatial position (x, y) of each node, and then by
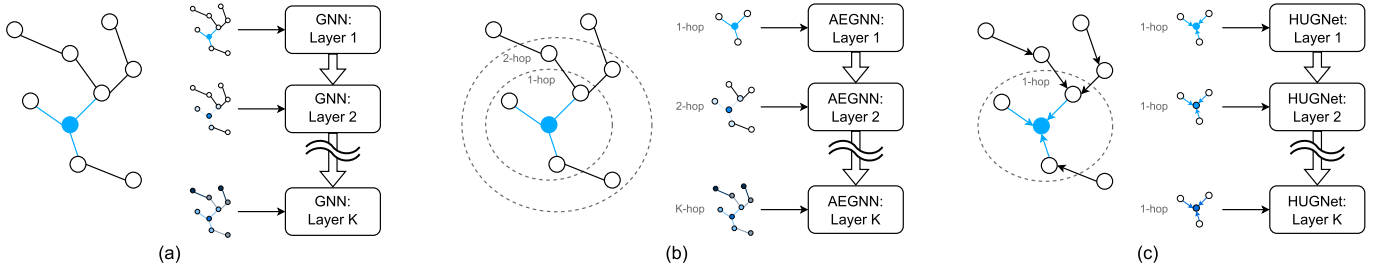
Fig. 4. Event-driven K-layer GNN processing schemes. (a) Naive scheme – A dynamic event graph, with the blue dot representing the new event node, is processed entirely for each layer of a conventional GNN. The colored nodes represent the updated features along the GNN, while uncolored ones depict unchanged features. (b) AEGNN scheme [25] – The same event graph is processed within a k-hop subgraph by the kth layer of the event-driven GNN, only involving nodes whose features will be effectively updated. (c) HUGNet scheme [26] – 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, because the features of neighboring nodes remain untouched.

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 generating the final prediction results.

C. Event Graph Construction and Processing Strategies

An event graph first needs to be constructed before it can be processed by a GNN. The *graph construction* (also referred to as *graph building*) step populates an event 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 [25], [26]. 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 on-the-fly: 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 occurs in an event-driven manner.

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 [24], [32], [33]. 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 [25], [26]: 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 a high computational overhead [25]. 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) [43],

[44]. 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. This technique, introduced in [25] as the asynchronous event-based graph neural network (AEGNN) method and illustrated in Fig. 4(b), significantly reduces computation while maintaining mathematical equivalence.

In their hemi-spherical update graph neural network (HUGNet) solution [26], 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, thereby 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: they do not need to be stored, nor computed beyond this range.

III. HARDWARE-AWARE GRAPH CONSTRUCTION AND PROCESSING

Previous event-driven GNN algorithms (*e.g.,* AEGNN [25] and HUGNet [26]) have been developed without a direct consideration for actual hardware limitations faced by a deployment at the edge, thereby making existing algorithms challenging to map on platforms with restricted memory and computational resources. We introduce the first dedicated hardware platform for event-driven GNNs. In this section, we revisit the key steps of graph construction and processing towards an efficient hardware deployment, while minimizing the penalty on accuracy (see Fig. 5 for an illustration of the overall processing

(a) Event stream with a new event

(b) Event graph construction

(c) Subgraph processing by a multilayer GNN

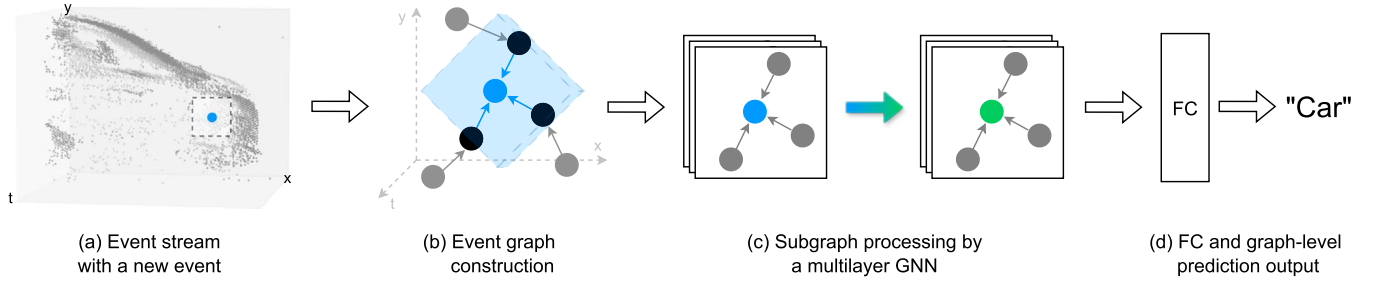(d) FC and graph-level prediction output

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.
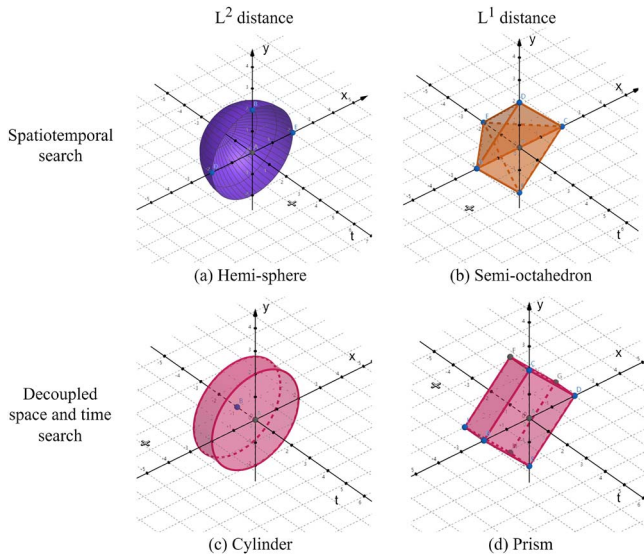

Fig. 6. Illustration of four spatiotemporal neighbor search ranges: (a) hemisphere of radius r, (b) semi-octahedron, (c) cylinder with base radius r_s and height r_t, and (d) prism, obtained by combining two different search schemes with two different L^p distance metrics. Note that the range parameters 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.

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 [24]. To solve this challenge, we propose to leverage directed edges and simplified neighborhood search ranges.

