MSc Thesis

Event-Driven Graph Neural Network Accelerators for Low-Power Vision

by

Yufeng Yang

Student number:	5579309	
Project duration:	December 1, 2022 – September 5, 2023	
Thesis committee:	Prof. Kofi Makinwa,	TU Delft, Thesis advisor
	Dr. Charlotte Frenkel,	TU Delft, Daily supervisor
	Dr. Chang Gao,	TU Delft

TU Delft

Abstract

Event-based cameras promise new opportunities for smart vision systems deployed at the edge. Contrary to their conventional frame-based counterparts, event-based cameras generate temporal light intensity changes as events on a per-pixel basis, enabling ultra-low latency with microsecond-scale temporal resolution, low power consumption at milliwatts level, and sparse information encoding where only dynamic objects trigger events, effectively excluding static background data. However, mainstream computer vision algorithms based on convolutional neural networks (CNNs) hardly exploit these advantages of event-based cameras. Recently, event graph neural networks (event-GNNs) have been proposed as the backbone for novel event-based vision algorithms. By treating events as graph data, GNNs are able to process events while preserving their spatiotemporal information and sparse characteristics. Further studies also revealed an event-driven computation workflow that translates an event stream into a dynamic, evolving graph, outlining a path toward low-latency event-based vision. Despite these promises, event-GNNs are still calling for dedicated hardware accelerators toward integrated solutions with real-time prediction latency and low power consumption for real-world edge intelligence.

In this thesis, for the first time, we proposed an event-driven GNN accelerator for low-power, high-speed edge vision. Through hardware-algorithm co-design, an event-driven GNN model is adopted for deployment on an edge FPGA platform without prediction accuracy loss. We also pointed out two novel optimizations, edge-free storage and layer-parallel computation, to further decrease memory footprints and processing latency. The proposed accelerator is implemented on the Xilinx KV260 System-On-Module (SOM) platform containing an UltraScale+ MPSoC FPGA, and benchmarked on-board. Targeting a car recognition task based on the NCars dataset, our accelerator achieves a prediction accuracy of 87.8%. Meanwhile, operating with a $6.86W$ board-level system power, the accelerator reaches an average $16\mu s$ prediction latency per event and runs $9.2\times$ faster than its software counterpart running on an NVIDIA RTX A6000 GPU platform. Therefore, our event-driven GNN accelerator efficiently allows for both real-time and microsecond-resolution event-based vision at the edge.

Acknowledgments

Though not seem like a long time, I have gained valuable insights during my master's thesis journey during these nine months. First and foremost, I would like to express my sincere gratitude to my supervisor, Dr. Charlotte Frenkel. From the inception of my thesis to the exploratory phases, and ultimately the completion of the design, she has provided meticulous guidance every step of the way. Furthermore, her thorough review and feedback comments on my thesis report have also contributed significantly to my progress. It is through her rigorous and diligent approach to research that I can have the opportunity to knock on the door of the scientific kingdom.

Additionally, I would like to extend my gratitude to the professors and Ph.D. students who have provided guidance. I appreciate the constructive feedback offered by Ph.D. students Nicolas Chauvaux and Fabrizio Ottati on my thesis. I am also thankful to Prof. Kofi Makinwa and Dr. Chang Gao for taking the time to participate in my thesis defense and provide me with valuable guidance.

Furthermore, I am thankful for the friends I have met during my master's program, NingChao, Xinhu, Fengwei, Lorenzo... The help and encouragement from everyone have touched me deeply. Lastly, I would like to extend special thanks to my parents. Despite being far away, they still consistently provided me with the most timely and warm comfort, letting me know that they are forever my most loyal supporters.

Contents

1

Introduction

Edge vision systems, combining digital cameras and edge computing devices deployed near them, aim to capture environmental visual information and provide basic decisions with a limited power budget in the order of milliwatts, which have been widely used in various fields, such as the Internet-of-Things or robotics sensing. However, in scenarios needed for microsecond-level low-latency awareness, such as autonomous navigation [33], edge vision systems leveraging traditional cameras may not be optimal choices. Indeed, normal video cameras capturing 30-60 frames per second (FPS) may not meet these speed requirements, while the power dissipation of 1000-FPS high-speed camera systems can reach tens of Watts [12], easily exceeding power budgets of common edge vision systems.

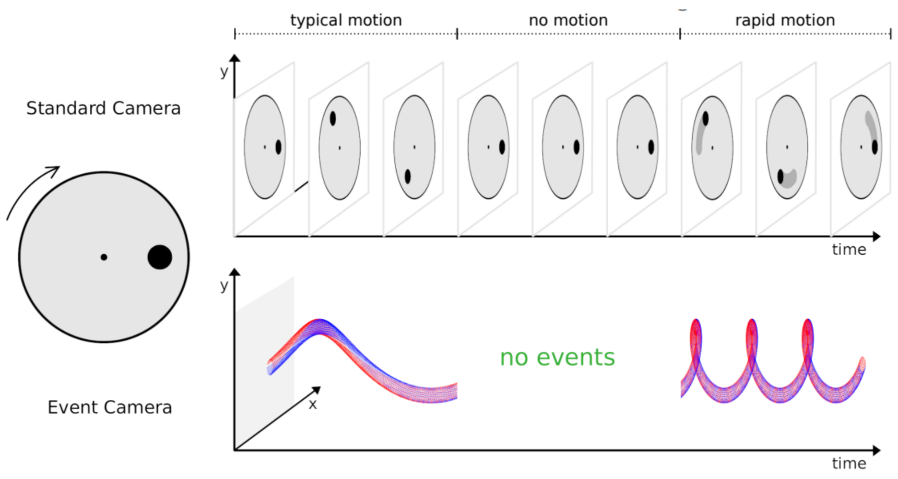


Figure 1.1: Data generated by a standard frame-based camera and by an event-based camera when recording a video about a black dot on a rotating plate. Top: the frame-based camera continuously outputs image-frame data even when the plate stops, and encounters motion blur for a rapid motion. Bottom: the event-based camera only outputs events (denoted as blue and red dots) corresponding to the rotating black dot, which are not generated by the static background (the plate), and stop generating when there is no motion. Adopted from [15].

The event-based camera, also called silicon retina [22] or dynamic vision sensor (DVS) [20], has obtained growing interest as a candidate for establishing low-power high-speed edge vision. The working principle of event-based cameras is demonstrated in Fig. 1.1. As opposed to their frame-based counterparts, which record absolute light intensity at every pixel in the field of view at a fixed sampling rate, event-based cameras are locally sensitive: the pixels in event-based cameras are asynchronously activated only when the received relative light intensity changes exceed a preset threshold, and emit a binary signal, named an event, to indicate the changes. Therefore, the static background information in video stream data delivers no light intensity changes to the event-based camera and is filtered out, leaving only the motion objects detected, thus causing inherent spatiotemporal sparsity of the event stream data. This sparsity also provides opportunities for event-based cameras to achieve low-power sensing at tens of milliwatts level, low latency and high temporal resolution (both at a microsecond scale) that eliminates motion blur in frame-based high-speed vision.

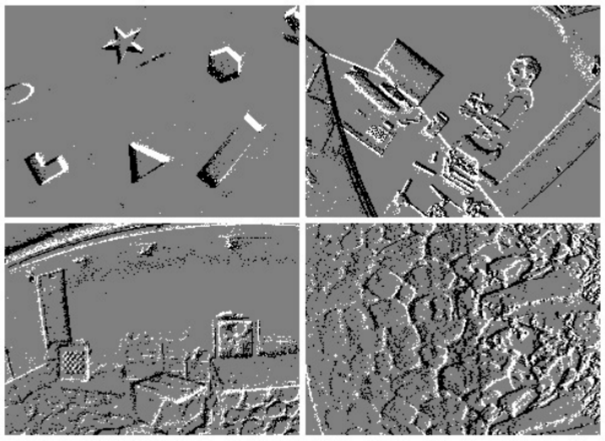
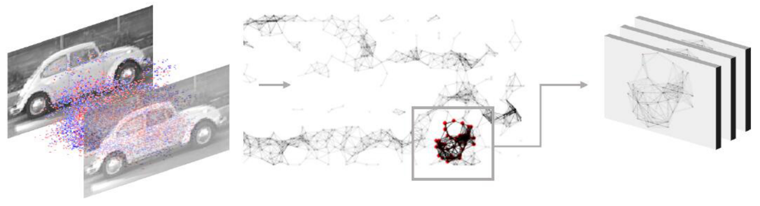


Figure 1.2: Images generated by the dense-frame event data processing approach. Events are accumulated and form high-contrast edge lines on a grey background. Adopted from [24]

However, current visual processing algorithms have difficulty processing event stream data generated from event-based cameras efficiently. Though the mainstream artificial intelligence (AI) approaches based on neural network algorithms, *e.g.* convolutional neural networks (CNNs) [18], have already demonstrated excellent performance for various computer vision tasks like object recognition or detection, their frame-based natures are still incompatible with sparse, asynchronous event data. One straightforward solution is converting the event data into image-like data, which is also known as the dense-frame approach [10, 21, 24], *i.e.* accumulating every event from each pixel within a given accumulation time window (Fig. 1.2). While these methods are able to leverage mature frame-based algorithms to process the transformed event data, the high temporal resolution and the sparse nature of event data are lost. In fact, to obtain processable image-like data, the accumulation window in the dense-frame approaches needs to be on the order of milliseconds, [21] thus discarding the original microsecond-level resolution of event data. Besides, to perform computer vision tasks, CNN algorithms have to process the whole transformed images naturally containing static background pixels, thereby directly translating into redundant computation, leading to higher power consumption, and missing key advantages of event-based sensors, which are not favorable characteristics for developing edge vision systems.

To solve these issues, graph neural networks (GNNs) [16, 8, 30, 19] have recently been proposed as the backbone structure of novel event-based vision algorithms that aim to process event data directly. A *graph* is a specific data structure consisting of *nodes* and *edges*, representing data points and their relationship respectively. By regarding individual events as nodes and their spatiotemporal relationship as edges, we can establish a specific graph that stores the spatial location information and the triggering history of all events. Therefore, event stream data is able to convert into a fully equivalent graph without impairment on fine-grained time resolution and inherent data sparsity, which is called the *event graph*. Subsequently, a



(a) Event Stream (b) Sparse Event Graph (c) Event-GNN

Figure 1.3: Relationship between event stream data, event graphs, and event-GNNs, modified from [30].

following GNN can learn patterns inside the event graphs and deliver prediction results, which establishes a system called *event-GNN*, as shown in Fig. 1.3. Event-GNNs also allow for *event-driven* computation: new events in a stream can be processed immediately, leading to continuously updating prediction results, which can further decrease the power consumption while maintaining the prediction accuracy [30, 5]. Recent research has demonstrated a near-equal performance between the sparse event-GNNs and dense-frame approaches in various computer vision tasks, such as object recognition, detection, and segmentation [3, 23, 30].

Despite its advantages, the event-GNN is still an emerging approach calling for appropriate hardware for real-world applications. Traditional neural networks leverage CPUs or GPUs, but their power consumption may reach hundreds of watts that significantly exceed the milliwatt-to-watt power budgets of edge devices. Though several general GNN hardware accelerators have already achieved energy efficiency improvement by more than two orders of magnitude over the CPU or GPU platforms [29, 39], they still focus on static graph computation, thus are unable to exploit the potential of dynamic event graphs with event-driven computation. The hardware-algorithm co-design landscape of event-driven GNN accelerators is therefore still open for integrated solutions with milliwatt- to watt-level power consumption and microsecond-scale latency.

In this thesis, for the first time, we propose an event-driven GNN accelerator for low-power high-speed edge vision. By leveraging hardware-algorithm co-design, this prototype is successfully implemented inside an edge FPGA platform with microsecond-level running time and acceptable object recognition accuracy. We summarize our contributions as follows:

- We adopt and combine two state-of-the-art event-driven GNNs, AEGNN [30] and HUGNet [5], and according to the target FPGA platform, we transform the core algorithms into a hardware-friendly version with full-integer computation.
- Based on this, we propose a novel event-driven GNN accelerator architecture exploiting two novel techniques: edge-free storage, and layer-parallel computation, to reduce the memory footprint and prediction latency.
- We design and implement the aforementioned event-driven GNN accelerator into the Xilinx KV260 System-On-Module (SOM) platform, which contains an UltraScale+ MPSoC FPGA. We evaluate our design with the real-world dataset NCars [32] in a car recognition task. Our design achieves a prediction accuracy of 95.6% on the validation dataset, and a prediction accuracy of 87.8% on the test dataset. Meanwhile, operating in a total $6.86W$ SoC system power on the Xilinx KV260 edge FPGA platform, the accelerator reaches an average $16\mu s$ prediction latency per event, and runs $9.2\times$ faster than its software counterpart running on an NVIDIA RTX A6000 GPU platform.

This thesis report is organized as follows. Chapter 2 will first introduce basic knowledge and algorithms of event data, event graphs, GNNs and event-driven GNNs. Chapter 3 will introduce the hardware-algorithm co-design approach of this project to achieve a hardware-friendly algorithm. Chapter 4 will demonstrate the concrete accelerator implementation workflow, as well as on-board validation and benchmarking results. Finally, Chapter 5 will conclude the entire project.

2 Preliminaries

In this chapter, we first introduce the preliminaries of event graphs and graph neural networks (GNNs), then the methods enabling GNNs to predict in an event-driven fashion are also presented.

2.1. Event Graphs and Graph Building Algorithms

Event graphs are the foundations of event-driven GNNs. In this section, we will first provide several key concepts of event graphs, then the classifications of event graphs will also be introduced.

2.1.1. Key Concepts of Event Graphs

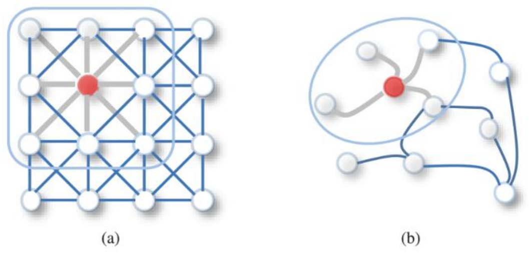


Figure 2.1: (a) Euclidean, structural data, *e.g.* conventional image data. (b) Non-Euclidean graph data. The dots represent nodes, and the blue lines between them represent edges in the graph. Adopted from [35].

A graph, $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$, is a data structure where objects are abstracted as nodes (vertices) $\mathcal{V}$ and their relationship are captured as edges $\mathcal{E}$. Here, the relationship can be the distance, *i.e.* if two nodes' spatiotemporal distance is within a certain threshold, then they are connected by an edge. The nodes can be assigned with additional information, called *features* or *feature vector* made of various numbers. Unlike conventional 2-dimension image data, where one pixel owns fixed 8 pixels around it in a square region, a node in a graph can have an arbitrary number of neighbors, causing difficulty to define its dimension. Therefore, the graph data is *non-structural* or *non-Euclidean*, contrary to the structural image-like data, as shown in Fig. 2.1.

Event stream data is a series of events generated from the activated pixels of event-based cameras, in which an event e is represented as a combination of the pixel position (x, y), the microsecond-scale timestamp t, and the binary polarity $p = \{0, 1\}$ to indicate decreasing or increasing light intensity respectively, as shown in Eq. 2.1.

$$e = (x, y, t, p) \tag{2.1}$$

Considering each event as a node in a graph, then each node carries two types of features: the spatiotemporal position/location, *i.e.* (x, y, t), which is considered when building the graph edges as explained hereafter, and the polarity feature, *i.e.* p, as the additional node information.

For one node, other nodes that connect direct edges with it are named *neighbors*, and these neighbors and edges form a sub-graph called the *neighborhood* of the node. There are two approaches to define the range of the neighborhood:

1. A uniform spatiotemporal distance $r = \sqrt{x^2 + y^2 + (\beta t)^2}$ can be defined, where the timestamp t is first scaled with preset a factor β to fit the range of the pixel positions (x, y), as adopted in [30].
2. Otherwise, the spatial and temporal range of the neighborhood can be chosen separately, usually several pixels in spatial radius r_s and tens to hundreds of milliseconds in temporal radius r_t, as in [5].

Equipped with the edges and node features, the event stream data now can be fully transformed into a graph without any information loss, thus this type of graph is known as an *event graph*.

2.1.2. Classifications of Event Graphs

It will be demonstrated by the following contents that the results of GNNs are sensitive to the structure of the graphs, therefore it is necessary to first elaborate on two different classifications of event graphs in this sub-section. We categorize event graphs according to two aspects: static/dynamic, and undirected/directed.

Static and Dynamic Event Graphs

Every event graph is compiled from an event stream, however, the time of processing the event graph is different, and we separate event graphs as static or dynamic.

Conventional GNNs usually process static graphs, which requires pre-processing the dataset to build the static graphs: the series of all event data is scanned, then the edges are established according to each event's spatiotemporal locations, and finally the generated event graph can be processed by the GNN (Fig. 2.2(a)). During the entire process, the event graph is only built once and has no change after the construction, corresponding to a ”static” event graph.

Contrary to the static method, the event graph can be both built and processed by GNNs simultaneously, as shown in Fig. 2.2(b). Whenever a new event is derived from the event stream, it will first find its neighbors and connect to the current event graph. Afterwards, instead of waiting for all events to build the graph, the event-GNN will immediately process this updated event graph and then generate a prediction. Since the event graph keeps evolving with the newly added events, this type is called the ”dynamic” event graph. With more events processed, the dynamic event graph is converging to its static counterpart, finally ending up with the same graph.

Undirected and Directed Event Graphs

Event graphs can also be categorized according to their connectivity (Fig. 2.3). Graphs default to be undirected, which means that their edges have no direction limitation. On the contrary,

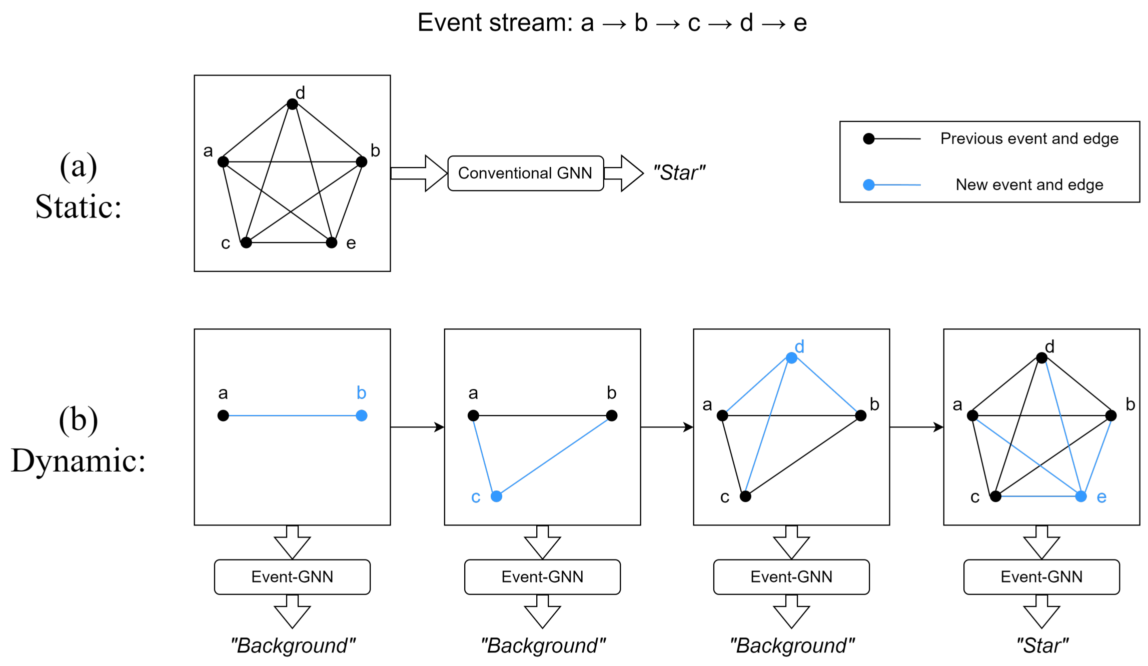


Figure 2.2: Static and dynamic event graphs with respect to the same event stream. (a). The whole event stream is first transformed into a static event graph, then uses GNN to provide a prediction result. (b). Whenever a new event is generated, the dynamic event graph will update, and be processed by an event-GNN immediately to obtain an updated prediction result.

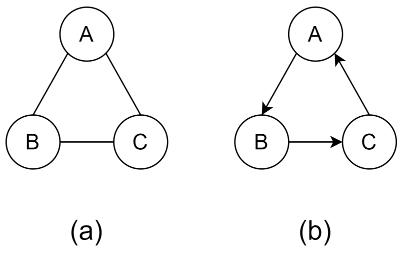


Figure 2.3: Examples of undirected (a) and directed (b) graphs. In (a), one can traverse all nodes in any order, but in (b), only counter-clockwise traversing is permitted due to the directed edges (the direction is denoted as an arrow).

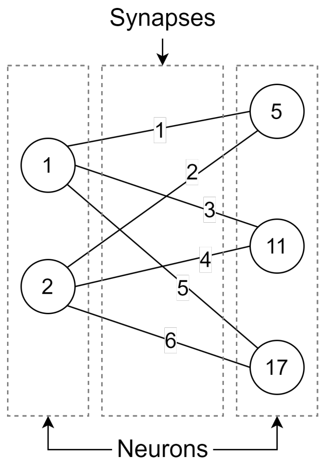
directed graphs contain edges in a certain direction, denoted by an arrow, in which operations (such as traversing) can only be processed in one direction. This extra limitation actually provides advantages to event-GNNs that leverage direct event graphs, which will be introduced hereafter in Section 2.4.3.

2.2. Neural Network Fundamentals

Graph neural networks (GNNs) are developed based on standard neural networks and include ideas from convolutional neural networks (CNNs). Therefore, before moving to GNNs, we first introduce several key concepts of neural networks and CNNs.

