PAPER

Performance and Power-Efficiency Improvements on Graph Embedding Using Sequential Training Algorithm and FPGA

Kazuki SUNAGA[†a)], **Keisuke SUGIURA**[††b)], ***Nonmembers***, ***and*** **Hiroki MATSUTANI**[†c)], ***Member***

SUMMARY Recently, graph structures have been utilized in IoT (Internet of Things) environments such as network anomaly detection, smart transportation, and smart grids. A graph embedding is a representation of a graph as a fixed-length, low-dimensional vector, which can concisely represent the characteristics of the graph. node2vec is one of the well-known algorithms for obtaining such a graph embedding by sampling neighboring nodes on a given graph using a random walk technique. However, the original node2vec algorithm relies on a conventional batch training using backpropagation algorithm. In other words, we have to retain the training data to retrain the model, which makes it unsuitable for real-world applications where the graph structure changes after the deployment. To address the changes of graph structures after the IoT devices are deployed in edge environments, this paper proposes a combination of an online sequential training algorithm and node2vec. The proposed model is implemented on an FPGA (Field-Programmable Gate Array) device for efficient sequential training. The proposed FPGA implementation achieves up to a 205.25 times speed improvement compared to the original model on an ARM Cortex-A53 CPU. We also evaluate the performance of the proposed model in the sequential training task from various perspectives. For example, evaluation results on dynamic graphs show that while the accuracy decreases in the original model, the proposed sequential model can obtain better graph embedding that achieves a higher accuracy even when the graph structure changes. In addition, the proposed FPGA implementation is evaluated in terms of the power consumption, and the results show that it significantly improves the power efficiency compared to the CPU and embedded GPU implementations.

key words: *graph embedding, FPGA, OS-ELM, IoT*

1. Introduction

Graph structures, in which nodes are connected by edges, can be found everywhere in our lives. For example, friendships in social networking services, user-item relationships on e-commerce sites, and citation networks in academic papers can all be represented as graph structures. Thus, applications that extract, analyze, and utilize information from graph structures are in high demand. Although a graph structure can be represented by an adjacency matrix, it is unsuitable for direct use in statistical or machine learning-based methods especially for large and sparse graphs. To address this issue, a graph embedding has emerged as a representation so that it can be directly used with statistical or machine learning methods.

Using the graph embedding, graph structures can be represented with fixed-length, low-dimensional vectors. node2vec [1] is a well-known algorithm for obtaining the graph embedding by sampling neighboring node information on a given graph using a random walk technique. However, the original node2vec algorithm typically relies on batch training, not online sequential training; thus, it is not suited for applications where the graph structure changes after the deployment. In this paper, we assume node2vec applications for IoT environments. To address the changes of graph structures after the IoT devices are deployed in edge environments, we propose to combine an online sequential training algorithm with node2vec. Since low-cost and low-power execution is essential for such IoT applications, the proposed sequential model is implemented on an FPGA device to significantly reduce the training time in the deployed environment. We also compare our FPGA implementation with an embedded GPU implementation. The major contributions of this paper are summarized as follows*:

1. To train the graph embedding sequentially, we combine node2vec with an online sequential training algorithm and then accelerate it using FPGA.
2. To combine node2vec with the online sequential algorithm and lighten the model for FPGA implementation, we utilize scale-multiplied output-side weights for the input-side weights in our proposed model.
3. To further accelerate the FPGA implementation, we modify the original sequential training algorithm to be complied with a dataflow optimization.
4. To demonstrate benefits of the FPGA implementation, it is compared with the CPU and embedded GPU implementations in terms of the performance and power efficiency.

The rest of this paper is organized as follows. Section 2 introduces related works. Section 3 proposes a sequentially-trainable graph embedding model and its FPGA-based accelerator. It also illustrates the FPGA implementation. Section 4 evaluates the model and the accelerator in terms of

Manuscript received December 12, 2024.
Manuscript publicized April 4, 2025.

[†]Graduate School of Science and Technology, Keio University, Yokohama-shi, 223–8522 Japan.
[††]Institute of Systems and Information Engineering, University of Tsukuba, Tsukuba-shi, 305–8573 Japan.
a) E-mail: sunaga@arc.ics.keio.ac.jp
b) E-mail: sugiura@lila.cs.tsukuba.ac.jp
c) E-mail: matutani@arc.ics.keio.ac.jp

DOI: 10.1587/transinf.2024EDP7319

*This paper is an extended version of our previous work presented at an international workshop [2]. This edition includes more explanations about the hardware implementation and compares the FPGA implementation with the CPU and embedded GPU implementations in terms of the performance and power efficiency. We also analyze the execution time breakdown of our FPGA implementation and the impacts on update frequency of the sampling table.

the execution time, accuracy, model size, FPGA resource utilization, and power consumption. Section 5 summarizes our contributions.

2. Related Work

2.1 node2vec Algorithm

node2vec [1] is a well-known algorithm for obtaining a graph embedding. As shown in Fig. 1, a random walk is performed from a selected start node (e.g., node-t) in the graph in order to collect neighboring nodes information. Assuming that a transition has been performed from node-t to node-u, Fig. 1 illustrates the probabilities of the next transition from node-u to one of its adjacent nodes. The transition probabilities to the adjacent nodes are determined as follows.

$$P(c_i = x | c_{i-1} = u) = \begin{cases} \dfrac{\alpha_{pq}(t,x) w_{ux}}{Z} & \text{if} \quad ((u,x) \in E) \\ 0, & \text{if} \quad ((u,x) \notin E) \end{cases} \tag{1}$$

where c_i represents the i-th node during a random walk. (u, x) represents an edge between node-u and node-x, and w_{ux} represents the weight of this edge. E represents all the edges in the graph. Z is a normalizing constant. α is defined using parameters p and q as follows.

$$\alpha_{pq}(t,x) = \begin{cases} 1 \,/\, p & \text{if} \quad d_{tx} = 0 \\ 1 & \text{if} \quad d_{tx} = 1 \\ 1 \,/\, q, & \text{if} \quad d_{tx} = 2 \end{cases} \tag{2}$$

where d_{tx} represents a distance between the previous node (i.e., node-t) and the next node (i.e., node-x). That is, d_{tx} is set to 0 when transitioning back to node-t; d_{tx} is set to 1 when transitioning to an adjacent node of node-t; otherwise d_{tx} is set to 2. Let RW be the result of a single random walk. The training samples can be efficiently generated by partitioning RW with a given context size (i.e., window size). In Fig. 1, $N_S(u)$ is an example of the training samples. $N_S(u)$ represents neighboring nodes of node-u obtained by a random walk started from node-u based on a random walk strategy S. These training samples are trained using the skip-gram model [3] illustrated in Fig. 2 (a).