1) Directed Graph Adaptation: Inspired by HUGNet [26], 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* [46]. For nodes in the $\mathcal{R}^3$ spatiotemporal space with position $\mathbf{pos}_i = (x_i, y_i, t_i)$, the raw L^1 and L^2 distances are

$$L^1(\mathbf{pos}_1, \mathbf{pos}_2) = |x_1 - x_2| + |y_1 - y_2| + |t_1 - t_2|$$
$$= |dx| + |dy| + |dt|,$$
$$L^2(\mathbf{pos}_1, \mathbf{pos}_2) = \sqrt{dx^2 + dy^2 + dt^2}.$$

While the L^2 distance is prevalent across previous graph-based event vision algorithms [24], [25], [26], [32], [33], it involves the computationally costly square root function to calculate the spatiotemporal neighborhood range of a node. On the contrary, the L^1 distance only involves addition and absolute value functions, both of which are hardware-friendly.

pipeline). We select the real-world task of car recognition with event-based vision, benchmarked on the N-CARS dataset [31]. This dataset contains several 100-ms samples of event-based camera recordings, representative of an edge-vision use case. As of today, AEGNN is currently the state-of-the-art event-driven GNN approach on N-CARS [25]. It 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 [45]. All experiments below report the average accuracy obtained over five trials.

A. Graph Construction

During the first step of graph construction (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

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%

TABLE III
PERFORMANCE METRICS OF DIFFERENT TYPES OF
GRAPH CONVOLUTION

Convolution Types	Acc. $\pm$ S.D.	#Parameters
Baseline (SplineConv)	95.4% $\pm$ 0.49%	30.4k
GCN conv	92.5% $\pm$ 1.09%	4.7k
This work	95.1% $\pm$ 0.43%	4.8k

TABLE IV
PERFORMANCE METRICS FOR DIFFERENT NETWORK
STRUCTURE SIMPLIFICATIONS

Structure	Acc. $\pm$ S.D.	Param. Mem. Footprint
Baseline (AEGNN, FP32)	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

As illustrated in Fig. 6(a) and 6(b) for directed edges, switching from an L^2 to an L^1 distance transforms the search range from a hemi-sphere to a semi-octahedron.

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 [25], [26] adopted a modified position $\mathbf{pos}_i^*$ with a preset time-scaling factor β in the L^2 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 of following a rescaling approach with a single spatiotemporal distance metric of 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 Fig. 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 with an accuracy drop limited to 0.6%.

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) [27]. Based on Eq. (1), its node feature update at layer

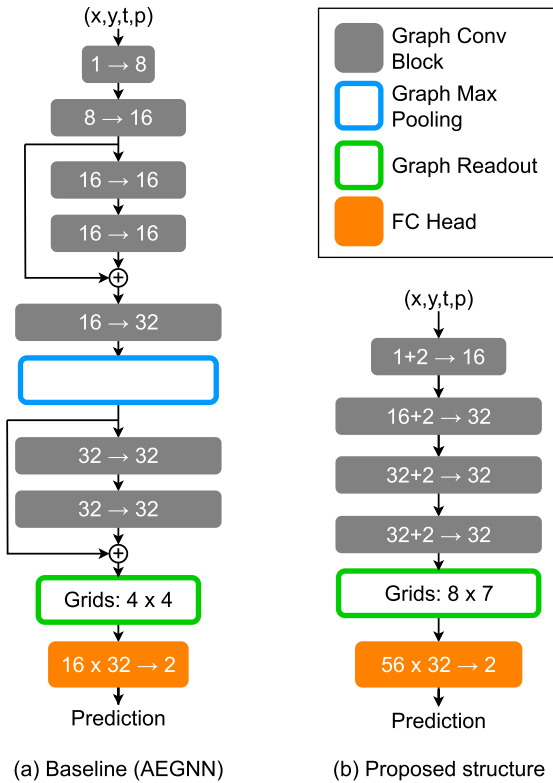l can be expressed for unitary edge and linear operators as

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

where d_i, d_j are the *degrees* (*i.e.* the numbers of neighbors) of nodes i and j, respectively, serving to normalize the features of each neighbor j (*i.e.* message generation), before summing them (*i.e.* aggregation) and applying a linear transformation with learnable parameters $\mathbf{\Theta}_l \in \mathbb{R}^{C_{in} \times C_{out}}$ (*i.e.* feature update). This operation is applied for each convolutional layer l of the GCN. While Eq. (5) only depends on the neighbors' features, and thus hardly exploits information related to the relative positions between two nodes, recent event-based GNNs such as AEGNN [25], [26] have aimed to explicitly exploit this information via *SplineConv*, a graph convolution method derived from [28] 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 classificaton 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* [38], 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

$$\mathbf{x}_i^{l+1} = \gamma_{\mathbf{\Theta}_0}^l \left(\max_{j \in \mathcal{N}(i) \cup \{i\}} \phi_{\mathbf{\Theta}_1}^l (\mathbf{x}_j^l, \mathbf{pos}_j - \mathbf{pos}_i) \right), \quad (6)$$

where $\gamma_{\mathbf{\Theta}_0}^l, \phi_{\mathbf{\Theta}_1}^l$ are multi-layer perceptrons (MLPs) with weights $\mathbf{\Theta}_0$ and $\mathbf{\Theta}_1$, respectively. Such a graph convolution avoids the resource-intensive B-splines while properly beholding spatial information. Still, it requires to store large end-to-end MLP models. Therefore, we introduce here a simplified

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 [47] can be applied. Numbers labeled on graph convolution blocks indicate the corresponding *input → output* channel dimensions. The +2 part in (b) corresponds to the concatenated position differences in Eq. (7).

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

Fig. 9. Hardware diagram of the graph-building module, highlighting the execution flow. For each new event i, a two-step neighbor-search process is carried out among past events, stored in an on-chip buffer following an event-queue format (W×H, i.e. 120×100 queues of 16 entries each in the N-CARS case, to match the DVS camera's dimensions). The search for neighbors follows the spatiotemporally decoupled prism shape, using the event's coordinates (x_i, y_i, t_i). The coordinates of the valid neighbors are buffered and sent to the graph convolution unit for subsequent processing.

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

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

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

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 [47] together with post-training quantization [48] to INT8, allows reducing the required number of parameters by $5\times$ without adversely affecting the classification accuracy (Table IV). Note that, since quantizing from 32-bit floating-point (FP32) to INT8 only induces a 0.1-% accuracy drop, FP8 has not been considered an option worth investigating. Indeed, FP8 MAC units require $23\times$ more resources than their INT8 counterpart when targeting an FPGA implementation [49] and suffer from a $3\times$ higher delay per

MAC operation [50], which would adversely impact latency for negligible accuracy gains over INT8.