2.2.1. Neural Networks

Fully Connected Layer



(a)

$$\begin{bmatrix}1 & 2\\ 3 & 4\\ 5 & 6\end{bmatrix} \times \begin{bmatrix}1\\ 2\end{bmatrix} = \begin{bmatrix}5\\ 11\\ 17\end{bmatrix}$$

$$\Theta \quad \times \quad \mathbf{x} \quad = \quad \mathbf{y}$$

(b)

Figure 2.4: (a) The fully connected (FC) layer abstracted from the neuron system. (b) Equivalent mathematical representation (matrix multiplication).

The cores of the mammalian nervous system are neurons and synapses, whose behavior can be simplified as follows: a neuron sends out an electrical signal of a certain strength, which is weighted by the synapse and transmitted to another neuron [28]. Instead of being point-to-point, this connection forms a network: an upstream neuron can send a signal to multiple downstream neurons, while the downstream neurons sum up weighted signals from different neurons. This network can be further abstracted into the most basic structure of a neural network, called the fully connected (FC) layer, as shown in Fig. 2.4(a). The circles represent neurons, where the numbers are their signal values, and the connecting lines between the circles represent synapses, where the numbers represent the weighted values of the synapses. Furthermore, as the values of all neurons located in the same layer can be represented as a vector, and the weights of all synapses as a matrix, the behavior of the above neural system can be abstracted as a linear transformation, *i.e.* a matrix multiplication, as shown in Fig. 2.4(b) and Eq. 2.2.

$$\mathbf{y} = \Theta \times \mathbf{x} \tag{2.2}$$

Here, the values of neurons together are called features or feature vectors, each of whose elements is called a feature; thus, the upstream and downstream neurons of the network become the input and output feature vectors, represented by $\mathbf{x}$ and $\mathbf{y}$ respectively, and the weights of the synapses are transformed to the weight matrix Θ.

Activation Layer

The goal of establishing a neural network is to generate predictions according to the input features, *e.g.* giving a picture of a handwritten number and predicting the number, which can be regarded as a function-fitting problem. However, a single FC layer is only able to fit a linear function due to its linear transformation nature. To fit more complex functions, a non-linear transformation is attached after the FC layer, which is called the activation layer. Various activation functions can be applied to the activation layer, *e.g.* the exponential linear unit (ELU) and the rectified linear unit (ReLU), which follow Eq. 2.3 and Eq. 2.4 respectively ($a \geq 0$ is a preset parameter), and are illustrated in Fig. 2.5:

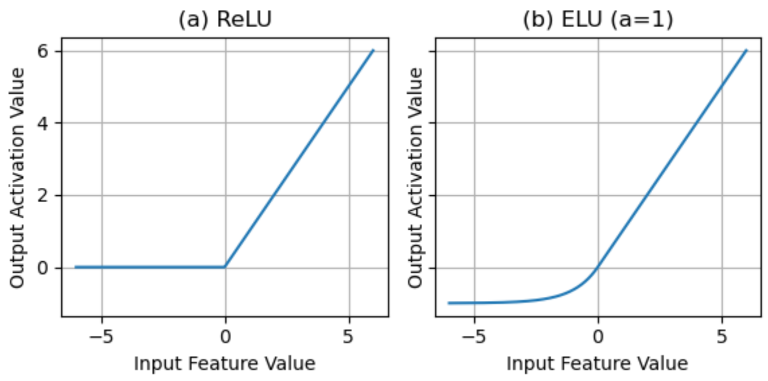


Figure 2.5: The (a) ReLU and (b) ELU (a=1) activation functions.

$$ELU(x) = \begin{cases} x & \text{if } x > 0, \\ a(e^x - 1) & \text{otherwise} \end{cases} \tag{2.3}$$

$$ReLU(x) = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{otherwise} \end{cases} = \max(0, x) \tag{2.4}$$

Batch Normalization

However, directly implementing an activation layer after an FC layer may impair the information extraction ability of the FC layer. For example, if the output feature vector $\mathbf{y}$ in Eq. 2.2 contains a majority of negative components, sending it to a ReLU activation layer would result in large parts of the output becoming 0, thereby losing information.

This problem can be alleviated by inserting a Batch Normalization layer between the FC layer and the activation layer [13]. Batch Normalization, often called BatchNorm or BN, is a layer responsible for correcting an output data distribution into a normal distribution, which is equivalent to a scaling-and-shifting operation for each feature in the output of the layer. BN is described in Eq. 2.5 as:

$$
\begin{aligned}
BN(\mathbf{x}) &= \gamma \circ \frac{\mathbf{x} - \mu(\mathbf{x})}{\sigma(\mathbf{x})} + \beta \\
&= \frac{\gamma}{\sigma(\mathbf{x})} \circ \mathbf{x} + (\beta - \frac{\gamma}{\sigma(\mathbf{x})} \cdot \mu(\mathbf{x})) \\
&= W_{BN} \circ \mathbf{x} + B_{BN},
\end{aligned}
\tag{2.5}
$$

where

$$
W_{BN} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_{C_{out}} \end{bmatrix} = \frac{\gamma}{\sigma(\mathbf{x})},\ B_{BN} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{C_{out}} \end{bmatrix} = \beta - \frac{\gamma}{\sigma(\mathbf{x})} \cdot \mu(\mathbf{x}), \tag{2.6}
$$

therefore

$$
W_{BN} \circ \mathbf{x} + B_{BN} = \begin{bmatrix} w_1 x_1 + b_1 \\ w_2 x_2 + b_2 \\ \vdots \\ w_{C_{out}} x_{C_{out}} + b_{C_{out}}. \end{bmatrix} \tag{2.7}
$$

Here, $\mathbf{x}$ represents the output feature vector of a layer, $\mu(\mathbf{x})$ and $\sigma(\mathbf{x})$ represent the mean value and the standard deviation of $\mathbf{x}$ respectively; γ and β are parameters, and $\circ$ represents element-wise multiplication. In most cases, elements in γ are greater than 0, thus w_i in W_{BN} is also positive.

Training and Inference

However, to predict correctly, parameters in neural networks need to be optimized. The process leveraging a certain algorithm to automatically tune the parameters is called the training step, where these parameters can ”learn” to extract hidden information from the input. After training, the neural network can actually perform the prediction process, which is also known as the inference step. Typically, hardware neural network accelerators are designed to accelerate the inference step. Therefore, we will mostly focus on this step hereafter.

2.2.2. Convolutional Neural Networks (CNNs)

To more efficiently predict the image data, convolutional neural networks (CNNs) have rapidly become the most prevalent algorithms [27, 1]. The fundamental structure is shown in Fig. 2.6.

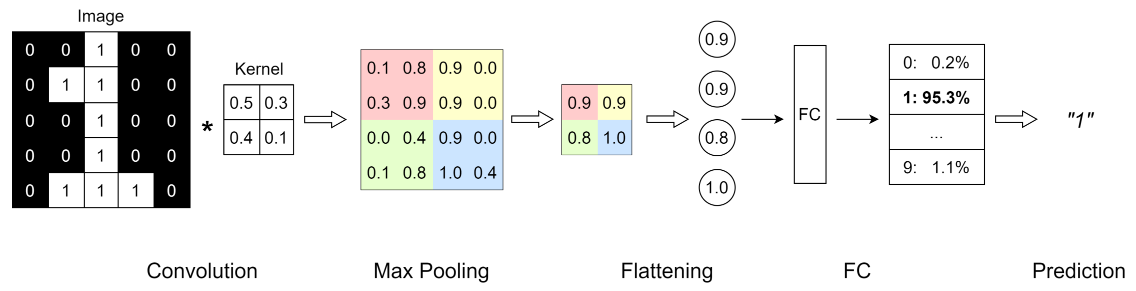


Figure 2.6: An example number-recognition CNN with one convolution layer, one 2×2 max pooling layer, one flattening layer, and one FC layer (the activation layer is not depicted). The image data is first convoluted by a kernel matrix. Then, the output image is sent to a 2×2 max pooling layer to be down-sampled into a smaller image (relative regions are drawn in the same colors). After reshaping to a vector by the flattening layer, the data is processed by an FC layer. The prediction is finally generated by selecting the result with the maximum confidence level.

CNNs introduce two novel layers, the convolution layer and the pooling layer. A convolution layer can extract information from the image data through a convolution operation based on a learnable parameter matrix, called a convolutional kernel. An activation layer typically follows a convolution layer.

Usually after several convolution and activation layers, a pooling layer is added to down-sample the image size. A pooling layer uses a window sliding across the image, and performs various operations to the data inside the window to condense the image, *e.g.* averaging all values (*average pooling*) or selecting the maximum values (*max pooling*), the latter being illustrated in Fig. 2.6

Finally, the down-sampled image is flattened into a vector, then an FC layer is attached to transform it into another vector, where each element represents a possible result with a corresponding confidence level [1] , and the maximal one is selected as the prediction. Therefore, this final FC layer is also known as the prediction head.

2.3. Graph Neural Networks (GNNs)

Tightly coupled with the graph, GNNs gather information from nodes and edges to generate a new representation of the graph. For GNNs designed for object recognition tasks, the aim is to learn a mapping relation, $f(\mathcal{G}) = y$, where $\mathcal{G}$ represents the event graph, and y is the GNN prediction results, *i.e.* the inferred class of the object.

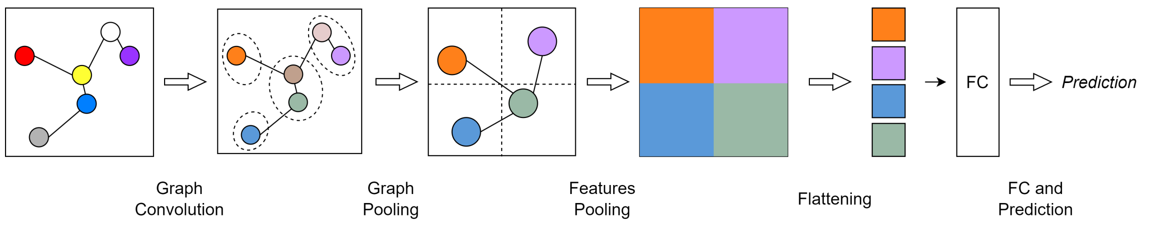


Figure 2.7: An illustration of the GNN network structure. The colors of nodes in the input graph represent their features. After the graph convolution, the features are updated (color changed). Then, the nodes are sent to the graph pooling layer, where certain nodes are selected and edges are re-arranged, thus the graph structure is further simplified. Next, the pooled graph is rasterized by the features pooling layer, which transforms the graph into several grids assigned with node features. Finally, after flattening, the features of grids are processed by the FC layer to derive the prediction results.

Inspired by the mature CNN algorithms introduced in Section 2.2.2, a typical GNN is composed of various similar functional layers, including graph convolution layers, graph pooling layers, features pooling / graph rasterization layers, and standard FC layers for the final prediction head, as shown in Fig. 2.7.

2.3.1. Graph Convolution

The general description of various graph convolution types, typically expressed as the *message passing algorithm*, consists of three major steps: message generation, message aggregation, and feature update, which can be described using one general equation in Eq. 2.8 [4]:

$$\mathbf{x}_i' = \gamma_{\mathbf{\Theta}} \left(\mathbf{x}_i, \bigoplus_{j \in \mathcal{N}(i)} \phi_{\mathbf{\Theta}} \left(\mathbf{x}_i, \mathbf{x}_j, \mathbf{e}_{j,i} \right) \right), \tag{2.8}$$

[1]Typically, a softmax layer is also needed to extract this confidence level, but we omit the details here.

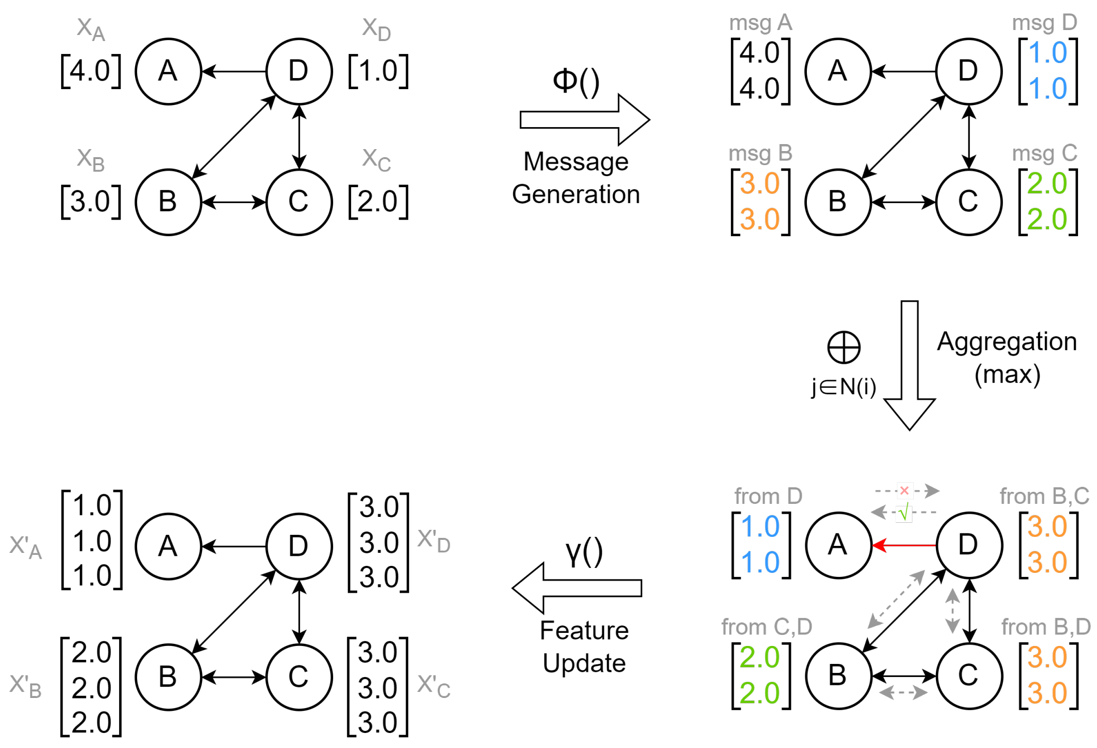


Figure 2.8: An example of the message passing algorithm. The message passing algorithm of the graph convolution consists of 3 steps: message generation, aggregation, and feature update. In the graph, each node A, B, C, D has a feature vector $\mathbf{x}_A, \mathbf{x}_B, \mathbf{x}_C, \mathbf{x}_D$. First, every node generates its own message ($msg\ A/B/C/D$) by the linear transformation $\phi()$ (*Message Generation*) (take a replication operation as an example). Next, messages are exchanged through the edges, and nodes aggregate messages they receive (*Aggregation*). Note that the message from A cannot be aggregated by D due to the directed edge. Finally, the aggregated features are transformed by $\gamma()$, thus the updated feature vectors $\mathbf{x}'_i$ are generated (*Feature Update*) (take a replication operation as an example).

where

$$\mathbf{x}'_i = [x'_{i,1}, x'_{i,2}, ..., x'_{i,C_{out}}]^T; \mathbf{x}_n = [x_{n,1}, x_{n,2}, ..., x_{n,C_{in}}]^T, n = \{i, j\} \in \mathcal{V} \tag{2.9}$$

Here, $\mathbf{x}_i$ represents one node's feature vector with C_{in} features in the current convolution layer, while $\mathbf{x}'_i$ is the convolution output node feature vector with C_{out} dimensions, passing to the next layer, where C_{in} and C_{out} are called the input and output *channels* in a graph convolution layer. Subscript j belongs to node i's neighbors denoted by the neighborhood function $\mathcal{N}(i)$, thus $\mathbf{x}_j$ represents the feature vector of node i's j^{th} neighbor. $\mathbf{e}_{j,i}$ is the edge feature (or attribute) vector between node i and its j^{th} neighbor. The $\bigoplus$ denotes a differentiable, permutation-invariant function (*e.g.* , summation, average, maximum, etc.,) that is insensitive to the node processing order. Finally, γ_Θ and ϕ_Θ denote differentiable functions such as multilayer perceptrons (MLPs) with learnable parameters Θ. A trivial pass-through layer is also a legal choice for γ_Θ and ϕ_Θ.

The three major steps of the general graph convolution are illustrated in Fig. 2.8. These steps must be applied to all nodes in a graph. For one node (node i), Eq. 2.8 works as follows:

1. Message generation: the features of all neighbors of node i, including both node features and edge features, are obtained and transformed by ϕ_Θ.
2. Aggregation: each neighbor of node i generates one message, and all messages are aggregated by a chosen $\bigoplus$ function. For directed graphs, messages are only allowed to be passed in a certain direction restricted by the direction of edges.
3. Update (optional): the aggregated message is transformed by γ_Θ to generate the new feature vector of node i.

We present three specific algorithms of graph convolution hereafter:

Graph Convolution Network

$$\mathbf{x}'_i = \mathbf{\Theta} \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{e_{j,i}}{\sqrt{\hat{d}_j \hat{d}_i}} \mathbf{x}_j \tag{2.10}$$

As one of the most prevalent types of GNNs, the Graph Convolution Network (GCN) [16] illustrates the fundamental core of graph convolution. Its graph convolution algorithm, GCNConv, is shown in Eq. 2.10, where $\hat{d}_i = 1 + \sum_{j \in \mathcal{N}(i)} e_{j,i}$ denotes the node degree, *i.e.* the number of neighbors of a node including itself, weighted by a scalar edge feature $e_{j,i}$. Together with $e_{j,i}$, the generated messages are then aggregated by the summation function. Finally, a linear transformation weight matrix $\mathbf{\Theta}$ is multiplied as the update step.

B-Spline Curve Convolution Network

$$\mathbf{x}'_i = \frac{1}{|\mathcal{N}(i)|} \sum_{j \in \mathcal{N}(i)} \mathbf{x}_j \cdot h_{\mathbf{\Theta}}(\mathbf{e}_{i,j}), \tag{2.11}$$

The B-Spline Curve Convolution Network [8] introduces a new graph convolution algorithm, SplineConv, which extends the scalar edge feature $e_{j,i}$ in GCNConv into a vector $\mathbf{e}_{i,j}$, which is then processed by a specific function used in the B-Spline curve (see Eq. 2.11, where $h_{\mathbf{\Theta}}()$ denotes a specific function defined over the weighted B-Spline basis). Although promising a faster convergence in the training step of neural networks, this type of convolution has a heavy computational burden in the inference phase due to the high computational complexity of B-Spline curve functions.

PointNet

$$\mathbf{x}'_i = \gamma_{\mathbf{\Theta}} \left(\max_{j \in \mathcal{N}(i) \cup \{i\}} \phi_{\mathbf{\Theta}}(\mathbf{x}_j, \mathbf{p}_j - \mathbf{p}_i) \right), \ \mathbf{p}_i = (x_i, y_i) \tag{2.12}$$

In PointNet [26], Qi *et al.* proposed a new graph convolution type described in Eq. 2.12, which we named PointNetConv. PointNetConv leverages the relative Cartesian coordinates of nodes, $\mathbf{p}_j - \mathbf{p}_i$, together with feature vectors of nodes, to generate messages by $\phi_{\mathbf{\Theta}}$, with learnable parameters $\mathbf{\Theta}$. Meanwhile, the complexity of this algorithm can be controlled by the choices of $\gamma_{\mathbf{\Theta}}$ and $\phi_{\mathbf{\Theta}}$. For example, one simplified yet feasible version consists in defining $\phi_{\mathbf{\Theta}}$ as one FC layer, with $\gamma_{\mathbf{\Theta}}$ being a trivial passthrough layer, as shown in Eq. 2.13:

$$\mathbf{x}'_i = \max_{j \in \mathcal{N}(i)} \left(\mathbf{\Theta} \cdot (\mathbf{x}_j \, , \, |\mathbf{p}_j - \mathbf{p}_i|) \right) \tag{2.13}$$

2.3.2. Cluster-based Graph Pooling

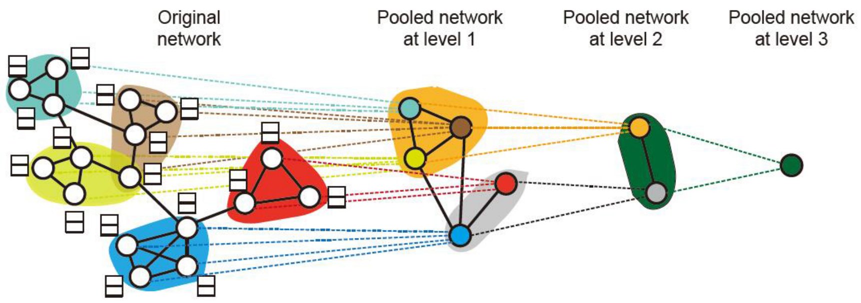

Original network
Pooled network at level 1
Pooled network at level 2
Pooled network at level 3

Figure 2.9: An illustration for hierarchical cluster-based graph pooling layer. Nodes are divided into different clusters (different colors). A cluster extracts node features (small white rectangles) inside to form a new node after a graph pooling layer, with new edges reconnected between clusters. Modified from [38].

Graph pooling layers transform the input graph $\mathcal{G}$ into a more coarse version $\mathcal{G}_C$, which facilitates GNNs to learn hierarchical representations. Proposed by [38], the cluster-based graph pooling algorithm leverages groups of nodes, named *clusters*, to coarsen the graph. Illustrated in Fig. 2.9, clusters gather the features of inner nodes, and process them with a certain pooling function (*e.g.* average or max pooling function), to derive new representations for the clusters. Next, the clusters are further abstracted as new nodes, with new representations as their features, while edges are reconnected between these new cluster nodes, thus finally forming a coarsened graph $\mathcal{G}_C$.

2.3.3. Grid-based Node Features Pooling

The graphs provide an uncertain number of nodes and their features. However, the final FC layer requires fixed dimensions for input data. To solve this mismatch, one straightforward idea is to rasterize the graph according to the physical position (x, y) of each node, with a certain type of pooling operation (*e.g.* max pooling) to extract general information.