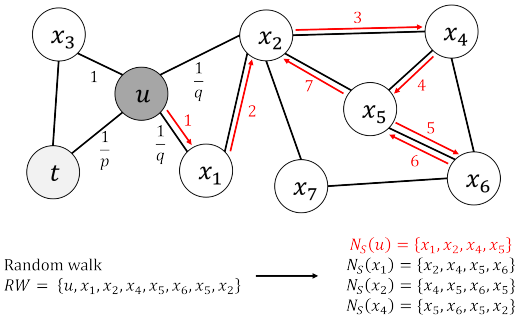
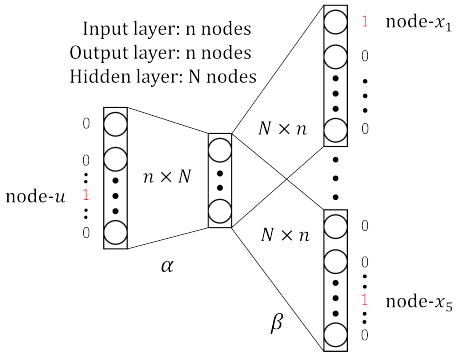


Fig. 1 An example of random walk in node2vec.

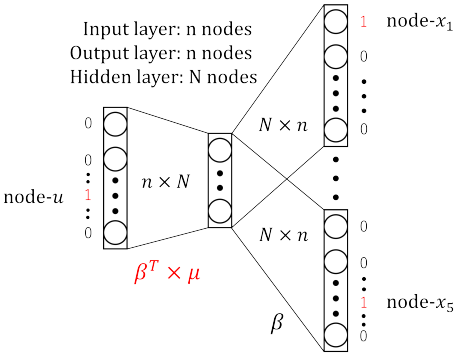
In the skip-gram model, the numbers of dimensions in the input-layer and output-layer are the same as the number of nodes in the graph. The number of hidden-layer dimensions corresponds to the number of the graph embedding dimensions to be trained. An input data to the skip-gram model is a one-hot vector, where only the element corresponding to node-u is set to 1, and all other elements are set to 0. An output of the model is a vector, where each element represents the probability that the corresponding node appears as an adjacent node of node-u. From $N_S(u)$ we can obtain four different teacher labels, each of which is a one-hot vector, where one of the nodes x_1, x_2, x_4, and x_5 is set to 1 and all other nodes are set to 0. Since there are four different teacher labels, the final loss value is calculated by summing the four individual loss values, each computed using the same weights but different one-hot teacher labels.

2.2 node2vec for Dynamic Network

Dealing with dynamic graphs is one of the important challenges in graph-based learning, and there are many approaches to learn graph embedding for dynamic graphs. Continuous-Time Dynamic Network (CTDN) [4] addresses dynamic graphs by incorporating temporal information to the graph and imposing temporal constraints during node2vec random walks. Specifically, CTDN prohibits a movement



(a) Original skip-gram model (input-side weights are fixed at random values)



(b) Proposed model (input-side weights are a constant multiple of β)

Fig. 2 Original skip-gram model and proposed model.

in the time-decreasing direction during the random walk, thereby preserving a temporal consistency. In [5], graph snapshots are used as temporal information and combined with node2vec for a link prediction. Specifically, a link prediction at time t utilizes the temporal information up to time $t-1$. In dynnode2vec [6], a graph embedding at time t is trained using the graph embedding at time $t-1$ as initial values. Although there are many prior works that learn graph embedding on such dynamic graphs, most of them rely on a skip-gram based model and a conventional batch training with backpropagation algorithm. dynnode2vec also learns graph embedding sequentially using this approach. However, in general, a sequential training using the conventional backpropagation algorithm can result in the loss of previous learning results. This phenomenon is known as a catastrophic forgetting, which reduces the accuracy. In this paper, we address this issue by utilizing an online sequential training algorithm for graph embedding on dynamic graphs. The details are described in the next section.

2.3 Sequential Training Algorithm

OS-ELM (Online Sequential Extreme Learning Machine) [7] is an online sequential training algorithm for neural networks with a single hidden layer. Figure 3 illustrates the network structure and its training algorithm. The input-layer, hidden-layer, and output-layer dimensions are n, N, and m, respectively. In the OS-ELM algorithm, the input-side weights $\boldsymbol{\alpha} \in \mathbb{R}^{n \times N}$ are fixed at random values at the initialization time, and only the output-side weights $\boldsymbol{\beta} \in \mathbb{R}^{N \times m}$ are trainable and sequentially updated. Assuming that the i-th input data is fed to the neural network, hidden-layer outputs $\boldsymbol{H_i} \in \mathbb{R}^{N}$ are generated. Then, new output-side weights $\boldsymbol{\beta_i}$ are calculated based on previous weights $\boldsymbol{\beta_{i-1}}$ and temporary values $\boldsymbol{P_i} \in \mathbb{R}^{N \times N}$, which are also calculated based on previous values $\boldsymbol{P_{i-1}}$ and $\boldsymbol{H_i}$. As shown in the equations in Fig. 3, the OS-ELM algorithm derives an optimal $\boldsymbol{\beta}$ that can minimize a loss between final outputs $\boldsymbol{y_i} \in \mathbb{R}^{m}$ and teacher labels $\boldsymbol{t_i} \in \mathbb{R}^{m}$ analytically, where $\boldsymbol{y_i} = G(\boldsymbol{x_i}\boldsymbol{\alpha} + \boldsymbol{b})\boldsymbol{\beta}$. This sequential training is simple and fast since the sequential training is done in a single epoch, which is different from a conventional backpropagation based training. An FPGA-based acceleration of OS-ELM is reported [8].