IV. PROPOSED HARDWARE ARCHITECTURE FOR EVENT-DRIVEN GNNS

In this section, we design a hardware architecture that accelerates the graph construction (*i.e.* building) 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, serving here as demonstrators for μs-level latency in low-cost edge vision.

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 EvGNN hardware accelerator is located in the PL, while an ARM core in the PS acts as a host CPU that
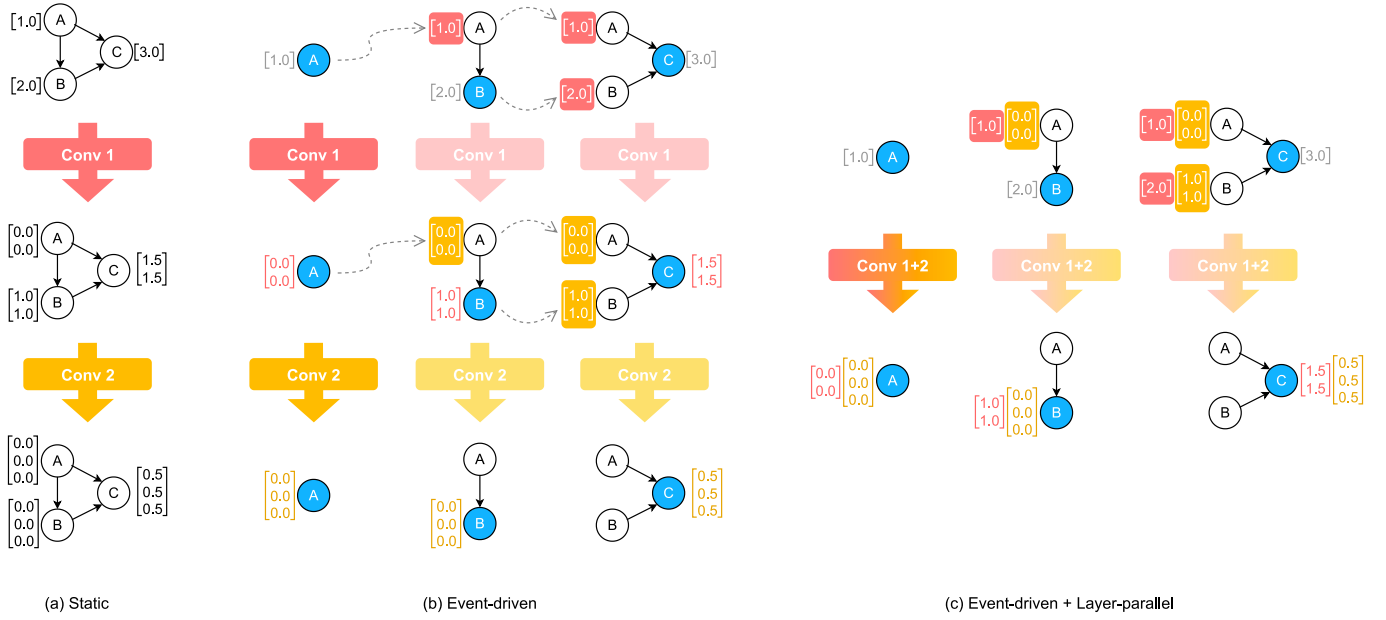
Fig. 10.   Event graphs processed by an example GNN containing two graph convolutional layers, *Conv 1* and *Conv 2*, both of which consist of a simple average-and-replicate operation for illustration purposes. (a) A directed *static* event graph is processed by the GNN. Each graph convolutional layer is applied sequentially, while 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 as an input by the *next* layer for *future* events, as shown with dotted gray arrows. (c) The proposed layer-parallel execution of the same directed graph processed by an event-driven GNN, which is mathematically equivalent to (b).

is responsible for (i) loading dataset samples from the external DRAM, (ii) feeding data to and reading prediction results from EvGNN through AXI system buses, and (iii) calculating the prediction accuracy and measuring the runtime.

The EvGNN accelerator consists of three major blocks. The *Datapath* block is the core implementation of our event-driven GNN algorithm in hardware. It mainly consists of a graph building 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, broadcasts them to the *Datapath* block, then sends the prediction results 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 *Datapath* block for graph building and GNN computation.

B. Graph-Building 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 block diagram of the graph-building module is shown in Fig. 9, together with an illustration of its execution flow. This module 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* [24],