The algorithm demonstrated in Fig. 2.10 works as follows. First, the whole field of view of the event-based camera is sliced into several spatial grids, *i.e.* small rectangular areas that are several pixels in length and width. Next, if a node's position (x_i, y_i) belongs to a grid's pixel ranges, the node is put into that grid. Finally, all nodes are traversed, then the features of nodes within each grid are collected for a pooling operation.

This feature pooling layer discards the timestamps of nodes and removes the edges between them, transforming event graphs into Euclidean, image-like data to be processed by the FC

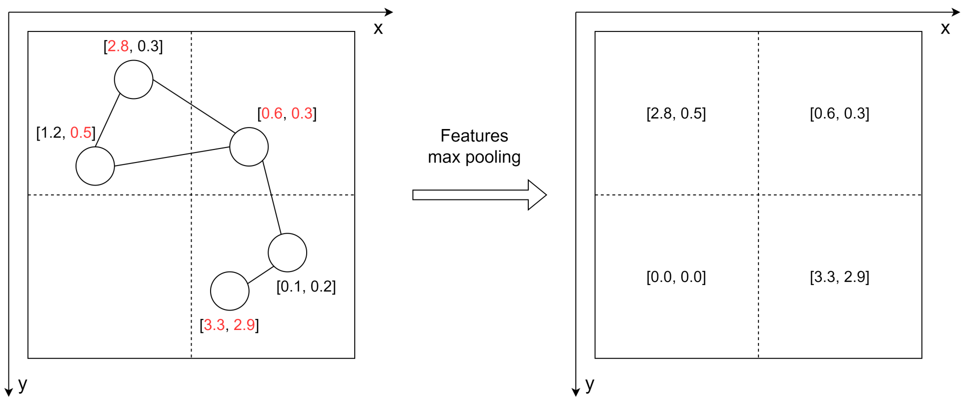


Figure 2.10: Grid-based node features max pooling layer in 4 grids. Nodes belonging to a grid are collected together and the pooling layer only takes the maximal feature values as the output, discarding nodes and edges of the graph.

layer. The features pooling layer only functions as a bridge between GNN backbone layers and the final FC prediction head.

2.4. Asynchronous Event-driven GNNs

2.4.1. Event-driven GNNs with Dynamic Event Graphs

As mentioned in the event graph section (Section 2.1), current GNNs are specialized for processing static graphs. However, this paradigm only works for situations where a dataset already exists, which is not the case for real-time edge vision systems in the real world, which have to directly process the data generated by an event-based camera. While a dataset is indispensable for training the neural network, the computing device will face a totally different environment after its deployment. Indeed, it no longer processes fixed-length event streams from a dataset, but has to receive continuously generated new events from the event-based camera. Therefore, it is not possible to wait until all events are generated to build the graph, as the conventional GNNs using static graphs do.

To solve this problem, a trade-off solution still using static graphs is to set a processing time window for the vision systems. In that time window, every generated new event is simply buffered. Only at the end of the window, all buffered events are leveraged to build a static event graph. Then, the graph is processed by the event-GNN to deliver a prediction for this time window. Finally, the graph and the event buffer are cleaned up, waiting to accommodate new events for the next time window. However, this solution proposed in [3] has two problems. First, it is fundamentally an accumulation method that, although avoiding transforming the event data into dense image frames, still sacrifices the high temporal resolution of event-based cameras. Second, since the static graphs cannot evolve, they have to be destroyed and rebuilt after each time window. However, in real-world scenarios, the graphs from two continuous time windows might be similar. This destroying-and-rebuilding scheme thus leads to redundant computation, causing unnecessary energy overhead.

The characteristics of dynamic event graphs are much preferable to solve this difficulty, as they avoid losing previous information, and keep evolving when new events are generated. An event-GNN is designed to keep track of new events and to update the prediction. This solves the issue of temporal resolution loss. However, it leaves unsolved the fact that, each time the graph is updated, the event-GNN has to process the new event together with all previous events, which still entails redundant computation.

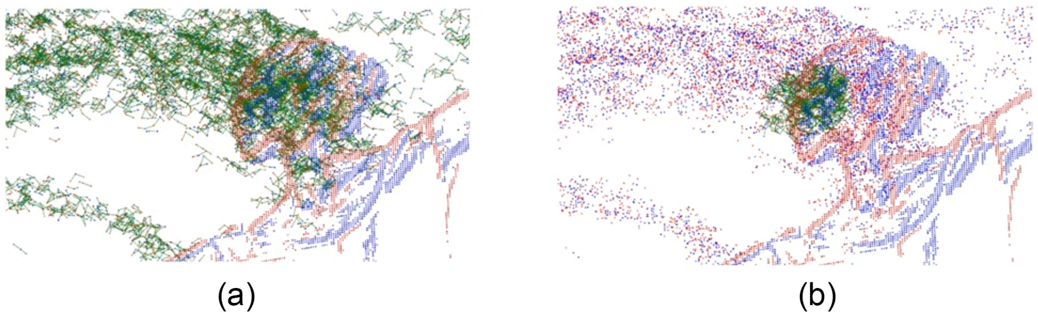


Figure 2.11: (a). Normal event-GNNs process the entire dynamic event graph. The processed events and edges are highlighted with green color. (b). Event-driven GNNs only process the new event and its limited locality. The range of green region in (b) is significantly smaller than in (a). Adopted from [31].

However, in recent studies, AEGNN [30] and HUGNet [5] have proposed novel schemes to eliminate this redundant computation. They show that the event-GNN processing range can be decreased from the entire dynamic event graph to only small neighborhood subgraphs around the newly generated event, as shown in Fig. 2.11. This local computation scheme for event-GNNs is called *event-driven GNN*, which achieves $12\times$ lower computational complexity [30] than [19], providing the possibility to establish low-cost real-time edge vision systems. Therefore, in the following sections, we will elaborate on these two event-driven GNN designs, AEGNN and HUGNet.

2.4.2. A K-hop Locality Event-driven GNN: AEGNN

In graph theory, the term ”hop” represents a unit distance moving from one node to one of its direct neighbors. Therefore, a 1-hop subgraph contains a node, all of its neighbors, and their common edges. Similarly, a K-hop subgraph consists of all elements within one node’s first-to-Kth-order neighborhood.

The information that can be processed by a K-layer GNN is actually only inside a K-hop region. Due to the message passing model, in one graph convolution layer, a node can only update features through the messages aggregated from its 1-hop neighborhood. However, when stepping into the next layer, since the node’s neighbors also aggregate their neighbors’ messages, the node actually obtains the information from a 2-hop radius. The deeper the network, the larger the region of the neighborhood that is processed. Therefore, the receptive field of a GNN, in which the GNN can effectively process the information, is tightly coupled to the number of layers.

This characteristic brings new insights into the design of event-GNNs with dynamic event graphs. The node’s final output features are only decided by the K-hop neighborhood, which also means that, for a dynamic event graph built by only adding one new event at a time, the GNN can only process information within its K-hop subgraph, rather than updating all the nodes when a new event comes. Introduced in AEGNN [30], this method points out the way to construct an asynchronous event-driven GNN depicted in Fig. 2.12(c), which is as follows:

1. When an asynchronous event comes, it triggers the dynamic graph building process to search its neighbors and assign edges, generating its new 1-hop neighborhood, and updating the event graph. (The new event and new edges are depicted in blue in Fig. 2.12.)
2. Dynamic graph updating further triggers the event-driven GNN: In the first layer, it processes the new event and all its 1-hop neighbors. Then in the second layer, the 2-hop locality is affected and updated, and so on.

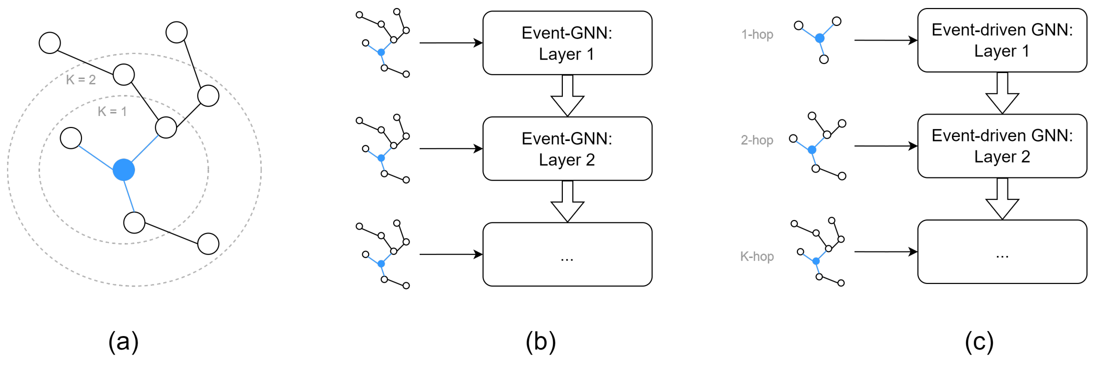


Figure 2.12: (a). An event graph, with the blue dot representing the new event, the blue lines representing the new edges between neighbors and the new event, and $K = 1, 2, ...$ showing its 1- to K-hop subgraphs. (b). For a normal event-GNN with multiple graph convolution layers, each layer will take the entire event graph as input. (c). For an event-driven GNN (*e.g.* AEGNN), the first layer only takes the 1-hop subgraph as input, the second layer only takes the 2-hop subgraph, and so on until the K^{th} layer.

3. When reaching the last K^{th} layer of the GNN, the entire K-hop neighborhood is updated, and the system waits for the next new event.

In this manner, the event graph is able to iteratively add events, and the event-driven GNN can only process a K-hop subgraph of an event graph, with the final output being the same as from normal event-GNNs with static event graphs.

Revealing the locality nature of the GNN algorithms, AEGNN significantly reduces the computation workload, measured in Million Floating-point Operations per event ($MFLOP/ev$), by $12\times$ compared with conventional event-GNN algorithms ($6.1MFLOP/ev$ in [19] to $0.47MFLOP/ev$ in AEGNN), with even higher prediction accuracy (93.1% in [19] and 94.5% in AEGNN). However, this method containing the task of finding K-hop neighbors with $0.47MFLOP/ev$ still represents a significant burden on embedded devices targeting high-speed event-based vision scenarios.

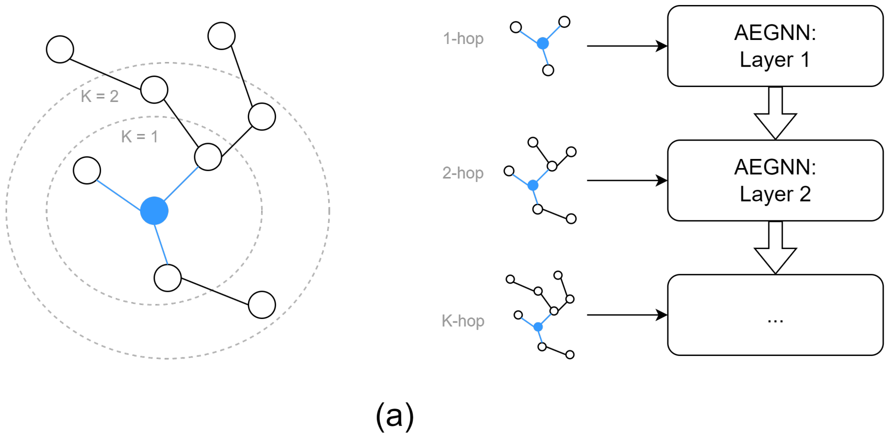
2.4.3. A 1-hop Locality Event-driven GNN: HUGNet

In HUGNet [5], Dalgaty *et al.* re-scrutinized the necessity of updating all nodes' features within the K-hop subgraph. It turns out that, in AEGNN, a new event comes and is updated by its older event neighbors, and at the same time, the older events also change their presentations due to the new event's features. This is due to the fact that edges are undirected, thus the messages are able to freely pass from the past to the present, and vice versa. The update of 1-hop nodes affects their neighbors in the same manner, then the whole K-hop locality is influenced by the new event.

A simple solution is thus proposed: if the messages cannot flow from the present to the past, *i.e.* the edges are restricted in one direction, the older neighbors within the 1-hop subgraph are not needed to update, and the rest of the K-hop locality can stay static. This heuristic solution is also causal in real-world scenarios: the information should only depend on past events, and should not be shaped by future ones.

Therefore, by using the directed graph building technique introduced in Section 2.1.2, HUGNet allows restricting the feature update only within the 1-hop neighborhood. The workflow is shown in Fig. 2.13(b):

1. As in AEGNN, the new event (blue) triggers the graph building algorithm to select its spatiotemporal neighbors among the previous events. However, this time the new event



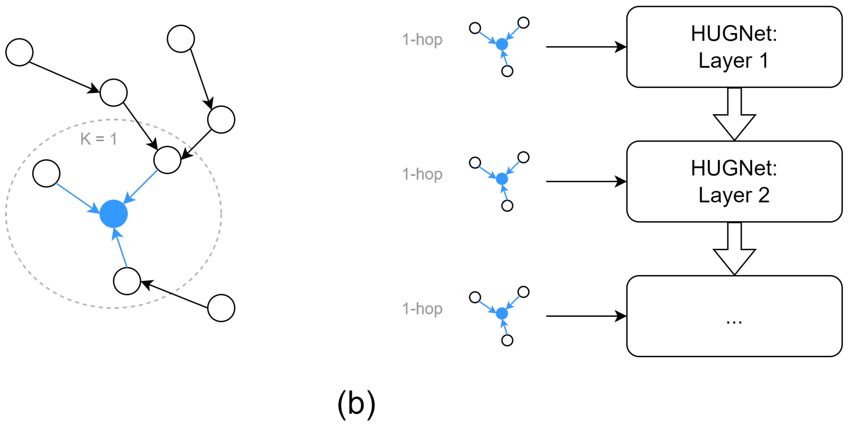


Figure 2.13: (a). K-layer AEGNN processes 1- to K-hop subgraphs of a dynamic event graph (same as in Fig. 2.12(c)) [30]. (b) K-layer HUGNet processes only the 1-hop subgraph of a directed dynamic event graph [5].

connects directed edges with its neighbors, pointing from the neighbors to itself.

2. The updated directed dynamic graph also triggers the GNN, but for each layer of the GNN, only the new event and its 1-hop directed subgraph are processed. Due to the directed edges, only the features of the new event are actually updated by the GNN, leaving the features of neighbors untouched.

It is worth noting that, since the graph is dynamically and iteratively built with directed edges, the older neighbor nodes already carry their final node features, which are unlike the ones in AEGNN waiting to be updated by the new event. Therefore, though it seems that the new node is obtaining information only from its 1-hop neighborhood, the information from the rest of the K-hop region is already aggregated into the older nodes, thus the actual receptive field still holds to K for a K-layer GNN.

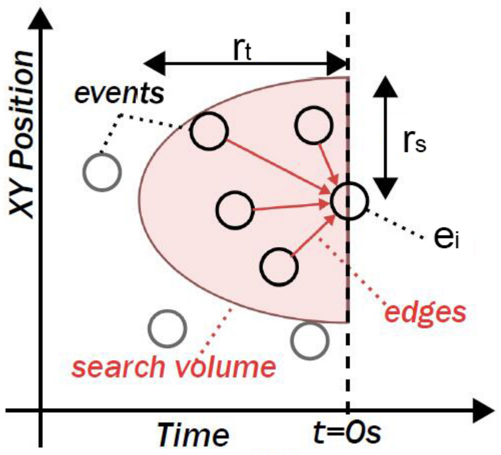


Figure 2.14: Hemi-sphere search range used in HUGNet. Modified from [5].

This novel workflow leads to the following advantages:

1. The K-hop-neighbor search function can be removed, as only 1-hop subgraphs are processed.
2. If the chosen graph convolution type in GNNs contains no edge features (*e.g.* the $e_{j,i}$ in GCN (Eq. 2.10) and the $\mathbf{e}_{j,i}$ in SplineConv (Eq. 2.11)), then edges do not need to be stored. Indeed, as only the 1-hop subgraph is processed, it is safe to discard the edges of the new event after information from its neighbors has been fully processed by the event-driven GNN.
3. Older features do not need to be written back, as they are not changed due to the directed edges, leading to lower computational burden and latency.
4. A hemi-sphere search range can be used, as shown in Fig. 2.14. In other event-driven GNNs using undirected graphs, such as AEGNN [30], the neighbor search range for a new event is a spherical region: after calculating the spatiotemporal distance with other nodes, a new event e_i not only needs to search for potential neighbors within a spatial radius r_s and a temporal radius r_t among previous events, it also has to wait for future events in the other half of the temporal radius r_t, since the future ones may influence the new event as well. In HUGNet, thanks to the directed edges, the future events cannot affect the new one, thus restricting the neighbor search range within a hemi-sphere spatiotemporal region, which avoids buffering and latency overhead.

5. Executing feature pooling layers and FC layers asynchronously. The 1-hop locality update characteristic of HUGNet also facilitates the process of feature pooling layers and FC layers, which can be referred to in Appendix A.1 and Appendix A.2 respectively.

2.5. Edge Vision Systems Application Scenarios

As mentioned in Chapter 1, edge vision systems with event-based cameras are able to process various computer vision tasks, such as optical flow estimation [5, 37], and object recognition [30, 32]. In this section, we will briefly introduce the purposes of these tasks, and the dataset they use in training and benchmarking the GNNs.

2.5.1. Optical Flow Estimation

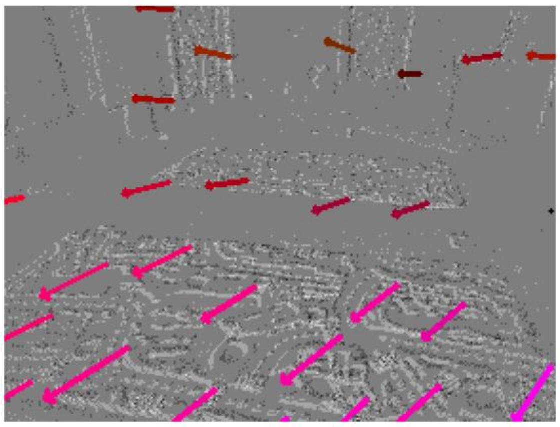


Figure 2.15: Optical flow estimation results based on event data. Arrows represent predicted velocity vectors. Adopted from [9].

Optical flow estimation is a task aiming at estimating the velocity vectors of moving objects in a video stream, as shown in Fig. 2.15. For example, HUGNet [5] and EV-Flow [37] are solutions to process this task. Relevant event-based datasets are MVSEC [40] and Rock Scenes [5].

2.5.2. Object Recognition

Object recognition tasks aim at identifying the category of an object in a video stream, which corresponds to a classification task in the computer vision field. Event-based methods have also been introduced for recognition tasks, such as AEGNN [30] and HATS [32].

There are two prevalent event-based datasets for object recognition tasks, NCaltech101 [25] and NCars [32], with example samples illustrated in Fig. 2.16. NCaltech101 is an event-based version of the famous Caltech101 [6] dataset used for standard frame-based image recognition. NCaltech101 is obtained by displaying samples of Caltech101 on an LCD monitor and recording them with an event-based camera, thus also containing the same 101 object categories as the frame-based Caltech101.

Different from NCaltech101, objects in the NCars dataset [32] only have 2 categories: either ”Car” or ”Background”, which corresponds to a binary classification task for car recognition. However, as opposed to NCaltech101, NCars is not derived from an existing dataset: samples in NCars are recorded by an event-based camera mounted behind the windshield of a car driving in real urban environments [32]. Therefore, though the number of categories is significantly reduced compared with NCaltech101, the complex real-world road conditions still lead to a challenging dataset.

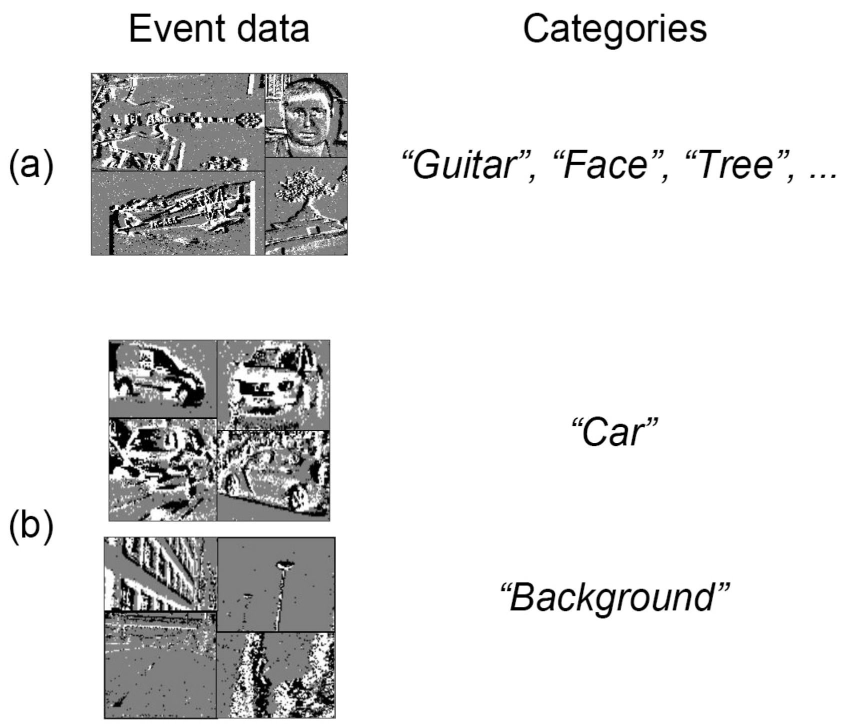


Figure 2.16: Example samples from (a) NCaltech101 and (b) NCars dataset, both obtained by accumulating $100ms$ of events. NCaltech101 (a) has multiple categories, while NCars (b) has only 2 categories. Modified from [32].