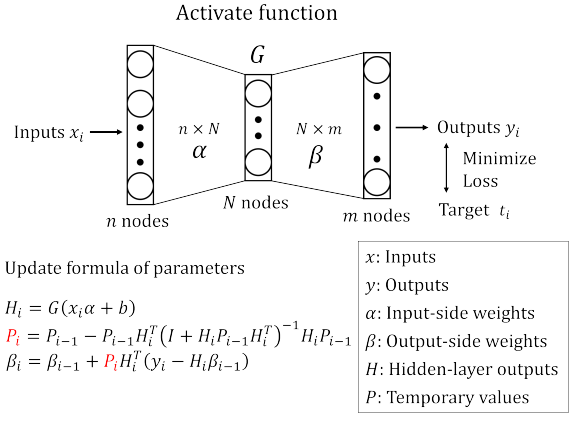


Fig. 3 OS-ELM algorithm.

2.4 Graph Embedding for IoT Environments

Although the graph-based learning has been actively studied in recent years, how to utilize graph embedding and graph neural networks (GNNs) for IoT applications remains an emerging research topic. For example, there are many prior works that utilize graph structures in IoT environments such as network anomaly detection, malware detection, smart transportation, and smart grids [9], [10]. It has been reported that using graph structures can achieve favorable results in these applications. On the other hand, real-world applications typically require a low latency while training requires large amounts of data; thus, acceleration methods of the training have been studied so far. These acceleration methods include algorithmic approaches and hardware-based approaches, and they are classified in [11]. In this paper, we utilize node2vec since several prior works that utilize node2vec for IoT tasks have been reported. For example, node2vec is used to detect malware in IoT environments [12] and to predict usage patterns in IoT environments [13].

2.5 node2vec Accelerator

Hardware accelerations for graph learning methods have been studied so far. In this section, we introduce those related to node2vec. For example, an FPGA-based acceleration for random walk in node2vec has been reported [14]. Additionally, an FPGA-based acceleration of word2vec [3] that uses the skip-gram model similar to node2vec has been reported [15]. Please note that in [14], the random walk process is accelerated, while in this paper we accelerate the training algorithm of node2vec. Although the training algorithm of word2vec is accelerated using FPGA in [15], in this paper we propose a new approach for sequential training of node2vec on dynamic graphs by combining OS-ELM with the skip-gram model. This enables an efficient on-device training on FPGA devices.

3. Sequential Graph Embedding Accelerator

3.1 Sequentially-Trainable Skip-Gram Model

Figure 2 (b) illustrates the proposed OS-ELM based training model for graph embedding, and Algorithm 1 describes the training algorithm. Since both the skip-gram and OS-ELM assume neural networks with a single hidden layer, the OS-ELM algorithm can be theoretically applied to the skip-gram. In the original OS-ELM, $\boldsymbol{H_i}$ in Algorithm 1 are calculated using $\boldsymbol{x_i} \in \mathbb{R}^{n}$; specifically, $\boldsymbol{H_i} = G(\boldsymbol{x_i}\boldsymbol{\alpha} + \boldsymbol{b})$. Since the input vector $\boldsymbol{x_i}$ is one-hot, $\boldsymbol{H_i}$ can be calculated as the row vector corresponding to the center node which is extracted from $\boldsymbol{\alpha}$ assuming that $\boldsymbol{b}$ is zero.

In the skip-gram model, a desired graph embedding can

Algorithm 1 Proposed algorithm

1: **for** each context **do**
2: $\quad \boldsymbol{H_i} \leftarrow \boldsymbol{\beta}$[center node] $\times \mu$
3: $\quad$ Compute $\boldsymbol{P_{i-1}H_i^T}$ and $\boldsymbol{H_iP_{i-1}}$
4: $\quad$ Compute $\boldsymbol{P_{i-1}H_i^TH_iP_{i-1}}$ and $\boldsymbol{H_iP_{i-1}H_i^T}$
5: $\quad hpht\_inv \leftarrow \frac{1}{\boldsymbol{H_iP_{i-1}H_i^T}}$
6: $\quad \boldsymbol{P_i} \leftarrow \boldsymbol{P_{i-1}} - \boldsymbol{P_{i-1}H_i^TH_iP_{i-1}} \times hpht\_inv$
7: $\quad$ Compute $\boldsymbol{P_iH_i^T}$
8: $\quad$ **for** each window **do**
9: $\quad\quad$ **for** $itr = 1$ to $ns + 1$ **do**
10: $\quad\quad\quad$ **if** $itr = 1$ **then**
11: $\quad\quad\quad\quad sample \leftarrow$ positive sample
12: $\quad\quad\quad$ **else**
13: $\quad\quad\quad\quad sample \leftarrow$ negative sample
14: $\quad\quad\quad$ Compute $\boldsymbol{t_i} - \boldsymbol{H_i\beta_{i-1}}[sample]$
15: $\quad\quad$ **end for**
16: $\quad$ **end for**
17: $\quad \boldsymbol{\beta_i} \leftarrow \boldsymbol{\beta_{i-1}} + \boldsymbol{P_iH_i^T}(\boldsymbol{y_i} - \boldsymbol{H_i\beta_{i-1}})$
18: **end for**

be obtained from weights of the neural network. Specifically, the following weights can be used for the graph embedding: 1) the input-side weights $\boldsymbol{\alpha}$, 2) the output-side weights $\boldsymbol{\beta}$, and 3) the average of $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$. Among them, the input-side weights are typically used for graph embedding. However, since the input-side weights of the original OS-ELM are statically fixed at random values, we cannot directly use these weights for the graph embedding in the proposed model. Although the original skip-gram model uses the input-side weights for the graph embedding, in the proposed model, we utilize the trainable weights of OS-ELM (i.e., $\boldsymbol{\beta}$) to build the input-side weights as described in [16]. Please note that although this technique is not suited for word2vec [16], it can be applied to node2vec algorithm. Assume an activation function of the first layer is a linear function without bias vectors. When we utilize $\boldsymbol{\beta}^T \in \mathbb{R}^{n \times N}$ as the input-side weights[†], the output probabilities are simply obtained by $O(\boldsymbol{x_i\beta}^T\boldsymbol{\beta})$, where O is an activation function of the last layer such as sigmoid function. In this case, since $\boldsymbol{x_i}$ is a one-hot vector where only a given center node is 1 and the others are 0, the output probability of the center node tends to be high. This is not suited for word2vec, because in the case of word2vec, probabilities that the center word appears as its neighboring words should be low; for example, when "dog" is the center word, "dog" rarely appears as neighboring words of the center word. In the case of node2vec, on the other hand, because of the nature of random walks described in Sect. 2.1, the same node often appears as its neighboring nodes.