[51], as depicted in Fig. 9. All queues are stored in a *global event-queues buffer*, where each queue is associated with a pixel of the input DVS 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, whereas 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, while temporal information is found inside the entries of a queue. This allows the graph-building module to perform the spatial and temporal search steps of Eq. (4) independently. First, according to the (x_i, y_i) location of the new event, *spatial search* (Fig. 9, orange) is performed to pick all event queues within an L^1 spatial distance r_s from the new event. They are accessed from the event-queues buffer and transferred to a *candidate-events buffer*, skipping out-of-bound locations around the target node. Then, the *temporal search* process (Fig. 9, green), which can be pipelined with the spatial one, subtracts the timestamps t_j of each candidate event from the current timestamp t_i. The obtained time difference dt_{ij} is then compared to the temporal radius r_t to select valid neighbors amongst candidates, which are sent to the graph-convolution module and stored in the graph
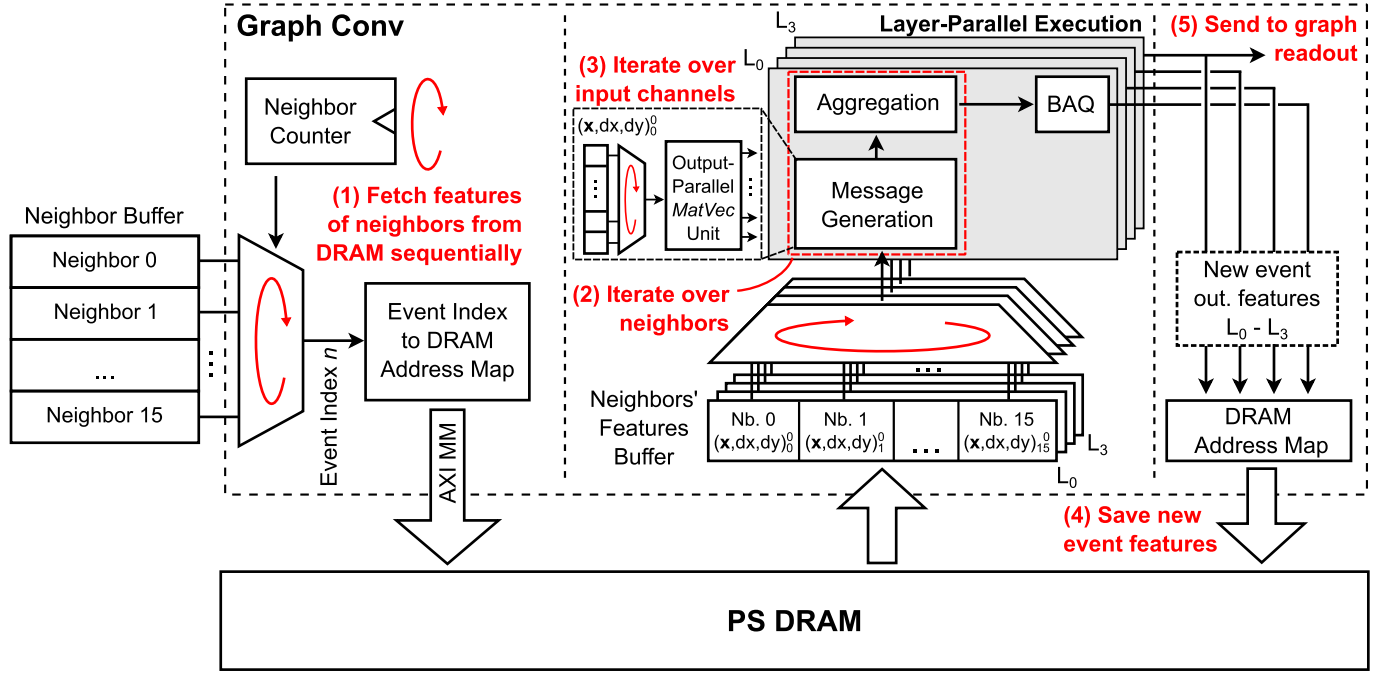
Fig. 11.   Hardware diagram of the graph convolution module, corresponding to the four-layer network architecture in Fig. 7. Node features are first loaded from off-chip DRAM to the graph convolution module through AXI MM buses (i), according to the nodes in the neighbor buffer. Afterwards, for every 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 is parallelized in the *MatVec* unit 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 sent to the readout unit (v).

convolution unit's neighbor buffer (Fig. 11, left). The buffer's depth $D_{max} = 16$ bounds the maximum number of neighbors per new event, triggering an early stopping of the search process once full.

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. 10(a)), the features of all nodes change after each convolutional layer. As updating the node features of the current layer depends on the new features of the previous one, layers need to be executed sequentially. However, when using dynamic directed event graphs, 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. 10(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. 10(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 bounds the total execution time to that of the largest GNN layer, and thus drastically shortens the overall per-event latency in multi-layer GNNs while leaving the total number of operations unchanged.

2) Graph-Convolution Hardware Architecture: The hardware diagram of our graph-convolution module with layer-parallel execution is shown in Fig. 11. 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 (1). 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 of Eq. (7), parallelized across all four GNN layers as per the selected network architecture (Fig. 7). The features vector $(\mathbf{x}_j, |dx_{ij}|, |dy_{ij}|)^l$ of each neighbor, including its relative position information, are sequentially broadcast to the message generation and aggregation sub-units for every layer l in parallel (2). As detailed in Fig. 12, each message-generation unit performs a 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 and is parallelized across all C_{out}^l output channels by means of a *MatVec* unit, which embeds one digital multiply-and-accumulate
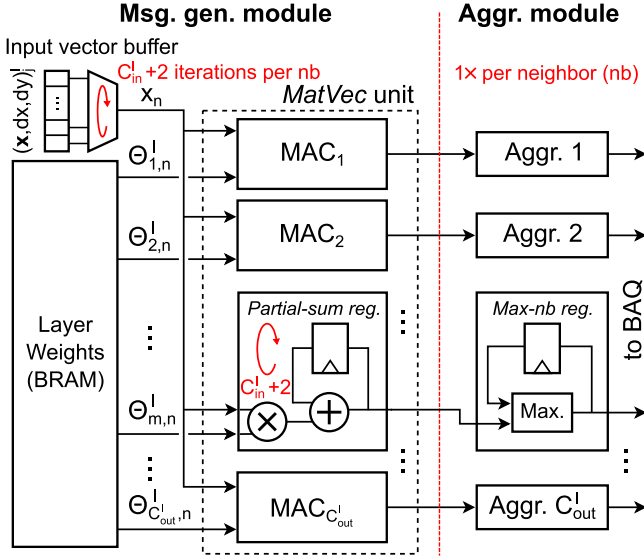
Fig. 12. Hardware diagram of the message generation (Msg. gen.) and aggregation (Aggr.) sub-units, including a *MatVec* unit and BRAM weight storage. The input vector $(\mathbf{x}_j, dx_{ij}, dy_{ij})$ of the selected neighbor j contains the C_{in} concatenated node features of the current layer l and 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.

(MAC) engine per output channel (3). Then, the aggregation unit 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 (2). Finally, a *bias* term, ReLU *activation* function, and *quantization* to INT8 is applied to each selected output, which are abbreviated as *BAQ* in Fig. 11.

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 (4), while the last layer's output is sent to the graph-readout module to serve in the graph prediction update (5).

D. Graph-Readout and FC Modules

Once a new event has been received and processed by the *Graph-building* 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. 12.

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%

V. RESULTS

We evaluate the performance of the designed EvGNN accelerator on the N-CARS dataset after implementing it on a resource-constrained edge 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 carried out on NVIDIA RTX A6000 GPUs while implemented, trained, and tested using the PyTorch Geometric (PyG) library [36]. Similarly to AEGNN [25], 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.

On the hardware side, the EvGNN accelerator 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 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-convolution 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-building module.

State-of-the-art event-driven algorithms are summarized in Table VI and compared to EvGNN in terms of computational complexity (MFLOPs per event) and classification accuracy. Methods described as *local* only process a neighboring region around the new event. H-First [52] uses spiking neural networks

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

Networks	Type	Local	MFLOPs/ev[+]	Accuracy
H-First [52]	Spikes	✗	N/C	56.1%
HATS [31]	TS	✗	N/C	90.2%
YOLE [53]	EH	✓	328	92.7%
AsyNet [54]	EH	✓	21.5	94.4%
NVS-S [24]	Graph	✓	5.2	91.5%
EvS-S [24]	Graph	✓	6.1	93.1%
AEGNN [25]	Graph	✓	0.6[*]	86.7%[*]
Ours (software)	Graph	✓	0.07	88.0%
Ours (hardware)	Graph	✓	0.07 (INT8)	87.8%

[+]Algorithmic computational efficiency as the total number of floating-point operations (MFLOPs) per event, at the exception of the EvGNN hardware using INT8 OPs. 1 MAC = 2 OPs.
[*]Based on the AEGNN open-source code on the N-CARS test set and on updated work [55].