2.5.3. Selected Target Task and Dataset

In this thesis, we choose object recognition and NCars as our target task and dataset for the following reasons:

1. Compared with optical flow estimation requiring multiple output results (predicted velocity vectors), the object recognition algorithms are less complex and thus more suitable for a resource-limited edge hardware platform. Meanwhile, as a mature computer vision task, it is still a proper benchmark for testing performance metrics (*e.g.* prediction accuracy, runtime, *etc.*) of our hardware event-driven GNN accelerator.
2. Concerning datasets, NCars is more representative of real-world environments than NCaltech101, which is consistent with the ultimate purpose of applying edge vision systems to real-world scenarios.

3 Hardware-algorithm Co-design

The original event-driven GNN algorithms are hardly compatible with hardware accelerators as they are usually optimized for high-performance computing platforms such as dedicated CPU + GPU environments. These algorithms tend to involve: (i) large amounts of memory usage and frequent data movement; (ii) parallelizable yet coarse-grained computation workflow; (iii) complex and non-linear algebra, such as square-root and exponential operations; (iv) floating-point calculation. Directly mapping algorithms with these characteristics on edge FPGA platforms may thus lead to sub-optimal efficiency and performance. Therefore, a hardware-aware optimization of these algorithms is critical, which is a part of hardware-algorithm co-design.

The purpose of hardware-algorithm co-design is different from conventional software design aiming at finding a novel algorithm with higher performance. Instead, the co-design part aims at developing an alternative model that is mathematically approximately equivalent to the original algorithm, while the operations inside have been simplified toward an efficient implementation in custom hardware.

In this chapter, we will first introduce the baseline software design, together with the details of our co-design experiments. Afterwards, we will elaborate on our hardware-algorithm co-design in three key aspects: graph building algorithms, GNN structure optimization, and full-integer quantized computation.

3.1. Baseline: AEGNN

To prove that our hardware-algorithm co-design implementations are indeed effective alternatives to the original algorithms while introducing no significant prediction accuracy loss, we need to select a proper baseline.

In this chapter, we choose AEGNN (Section 2.4.2) as the baseline design. AEGNN is designed for the object recognition task using the NCars dataset, which corresponds to our target task and dataset in Section 2.5.3. Meanwhile, the prediction accuracy of AEGNN is 94.5%, which is the state-of-the-art result among all event-based object recognition algorithms with NCars [30].

All experiments in this chapter are based on the *validation* dataset, which is a random-selected subset of the NCars dataset (16% of the total), while the rest part of dataset is used for training. The prediction accuracy is also derived by calculating the percentage of correct predicted samples among the entire validation dataset. NCars also provides another independent test dataset, containing around $4\times$ more samples than the validation dataset. However, for convenience, in this chapter we will not evaluate our design results on the test dataset due to its large volume, until the final hardware accelerator benchmarking in the next chapter.

All experiments are deployed on a NVIDIA RTX A6000 GPU hardware platform, and are established, trained, and tested using the PyTorch Geometric library (PyG) [7] in the Python software environment.

3.2. Graph Building Algorithms

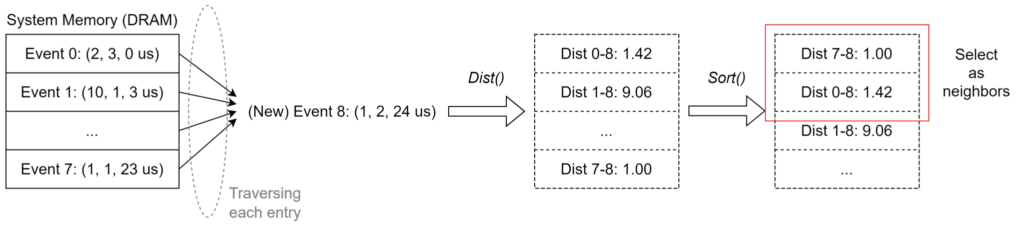


Figure 3.1: In software algorithms, event position information is stored in the system memory (DRAM) in (x, y, t) format. When a new event arrives, the software neighbor selection algorithm requires traversing the position information of all previous events, and computes spatiotemporal distances between the new event and older ones (*Dist()* function), following a sorting operation to finally select the neighbors (*Sort()* function). Therefore, this process is memory-intensive.

The first step of event-driven GNN algorithm design is the graph building part, which aims at searching and selecting spatiotemporal neighbors for each new event. As shown in Fig. 3.1, for software algorithms, selecting neighbors for a new event within a certain spatiotemporal distance can be a memory-intensive workload since it needs to traverse all the previous nodes' positions [1]. However, with proper simplification and limitations on the neighbor searching range, it is possible for this part to be implemented efficiently on custom hardware.

In the graph building co-design part, we first adopted the directed graph and the hemi-sphere search range ideas of HUGNet into the baseline design, then explored two sub-problems: the total number of neighbors to select (*degree limitation* problem), and the range of the neighbor search (*search range* problem). These three aspects will be elaborated upon in the following sections.

3.2.1. Graph Building Improvements

For two major parts of event-driven GNNs, the dynamic graph building algorithms and the GNNs, only the latter is task-specific. Therefore, even though aiming at a different target task from us (Section 2.5.3), the techniques proposed in HUGNet for the former can still facilitate our event-driven GNN design, which contains two core innovations of HUGNet, the directed event graphs, and the hemi-sphere search range, as introduced in Section 2.4.3.

Based on the above observations, we proposed to fuse HUGNet into the baseline design, AEGNN, by replacing the graph building part with the HUGNet fashion, thus we can still maintain the target object recognition task while obtaining the advantages brought by HUGNet. The detailed adoption process is as follows:

1. Replacing the undirected event graphs used in AEGNN to the directed version, as in HUGNet.

[1]In the following contents, unless otherwise indicated, the terms ”position” or ”location” of an event both refer to its spatiotemporal position (x, y, t). ”Spatial position” or ”physical position” both refer to its spatial coordinates (x, y), and ”timestamp” or ”time” both refer to its temporal coordinate t.

2. According to Section 2.4.3, restricting the processing subgraphs from the K-hop neighborhood (AEGNN) into the 1-hop neighborhood (HUGNet).
3. Limiting the neighbor search range from a sphere spatiotemporal region (AEGNN) into a hemi-sphere region (HUGNet).

Table 3.1: Prediction accuracy for the baseline and the improved graph building on NCars dataset

Methods	**Accuracy**
Baseline (AEGNN)	94.5%
AEGNN + HUGNet	94.7%

Table 3.1 demonstrates that the AEGNN+HUGNet scheme achieves 0.2% higher accuracy, which proves that this improved graph building scheme is also suitable for the object recognition tasks.

3.2.2. Degree Limitation

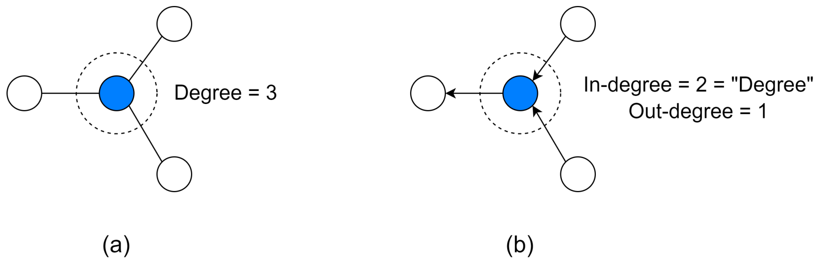


Figure 3.2: (a). In an undirected graph, the blue node has 3 neighbors, thus its degree $D = 3$. (b). In a directed graph, the blue node has 2 neighbors pointing to it, and 1 neighbor pointing from it, thus its in-degree $D_{in} = 2$, and out-degree $D_{out} = 1$. Note that, in the following contents, we still use the term ”degree” referring to the in-degree of nodes in direct graphs.

In graph theory, the degree D of a node is the number of its 1-hop neighbors, also equal to the number of edges that connect with the node. For a directed event graph, we instead define the in-degree D_{in} of a node as the number of edges coming into the node. There is also a counterpart, the out-degree D_{out}, which represents the number of edges departing from a node. However, in directed event graphs, we only care about the message flowing from the past to the future, thus in the following, the term ”degree” in the directed graph context will refer exclusively to the in-degree, unless otherwise specified (Fig. 3.2).

Event-driven GNNs do not impose any limit to the node degree by definition. Indeed, theoretically, more neighbors provide more information to a node. However, concerning the actual hardware resources, a proper upper limit of D_{max} is also needed to bind the computation and memory footprints of the neighbor search process. These characteristics render D_{max} as a use-case-dependent parameter.

Since, as mentioned in Section 2.5.3, we use the NCars dataset in this work, we converted each sample in the dataset into an event graph and counted the degree distribution of each node. The statistics shown in Fig. 3.3 demonstrate that most nodes have degrees smaller than 75, taking around 53.8% of total nodes, and highlight an obvious power law decay trend. Considering both the graph representation ability and the hardware efficiency, we chose three power-of-2 maximal degrees as candidates, *i.e.* $D_{max} = 16, 32, 64$.

We tested these candidates, and the prediction results are listed in Table 3.2. Note that $D_{max} = 32$ is actually the baseline design, AEGNN. It is clear that, $D_{max} = 32$ is not the best choice,

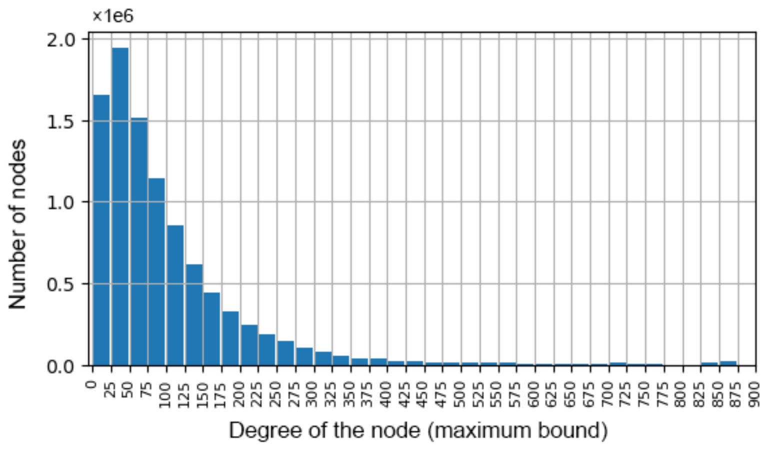


Figure 3.3: Maximal node degree distribution of the NCars [32] dataset

Table 3.2: Prediction accuracy for different maximal degrees on NCars dataset

D_{max}	**Accuracy**
16	95.8%
32 (Baseline)	94.5%
64	96.0%

while $D_{max} = 16, 64$ have similar accuracies. Considering smaller D_{max} leads to smaller memory footprints on edge devices (roughly proportional), we finally select $D_{max} = 16$ as the final choice.

3.2.3. Neighbor Search Range

The neighbor search range defines within what spatiotemporal range a previous event can be regarded as a neighbor to a new event. We scrutinized this problem from two perspectives. First, we checked the spatiotemporal distance definitions and proposed an L^1 distance alternative. Next, we decoupled the spatial and temporal coordinates of the search range to deliver a novel *cylinder* search scheme. Finally, we combined the aforementioned improvements to create a prism neighbor search range for efficient hardware graph building implementations.

L^1 Search Range

The method for calculating distance decides whether a previous node can be regarded as a legitimate neighbor to a new node. The distance is defined by the L^p*-norm* of a difference vector of two position vectors, thus this type of distance is also known as the L^p distance. Choosing a proper L^p distance can simplify the search algorithm without sacrificing performance.

For an n-dimensional vector $\mathbf{x} = [x_1, x_2, ..., x_n]^T$, the L^p-norm, $\|\mathbf{x}\|_p$, is defined as:

$$\|\mathbf{x}\|_p = (|x_1|^p + |x_2|^p + ... + |x_n|^p)^{1/p},\ p \geq 1 \tag{3.1}$$

Especially, for L^1-norm and L^2-norm respectively:

$$\begin{aligned} \|\mathbf{x}\|_1 &= |x_1| + |x_2| + ... + |x_n|, \\ \|\mathbf{x}\|_2 &= \sqrt{(x_1^2 + x_2^2 + ... + x_n^2)} \end{aligned} \tag{3.2}$$

Therefore, for points in the $\mathcal{R}^3$ spatiotemporal space, $\mathbf{p_i} = [x_i, y_i, t_i]^T$, we can use the L^p-norm to define the L^p distance between two points. Taking the L^1 and L^2 distances as examples, which are:

$$\begin{aligned} dist_1(\mathbf{p_1}, \mathbf{p_2}) &= \|\mathbf{p_1} - \mathbf{p_2}\|_1 \\ &= |x_1 - x_2| + |y_1 - y_2| + |t_1 - t_2| \\ &= |dx| + |dy| + |dt|, \end{aligned} \tag{3.3}$$

and

$$\begin{aligned} dist_2(\mathbf{p_1}, \mathbf{p_2}) &= \|\mathbf{p_1} - \mathbf{p_2}\|_2 \\ &= \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (t_1 - t_2)^2} \\ &= \sqrt{dx^2 + dy^2 + dt^2}, \end{aligned} \tag{3.4}$$

the L^2 distance, also known as the Euclidean distance, is the most prevalent distance definition leveraged by software event-GNN algorithms [30, 5, 19] to calculate the spatiotemporal neighborhood range of a node. However, it requires computing a square root function, which is a costly operation in edge hardware implementations. On the other hand, the L^1 distance involves only additions and taking absolute values, both of which are hardware-friendly functions.

Combining the hemi-sphere search scheme we adopted in Section 3.2.1, we illustrate different L^p distance search ranges in Fig. 3.4. While the original L^2 search range in HUGNet is a hemisphere region as shown in Fig. 3.4(a), our L^1 alternative is a semi-octahedron region depicted in Fig. 3.4(b).

Spatiotemporal Decoupling: Cylinder Search Scheme

Common event-driven GNN algorithms [30, 5] usually adopt a uniform L^2 spatiotemporal distance limit, r, to define the range of a node's spatiotemporal neighborhood, *i.e.* every previous event within the distance r can be selected as a neighbor of a new event. However, there is a scale mismatch between the spatial and temporal coordinates: the spatial coordinates of an event, (x, y), correspond to the pixel physical locations in event-based cameras, which are integers within a few hundreds; while the timestamps t are already on the order of $10^6 \mu s$ in an event stream lasting for seconds.

$$dist_2^*(\mathbf{p_1}, \mathbf{p_2}) = \sqrt{dx^2 + dy^2 + (\beta dt)^2} \leq r \tag{3.5}$$

As shown in Eq. 3.5, in HUGNet, this mismatch is alleviated by multiplying a factor β on the timestamps to scale down their ranges until comparable to spatial coordinates. Therefore, the search range is actually transformed into a half-ellipsoid region [5]. [2] However, if we want to leverage this hemi-sphere search scheme while maintaining the microsecond-scale temporal resolution from event-based cameras, βdt will require a high-precision fixed-point, or even floating-point, number representation, which is not an efficient solution for edge hardware implementations.

Instead of searching neighbors with the above hemi-sphere scheme, which forces timestamps to fit the scale of spatial coordinates so that one can use a uniform spatiotemporal distance r

[2]To avoid confusion, we still named it the "hemi-sphere" search scheme as consistent in the original HUGNet.

to define the range, we proposed a novel search scheme, where we separated the distance limit into two independent parts, the spatial distance range r_s and the temporal distance range r_t. Based on the most prevalent L^2 distance, our spatiotemporal decoupled search scheme is expressed as:

$$dist_2(\mathbf{p_1}, \mathbf{p_2})|_s = \sqrt{dx^2 + dy^2} \leq r_s, \tag{3.6}$$

and

$$dist_2(\mathbf{p_1}, \mathbf{p_2})|_t = dt \leq r_t \tag{3.7}$$

In our scheme, an event can only be a neighbor of a new event if its relative spatiotemporal coordinates satisfy both the spatial condition (Eq. 3.6) and the temporal condition (Eq. 3.7). In this way, the spatial metric and the temporal metric can be decoupled, and each of them can be processed in different units, *e.g.* r_s can be a few pixels while r_t can cover a temporal range of $10^5 \mu s$.

The visualization of this search scheme is shown in Fig. 3.4(c). As opposed to the hemi-sphere search range of HUGNet in Fig. 3.4(a), this scheme searches neighbors in a cylinder range, thus we named our spatiotemporally decoupled scheme the *cylinder* search scheme. In Fig. 3.4(c), the base of the cylinder represents the spatial search range r_s, while the height of the cylinder represents the temporal search range r_t, both of which can be selected according to the actual use case.

Combination of the Two Approaches: Prism Search Range

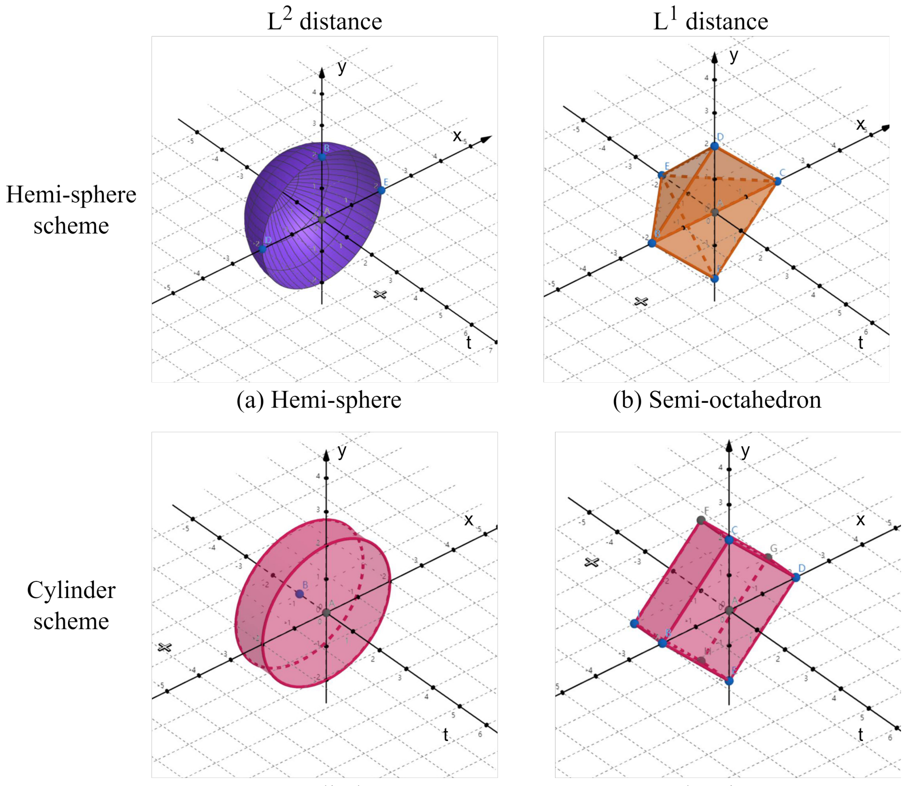


Figure 3.4: Illustrations of four spatiotemporal search ranges, by combining two different search schemes, the hemi-sphere and the cylinder schemes, with two different L^p distance.

It is worth noticing that, our cylinder search scheme can be extended to any L^p distance. Therefore, we combined our L^1 search range with the cylinder search scheme, which is:

$$|dx| + |dy| \leq r_s \text{ and } dt \leq r_t \tag{3.8}$$

Table 3.3: Prediction accuracy for different neighbor search ranges on NCars dataset. For the hemi-sphere scheme, $\beta = 5 \times 10^{-6}$, $r = 3$; for the cylinder scheme, $r_s = 3$, $r_t = 65,535\mu s$.

Search Ranges	Search Scheme	L^p	Accuracy
Hemi-sphere	Hemi-sphere	L^2	94.7%
Semi-octahedron		L^1	95.6%
Cylinder	Cylinder	L^2	95.1%
Prism		L^1	95.6%

We also visualize this search range in Fig. 3.4(d), which is a prism region. Table 3.3 provides the prediction accuracies of all four search ranges in Fig. 3.4 with parameters derived from [30]. It is clear that the proposed prism search range promises easier computing operations without compromising performance, which is a key toward deployment on custom hardware.

3.3. GNN Architecture Search and Optimization

After building the event graph according to the algorithms introduced in the previous section, a GNN performs the actual computer vision task, *i.e.* the object recognition task in this thesis (see Section 2.5.3). In this section, we will first compare different activation functions and different graph convolution types to minimize hardware requirements. Then, we will introduce the network architecture of our GNN, which is inspired from the AEGNN [30] with simplifications.

3.3.1. Activation Functions

In AEGNN, the authors leverage the exponential linear unit (ELU) to work as the activation function. To avoid the hardware-intensive exponential computation, we change it into a rectified linear unit (ReLU). Both functions are introduced in Section 2.2.1, following Eq. 2.3 and Eq. 2.4 respectively.

Table 3.4: Prediction accuracy for different activation functions on NCars dataset

Activation Functions	Accuracy
ELU (Baseline)	94.5%
ReLU	95.2%

The resulting classification accuracies on the NCars dataset are provided in Table 3.4. Statistics prove that, the usage of ReLU does not affect the prediction accuracy, while providing a simpler linear activation function compared with the ELU version, which is more suitable for a hardware implementation.

3.3.2. Alternative Graph Convolution

In Section 2.3.1, we have introduced 3 types of typical graph convolution algorithms: GCNConv, SplineConv, and PointNetConv. In AEGNN [30], the adopted convolution algorithm is the SplineConv.

However, we argue that SplineConv may not be the optimal choice as shown in Table 3.5. Here, the *#Parameters* column represents the total number of parameters needed. The fewer

Table 3.5: Performance metrics of 3 types of graph convolution algorithms on NCars dataset

Convolution Types	Accuracy	#Parameters	Runtime
Baseline (SplineConv)	94.5%	33.4k	16s
GCNConv	92.6%	7.8k	11s
PointNetConv (Eq. 2.13)	95.7%	12.3k	11s

parameters in an algorithm, the less memory space it requires. For the *Runtime* column, we counted the time spent by the 3 algorithms after performing inference on the entire validation dataset, where a faster runtime typically implies a lower computational complexity.