In Fig. 2 (b), μ is a scale factor to transform $\boldsymbol{\beta}$ into the input-side weights. In this case, the input-side weights become a constant multiple of $\boldsymbol{\beta}$; thus, the hidden-layer outputs $\boldsymbol{H_i}$ also become a constant multiple of the column vector corresponding to the center node which is extracted from $\boldsymbol{\beta}$. This eliminates the original random weights $\boldsymbol{\alpha}$ from OS-ELM, so we can reduce the model size and memory utilization.

[†]In the skip-gram model we can assume $n = m$.

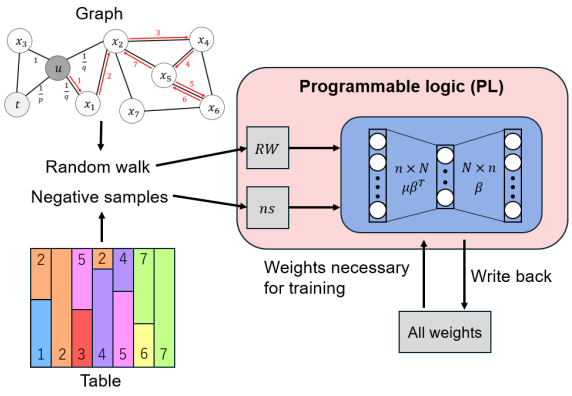


Fig. 4 Overview of our proposal.

The proposed model adopts the negative sampling method [17]. In this case, only a fraction of samples from negative nodes of teacher labels (i.e., nodes with a value of 0 in the one-hot vector) is trained instead of training all the negative samples. This can significantly reduce the training time by limiting the number of nodes to update, even if the number of nodes in the graph is huge. In general, 5 to 20 negative samples are sufficient for small datasets, while 2 to 5 negative samples are enough for large datasets [17]. In Algorithm 1, the innermost loop starting from line 9 corresponds to the negative sampling. In this loop, ns denotes the number of negative samples to be trained. The outermost loop starting from line 1 processes RW obtained from a random walk of node2vec. In the training phase, as described in Sect. 2.1, RW is partitioned into samples (e.g., $N_S(u)$) by a given window size. In the case of $N_S(u)$, for example, node-u is the center node, and nodes included in $N_S(u)$ are trained as positive nodes. Only a fraction of negative nodes is sampled randomly by the negative sampling method. The sampled frequency as negative nodes depends on the number of appearances of each node in the entire RW. This sampling is done by the Walker's alias [18], which is a weighted sampling method. In this case, although the time complexity to build a table used in the sampling is proportional to the number of nodes, the time complexity of the sampling is $O(1)$. In Algorithm 1, lines 2 to 7 and lines 14 to 15 describe the training algorithm of OS-ELM.

Figure 4 illustrates an overview of our proposal. The training data are sampled by random walks on the "Graph" in Fig. 4. "Table" represents the table for weighted sampling used in Walker's alias, which is used to sample negative samples. These data are transferred to the programmable logic part, and the weights necessary for training are read. The updated weights are written back into DRAM to enable the sequential training of the graph embedding.

3.2 FPGA Implementation

In this section, we describe an FPGA implementation (including board and IP core levels) of the proposed model. We use Xilinx Zynq MPSoC series as a target FPGA platform.

Figure 5 illustrates a block diagram of the board-level implementation, which is divided into a processing system (PS) part and a programmable logic (PL) part. Figure 6 illustrates a block diagram of the implementation of the IP core. Our sequentially-trainable node2vec accelerator is implemented in the PL part of the FPGA, which is denoted as "Core" in Fig. 5. This implementation of Core has two AXI interfaces. One interface is an AXI4-Lite with a 32-bit data bus connected to the High-Performance Master (HPM0) port, which allows the PS part to access control registers. The other interface is an AXI4 with a 128-bit data bus connected to the High-Performance Coherent (HPC0) port. In Fig. 6, Parameter Buffer manages the model parameters such as $\boldsymbol{\beta}$ and Input Buffer manages the training data, i.e., RW, which is the result of a random walk, and ns, which is the result of negative sampling. All these parameters are implemented using BRAMs.

As graphs become larger and the dimensions of graph embedding to be learned increase, it becomes challenging to implement all the weights on resource-limited FPGA devices. In the proposed model, since only a fraction of weights is updated by each training data by the negative sampling method, only weights necessary for training are implemented on BRAM cells of the PL part.

The training is executed as follows. First, nodes are sampled from a graph using random walk by a host CPU in the PS part. The result of a single random walk and negative samples necessary for the training are pre-sampled by the CPU. Then, the following five steps are executed (the steps correspond to the numbers in Fig. 6). (1) The mode of operation (e.g., read training data, learn graph embedding, etc.) is set by the control registers and the IP core is started from the CPU. (2) These training samples are transferred from a DRAM to the Input Buffer. (3) After transferring the training data and negative samples, weights necessary for the training (e.g., $\boldsymbol{\beta}$) are transferred from the DRAM to the Parameter Buffer. (4) Then the model is sequentially trained in the PL part so that the weights are updated using these data. The result (a flag if the training is successful or not) is written to the control register. (5) Finally, the trained weights are written back to the DRAM. By repeating this procedure, the graph embedding can be trained. In our implementation, the same negative samples are used for multiple sets of training data as described in [19] to reduce the data transfer size between DRAM and BRAM; in this case, training samples obtained by a single random walk are trained using the same negative samples.

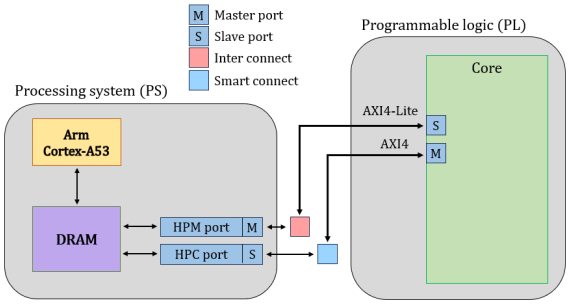


Fig. 5 Block diagram of board-level implementation.