Fig. 13. Classification accuracy obtained with EvGNN as a function of the number of events on samples of the N-CARS test set.

(SNNs) to process event-based data, which support event-driven processing but are updated in a global fashion for each new event. HATS [31] exploits time-surface (TS) representations of event streams, which are processed by global classifiers. YOLE [53] and AsyNet [54] 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 [24], as well as AEGNN from [25] 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. In contrast to these methods, our approach cuts the computational complexity of convolutional operations by relying on local computation within 1-hop subgraphs, in an event-driven fashion. Despite being optimized for low-footprint edge applications, we achieve classification accuracies of 88.0% (FP32, software) and 87.8% (INT8, hardware) on the N-CARS test set, which outperforms the classification accuracy of AEGNN. The achieved accuracy thus remains competitive with state-of-the-art works, while EvGNN enables an improvement in computational efficiency from two to three orders of magnitude. The dynamic, event-by-event classification accuracy of EvGNN is reported in Fig. 13, which shows that, for low-latency applications, reliable classification can be obtained without having to process the full duration of N-CARS samples, with an average latency per event of $16\mu s$. Hence, the event-driven GNN hardware acceleration enabled by EvGNN allows exploiting the μs-level resolution of event-based cameras at the edge.

C. Synchronous vs. Asynchronous GNN Accelerators

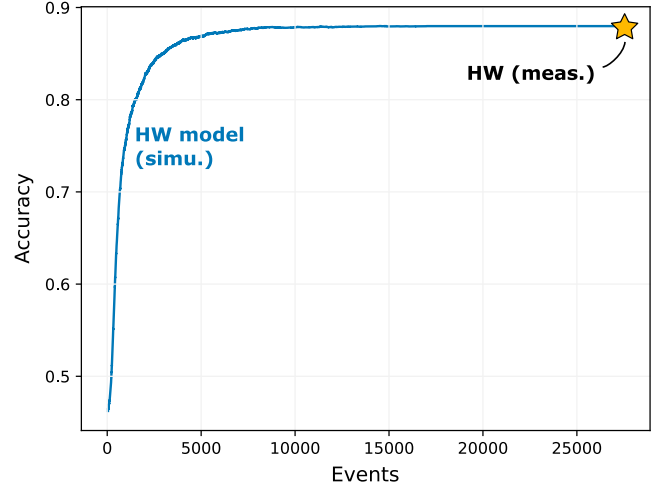While hardware acceleration for GNNs has started gaining interest [29], [56], we would like to further emphasize that prior designs all target the *synchronous* processing of large, *static* graphs. Their typical target workloads consist of large-scale databases [57] or molecular structures [58], for which the real-time constraint is typically on the order of milliseconds or more, orders of magnitude above the few-μs target of the latency-critical event-driven systems considered in this work. Synchronous GNN accelerators can therefore afford a priori analysis of the graph to harness partitioning and execution-rescheduling techniques, which reduce irregular data transfers and thereby improve the energy efficiency [56], [59] at the expense of end-to-end latency, from graph loading to inference update. Besides, the synchronous processing in these accelerators requires the storage of the entire graph structure, including all edge-related information. This storage amounts to a $>50\%$ memory overhead [56], which is not desirable for a compact deployment at the edge. These fundamental differences in constraints and use cases preclude a direct performance comparison between both kinds of architectures. Recent partial asynchronous acceleration in [60] resulted in a latency per event of 4.5ms, which is still beyond the target μs-level range.

D. Silicon Implementation and Scalability Perspectives

Our FPGA demonstrator paves the way to energy-efficient on-silicon accelerators based on the EvGNN architecture. To estimate the expected latency and energy of such a system, we developed a cycle-accurate model of the accelerator using Python, as described in Fig. 14(a). We consider a 28nm bulk CMOS process node, a unified core voltage of 0.8V, a frequency of 200MHz, and an off-chip bandwidth for DRAM transfers of 3.2Gb/s. For energy estimates, logic units (MACs) rely on a scaled version of [61] to 28nm, while both on-chip SRAM buffers and DRAM rely on the CACTI estimator [62], which is the de-facto tool for modeling memories. The resulting EvGNN application-specific integrated circuit (ASIC) accelerator would occupy a silicon area of 2.5mm², while achieving a per-event latency and energy of 10.7μs and 305nJ on the
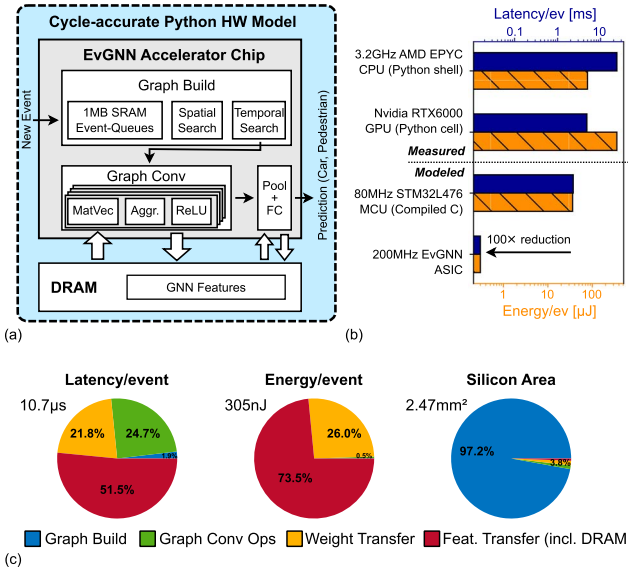
Fig. 14. (a) Block diagram of the EvGNN-based system considered for cycle-accurate modeling in a 28nm bulk CMOS process using Python. (b) Per-event latency and energy comparison between EvGNN's algorithm execution on different platforms: a high-performance CPU, a GPU (measured), a commercial MCU, and the EvGNN accelerator (simulated model). (c) Latency, energy and area breakdown of the modeled EvGNN ASIC accelerator. Feature and weight transfers dominate the latency and energy fraction, whereas the event-queue buffer occupies most of the silicon area.