Among all 3 metrics, though accuracy is high, SplineConv has the largest number of parameters and the slowest runtime, due to the computationally intensive B-Spline convolution algorithms. GCNConv has the least parameters, but the accuracy is around 2% lower than others, partially because it does not consider position parameters in its calculation. Finally, PointNetConv achieves the highest accuracy, the same runtime as GCNConv, and a balanced number of parameters. We thus select PointNetConv, as it takes position information into account at a much lower computational overhead than SplineConv (see Eq. 2.13), while further increasing accuracy.

3.3.3. Network Structure and Simplification

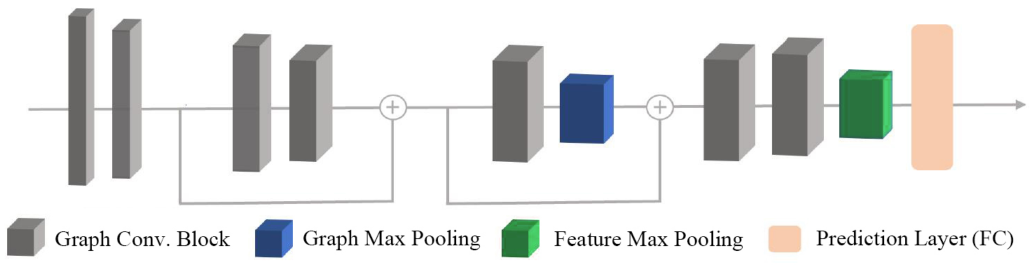


Figure 3.5: The AEGNN network structure. The grey blocks represent graph convolution blocks including a graph convolution layer, a batch normalization layer, and an activation layer. The blue block represents a graph max pooling layers, the green block represents a feature max pooling layer, and the orange block represents the final FC layer. Modified from [30].

Fig. 3.5 shows the original AEGNN network structure for the car recognition task, as proposed in [30]. Here, a grey block represents a sequence of layers containing a graph convolution layer, an activation layer, and a batch normalization layer, named a *graph convolution block*. The $\bigoplus$ symbols indicate residual connections, which add the current node features with previous features from shallower layers. A blue block represents a cluster-based graph max pooling layer, and a green block represents a grid-based feature max pooling layer, both of which are introduced in Section 2.3. Finally, a conventional FC layer outputs the final GNN prediction results for the input event graphs.

We implement 2 simplifications for a more hardware-friendly design:

1. Removing residual connections and the middle graph max pooling layer (*Component Removing*). The residual connection needs extra storage and memory access to fetch the previous features. To keep a low memory footprint, we removed these components. The graph max pooling layer coarsens the input graph, as mentioned in Section 2.3.2, thus involving node deleting and edge re-connection, which would introduce frequent memory accessing and overwriting in hardware. Therefore we also remove it, leaving

only the feature pooling layer.

2. Graph convolution block number and feature channels adjustments (*Block Adjustments*). We eliminated several graph convolution blocks in the middle of the original network, reducing the total number of graph convolution blocks from 7 in AEGNN to 4 in our design. Input and output feature channels of certain graph convolution layers have also been adjusted accordingly.

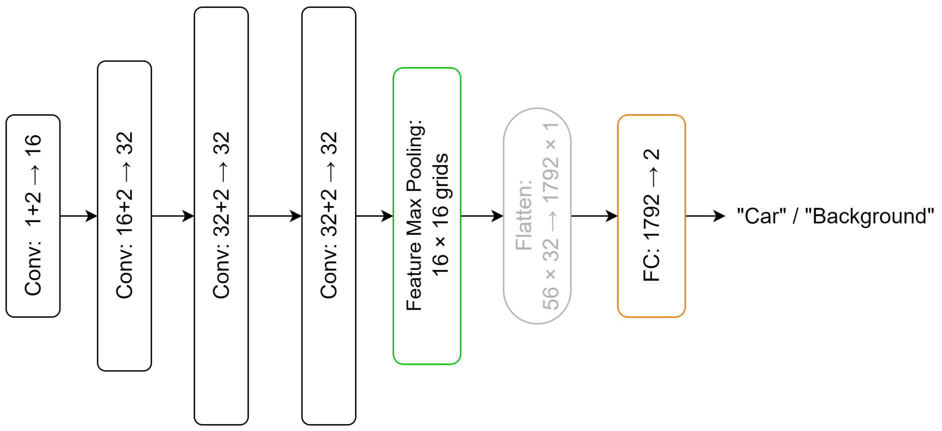


Figure 3.6: Our optimized event-driven GNN inspired from the AEGNN structure, including 4 graph convolution blocks (*Conv* in the figure), a feature max pooling layer, and an FC layer for prediction.

The proposed network structure is depicted in Fig. 3.6. A *Conv* represents a graph convolution block, where the numbers follow the format "$C_{in} \to C_{out}$". Notice that for input feature channels C_{in}, there is a "+2" representing the event's spatial position information (x, y) used in PointNetConv (Eq. 2.13).

Table 3.6: Prediction accuracy for different network structure simplifications on NCars dataset

Simplifications	**Accuracy**
Baseline (AEGNN)	94.5%
Components Removing	94.9%
Block Adjustments	96.0%

Table 3.6 demonstrates the prediction accuracy of these two simplifications. In this table, we found that purely removing the graph max pooling layer and residual connections does not provide notable accuracy changes. However, when combined with a reduction of the number of convolution blocks (*i.e.* shallower network), the final simplified scheme achieves a significant accuracy improvement, which we abbreviate to *optimized network*. Meanwhile, the decreased number of graph convolution blocks also promises less resource usage in custom hardware.

3.4. Quantization

In this section, we will introduce the quantization technique, which is necessary to achieve low-cost full-integer calculation in the edge hardware, thereby reducing the computational and memory requirements

The default precision in neural networks (weights, bias, features, *etc.*), for both training and inference, is the 32-bit floating-point (FP32) format. Nevertheless, studies [17, 11, 14] have demonstrated that applying simpler, lower-precision data formats, such as 8-bit integer (INT8) or even 4-bit integer (INT4), can further compress the networks without apparent accuracy loss for the inference results. Changing the data format from a high precision to a lower one is called quantization. For example, quantizing the network from FP32 to INT8 allows for $4\times$ less memory usage and bandwidth while avoiding complex floating-point algebra units to perform calculations, leading to $3-4\times$ acceleration and $3-7\times$ energy efficiency improvements in hardware [11], which is highly favorable for edge systems.

However, emerging GNNs lack a standard to perform quantization, therefore we developed a feasible two-step quantization workflow for GNNs, which we will introduce in this section.

3.4.1. Graph Convolution - Batch Normalization Folding

Quantization introduces errors due to a loss in data precision. For neural network quantization, if we quantize the convolution layer and the BN layer separately, quantization errors will be introduced twice. To alleviate this issue, an approach is to fuse the convolution computation with the BN computation, *i.e.* convolution-BN Folding, and then to quantize the fused layer [14]. Indeed, the basis of the convolution algorithm is a matrix multiplication, which can be easily combined with the scaling-and-shifting operation in the BN layer introduced in Section 2.2.1.

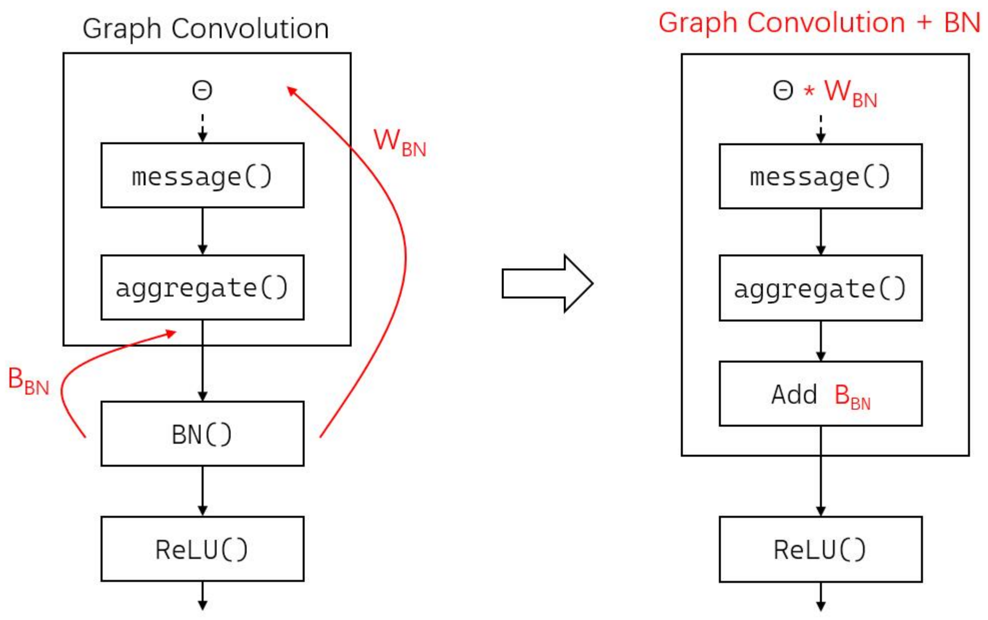


Figure 3.7: Graph convolution - BN folding. The $message()$ function is the message generation step, and the $aggregate()$ function is the aggregation step.

However, as introduced in Section 2.3.1, our simplified PointNetConv (Eq. 2.13) is actually a linear transformation (the message generation step) wrapped by a non-linear max function (the aggregation step), therefore it is not trivial to implement convolution-BN folding on it.

Based on Eq. 2.5 and Eq. 2.13, a uniform equation for a PointNetConv followed by a BN layer is described as:

$$\begin{aligned} \mathbf{x}'_i &= BN\left(\max_{j\in\mathcal{N}(i)} \mathbf{\Theta}\cdot(\mathbf{x}_j\,,\,|\mathbf{p}_j-\mathbf{p}_i|)\right) \\ &= W_{BN}\circ\left(\max_{j\in\mathcal{N}(i)} \mathbf{\Theta}\cdot(\mathbf{x}_j\,,\,|\mathbf{p}_j-\mathbf{p}_i|)\right)+B_{BN} \end{aligned} \tag{3.9}$$

Since elements in W_{BN} are positive (see Section 2.2.1), the order of scaling operation ($W_{BN} \circ$) and aggregation ($\max()$) can be reversed, therefore:

$$\begin{aligned} \mathbf{x}'_i &= W_{BN} \circ \left(\max_{j \in \mathcal{N}(i)} \mathbf{\Theta} \cdot (\mathbf{x}_j \, , \, |\mathbf{p}_j - \mathbf{p}_i|) \right) + B_{BN} \\ &= \max_{j \in \mathcal{N}(i)} (W_{BN} * \mathbf{\Theta}) \cdot (\mathbf{x}_j \, , \, |\mathbf{p}_j - \mathbf{p}_i|) + B_{BN} \\ &= \max_{j \in \mathcal{N}(i)} W \cdot (\mathbf{x}_j \, , \, |\mathbf{p}_j - \mathbf{p}_i|) + B_{BN}, \end{aligned} \tag{3.10}$$

where we define the column-wise matrix-vector multiplication $*$ as

$$\begin{aligned} W &= W_{BN} * \mathbf{\Theta} \\ &= \begin{bmatrix} w_1 \\ \vdots \\ w_{C_{out}} \end{bmatrix} * \begin{bmatrix} \theta_{1,1} & \cdots & \theta_{1,C_{in}} \\ \vdots & \ddots & \vdots \\ \theta_{C_{out},1} & \cdots & \theta_{C_{out},C_{in}} \end{bmatrix} \\ &= \begin{bmatrix} w_1 \cdot \theta_{1,1} & \cdots & w_1 \cdot \theta_{1,C_{in}} \\ \vdots & \ddots & \vdots \\ w_{C_{out}} \cdot \theta_{C_{out},1} & \cdots & w_{C_{out}} \cdot \theta_{C_{out},C_{in}} \end{bmatrix} \end{aligned} \tag{3.11}$$

The complete workflow described in Eq. 3.10 is depicted at Fig. 3.7.

3.4.2. Post-Training Quantization

To achieve full quantization of the network, post-training quantization (PTQ) is the most straightforward solution. PTQ, as its name suggests, will first perform a normal network training phase in the original data precision (FP32), then quantize the FP32-format weight and bias matrices into a lower precision, such as INT8, its unsigned counterpart UINT8, or even fewer bits. Because PTQ quantizes the network after the training, it can only be implemented in the inference phase, which is appropriate for an inference-only hardware accelerator.

There are two phases in the PTQ [17]: calibration and quantization, shown in Fig. 3.8. To quantize weights, bias, and features in the GNN calculation, we first need to obtain the maximal and minimal values of these data by running the GNN inference in the original FP32 format on the specific dataset we used, *i.e.* the NCars. This step is called the calibration.

$$Q_i = round(r_i/S), \text{ where } S = \frac{|r|_{max}}{|Q|_{max}} \tag{3.12}$$

After calibration, the quantization step can be performed according to Eq. 3.12. The basic method for quantizing a floating-pointer number to an integer described in Eq. 3.12 [14] is called scale quantization. Here, r_i is a floating-point real number belonging to set r, which represents a set of floating-point data, such as all weights, all bias, or all features, of one network layer. Its maximal absolute value, $|r|_{max}$, is derived from the above calibration step. Q_i is the quantized integer belonging to set Q, which includes all possible numbers within the target quantization precision, *e.g.* for INT8, $Q = [-128, 127] \subseteq \mathbb{Z}$, $|Q|_{max} = 127$; and for UINT8, $Q = [0, 255] \subseteq \mathbb{Z}$, $|Q|_{max} = 255$. Together with the scaling factor S, each floating-point number r_i can be quantized into an integer Q_i without overflow nor underflow, while a zero in r still maps to a 0 in Q.

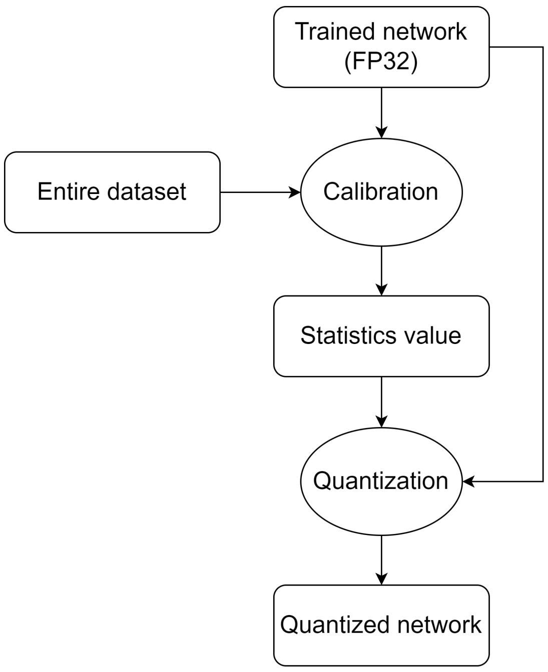


Figure 3.8: Flow chart of the PTQ, including calibration and quantization after the training phase.

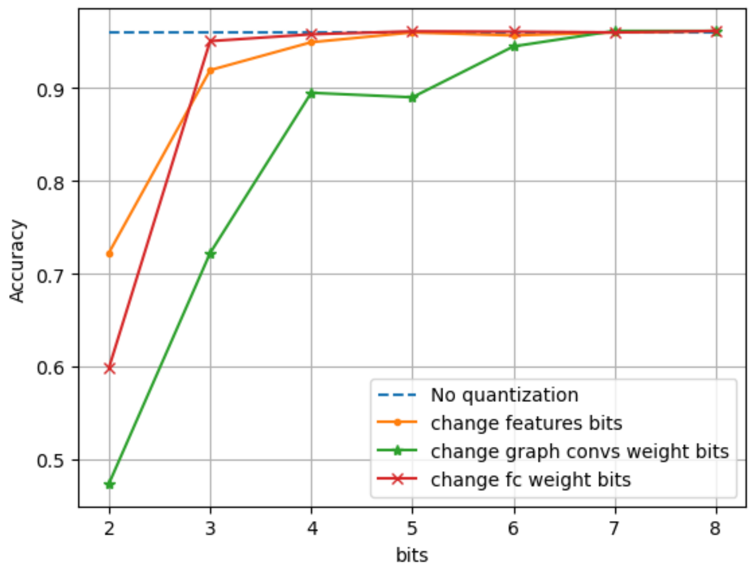


Figure 3.9: Inference accuracy after different bit-width quantization.

This idea can be easily applied to linear transformation, which is the core of neural networks. It can be proved that [14], quantizing a linear transformation is equivalent to multiplying a quantization factor M by the transformation.

Following this idea, we quantized each graph convolution - BN folding layers and the final FC layer in our GNN, by transforming all features into unsigned integers (since they are the output of the ReLU function which yields no negative data) and all weights and biases into signed integers. To decide the precision, we performed the network inference with various bit widths, ranging from 8 bits to 2 bits. As demonstrated in Fig. 3.9, when setting the bit width to 8, the accuracy is basically the same as the result without quantization. Therefore, we chose the INT8-quantization for all weights, and UINT8-quantization for all features.

3.5. Overall Co-design and Event-driven Experiments Results

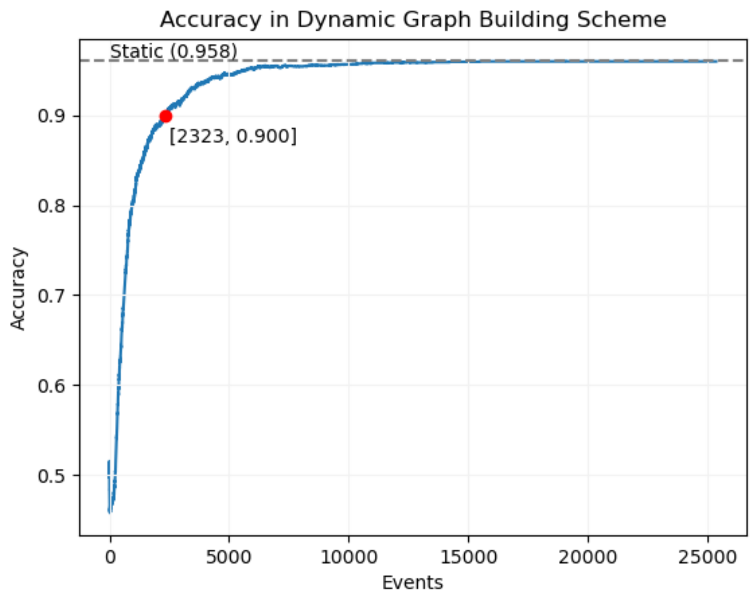


Figure 3.10: The accuracy changing when using dynamic event graphs.

From the above discussion, our final co-design schemes are: (i) graph building: prism neighbor search range with $D_{max} = 16$; (ii) GNN network optimization: simplified structure with ReLU activation layers and PointNet graph convolution; (iii) full-integer network quantization. Combining all co-design schemes, our final proposed event-driven GNN reaches a prediction accuracy of 95.8% on the validation dataset, which is even higher than the AEGNN baseline (94.5%).

However, the above accuracy results are still derived by using the static graph building method (see Section 2.1.2). Since we are designing an event-driven GNN, we also tested the accuracy changing using dynamic event graphs. Fig. 3.10 shows the average GNN prediction accuracy changing across the entire validation dataset, according to the number of events processed. It is clear that the accuracy is increasing as more events are processed by the event-driven GNN. The final average accuracy is 96.0%, which matches the accuracy obtained with static graph building. We also highlight the event number when the accuracy reaches 90%. The 90%-correct accuracy only requires 2,323 events, which provides an opportunity for an early-stopping inference scheme, if a faster though slightly less accurate prediction is further needed in the selected real-world application scenario.

For a fast and convenient co-design workflow, we did not test the prediction accuracy of our event-driven GNN on the test dataset (8,606 samples) of NCars, since the processing time is much longer than on the smaller validation dataset (2,462 samples). However, the test dataset was still leveraged to benchmark our final hardware accelerator, which will be introduced in the next chapter.

4 Hardware Implementation

Thanks to the hardware-friendly scheme derived from the hardware-algorithm co-design part in Chater 3, we can easily map the software code into an actual hardware solution. In this thesis, we implemented our design on the Xilinx KV260 development board, which contains a Zynq UltraScale+ MPSoC in the Kria K26 System-On-Module (SOM) platform, targeting edge vision applications.

This chapter will first introduce our overall design diagram. Afterwards, components of the system, including the graph building module, the graph convolution module, the pooling module, the FC computation module, and the host CPU controlling software are introduced. Finally, the performance of the on-board implementation is tested, with the results provided at the end of this chapter.

4.1. Overall System Architecture

There are two major parts in the Xilinx Zynq MPSoC: the Programmable Logic (PL) part containing the field-programmable-gate-array (FPGA), and the Processing System (PS) part containing an ARM CPU and 4GB of external DRAM as the main memory for the CPU. The hardware in the PL is also able to access the DRAM in the PS. In the following contents, we name the PS DRAM as the ”off-chip” memory, while the SRAM memory resources (total 3.32MB) in the PL part, including the Distributed-RAM (LUTRAM, 0.44MB), Block-RAM (BRAM, 0.63MB), and Ultra-RAM (URAM, 2.25MB), are regarded as the ”on-chip” memory [36].

Fig. 4.1 shows the overall system architecture. The entire hardware accelerator is implemented on the PL part using the Verilog and SystemVerilog hardware description languages (HDLs). The ARM core in the PS works as a host CPU, responsible for loading dataset samples, feeding data to the accelerator, reading accelerator prediction results, calculating the prediction accuracy, and measuring the runtime.

The PS and the PL can communicate through bi-directional AXI system buses. The AXI protocol family is established by the ARM company [2], and we used two sub-protocols for communication: AXI-Lite and AXI-Memory-Mapped (AXI-MM). The AXI-Lite is suitable for transmitting a single data pack, such as event data, control signals, and status signals. The other bus, the AXI-MM bus, can transmit data packs with larger bit-width (in our case $4\times$), thus we use it as the off-chip memory bus for the accelerator to load intermediate data from, and store the data into the off-chip memory.