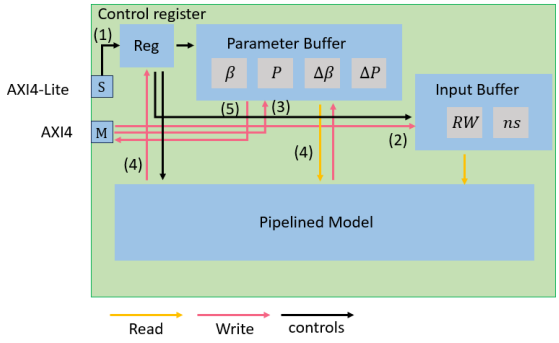


Fig. 6 Block diagram of IP core (numbers in this figure indicate steps for graph embedding).

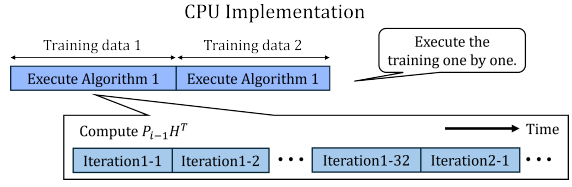
To further speedup, a dataflow optimization is applied by modifying the update procedure of $\boldsymbol{\beta}$ in Algorithm 1. Algorithm 2 shows the modified procedure. In Algorithm 1, $\boldsymbol{P_i}$ and $\boldsymbol{\beta_i}$ are updated sequentially in each iteration of the outermost loop starting from line 1. Since there is a dependency between two successive iterations, a dataflow optimization cannot be applied in our original algorithm. In Algorithm 2, on the other hand, $\boldsymbol{P}$ and $\boldsymbol{\beta}$ are updated outside the outermost loop (lines 19 and 20), and only their accumulated differences (i.e., $\boldsymbol{\Delta P}$ and $\boldsymbol{\Delta\beta}$) are updated sequentially inside the loop. This modification enables the dataflow optimization. Please note that the proposed model is trained with the same output-side weights $\boldsymbol{\beta}$ and the same intermediate data $\boldsymbol{P}$ for the result of a single random walk. It is expected that the proposed model can maintain an accuracy if the number of training data is sufficient, which will be evaluated in

Algorithm 2 Modified algorithm for dataflow optimization

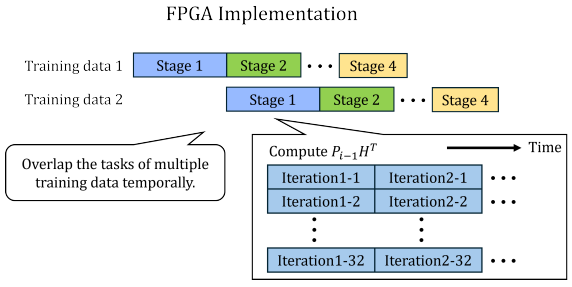
1: **for** each context **do**
2: $Stage1$:
3: $\boldsymbol{H_i} \leftarrow \boldsymbol{\beta}$[center note] $\times\, \mu$
4: Compute $\boldsymbol{P_{i-1}H_i^T}$ and $\boldsymbol{H_iP_{i-1}}$
5: $Stage2$:
6: Compute $\boldsymbol{P_{i-1}H_i^TH_iP_{i-1}}$ and $\boldsymbol{H_iP_{i-1}H_i^T}$
7: $Stage3$:
8: **for** each window **do**
9: **for** $itr = 1$ to $ns + 1$ **do**
10: **if** $itr = 1$ **then**
11: $sample \leftarrow$ positive sample
12: **else**
13: $sample \leftarrow$ negative sample
14: Compute $\boldsymbol{t_i - H_i\beta_{i-1}}[sample]$
15: **end for**
16: **end for**
17: $Stage4$:
18: $hpht\_inv \leftarrow \frac{1}{\boldsymbol{H_iP_{i-1}H_i^T}}$
19: $\boldsymbol{\Delta P} \leftarrow \boldsymbol{\Delta P - P_{i-1}H_i^TH_iP_{i-1}} \times hpht\_inv$
20: $\boldsymbol{\Delta\beta} \leftarrow \boldsymbol{\Delta\beta + P_iH_i^T(y_i - H_i\beta_{i-1})}$
21: **end for**
22: $\boldsymbol{P_i} \leftarrow \boldsymbol{P_{i-1} + \Delta P}$
23: $\boldsymbol{\beta_i} \leftarrow \boldsymbol{\beta_{i-1} + \Delta\beta}$

Table 1 Evaluation environments (desktop CPU, embedded GPU, and FPGA).

	Workstation	Nvidia Jetson Xavier NX	Xilinx ZCU104 Evaluation Kit
CPU	Intel Core i7-11700 @2.5GHz	Nvidia Carmel ARM v8.2 @1.4GHz	ARM Cortex-A53 @1.2GHz
GPU	–	384-core Nvidia Volta	–
FPGA	–	–	XCZU7EV-2FFVC1156
DRAM	32GB (DDR4)	8GB (DDR4)	2GB (DDR4)
OS	Ubuntu 20.04.6	Nvidia JetPack 5.1 (based on Ubuntu 20.04)	Pynq Linux v2.7 (based on Ubuntu 20.04)



(a) CPU implementation based on Algorithm 1



(b) FPGA implementation based on Algorithm 2

Fig. 7 CPU and FPGA implementations.

Sect. 4.

In our experiments, the CPU implementation is based on Algorithm 1 and the FPGA implementation is based on Algorithm 2. In addition to the dataflow optimization, loop unrolling and loop pipelining directives are used in each stage in Algorithm 2. This parallelism and dataflow optimization are key factors to accelerate the training in our FPGA implementation. Figure 7 (a) shows the training procedure of the CPU implementation. In the CPU implementation, each piece of training data is used for the training one by one based on Algorithm 1. As an example, the calculation of $\boldsymbol{P_{i-1}H_i^T}$ in Algorithm 1 is illustrated in this figure. This computation is executed in a double loop over the number of hidden-layer dimensions and is computed in order in the CPU implementation. Figure 7 (b) shows the training procedure of our FPGA implementation. In our FPGA implementation, multiple training data are temporally overlapped and executed simultaneously by the dataflow optimization. Furthermore, the calculation of $\boldsymbol{P_{i-1}H_i^T}$ is also executed simultaneously by employing the loop unrolling and loop pipelining directives.