N-CARS dataset, respectively. In particular, the proposed layer-parallel execution scheme reduces the latency footprint of the graph convolution from $6.9\mu s$ to $2.6\mu s$, an improvement by $2.5\times$. These numbers are orders of magnitude below what is achievable by high-performance CPUs (AMD EPYC 7763 64-Core Processor 3.2GHz) or GPUs (NVIDIA RTX A6000) running the developed algorithm in Python (Fig. 14(b)). They are also $100\times$ better than the estimated metrics for an off-the-shelf deployment on a modeled micro-controller unit (MCU) based on [63], where the event-driven algorithm is executed in compiled C code. This gap in performance further emphasizes the importance of building dedicated event-driven GNN accelerators to enable efficient irregular graph processing at the edge.

While our EvGNN accelerator can map event-driven GNNs of any size and achieves excellent latency and energy results on the N-CARS dataset, several challenges may appear when scaling up to larger, real-world scenarios (e.g., [55]). In particular, three main challenges would pressure the latency, energy and area breakdown of the accelerator in Fig. 14(c), for which we identify possible solutions. First, scaling to high-resolution DVS cameras (e.g., QVGA [64] or HD [65] resolutions) would induce a quadratic increase in the size of the event-queue buffer, demanding significant silicon area for an irregular, event-dependent utilization of the storage space. Moving the storage of event queues off-chip would alleviate this area bottleneck at the cost of $\sim 10\%$ additional latency and energy, from our model estimates. Avoiding the storage of one event queue per pixel would be another viable option, at the expense of a more complex neighbor-search process. Second, the continuous transfer of data features to/from the DRAM translates into

significant latency and energy overheads (Fig. 14(c)), which would linearly scale with the layer size. Adding smart caching strategies to efficiently reuse neighbor information between successive events would tackle this overhead. Finally, relying on a 2D array of processing elements with higher data stationarity (e.g., compute-in memory [66]) would improve the latency of every GNN layer execution, which would prove particularly critical for larger layers, as the latency of weight transfers scales quadratically with the layer size. These considerations pave the way for improving the efficiency of future on-silicon GNN accelerator designs.

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 real-time and microsecond-latency event-based vision intelligence at the edge.

ACKNOWLEDGMENT

REFERENCES

[1] Z. Wang, K. Xu, S. Wu, L. Liu, L. Liu, and D. Wang, "Sparse-YOLO: Hardware/software co-design of an FPGA accelerator for YOLOv2," *IEEE Access*, vol. 8, pp. 116569–116585, 2020.

[2] P. Bonazzi, T. Rüegg, S. Bian, Y. Li, and M. Magno, "TinyTracker: Ultra-fast and ultra-low-power edge vision in-sensor for gaze estimation," in *Proc. IEEE Sensors*, 2023, pp. 1–4.

[3] F. Conti et al., "PULP: A ultra-low power parallel accelerator for energy-efficient and flexible embedded vision," *J. Signal Process. Syst.*, vol. 84, pp. 339–354, Nov. 2016.

[4] J. Tang, S. Liu, L. Liu, B. Yu, and W. Shi, "LoPECS: A low-power edge computing system for real-time autonomous driving services," *IEEE Access*, vol. 8, pp. 30467–30479, 2020.

[5] S. Benninger, M. Magno, A. Gomez and L. Benini, "EdgeEye: A long-range energy-efficient vision node for long-term edge computing," in *Proc. Int. Green Sustain. Comput. Conf. (IGSC)*, 2019, pp. 1–8.

[6] H. Genc, Y. Zu, T. -W. Chin, M. Halpern, and V. J. Reddi, "Flying IoT: Toward low-power vision in the sky," *IEEE Micro*, vol. 37, no. 6, pp. 40–51, Nov./Dec. 2017.

[7] L. Huang, G. Wu, W. Tang, and Y. Wu, "Obstacle distance measurement under varying illumination conditions based on monocular vision using a cable inspection robot," *IEEE Access*, vol. 9, pp. 55955–55973, 2021.

[8] A. Suleiman, Z. Zhang, L. Carlone, S. Karaman, and V. Sze, "Navion: A fully integrated energy-efficient visual-inertial odometry accelerator for autonomous navigation of nano drones," in *Proc. IEEE Symp. VLSI Circuits*, 2018, pp. 133–134.

[9] M. Arafat, M. Alam, and S. Moh, "Vision-based navigation techniques for unmanned aerial vehicles: Review and challenges," *Drones*, vol. 7, no. 89, pp. 1–41, 2023.

[10] A. Gupta et al., "Deep learning for object detection and scene perception in self-driving cars: Survey, challenges, and open issues," *Array*, vol. 10, pp. 1–20, Jul. 2021.

[11] M. Hillebrand, N. Stevanovic, B. J. Hosticka, J. E. Santos Conde, A. Teuner, and M. Schwarz, "High speed camera system using a CMOS image sensor," in *Proc. IEEE Intell. Vehicles Symp. (Cat. No.00TH8511)*, 2000, pp. 656–661.

[12] M. A. Mahowald and C. Mead, "The silicon retina," *Sci. Ame.*, vol. 264, no. 5, pp. 76–82, 1991.

[13] P. Lichtsteiner, C. Posch, and T. Delbruck, "A 128× 128 120 db 15 μs latency asynchronous temporal contrast vision sensor," *IEEE J. Solid-State Circuits*, vol. 43, no. 2, pp. 566–576, Feb. 2008.

[14] G. Gallego et al., "Event-based vision: A survey," *IEEE Trans. pattern Anal. Mach. Intell.*, vol. 44, no. 1, pp. 154–180, Jan. 2020.

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, May 2017, doi: 10.1145/3065386.

[16] K. He, X. Zhang, S. Ren, J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[17] P. Adarsh, P. Rathi, and M. Kumar, "YOLO v3-Tiny: Object detection and recognition using one stage improved model," in *Proc. Int. Conf. Adv. Comput. Commun. Syst. (ICACCS)*, 2020, pp. 687–694.

[18] A. Howard et al., "Searching for MobileNetV3," in *Proc. IEEE/CVF Int. Conf. Comput. Vision*, 2019, pp. 1314–1324.

[19] J. Zhong, J. Chen, and A. Mian, "DualConv: Dual convolutional kernels for lightweight deep neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 34, no. 11, pp. 9528–9535, Nov. 2023.