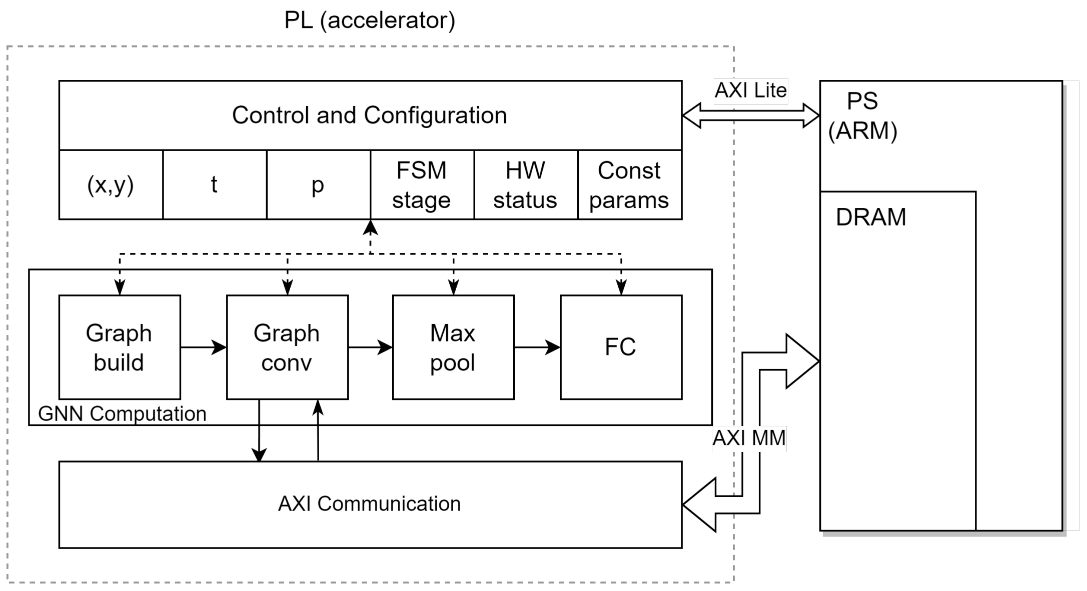


Figure 4.1: The overall system architecture. The designed accelerator is located in the PL part, while the PS host CPU is mainly responsible for benchmarking and monitoring.

Inside the PL accelerator, there are three major blocks: the control and configuration, the GNN computation, and the AXI communication.

There are two purposes for the control and configuration block. From/to the outside of the accelerator, represented by the PS, it receives the event data $e = (x, y, t, p)$, the accelerator control signals (start signal and data clean signal), and some constant parameters, while it sends back a binary prediction result (”Car” or ”Background”) and accelerator status signals (idle signal or done signal). From/to the inside of the accelerator, it broadcasts the event data to all modules in the GNN computation block and guides their running order using a finite-state machine (FSM).

The GNN computation block consists of 4 modules corresponding to 4 important steps in the event-driven GNN computation pipeline, which are: graph building, graph convolution, grid-based feature max pooling, and the FC prediction head, labeled as *Graph build*, *Graph conv*, *Max pool*, and *FC* in Fig. 4.1 respectively. These modules are functionally identical to the steps in the algorithm mentioned in previous chapters with the corresponding names.

The AXI communication block is designed to ease the design of the graph convolution module. The graph convolution module alone requires at least 3.1MB of memory to store the GNN network features, not to mention the memory requirements of other modules. Therefore, due to the limited on-chip memory resources inside the Zynq MPSoC (3.32MB), the graph convolution module has to offload its data to the off-chip DRAM through the AXI communication block, which is made of a large AXI buffer for the sending and receiving off-chip memory data, and a standard module implementing the AXI-MM protocol.

In the following sections, we will elaborate on the design details of the core of this accelerator, *i.e.* the 4 modules of the GNN computation block, in a top-down manner. We will also introduce the host software we designed for controlling and benchmarking the accelerator in an integrated environment. Finally, we will also provide the actual on-board performance experiment results in the last section.

4.2. Graph Building

As introduced in Section 3.2, in this thesis, the first step for establishing our event-driven accelerator is to build a directed dynamic event graph, followed by an event-driven GNN. In order to map the graph building algorithms into custom hardware, we address two important problems: where to store the event graph, and how to select neighbors of the new event. The graph building hardware diagram is shown in Fig. 4.2.

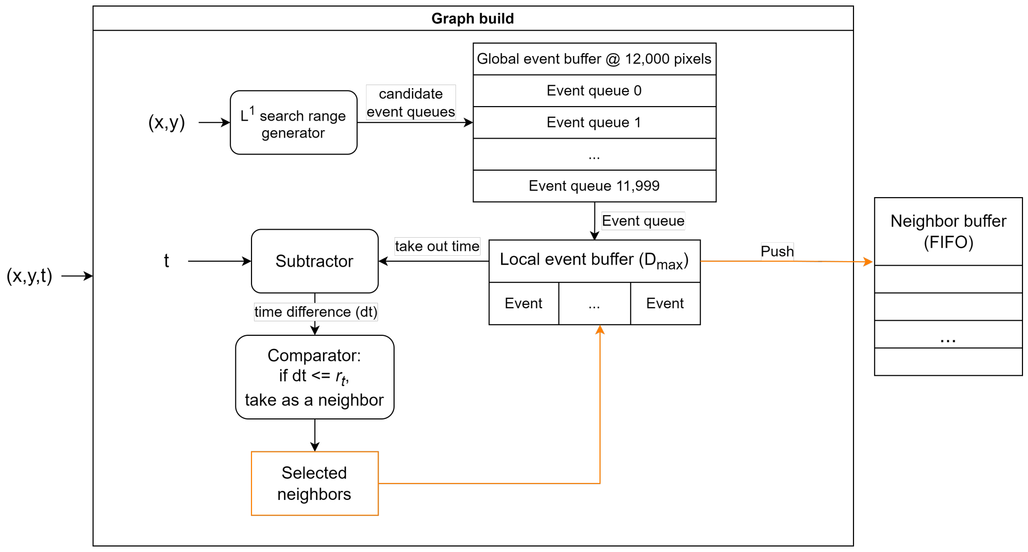


Figure 4.2: Hardware diagram of the graph building module, designed specifically for NCars, which uses an event-based camera with 12,000 pixels. The input is the new event's spatiotemporal position (x, y, t), and the output is stored into the neighbor buffer FIFO.

4.2.1. Edge-Free Event Graph Storage and Event Queues

There are two fundamental elements in an event graph: the nodes (or events) and the edges. Thanks to the directed graph scheme introduced in Section 3.2.1, we do not need to store and fetch the edges in the graph: since we only need the 1-hop locality information, *i.e.* the direct neighbors of each new event, edges only need to be connected and processed once. Therefore, once processed, edges do not have to be stored. Only the nodes, which contain spatiotemporal positions (x, y, t) closely linked to the graph building process, need to be stored.

To store and process the position information of events, we adopt the *event queue* technique, which has been used in several previous works [19, 34]. Event queues for each pixel are First-In-First-Out (FIFO) queues with a depth equal to the node maximal degree D_{max} (Section 3.2.2), which is illustrated in Fig. 4.3. At each pixel, an event queue FIFO is implemented to store events whose position is the same as this pixel's physical location (x, y). Besides spatial coordinates, other data of an event is also stored in the FIFO, including the timestamp t and event order n.

The event queue FIFOs have special storage and accessing behaviors contrary to standard FIFOs. When storing a new event into it, the event queue works like a FIFO, except that it is designed to be overflow-free: once a new event has to enter a full event queue FIFO, it will automatically pop out the first event in the FIFO, therefore the new event can always be stored unconditionally. However, when accessing the event queue, it then works like a normal memory unit where every entry inside can be read out independently and multiple times, contrary to the conventional FIFO which provides the data at the front end of the queue

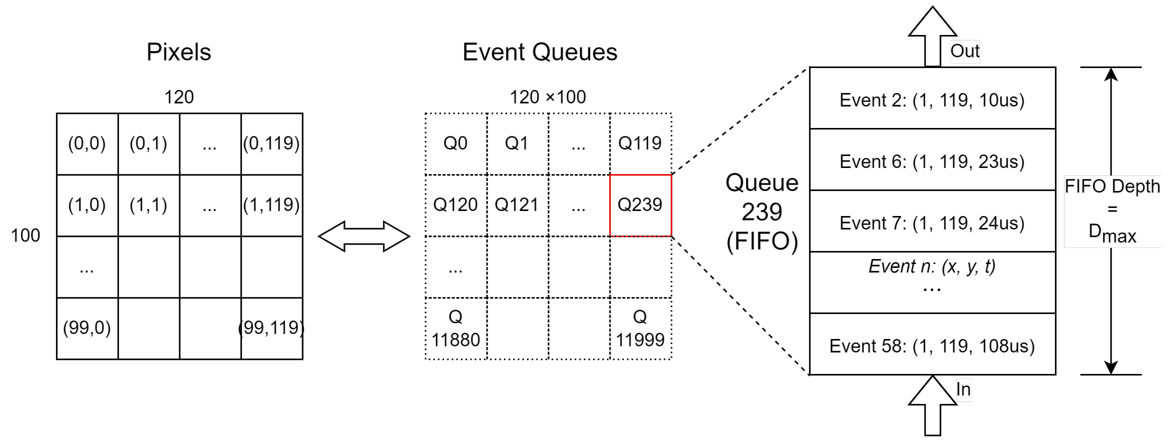


Figure 4.3: Illustrations for event queues. Left: Pixels from a 120×100 event-based camera. Each pixel has a physical location (x, y). Middle: 120×100 Event queues. Each queue corresponds to a unique pixel. Every event generated by that pixel will be collected by this queue. Right: An event queue FIFO. In a queue, event order n and its spatiotemporal coordinates (x, y, t) are stored. FIFO depth is equal to D_{max} in Section 3.2.2.

and discards it after being popped out.

In our hardware implementation, all event queues are stored in an on-chip memory buffer called the *global event buffer*, where each entry corresponds to an event queue.

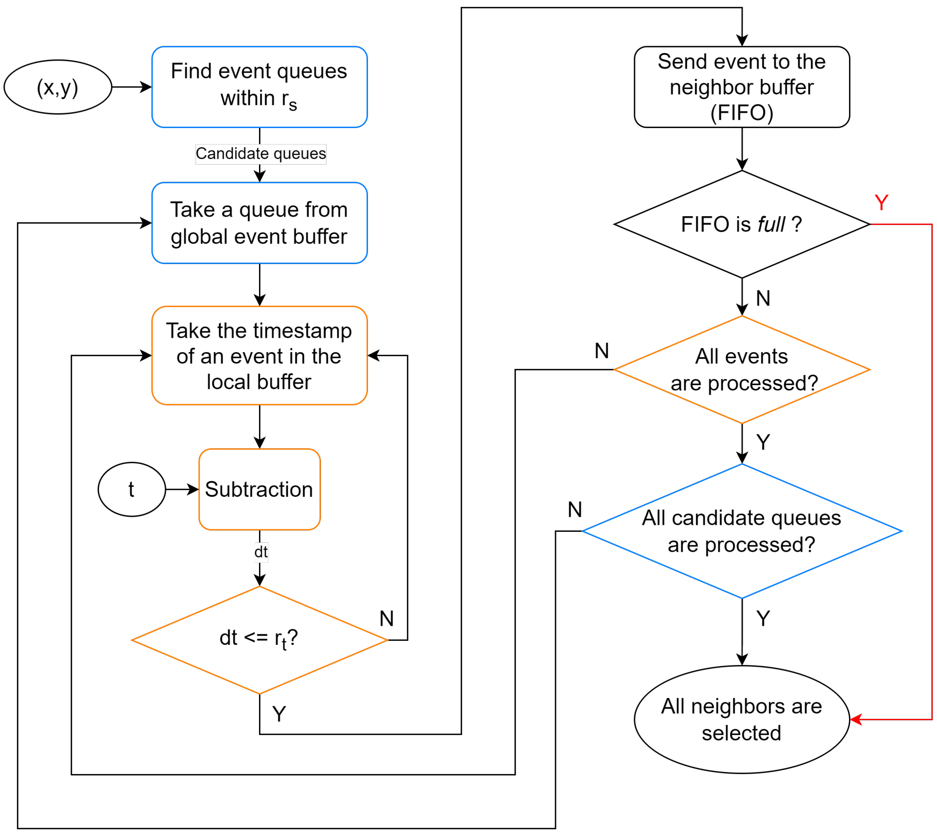
4.2.2. Event-queue-based Neighbor Selection

The aforementioned event queue storage scheme naturally separates space and time: each event queue corresponds to a pixel, thus its index directly maps the spatial location of the pixel; while inside an event queue, the timestamp is the only important difference across events. This spatiotemporal decoupling in storage facilitates our hardware design for implementing the spatiotemporally decoupled prism search range introduced in Section 3.2.3. Based on the event queues, the neighbor selection hardware sub-module can perform the spatial and temporal search step by step. The detailed workflow is shown in Fig. 4.4.

First, according to the position (x, y, t) of the new event, spatial search is performed to pick up all candidate event queues within r_s, which is equivalent to the first half of Eq. 3.8. For example, with $r_s = 3$, there are 25 event queues within the search range, which are accessed following the arrows shown in Fig. 4.5 to ease the queue-fetching process. If there is an invalid index, *i.e.* beyond the valid pixel ranges, it will be simply skipped, and the search for the next legal queue continues. Each time a queue is selected, it will be fetched from the global event buffer and stored in a smaller storage pool: the *local event buffer*.

Whenever a candidate queue is selected by the above process, the following temporal search is activated. The temporal search process first reads out each timestamp of events stored in the candidate queue and subtracts the timestamp of the current event to derive the time differences dt. Finally, according to Eq. 3.8, events that meet the constraint $dt \leq r_t$ will be selected as potential neighbors.

There is also a storage FIFO with a depth equal to D_{max}, named *neighbor buffer*, to store previous events selected as potential neighbors. The *full* signal of this FIFO can also control the spatial and temporal search processes, as shown in Fig. 4.4. After all neighbors are selected, the neighbor buffer will be further passed to the downstream module, the graph convolution module, for the following event-driven GNN computation.



(x,y)
Find event queues within r_s
Candidate queues
Take a queue from global event buffer
Take the timestamp of an event in the local buffer
t
Subtraction
dt
dt <= r_t?
N
Y
Send event to the neighbor buffer (FIFO)
FIFO is *full* ?
Y
N
All events are processed?
N
Y
All candidate queues are processed?
N
Y
All neighbors are selected

Figure 4.4: Graph building module workflow, from taking the new event position as the input, to all neighbors being selected. Blue blocks belong to the spatial search process, while orange blocks belong to the temporal search process. Note that if the neighbor buffer is full, it will immediately stop the spatial and temporal search processes.

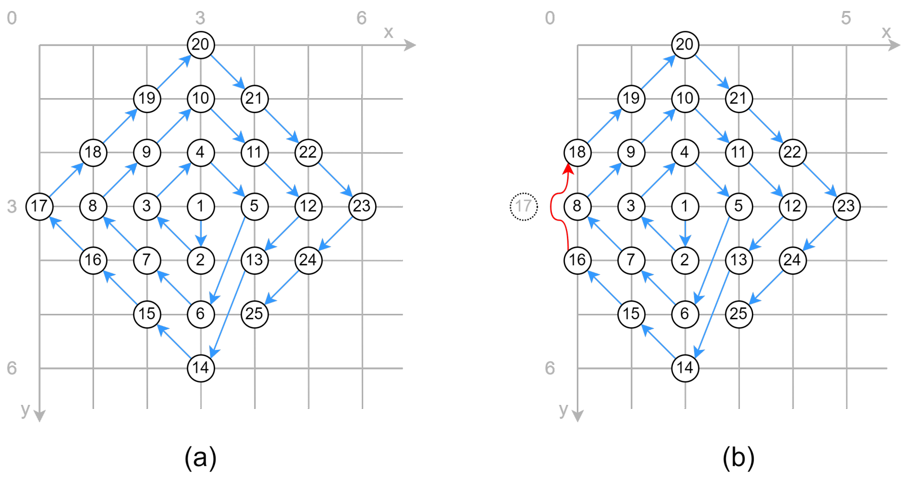


Figure 4.5: Event queue access order in L^1 search range within $r_s = 3$. (a). A new event locates at $(x, y) = (3, 3)$. Numbers in the circles represent the access order. (b). Another new event locates at $(x, y) = (2, 3)$. Illegal access (circle 17) will be directly skipped (red arrow).

4.3. Graph Convolution

In the previous section, the graph building module selected neighbors and passed them through the neighbor buffer. In this section, we will present the hardware design of the graph convolution module for our quantized event-driven GNN. We will first introduce a novel computation for multiple graph convolution layers: layer-parallel computation. Then, we will describe the detailed hardware sub-module design inside the graph convolution module, including message generation, aggregation, and biasing-activation-quantization.

4.3.1. Layer-Parallel Computation

For the first time, we point out that all graph convolution layers in an event-driven GNN using directed dynamic event graphs can be processed simultaneously, which we denoted as *layer-parallel computation*, and illustrate in Fig. 4.6.

To achieve parallel computation, there should be no data dependencies between graph convolution layers, *i.e.* the output of one layer cannot be the input of the next one. It is not possible for GNNs to use static graphs (Fig. 4.6(a)), as features of nodes will change after a convolution layer, thus the next layer has to use the updated features. The same problem occurs on event-driven GNNs using undirected graphs, such as AEGNN. As mentioned in Section 2.4.2, graph convolution layers in such GNNs will change the features of both the new event and its K-hop locality neighbors, and the next layer needs to process the changed neighbor features to process.

This problem is solved by event-driven GNNs leveraging directed dynamic graphs, such as HUGNet or our proposed GNN (Section 3.2.1). As mentioned in Section 2.4.3, directed edges prevent the graph convolution from changing the neighbor features. Meanwhile, features of the new event itself are not needed in the convolution computation, since the new event is forbidden to point back to older neighbors. Therefore, the input of such graph convolution layers, *i.e.* the features of the nodes neighboring each new event, can be considered as constants. The output of the convolution layers, *i.e.* the features of the new event, can just be stored, waiting for the next upcoming event. This process is depicted in Fig. 4.6(b).

Therefore, in this computation flow, the output of a convolution layer is not the input of the next layer, but the input of the next event, thus cutting the data dependencies between layers. Hence, as shown in Fig. 4.6(c), we can process all graph convolution layers simultaneously: we first collect all features of neighbors generated by every convolution layer, then parallelize layer computation. This series-to-parallel computation flow reduces the total runtime of the GNN, making it more suitable for real-time hardware implementations, targeting high-temporal-resolution event flows.

4.3.2. Graph Convolution Hardware Sub-modules

Based on Section 2.3.1 and the proposed layer-parallel computation, the workflow of the graph convolution sub-modules can be divided into 5 steps, shown in Fig. 4.7:

1. Loading features: the module first tries to read one neighbor's information in the neighbor buffer. If the buffer is not empty, the module will use the neighbor's event order n as the index to fetch the corresponding feature vectors stored in the off-chip DRAM. In this step, all feature vectors of layers are fetched and loaded for the layer-parallel computation.
2. Message generation: according to Eq. 2.13, for each neighbor in the input neighbor buffer, a message vector is generated by a linear transformation.
3. Aggregation: a max aggregation function is used according to PointNetConv. Among all the neighbors, this max aggregation function compares and selects the maximum

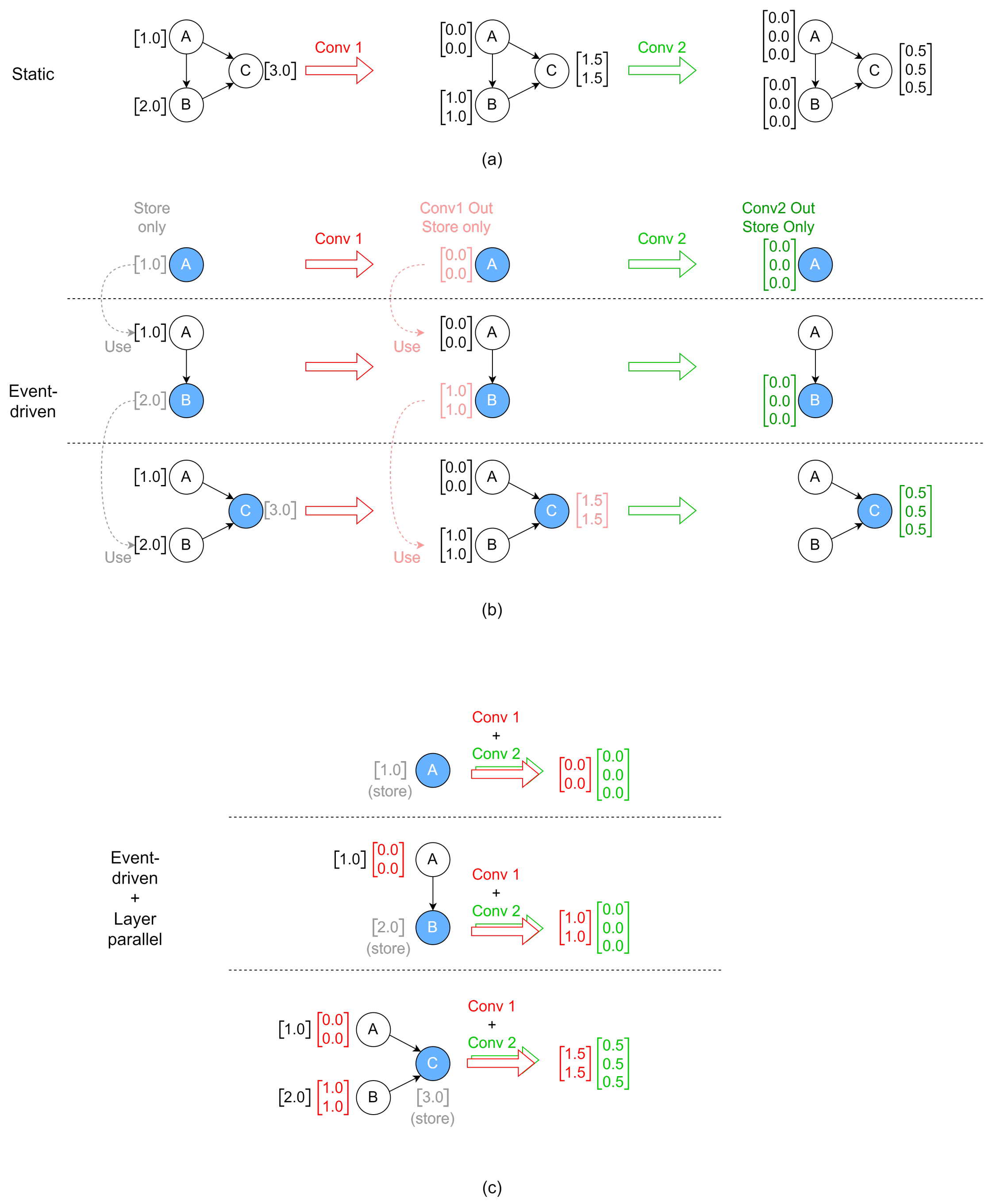


Figure 4.6: Event graphs processed by an example GNN containing two graph convolution layers, *Conv 1* and *Conv2*, both of which consist of a simple average-and-replicate operation for illustration. (a) A static event graph is processed by the GNN. The features are changed and the channels of features are increased by the graph convolution layers. (b) A directed dynamic event graph is processed by the GNN in an event-driven fashion, with new events colored blue. The outputs are identical to (a). Note that the output of a convolution layer is not used by the next layer, but by the next event in the same layer, denoted in dotted arrows. (c) The same directed graph is processed by a convolution-layer-parallel event-driven GNN, which is mathematically equivalent to (b).