4. Evaluations

The proposed accelerator is implemented with Xilinx Vivado v2022.1 and Xilinx Vitis HLS v2022.1. We choose Xilinx Zynq UltraScale+ MPSoC series as a target FPGA platform; specifically, ZCU104 evaluation board (XCZU7EV-2FFVC1156) is used in this paper. In the performance evaluation, our FPGA implementation is compared with an embedded CPU of the FPGA board (ARM Cortex-A53@1.2 GHz), a desktop computer (Intel Core i7-11700@2.5 GHz), and an embedded GPU (Nvidia Jetson Xavier NX). Their specifications and environments are summarized in Table 1. The clock frequency of the PL part of the FPGA board is set to 200 MHz. As for software counterparts running on CPU, we use C/C++ to implement the models and compile them with gcc 9.4.0. For the GPU implementation, we use PyTorch 2.0.0+nv23.05 and CUDA v11.4. Matrix multiplications such as $\boldsymbol{P_{i-1}H_i^T}$ are computed by many processing cores. We use Pybind11 [20] for the GPU implementation to accelerate the random walk up to the same level as the other implementations.

Table 2 Three datasets used in evaluations.

Dataset	# nodes	# edges	# classes
Cora	2,708	5,429	7
Amazon Photo	7,650	143,663	8
Amazon Electronics Computers	13,752	287,209	10

4.1 Datasets

Table 2 lists three datasets used in our evaluations. We use Cora [21], Amazon Photo [22], and Amazon Electronics Computers [22]. Cora is a paper citation network in a machine learning research field. Each node represents a paper, and each edge represents a citation relationship. Amazon Photo and Amazon Electronics Computers are subsets of Amazon co-purchase graph dataset [23]. Each node represents a product, and each edge represents that the two products are frequently bought together.

4.2 Execution Time

Here, we evaluate the execution time of the proposed accelerator. The execution time is an elapsed time to train RW, which is obtained by a single random walk as mentioned in Sect. 3. In our evaluation, the length of random walk l and the window size w are set to 80 and 8, respectively. Thus, the training time of a single random walk is measured over 73 iterations of the outermost loop starting from line 1 in Algorithm 2. Please note that the execution time includes the time required for the negative sampling but excludes the time for random walks, which are not the target of acceleration in this paper. Table 3 summarizes the hyper-parameters

Table 3 Hyper-parameters of node2vec.

Parameter	Value	Description
p	0.5	Parameter to define $\alpha_{pq}(t,x)$
q	1.0	Parameter to define $\alpha_{pq}(t,x)$
r	10	Number of random walks per node
l	80	Length of single random walk
w	8	Window size
ns	10	Number of negative samples

Table 4 Training time of a single random walk (vs. Cortex-A53 CPU).

	# graph embedding dimensions		
	32	64	96
Original model on CPU (ms)	35.357	100.291	202.175
Proposed model on CPU (ms)	18.753	35.941	72.612
Proposed model on GPU (ms)	60.129	65.137	65.326
Proposed model on FPGA (ms)	0.777	0.878	0.985
Speedup (vs. Original model on CPU)	45.504	114.227	205.254
Speedup (vs. Proposed model on CPU)	24.135	40.935	73.718

of node2vec in this evaluation.

Table 4 shows the execution times of the proposed accelerator and software implementations on ARM Cortex-A53 CPU and Nvidia Jetson Xavier NX. As shown, 1.89 to 2.77 times speedup is achieved by replacing the original skip-gram model with our OS-ELM based sequential model (Algorithm 1). By implementing the proposed model on the FPGA, the proposed accelerator achieves 24.14 to 73.72 times speedup compared to that on ARM Cortex-A53 CPU. Compared to the CPU implementation of the original skip-gram model, our accelerator achieves 45.50 to 205.25 times speedup. On the other hand, GPU-based implementation only achieves 0.31 to 1.11 times speedup compared to that on ARM Cortex-A53 CPU. This is because the GPU-based implementation suffers from latency overheads due to the communication cost between a GPU and a CPU in addition to software overheads, especially when the number of graph embedding dimensions is small. In fact, when the number of dimension is 256, the GPU-based implementation achieves 7.23 times speedup compared to ARM Cortex-A53 CPU since the proportion of the model's computational load in the overall processing increases. In the CPU implementation, the training time increases almost linearly as the number of graph embedding dimensions increases since the computation is done serially. In the GPU implementation, the training time is nearly constant since the GPU implementation can sufficiently parallelize the computation even when the number of embedding dimensions is 96. In the FPGA implementation, although the training time is quite small compared to the other two implementations, it increases gradually as the number of embedding dimensions increases since the computation was not fully parallelized due to the resource limitations. In addition, Table 5 shows the execution times of the proposed accelerator and software implementations on Intel Core i7 11700 CPU. Even when compared to the desktop computer, our small FPGA implementation achieves 1.01 to 3.34 times speedup.

Table 6 shows the execution time breakdown in our FPGA implementation. In this evalutaion, the number of graph embedding dimensions is 32. The unit is microseconds in this table. From this table, the total execution time of the PL part required to train *RW* is 355 microseconds. This is approximately the same as the execution time required for tasks such as a single random walk or a negative samples generation in the PS part. In the PL part, the training task begins with reading training data and weights. The most computationally expensive task is the training of the graph embedding, which is described in Algorithm 2. Since each stage is simultaneously executed in a pipeline by the dataflow operation, we do not evaluate the execution time of each stage from the start to the end of the training for *RW*. As an alternative, Table 7 shows the number of cycles required for each stage, calculated using Vitis HLS v2022.1. From this table, stages 3 and 4 are relatively computationally expensive. This is because stage 3 involves an increased number of loops due to the negative sampling, and stage 4 requires a single division operation.

Table 5 Training time of a single random walk (vs. Core i7 11700 CPU).