[20] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A survey of convolutional neural networks: Analysis, applications, and prospects," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 33, no. 12, pp. 6999–7019, Dec. 2022.

[21] D. Gehrig, A. Loquercio, K. G. Derpanis, and D. Scaramuzza, "End-to-end learning of representations for asynchronous event-based data," in *Proc. IEEE/CVF Int. Conf. Comput. Vis.*, 2019, pp. 5633–5643.

[22] M. Liu and T. Delbruck, "Adaptive time-slice block-matching optical flow algorithm for dynamic vision sensors," in *Proc. Brit. Mach. Vis. Conf. (BMVC)*, Sep. 2018, doi: 10.5167/uzh-168589.

[23] A. Nguyen et al., "Real-time 6DOF pose relocalization for event cameras with stacked spatial LSTM networks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR) Workshops*, Jun. 2019.

[24] Y. Li H. Zhou, B. Yang, Y. Zhang, Z. Cui, H. Bao, and G. Zhang, "Graph-based asynchronous event processing for rapid object recognition," in *Proc. IEEE/CVF Int. Conf. Comput. Vis.*, 2021, pp. 934–943.

[25] S. Schaefer, D. Gehrig, and D. Scaramuzza, "AEGNN: Asynchronous event-based graph neural networks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2022, pp. 12371–12381.

[26] T. Dalgaty T. Mesquida, D. Joubert, A. Sironi, P. Vivet, and C. Posch, "HUGNet: Hemi-spherical update graph neural network applied to low-latency event-based optical flow," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR) Workshops*, Jun. 2023, pp. 3952–3961.

[27] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016, *arXiv:1609.02907*.

[28] M. Fey, J. E. Lenssen, F. Weichert, and H. Müller, "SplineCNN: Fast geometric deep learning with continuous b-spline kernels," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 869–877.

[29] R. Sarkar, S. Abi-Karam, Y. He, L. Sathidevi, and C. Hao, "FlowGNN: A dataflow architecture for real-time workload-agnostic graph neural network inference," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, 2023, pp. 1099–1112.

[30] Z. Zhou, B. Shi, Z. Zhang, Y. Guan, G. Sun, and G. Luo, "Block-GNN: Towards efficient GNN acceleration using block-circulant weight matrices," in *Proc. ACM/IEEE Des. Automat. Conf. (DAC)*, 2021, pp. 1009–1014.

[31] A. Sironi, M. Brambilla, N. Bourdis, X. Lagorce, and R. Benosman, "HATS: Histograms of averaged time surfaces for robust event-based object classification," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 1731–1740.

[32] A. Mitrokhin, Z. Hua, C. Fermuller, and Y. Aloimonos, "Learning visual motion segmentation using event surfaces," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2020, pp. 14414–14423.

[33] Y. Bi, A. Chadha, A. Abbas, E. Bourtsoulatze, and Y. Andreopoulos, "Graph-based spatio-temporal feature learning for neuromorphic vision

sensing," *IEEE Trans. Image Process.*, vol. 29, pp. 9084–9098, 2020.

[34] W. Liu and Z. Li, "An efficient parallel algorithm of n-hop neighborhoods on graphs in distributed environment," *Frontiers Comput. Sci.*, vol. 13, pp. 1309–1325, Jul. 2019.

[35] M. M. Bronstein et al., "Geometric deep learning: Grids, groups, graphs, geodesics, and gauges," 2021, *arXiv:2104.13478*.

[36] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," 2019, *arXiv:1903.02428*.

[37] Z. Wu et al., "A comprehensive survey on graph neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 32, no. 1, pp. 4–24, Jan. 2021.

[38] C. R. Qi, H. Su, K. Mo, L. J. Guibas, "PointNet: Deep learning on point sets for 3D classification and segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 652–660.

[39] Z. Ying, J. You, C. Morris, X. Ren, W. Hamilton, and J. Leskovec, "Hierarchical graph representation learning with differentiable pooling," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 31, 2018, pp. 1–11.

[40] Y. Ma, S. Wang, C. C. Aggarwal, and J. Tang, "Graph convolutional networks with eigenpooling," in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discovery & Data Mining*, 2019, pp. 723–731.

[41] R. Li, S. Wang, F. Zhu, J. Huang, "Adaptive graph convolutional neural networks," in *Proc. AAAI Conf. Artif. Intell.*, vol. 32, no. 1, 2018, pp. 3546–3553.

[42] J. Atwood and D. Towsley, "Diffusion-convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 29, 2016, pp. 1–9.

[43] P. Quan et al., "A brief review of receptive fields in graph convolutional networks," in *Proc. IEEE/WIC/ACM Int. Conf. Web Intell.-Companion Volume*, 2019, pp. 106–110.

[44] Z. Zhang et al., "Multireceptive field: An adaptive path aggregation graph neural framework for hyperspectral image classification," *Expert Syst. with Appl.*, vol. 217, 2023, Art. no. 119508. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095741742300009X

[45] A. Rinaldo, L. Wasserman, and M. G'Sell, "Bootstrapping and sample splitting for high-dimensional, assumption-lean inference," *Ann. Statist.*, vol. 47, no. 6, pp. 3438–3469, 2019, doi: 10.1214/18-AOS1784.

[46] J. Brimberg and R. F. Love, "General considerations on the use of the weighted lp norm as an empirical distance measure," *Transportation Sci.*, vol. 27, no. 4, pp. 341–349, 1993.

[47] B. Jacob et al., "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proc. IEEE Conf. Comput. vision pattern Recognit.*, 2018, pp. 2704–2713.

[48] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," 2018, *arXiv:1806.08342*.

[49] Z. Carmichael et al., "Deep positron: A deep neural network using the posit number system," in *Proc. Des., Automat. & Test in Europe Conf. & Exhib. (DATE)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 1421–1426.

[50] Z. Carmichael et al., "Performance-efficiency trade-off of low-precision numerical formats in deep neural networks," in *Proc. Conf. Next Gener. Arithmetic*, 2019, pp. 1–9.

[51] S. Tulyakov, F. Fleuret, M. Kiefel, P. Gehler, and M. Hirsch, "Learning an event sequence embedding for dense event-based deep stereo," in *Proc. IEEE/CVF Int. Conf. Comput. Vis.*, 2019, pp. 1527–1537.

[52] G. Orchard, C. Meyer, R. Etienne-Cummings, C. Posch, N. Thakor, and R. Benosman, "HFirst: A temporal approach to object recognition," *IEEE Trans. pattern Anal. Mach. Intell.*, vol. 37, no. 10, pp. 2028–2040, 2015.