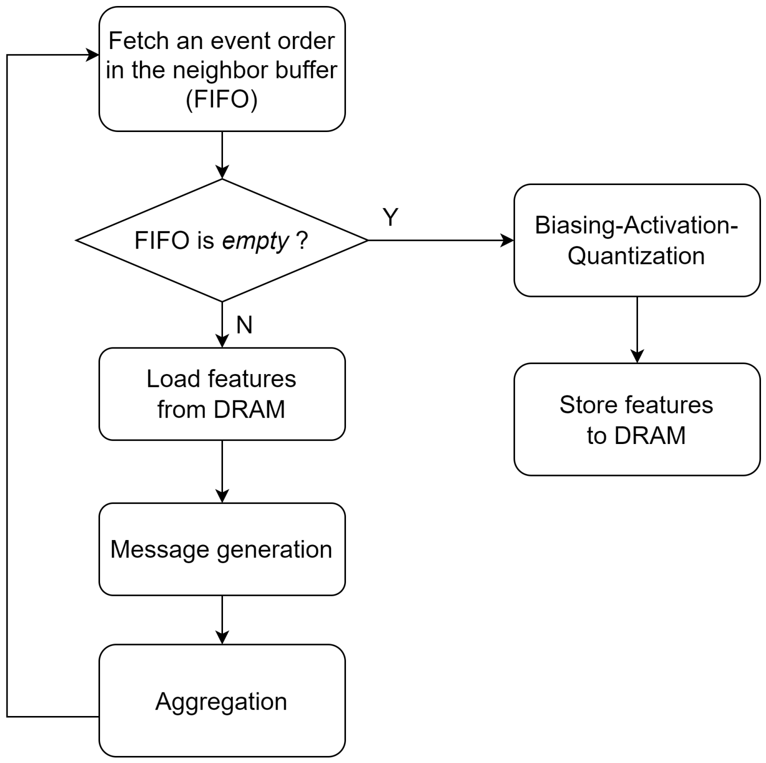


Figure 4.7: Graph convolution module workflow, including 5 major steps.

value for each element in the generated message vectors independently, forming a new feature vector. Then, the module proceeds to read every entry in the neighbor buffer until it is empty, which indicates that all neighbors have been processed.

4. Biasing-activation-quantization (BAQ): after all neighbors have been processed, a bias vector B_{BN} derived from batch normalization is added to the new feature vector. Then, the biased vector is rectified by the ReLU activation layer and, eventually, the activated vector is quantized, leading to the final feature vector.
5. Storing features: the new event's feature vectors derived from the graph convolution layer are sent back to the off-chip DRAM. The memory address is decided by the new event's order in the event stream.

Step 1 and Step 5 are feature data movements, which are performed by the AXI communication block introduced in Section 4.1. Steps 2 to 4 are mapped into 3 sub-modules in hardware, as shown in the hardware diagram Fig. 4.8, which are described hereafter.

Message Generation

The core of the message generation step is the linear transformation, *i.e.* the combination of matrix-vector multiplications, where the vector is one feature vector of the neighbor, and the matrix is the column-wise matrix-vector product (matrix W) of the trained weight Θ and the scaling factors W_{BN} due to the convolution-BN folding in Eq. 3.11, which is reminded below for convenience:

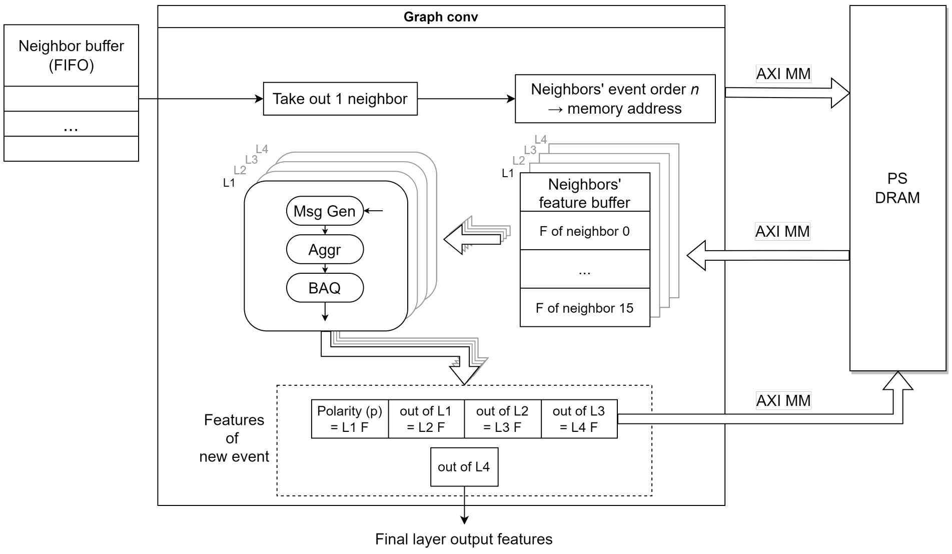


Figure 4.8: Hardware diagram of the graph convolution module. The letter "F" refers to features. This diagram is specific to our GNN structure, which contains four graph convolution layers denoted as L1, L2, L3, and L4. Each graph convolution module has three sub-modules: message generation (*Msg Gen*), aggregation (*Aggr*), and bias-activation-quantization (*BAQ*), taking features from the corresponding neighbors' feature buffer with depth equal to D_{max} (16). All convolution layers are computed simultaneously, following the proposed layer-parallel computation.

$$
\begin{aligned}
W &= W_{BN} * \mathbf{\Theta} \\
&= \begin{bmatrix} w_1 \cdot \theta_{1,1} & \cdots & w_1 \cdot \theta_{1,C_{in}} \\ \vdots & \ddots & \vdots \\ w_{C_{out}} \cdot \theta_{C_{out},1} & \cdots & w_{C_{out}} \cdot \theta_{C_{out},C_{in}} \end{bmatrix}
\end{aligned}
$$

$$
\mathbf{message} = W \times \mathbf{x} = \begin{bmatrix} w_{1,1} & \cdots & w_{1,C_{in}} \\ \vdots & \ddots & \vdots \\ w_{C_{out},1} & \cdots & w_{C_{out},C_{in}} \end{bmatrix} \times \begin{bmatrix} x_1 \\ \vdots \\ x_{C_{in}} \end{bmatrix} = \sum_{j=1}^{C_{in}} \begin{bmatrix} x_j \cdot w_{1,j} \\ \cdots \\ x_j \cdot w_{C_{out},j} \end{bmatrix}, \qquad (4.1)
$$

where $w_{i,j} = w_i \cdot \theta_{i,j}$ from the above mentioned Eq. 3.11.

To achieve matrix-vector multiplications, we designed the MatVec unit. This unit is based on a specific multiplication decomposition algorithm shown in Eq. 4.1, which we named *row-parallel-column-accumulated* (RP-CA) multiplication. Here, $\mathbf{x}$ represents the input feature vector with C_{in} input feature channels, and $\mathbf{message}$ represents the generated message vector with C_{out} output feature channel. The reason to choose this type of multiplication is two-fold: if we perform matrix-vector multiplication element-by-element (*i.e.* serial multiplication), then the overall computation latency will be unacceptably slow; otherwise, if we perform every operation in the multiplication at the same time (*i.e.* parallel multiplication), then the logic overhead will exceed the FPGA resources. Therefore, this RP-CA multiplication can reasonably balance both the resource usage and the computation latency in a reasonable range.

Based on this algorithm, the hardware design of the message generation sub-module is depicted in Fig. 4.9(a). Here, the weight matrix W, containing multiple weights $w_{i,j}$, is separately stored in C_{out} on-chip SRAM. Each memory stores C_{in} weight elements, and all memories

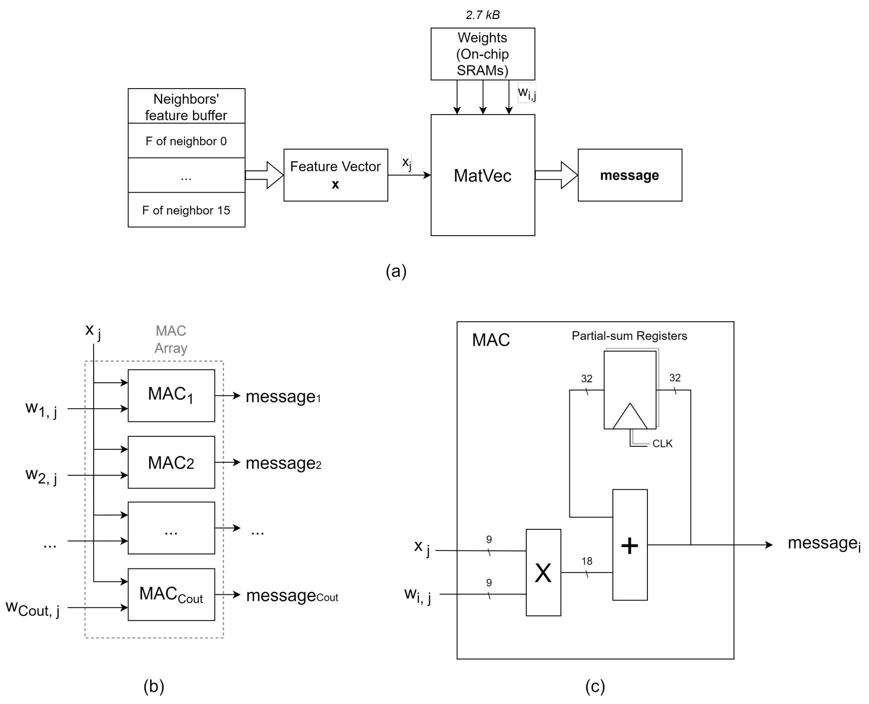


Figure 4.9: (a) Hardware diagram of the message generation sub-module, including a MatVec unit and on-chip weight memories. (b) MatVec inner structure. It repeats C_{in} times for accumulation ($j = 1, \cdots, C_{in}$). (b) MAC inner structure. Both x_j and $w_{i,j}$ have 8-bit data and 1-bit sign extension. The output $message_i$ is padded to 32-bit to avoid overflow in accumulation.

can send out 1 element at a time, thus forming a memory array able to output one column of the weight matrix at once.

The j^{th} column of the weight matrix, denoted as $w_{1,j}, \cdots, w_{C_{out},j}$, and the j^{th} element of $\mathbf{x}$, are sent together into the MatVec unit consisting of C_{out} multiplier-and-accumulator (MAC) units. x_j is broadcasted to all MAC units, while each weight in the column is dispatched to one MAC unit. The MAC unit first multiplies the weight element with the feature element, and then accumulates the product with the previous output, whose inner structure is shown in Fig. 4.9(c).

This process is repeated until all C_{in} feature elements and weight columns are processed. This operation can run in a pipelined fashion thanks to partial-sum registers in the MAC units, thus reducing the computation latency.

Aggregation

The aggregation sub-module is activated upon completion of the message generation sub-module, and keeps track of all neighbors' messages to find the maximum elements (Fig. 4.10). The aggregation sub-module is made of C_{out} processing elements (PEs), each of which includes a register file *History* (64B to 128B, according to the output channels of different graph convolution layers) to hold the last maximum value, and a multi-bit algebra comparator (*max()*) selects the largest value between the newly received message from the message generation sub-module ($message_i$) and the previous maximum value. After all neighbors are processed, all messages are aggregated, and a feature vector of the new event is automatically generated, waiting to be sent to the final biasing-activation-quantization sub-module.

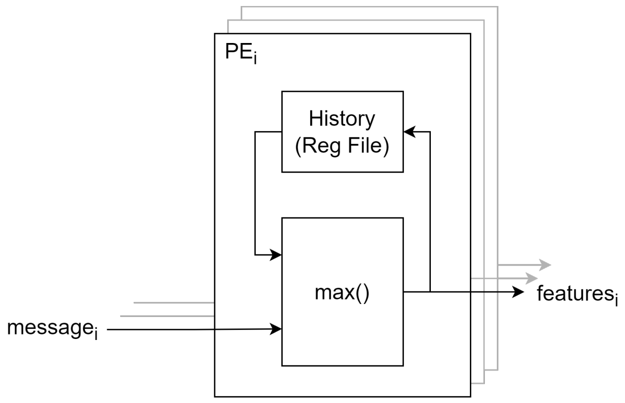


Figure 4.10: Hardware diagram of the aggregation sub-module.

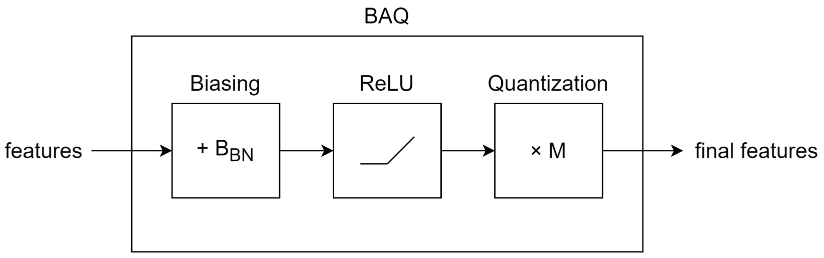


Figure 4.11: Hardware diagram of the biasing-activation-quantization (BAQ) sub-module.

Biasing-Activation-Quantization (BAQ)

Similar to the aggregation step, the biasing step, the activation step, and the quantization step process the new feature vector element by element, thus we combine them in a biasing-activation-quantization (BAQ) sub-module. The diagram is shown in Fig. 4.11.

$$\mathbf{x}_{final} = \max\left(0, (\mathbf{x} + B_{BN})\right) \cdot M \tag{4.2}$$

The BAQ sub-module can be described by a uniform equation, Eq. 4.2. After all neighbors are processed, the BAQ sub-module first adds a bias vector B_{BN}, derived from the BN layer, to the feature vector (Eq. 3.10). Then, the biasing vector passes through the ReLU activation function (Eq. 2.4). Finally, the activated vector is multiplied by the factor M to perform the quantization step, as introduced in Section 3.4.2. After these steps, the final quantized feature vectors are sent to the off-chip DRAM, which ends the graph convolution module operation.

4.4. Grid-based Feature Max Pooling

As outlined in Fig. 4.1, the output feature vector from the last graph convolution layer then goes through the grid-based feature max pooling layer, which derives grid features for the rest of the GNN.

Following Section 3.3.3, we split the full 120×100-pixel input range into 8×7 grids, each of which is a square region containing 16×16 pixels with a unique grid index.

The hardware diagram of the feature max pooling module is shown in Fig. 4.12. The new event's position (x, y) is first used to calculate the grid index it belongs to. According to this index, the module fetches the history of maximum grid features in a buffer. Although similar to the aggregation sub-module, here, the ”history” features are actually from a previous event,

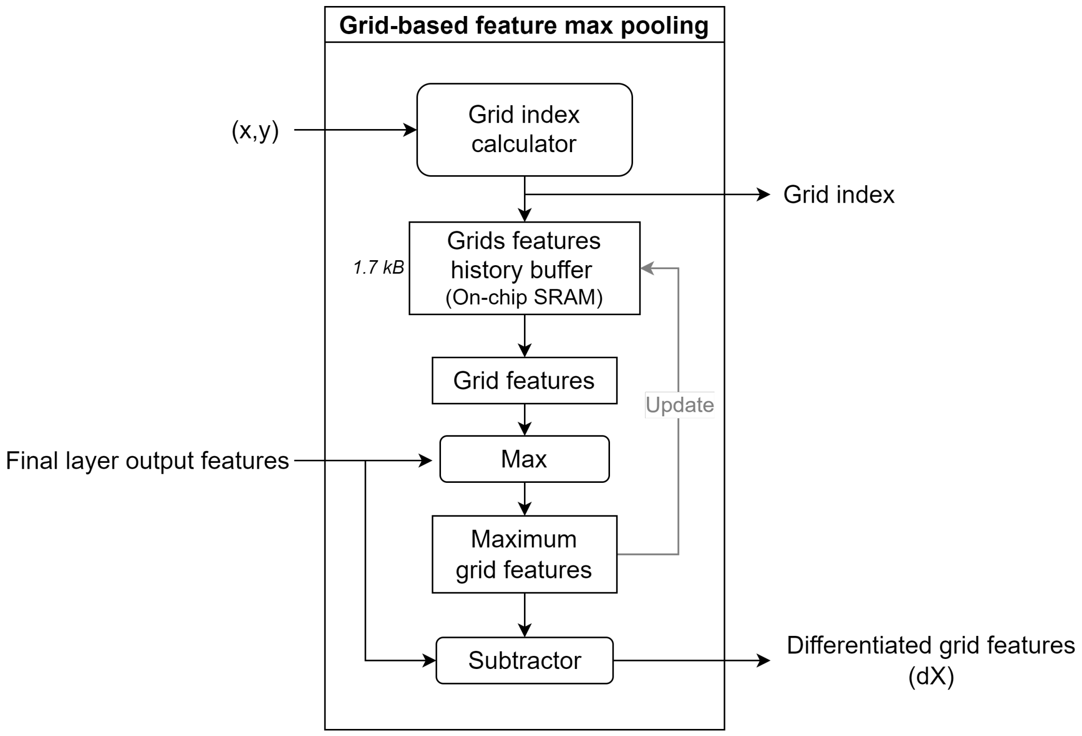

Grid-based feature max pooling
(x,y)
Grid index calculator
Grid index
1.7 kB
Grids features history buffer (On-chip SRAM)
Grid features
Update
Final layer output features
Max
Maximum grid features
Subtractor
Differentiated grid features (dX)

Figure 4.12: Hardware diagram of the grid-based feature max pooling module. Left-side arrows indicate the module input, consisting of the new event's pixel location (x, y), and the output node features from the final layer of the event-driven GNN. Right-side arrows indicate the module output, consisting of the grid index where the new event belongs, and the differentiated grid features, derived from subtracting the final layer features with the maximal grid features.

not from the last neighbor. After fetching the history maximum, a same *max()* comparator selects the new maximum features between the previous and the current features, which will not only update the history buffer, but also subtract from the current features to generate the differentiated grid features, *i.e.* $d\mathbf{X} = \mathbf{x}'_{grid} - \mathbf{x}_{grid}$ (Eq. A.3), used by the final FC layer.

4.5. Fully Connected Module

Guided by Eq. A.2:

$$\mathbf{y}' = \mathbf{y} + d\mathbf{y} = \Theta \times \mathbf{X} + \Theta_{partial} \times d\mathbf{X},$$

the fully connected (FC) module, whose diagram is shown in Fig. 4.13, is responsible for the final prediction. The module first leverages the grid index generated by the feature max pooling module to form the partial weight matrix $\Theta_{partial}$. The same MatVec unit is then reused to calculate the matrix-vector multiplication $\Theta_{partial} \times dX$, where dX corresponds to the differentiated grid features mentioned above.

The output results of the FC layer are 2 numbers. Finally, a comparator-based predictor follows, which compares these 2 numbers to find the largest one. The larger number corresponds to a ”Car” prediction, while the smaller one to a ”Background” prediction.

4.6. Processing System Integration

To control and benchmark the accelerator implemented in the PL, it is necessary to design a host software running on a host CPU, *i.e.* an ARM core in the PS

The host software written in C code runs in a bare-metal environment. This software has 4

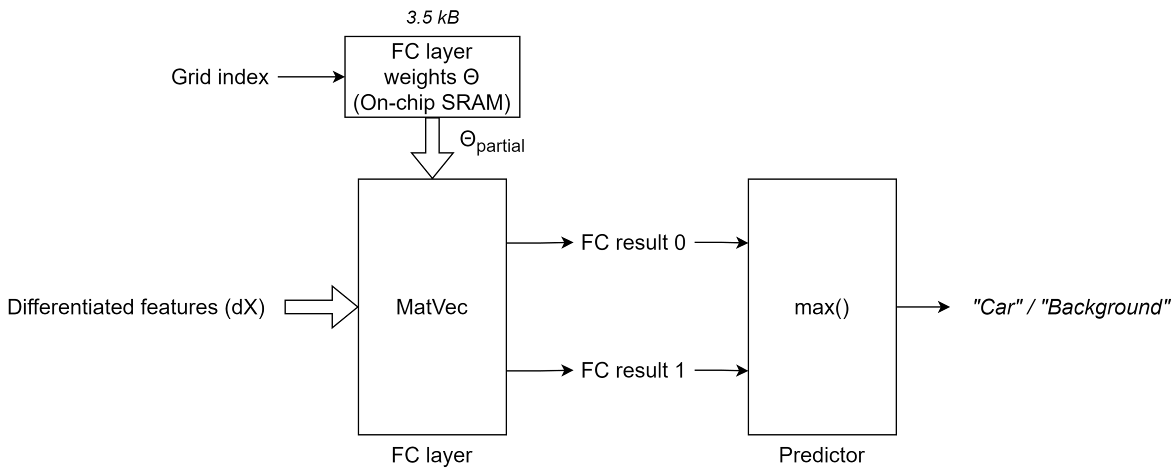


Figure 4.13: Hardware diagram of the FC sub-module. The MatVec (same as used in the graph convolution module) is re-used for the computation of the FC layer, and a predictor follows for generating the final prediction.