	# graph embedding dimensions		
	32	64	96
Original model on CPU (ms)	1.309	2.293	3.285
Proposed model on CPU (ms)	0.787	1.426	2.396
Proposed model on FPGA (ms)	0.777	0.878	0.985
Speedup (vs. Original model on CPU)	1.687	2.612	3.335
Speedup (vs. Proposed model on CPU)	1.013	1.624	2.432

Table 6 Execution time breakdown on FPGA (µs).

	PS part	PL part
Single random walk	253.3	-
Generating negative samples	272.5	-
Transfer of training data	116.6	-
Reading of training data	-	3.9
Reading of weights ($\boldsymbol{\beta}$)	-	28.2
Training of the model	-	294.8
Writing of weights ($\boldsymbol{\beta}$)	-	28.1

Table 7 The number of cycles for each stage in Algorithm 2 using Vitis HLS v2022.1.

	Stage1	Stage2	Stage3	Stage4
Cycles	152	102	169	177

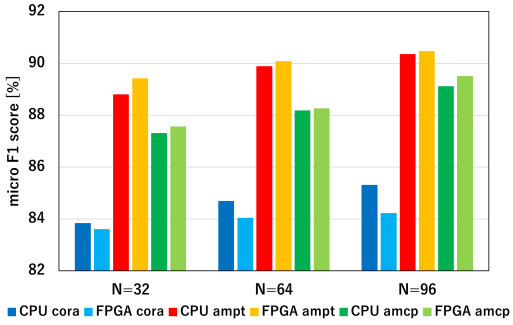
4.3 Accuracy

For the accuracy evaluation, our trained graph embedding should be tested with a machine learning task. Thus, in this evaluation, it is used for a one-vs-rest logistic regression. The F1 score by the logistic regression is used as an evaluation metric. The F1 score by the logistic regression is used as an evaluation metric of classification models as in [1]. It is denoted as the harmonic mean of the precision and recall. For multi-class classification tasks, micro F1 score is computed based on prediction results for all the classes. When the number of samples is biased depending on their classes, it is affected by majority classes. On the other hand, macro F1 score is computed by averaging F1 scores each of which is computed for each class. There are 4.54×, 5.86×,

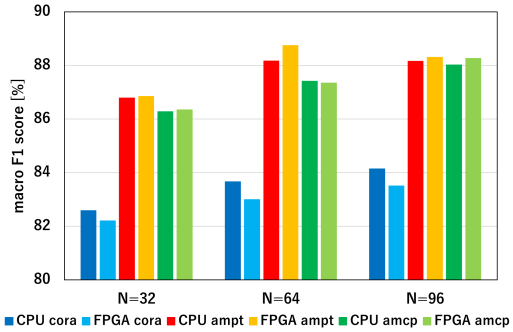
and 17.72× differences between the largest and smallest class samples in Cora, Amazon Photo, and Amazon Electronics Computers datasets. In this paper, we thus show both the micro and macro F1 score results. For the logistic regression, 90% of the data are used as training data, and 10% are used as test data for multiclass classification. SGD (Stochastic Gradient Descent) is used to train the original skip-gram model, and the learning rate is set to 0.01. In this evaluation, a graph embedding is trained three times. Then, an average F1 score over the three trials is reported as the evaluation result.

4.3.1 Impact of Dataflow Optimization

To evaluate the impact of dataflow optimization applied to our FPGA accelerator, the proposed algorithm (Algorithm 1) on CPU and the modified algorithm (Algorithm 2) on FPGA are compared in terms of the accuracy. The three datasets described in Sect. 4.1 are used for this evaluation. Figure 8 shows the evaluation results, where "ampt" and "amcp" represent Amazon Photo and Amazon Electronics Computers datasets, respectively. Both micro and macro F1 scores are reported. While the accuracy of the FPGA implementation decreases by up to 1.09% in Cora dataset, no accuracy degradation is observed in the other two datasets, which have a relatively large number of nodes. In our FPGA implementation, the number of weight updates is decreased due to the dataflow optimization, and this affects the accuracy of Cora, which is a relatively small graph.



(a) CPU vs. FPGA (micro F1 score)



(b) CPU vs. FPGA (macro F1 score)

Fig. 8 Impact of dataflow optimization on accuracy.