[53] M. Cannici, M. Ciccone, A. Romanoni, and M. Matteucci, "Asynchronous convolutional networks for object detection in neuromorphic cameras," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR) Workshops*, Jun. 2019.

[54] N. Messikommer, D. Gehrig, A. Loquercio, and D. Scaramuzza, "Event-based asynchronous sparse convolutional networks," in *Proc. Comput. Vis. (ECCV)*, A. Vedaldi, H. Bischof, T. Brox, and J.-M. Frahm, Eds., 2020, pp. 415–431.

[55] D. Gehrig and D. Scaramuzza, "Low-latency automotive vision with event cameras," *Nature*, vol. 629, pp. 1034–1040, May 2024.

[56] Z. Zhu, X. Liang, and J. Cheng, "MEGA: A memory-efficient GNN accelerator exploiting degree-aware mixed-precision quantization," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, 2024, pp. 124–138. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10476431

[57] Kaggle. "Journal ranking databases," Kaggle. 2024. [Online]. Available: https://www.kaggle.com/datasets/xabirhasan/journal-ranking-dataset

[58] P. Bongini et al., "Molecular generative graph neural networks for drug discovery," *Neurocomputing*, vol. 450, pp. 242–252, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231221005737

[59] W. L. Hamilton et al., "Inductive representation learning on large graphs," 2018, *arXiv:1706.02216v4*.

[60] K. Jeziorek et al., "Embedded graph convolutional networks for real-time event data processing on SoC FPGAs," 2024, *arXiv:2406.07318*.

[61] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *Proc. IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers (ISSCC)*, 2014, pp. 10–14.

[62] R. Balasubramonian et al., "CACTI 7: New Tools for interconnect exploration in innovative off-chip memories," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, pp. 1–25, 2017, doi: 10.1145/3085572.

[63] S. Microelectronics. "STM32L476xx: Ultra-low-power Arm ® Cortex ® -M4 32-bit MCU+FPU, 100DMIPS,up to 1MB Flash, 128 KB SRAM, USB OTG FS, LCD, ext. SMPS." 2019. [Online]. Available: https://docs.rs-online.com/d8ed/A700000008303040.pdf

[64] M. Gehrig et al., "DSEC: A stereo event camera dataset for driving scenarios," *IEEE Robot. Automat. Lett.*, vol. 6, no. 3, pp. 4947–4954, Jul. 2021.

[65] E. Perot et al., "Learning to detect objects with a 1 megapixel event camera," in *Proc. Adv. Neural Inf. Process. Syst.*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33, Long Beach, CA, USA: Curran Associates, Inc., 2020, pp. 16639–16652. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/c213877427b46fa96cff6c39e837ccee-Paper.pdf

[66] A. Kneip et al., "IMPACT: A 1-to-4b 813-TOPS/W 22-nm FD-SOI compute-in-memory CNN accelerator featuring a 4.2-POPS/W 146-TOPS/mm2 CIM-SRAM with multi-bit analog batch-normalization," *IEEE J. Solid-State Circuits*, vol. 58, no. 7, pp. 1871–1884, Jul. 2023.

Adrian Kneip (Member, IEEE) received the M.Sc. degree (*summa cum laude*) in electrical engineering from the Université catholique de Louvain (UCLouvain), Louvain-la-Neuve, Belgium, in 2019. He then joined UCLouvain's Electronics Circuits and Systems (ECS) Group, where he received the Ph.D. degree in 2024 under the supervision of Prof. D. Bol. His research interests notably include the design of ultra-low-power digital ICs, as well as analog/mixed-signal accelerators for edge-AI chips. He has a particular interest for SRAM-based in-memory computing and hardware/software co-design aspects. Recently, he has also shown a rising interest event-based computing techniques, with a focus on event-driven graph neural networks and their hardware acceleration. Dr. Kneip is the author or co-author of several research papers in IEEE conferences and journals, receiving the Best Student Paper Award for the 2022's ESSCIRC-ESSDERC conference. He also serves as reviewer for various IEEE SSCS and CAS journals, wherein IEEE ESSCERC, ISCAS, JSSC, and TCAS-I/-II. He also was a Student Representative to the IEEE Benelux Section from 2020 to 2023.

Charlotte Frenkel (Member, IEEE) received the M.Sc. degree (*summa cum laude*) in electromechanical engineering and the Ph.D. degree in engineering science from the Université catholique de Louvain (UCLouvain), Louvain-la-Neuve, Belgium, in 2015 and 2020, respectively. In 2020, she joined the Institute of Neuroinformatics, UZH and ETH Zurich, Switzerland, as a Postdoctoral Researcher. She is an Assistant Professor with Delft University of Technology, Delft, The Netherlands, since 2022, and holds a Visiting Faculty Researcher position with Google since 2024. Her research aims at bridging the bottom-up (bio-inspired) and top-down (engineering-driven) design approaches toward neuromorphic intelligence, with a focus on hardware-algorithm co-design for (Neuro)AI, digital hardware accelerators, and brain-inspired on-device learning. She received a best paper award at the IEEE International Symposium on Circuits and Systems (ISCAS) 2020 conference in the *Neural Networks* track, and her Ph.D. thesis was awarded the FNRS-FWO/Nokia Bell Scientific Award 2021 and the FNRS-FWO/IBM Innovation Award 2021. In 2023, she was awarded prestigious Veni and AiNed Fellowship Grants from the Dutch Research Council (NWO). She presented several invited talks, including keynotes at the tinyML EMEA technical forum 2021 and at the Neuro-Inspired Computational Elements (NICE) neuromorphic conference 2021. She serves or has served as a Program Co-Chair of NICE 2023–2024 and of the tinyML Research Symposium 2024, as a Co-Lead of the NeuroBench initiative for benchmarks in neuromorphic computing since 2022, as a TPC Member of IEEE ESSERC for 2022–2024, and as an Associate Editor for IEEE TRANSACTIONS ON BIOMEDICAL CIRCUITS AND SYSTEMS since 2022.

Yufeng Yang received the bachelor's degree in electronic and information engineering from Beihang University, in 2020, and the M.Sc. degree in electrical engineering from Delft University of Technology (TU Delft), in 2023 under the supervision of Prof. Charlotte Frenkel. His research focuses on the hardware-algorithm co-design of modern neural networks and on the design of efficient AI accelerators for resource-constrained embedded hardware platforms.