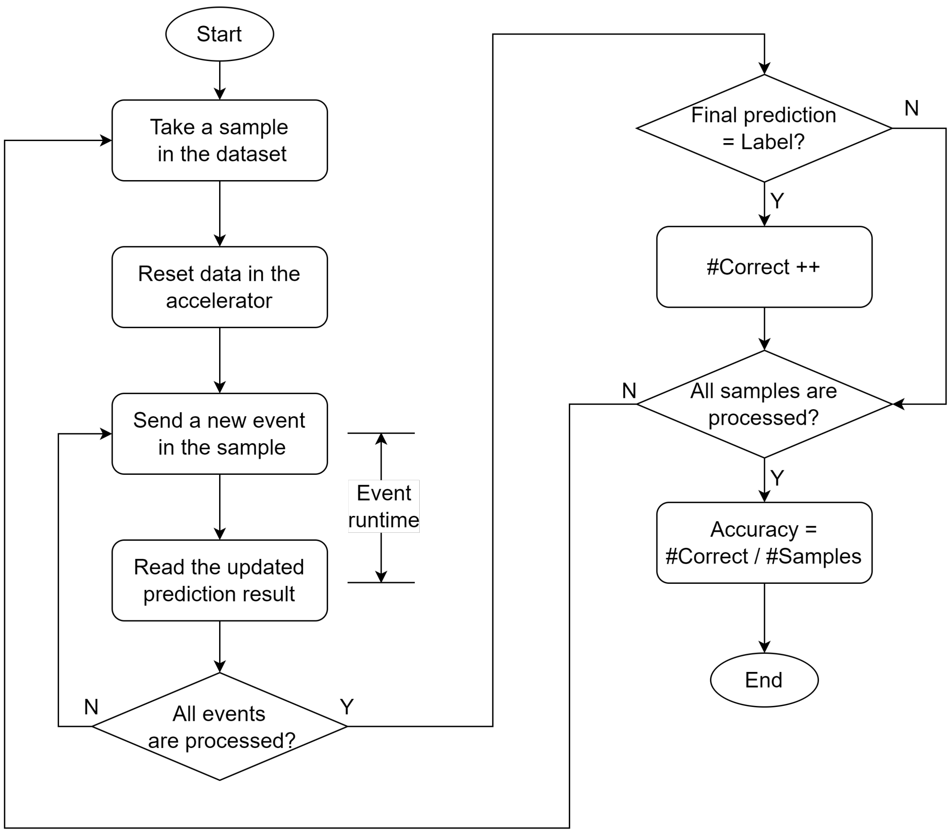

Start
Take a sample in the dataset
Reset data in the accelerator
Send a new event in the sample
Read the updated prediction result
Event runtime
All events are processed?
N
Y
Final prediction = Label?
N
Y
#Correct ++
All samples are processed?
N
Y
Accuracy = #Correct / #Samples
End

Figure 4.14: The host benchmarking workflow. #Correct means the number of correctly predicted samples, and #Samples represents the total number of samples in the validation/test dataset.

main functions: reading [1] the dataset in an SD card and storing it into the external DRAM, controlling the behavior of the accelerator, sending and reading data, and extracting performance results. The detailed workflow is depicted in Fig. 4.14 and described as follows:

1. At the beginning, the host CPU in the PS fetches a new sample from the dataset, and sends a *data clean* signal to the accelerator, to reset all data inside the accelerator to default values.
2. The actual inference process begins: the host sends one new event data from the selected sample. If the accelerator is idle, the host will start it, and start runtime recording at the same time.
3. The hardware accelerator selects neighbors for the new event and updates the graph inside the graph building module, then performs the GNN computation.
4. After the GNN computation, an updated prediction result is sent back to the host. The host stores the prediction and the accelerator runtime of this event, then sends the next event data in the sample.
5. Once all events in the sample are processed, the host takes the final prediction result to compare with the ground truth, *i.e.* the label of the selected sample. Then, the host sends a data clean signal, telling the accelerator to prepare for event data from a new sample.
6. If all samples in the dataset are processed, the host calculates the final prediction accuracy and average runtime for one event.

We benchmarked our accelerator with the validation dataset and the test dataset of the NCars. After benchmarking, the host software can report the statistics through a standard serial port (such as a USB) for further analysis.

4.7. Hardware Experiments and Results

In this section, we will discuss the experiments and results of our GNN accelerator implemented on the FPGA platform. As mentioned, we implemented our design on the Xilinx KV260 development board containing a Zynq UltraScale+ MPSoC, with the accelerator written in Verilog and SystemVerilog and implemented in the PL part, and the host software running in an ARM core located in the PS part. We will report the results in 4 aspects: prediction accuracy, runtime, FPGA resource usage, and board-level power consumption.

4.7.1. Prediction Accuracy

Table 4.1: Prediction accuracy for the object recognition task on the NCars dataset

Methods	Accuracy
NVS-S [19]	91.5%
EvS-S [19]	93.1%
AEGNN [30]	94.5%
Ours (software, validation set)	96.0%
Ours (hardware, validation set)	95.6%
Ours (hardware, test set)	87.8%

[1]From the viewpoint of the host CPU, the accelerator is regarded as a peripheral device, thus in the following, the term ”send” means passing data from the CPU to the accelerator, and the term ”read” means fetching data from the accelerator to the CPU.

The prediction accuracy results are provided in Table 4.1. For a total of 2,462 samples in the validation dataset, the overall prediction accuracy reaches 95.6% in hardware, which is consistent with the results of the hardware-algorithm co-design using dynamic event graphs on the same validation dataset (96.0%), proving that we successfully implemented and deployed our event-driven GNN algorithm in edge hardware.

For a total of 8,606 samples in the test dataset, the prediction accuracy drops to 87.8%, indicating reduced generalization capabilities for our event-driven GNN compared to state-of-the-art models in Table 4.1, which will be further investigated in future work.

4.7.2. Runtime

Table 4.2: Runtime comparison for the software and hardware solutions on the validation dataset.

Methods	Runtime
Event-driven GNN (software)	13h50m
Accelerator (hardware)	1h30m

FPGA acceleration also leads to a significantly faster runtime compared with the CPU or GPU platforms. The measurement results are shown in Table 4.2.

We first measured the total time for the whole benchmarking workflow, including the validation dataset loading time, and inference time of all samples (*i.e.* runtime from the *start* to the *end* in Fig. 4.14). Our proposed accelerator demonstrates significant speed-up compared with the software-only solution: it took $1h30m$ to process the validation dataset, achieving $9.2\times$ acceleration with respect to the event-driven GNN software counterpart implemented on an NVIDIA RTX A6000 GPU platform.

To further assess the potential of the accelerator when integrated into a real-world edge vision system, we also measured the event runtime, which is the time duration from receiving a new event to updating the prediction result. The event runtime reaches $16\mu s$ on average, basically matching the event generation time constant of the event-based camera, which is on the order of microseconds [9].

4.7.3. FPGA Resource Usage

Table 4.3: FPGA resource usage of our proposed hardware accelerator

Resources		Usage	Total	Percentage
Computation	LUT	30,908	117,120	26.4%
	DSP	228	1,248	18.3%
On-chip SRAM	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%

Table 4.3 demonstrates the resource usage of our proposed accelerator on the Xilinx KV260 platform. For the computational resources, we used 26.4% of look-up tables (LUTs) as logic, and 18.3% of digital signal processors (DSPs), which are mainly used for MAC units in the graph convolution module. For on-chip SRAM, we used 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.

4.7.4. Power Consumption

We measured the power consumption of the whole system with a power meter. The overall board-level system power is $6.86W$ on the Xilinx KV260 edge FPGA platform, which aligns with the watt-level low-power requirements for edge vision systems, and outlines a promising path for future ASIC implementations.

5 Conclusion

In this chapter, we will briefly conclude our works and results, with several perspectives for future works.

In Chapter 3, through hardware-algorithm co-design, we fined-tuned the entire event-driven GNN to enhance its hardware compatibility in three aspects: graph building algorithms, GNN architectures, and full-integer quantized computation.

1. We adopted the directed dynamic graph building algorithm proposed in HUGNet [5], with a restricted maximal number of neighbors and a customized prism neighbor search range.
2. We adjusted the GNN architecture in AEGNN, and leveraged more hardware-friendly alternatives for the activation layers and the graph convolution layers.
3. We implemented convolution-BatchNorm folding on GNNs. Based on that, we further applied a post-training quantization to implement the full-integer computation in our event-driven GNN.
4. Combining all designs, our event-driven GNN reached competitive 95.8% and 96.0% prediction accuracies on the NCars validation dataset for static and dynamic graph building schemes respectively.

In Chapter 4, we implemented our event-driven GNN algorithm into an edge FPGA MPSoC platform, then benchmarked our event-driven GNN hardware accelerator on-board with the validation dataset and the test dataset of NCars.

1. Thanks to directed dynamic event graphs, we introduced an edge-free event graph storage scheme and only stored nodes in event queues, which facilitated the hardware mapping of our graph building algorithms.
2. We pointed out a novel low-latency computation workflow for multiple graph convolution layers, the layer-parallel computation, which is enabled by the use of directed graphs for event-driven GNNs.
3. We designed every layer in an event-driven GNN in hardware, including the graph convolution module with layer-parallel computation, the feature max pooling module, and the FC prediction module.
4. Our accelerator achieved 95.6% prediction accuracy on the validation dataset, which is consistent with the dynamic result of the hardware-algorithm co-design in software,

96.0%. Meanwhile, operating in a total $6.86W$ SoC system power, the accelerator reached an average $16\mu s$ prediction latency per event, and ran $9.2\times$ acceleration compared with its software counterparts. Our accelerator also reached 87.8% prediction accuracy on the test dataset, which is acceptable but leaves room for further optimizations in the future.

Several aspects are worth further exploitation in the future.

1. In-depth research for the accuracy drop of 7.8% on the test dataset compared to the validation dataset. There are several possible reasons, such as the weak generalization capabilities of our GNN, or inappropriate system hyperparameters, which we will further investigate.
2. Configurable graph building hardware modules. For now, the parameters of the graph building module in our accelerator are fixed, which impedes scalability and flexibility.
3. More challenging computer vision tasks. For now, we target a car recognition task using the NCars dataset. In the future, adaptation to more challenging tasks, such as object detection or simultaneous localization and mapping (SLAM), will provide broader application scenarios for our accelerator.

References

[1] Pranav Adarsh, Pratibha Rathi, and Manoj Kumar. “YOLO v3-Tiny: Object Detection and Recognition using one stage improved model”. In: *2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS)*. 2020, pp. 687–694. DOI: `10.1109/ICACCS48705.2020.9074315`.

[2] ARM. *AMBA AXI Protocol Specification*. 2023. URL: `https://developer.arm.com/documentation/ihi0022/latest/`.

[3] Yin Bi et al. “Graph-based object classification for neuromorphic vision sensing”. In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2019, pp. 491–501.

[4] Michael M. Bronstein et al. “Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges”. In: *CoRR* abs/2104.13478 (2021). arXiv: `2104.13478`. URL: `https://arxiv.org/abs/2104.13478`.

[5] Thomas 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*. June 2023, pp. 3952–3961.

[6] Li Fei-Fei, Rob Fergus, and Pietro Perona. “Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories”. In: *2004 conference on computer vision and pattern recognition workshop*. IEEE. 2004, pp. 178–178.

[7] Matthias Fey and Jan Eric Lenssen. “Fast Graph Representation Learning with PyTorch Geometric”. In: *CoRR* abs/1903.02428 (2019). arXiv: `1903.02428`. URL: `http://arxiv.org/abs/1903.02428`.

[8] Matthias Fey et al. “SplineCNN: Fast Geometric Deep Learning with Continuous B-Spline Kernels”. In: *CoRR* abs/1711.08920 (2017). arXiv: `1711.08920`. URL: `http://arxiv.org/abs/1711.08920`.

[9] Guillermo Gallego et al. “Event-based vision: A survey”. In: *IEEE transactions on pattern analysis and machine intelligence* 44.1 (2020), pp. 154–180.

[10] Daniel 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.

[11] Song Han, Huizi Mao, and William J Dally. “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding”. In: *arXiv preprint arXiv:1510.00149* (2015).

[12] M. Hillebrand et al. “High speed camera system using a CMOS image sensor”. In: *Proceedings of the IEEE Intelligent Vehicles Symposium 2000 (Cat. No.00TH8511)*. 2000, pp. 656–661. DOI: `10.1109/IVS.2000.898423`.

[13] Sergey Ioffe and Christian Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *International conference on machine learning*. pmlr. 2015, pp. 448–456.

[14] Benoit 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.

[15] Hanme Kim, Stefan Leutenegger, and Andrew J Davison. “Real-time 3D reconstruction and 6-DoF tracking with an event camera”. In: *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part VI 14*. Springer. 2016, pp. 349–364.

[16] Thomas N. Kipf and Max Welling. “Semi-Supervised Classification with Graph Convolutional Networks”. In: *CoRR* abs/1609.02907 (2016). arXiv: `1609.02907`. URL: `http://arxiv.org/abs/1609.02907`.

[17] Raghuraman Krishnamoorthi. “Quantizing deep convolutional networks for efficient inference: A whitepaper”. In: *CoRR* abs/1806.08342 (2018). arXiv: `1806.08342`. URL: `http://arxiv.org/abs/1806.08342`.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25 (2012).

[19] Yijin 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.

[20] Patrick Lichtsteiner, Christoph Posch, and Tobi Delbruck. “A 128$\times$ 128 120 dB 15 μs Latency Asynchronous Temporal Contrast Vision Sensor”. In: *IEEE Journal of Solid-State Circuits* 43.2 (2008), pp. 566–576. DOI: `10.1109/JSSC.2007.914337`.

[21] Min Liu and Tobi Delbruck. “Adaptive time-slice block-matching optical flow algorithm for dynamic vision sensors”. In: BMVC. 2018.

[22] Misha A. Mahowald and Carver Mead. “The silicon retina”. In: *Scientific American* 264 5 (1991), pp. 76–82.

[23] Anton Mitrokhin et al. “Learning visual motion segmentation using event surfaces”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 14414–14423.

[24] Anh 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 Workshops*. 2019, pp. 0–0.

[25] Garrick Orchard et al. “Converting static image datasets to spiking neuromorphic datasets using saccades”. In: *Frontiers in neuroscience* 9 (2015), p. 437.

[26] Charles Ruizhongtai Qi et al. “PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation”. In: *CoRR* abs/1612.00593 (2016). arXiv: `1612.00593`. URL: `http://arxiv.org/abs/1612.00593`.

[27] Joseph Redmon et al. “You Only Look Once: Unified, Real-Time Object Detection”. In: *CoRR* abs/1506.02640 (2015). arXiv: `1506.02640`. URL: `http://arxiv.org/abs/1506.02640`.

[28] Frank Rosenblatt. “The perceptron: a probabilistic model for information storage and organization in the brain.” In: *Psychological review* 65.6 (1958), p. 386.

[29] Rishov Sarkar et al. “FlowGNN: A Dataflow Architecture for Real-Time Workload-Agnostic Graph Neural Network Inference”. In: *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2023, pp. 1099–1112.

[30] Simon Schaefer, Daniel Gehrig, and Davide Scaramuzza. “AEGNN: Asynchronous Event-Based Graph Neural Networks”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2022, pp. 12371–12381.

[31] Simon Schaefer, Daniel Gehrig, and Davide Scaramuzza. *AEGNN: Asynchronous Event-based Graph Neural Networks (Demo Video)*. 2022. URL: `https://www.youtube.com/watch?v=opbFE6OsAeA`.

[32] Amos Sironi et al. “HATS: Histograms of Averaged Time Surfaces for Robust Event-based Object Classification”. In: *CoRR* abs/1803.07913 (2018). arXiv: `1803.07913`. URL: `http://arxiv.org/abs/1803.07913`.

[33] Amr Suleiman et al. “Navion: A Fully Integrated Energy-Efficient Visual-Inertial Odometry Accelerator for Autonomous Navigation of Nano Drones”. In: *2018 IEEE Symposium on VLSI Circuits*. 2018, pp. 133–134. DOI: `10.1109/VLSIC.2018.8502279`.

[34] Stepan 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.

[35] Zonghan Wu et al. “A Comprehensive Survey on Graph Neural Networks”. In: *IEEE Transactions on Neural Networks and Learning Systems* 32.1 (2021), pp. 4–24. DOI: `10.1109/TNNLS.2020.2978386`.

[36] Xilinx. *Kria K26 SOM Data Sheet (DS987)*. 2023. URL: `https://docs.xilinx.com/r/en-US/ds987-k26-som/Overview`.

[37] Chengxi Ye et al. “Unsupervised learning of dense optical flow, depth and egomotion from sparse event data”. In: *arXiv preprint arXiv:1809.08625* (2018).

[38] Rex Ying et al. “Hierarchical Graph Representation Learning with Differentiable Pooling”. In: *CoRR* abs/1806.08804 (2018). arXiv: `1806.08804`. URL: `http://arxiv.org/abs/1806.08804`.

[39] Zhe Zhou et al. “Blockgnn: Towards efficient gnn acceleration using block-circulant weight matrices”. In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2021, pp. 1009–1014.

[40] Alex Zihao Zhu et al. “EV-FlowNet: Self-supervised optical flow estimation for event-based cameras”. In: *arXiv preprint arXiv:1802.06898* (2018).

A
Asynchronous Layers in HUGNet

The 1-hop locality event-driven GNNs, such as HUGNet [5], furthermore allow for executing features pooling layers and FC layers in an asynchronous, event-driven manner, which is necessary to build a fully event-driven GNN.

A.1. Asynchronous Event-driven Node Feature Pooling Layer

Since only the features of the new event are changed in HUGNet, asynchronizing grid-based feature pooling layers is straightforward, which is described in Eq. A.1:

$$\mathbf{x}'_{grid} = Pool(\mathbf{x}_{new}, \mathbf{x}_{grid}), \tag{A.1}$$

where the $\mathbf{x}_{new}$ represents the feature vector of the new event generated from the last graph convolution layer, the $grid$ represents the grid index, thus the $\mathbf{x}_{grid}$ is the previous grid-based feature pooling outputs, and $Pool()$ represents one type of pooling function, *e.g.* max pooling.

A.2. Asynchronous FC Layer

The conventional FC layer is a linear transformation $\mathbf{y} = \Theta \times \mathbf{x}$, as introduced in Section 2.2.1. However, each time $\mathbf{x}$ changes, the FC layer has to compute again to update $\mathbf{y}$. This progress is computationally intensive when vector $\mathbf{x}$ comprises numerous elements, which is usually the case if $\mathbf{x}$ is derived from a flattening layer. Therefore, an asynchronous FC layer is proposed to release the computational burden.

The asynchronous FC layer is based on the following equation:

$$\mathbf{y}' = \mathbf{y} + d\mathbf{y} = \Theta \times \mathbf{X} + \Theta_{partial} \times d\mathbf{X}, \tag{A.2}$$

where

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1 \\ \cdots \\ \mathbf{x}_{grid} \\ \cdots \end{bmatrix}, \; d\mathbf{X} = \begin{bmatrix} \mathbf{x}_1 \\ \cdots \\ \mathbf{x}'_{grid} \\ \cdots \end{bmatrix} - \mathbf{X} = \mathbf{x}'_{grid} - \mathbf{x}_{grid}, \tag{A.3}$$

and

$$
\begin{aligned}
\Theta &= \begin{bmatrix} \theta_{1,1} & \cdots & \theta_{1,grid_{start}} & \cdots & \theta_{1,grid_{end}} & \cdots \\ \vdots & & \ddots & & & \vdots \\ \theta_{out,1} & \cdots & \theta_{out,grid_{start}} & \cdots & \theta_{out,grid_{end}} & \cdots \end{bmatrix}, \\
\Theta_{partial} &= \begin{bmatrix} \theta_{1,grid_{start}} & \cdots & \theta_{1,grid_{end}} \\ \vdots & \ddots & \vdots \\ \theta_{out,grid_{start}} & \cdots & \theta_{out,grid_{end}} \end{bmatrix},
\end{aligned}
\tag{A.4}
$$

where $grid_{start} = (grid - 1) \cdot C_{out} + 1$ and $grid_{end} = grid \cdot C_{out}$.

The C_{out} is the number of elements of the $\mathbf{x}_{grid}$, while the out represents the output feature number of this FC layer. The capitalized $\mathbf{X}$ indicates that it is derived by flattening the grid features $\mathbf{x_{grid}}$. When the upstream $\mathbf{x}_{grid}$ changes, it will also change a part of $\mathbf{X}$ and generate a narrow vector called differentiated grid features $d\mathbf{X}$. Then multiplying $d\mathbf{X}$ with corresponding partial weights, an output increment $d\mathbf{y}$ can be produced and further added into the older $\mathbf{y}$ to deliver the new prediction $\mathbf{y}'$. In this workflow, only partial data is processed in the calculation, thus saving computational time.