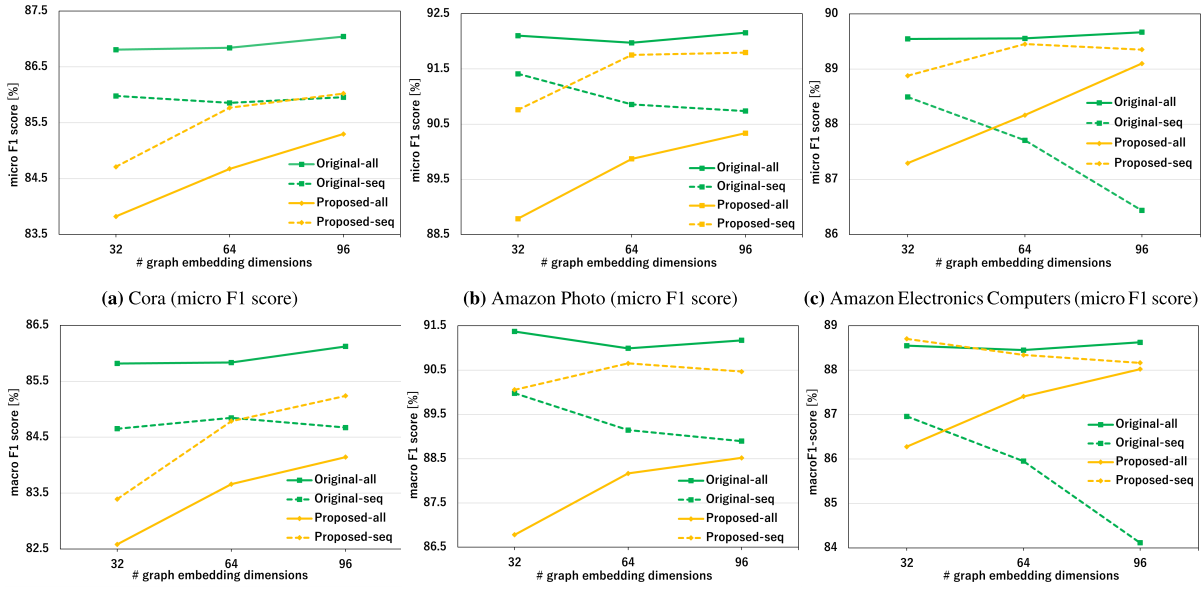
4.3.2 Impact of Sequential Training

Next, we evaluate the benefit of the sequential training, which is one of major contributions of this paper. Figure 9 shows the evaluation results, where "Original" represents the original skip-gram model and "Proposed" represents our proposed model (i.e., Algorithm 2). In addition, we examine two training scenarios: "all" and "seq". In the "all" case, an entire graph is trained assuming that all the edges exist from the beginning. In the "seq" case, only a fraction of edges is trained first; then, new edges are sequentially added to the graph, and a sequential training is executed every time a new edge is added. To build the initial graph of the "seq" case, we remove edges from an entire graph so that the initial graph becomes a forest without changing the number of connected components to the original entire graph. Subsequently, every time the removed edge is added, the random walk and training of node2vec are executed. In this case, the random walk starts from both the ends of an added edge.

As shown in Fig. 9, in the "all" case, the original skip-gram model achieves a higher accuracy compared with the proposed model for all the numbers of graph embedding dimensions (i.e., the numbers of hidden-layer nodes in the model) in all the datasets. In the "seq" case, on the other hand, the accuracy of our OS-ELM based sequentially-trainable model tends to be high compared to the original skip-gram model. In contrast, the accuracy of the original model drops when sequentially training the edges in the "seq" case. This implies that the sequential training using the backpropagation algorithm for the original model causes a catastrophic forgetting. This impact tends to be larger when the number of graph embedding dimensions increases and the graph becomes large. Although in this evaluation only a fraction of weights is updated by the negative sampling, the accuracy of the original model decreases due to the catastrophic forgetting. Please note that the proposed model in the "seq" case achieves a higher accuracy compared to the "all" case. Because in the "seq" case, a random walk and sequential training are executed every time a new edge is added, the number of training samples increases in the "seq" case; thus, the proposed sequential model successfully increases the accuracy. These results demonstrate that the graph embedding can be sequentially trained by using the proposed sequential model even if target graphs are large and dynamically updated. Overall, the macro F1 scores are approximately 1% lower than the micro F1 scores, which means that there are differences in classification accuracies depending on the classes. Specifically, the accuracy of small class samples tends to be low compared to those of large class samples.

4.3.3 Impact of Scale Factor μ

As proposed in Sect. 3.1, in our sequential model, the input-side weights are replaced with a constant multiple of β. Figure 10 shows the accuracy of the proposed model when



(a) Cora (micro F1 score) **(b)** Amazon Photo (micro F1 score) **(c)** Amazon Electronics Computers (micro F1 score)

(d) Cora (macro F1 score) **(e)** Amazon Photo (macro F1 score) **(f)** Amazon Electronics Computers (macro F1 score)

Fig. 9 Impact of sequential training on accuracy.

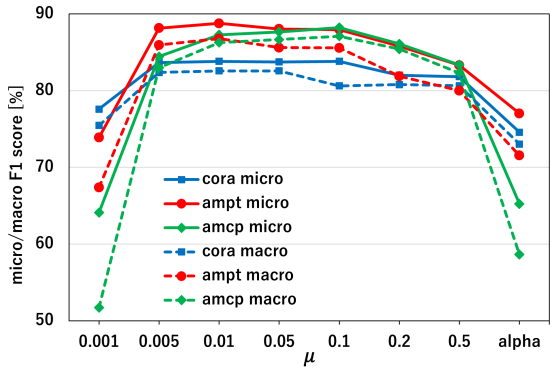


Fig. 10 Impact on scale factor μ on accuracy.

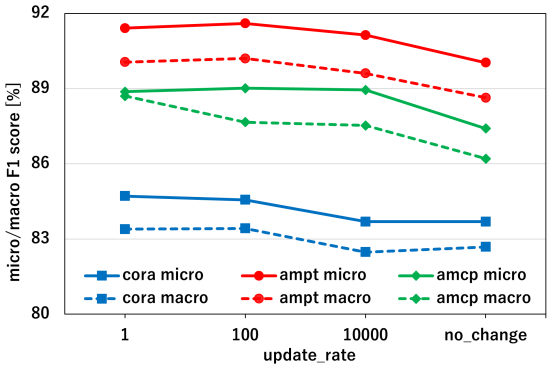


Fig. 11 Impact of update frequency of sampling table on accuracy.

the scale factor μ in Algorithm 2 is varied. Y-axis shows the accuracy, while X-axis shows the scale factor. The number of graph embedding dimensions is 32. In this graph, "alpha" represents an accuracy of a special case, where the input-side weights are fixed with random values as in the original OS-ELM algorithm. The accuracy of this "alpha" case is lower than our proposed model except when the scale factor μ is very small (i.e., 0.001). Actually, the accuracy of the proposed model when μ is 0.001 is quite low, indicating that a meaningful graph embedding may not be learned. On the other hand, we can see that the proposed model when $\mu \geq 0.005$ can learn a useful graph embedding. Especially, the accuracy is quite high when μ is ranging from 0.005 to 0.1, while it is gradually decreased when $\mu > 0.1$.

4.3.4 Impact of Update Frequency of Sampling Table

In our proposed model, a table for the negative sampling is updated during the sequential training. Here, we evaluate the impact of the update frequency of the sampling table in terms of the accuracy. For the negative sampling, as mentioned in Sect. 3.1, the sampling frequency depends on the number of appearances of each node in a given training dataset. Since the Walker's alias is used for the sampling, the time complexity to update the table is proportional to the number of graph nodes. Figure 11 shows the accuracy of the proposed model when the update frequency of the sampling table is varied. Y-axis shows the accuracy, while X-axis shows the number of added edges for each table update (i.e., update frequency of the table). In this graph, "no_change" represents an accuracy of a special case, where the table is not updated once the table is created. The accuracy when the table is updated every 100 edges added is almost the same as that when the table is updated every single edge added. On the other hand, the accuracy is dropped when the table is updated every 10,000 edges added, and that of "no_update" is also low. These negative impacts tend to be large in larger graphs.

Table 8 Model sizes of original model and proposed model (MB).

# graph embedding dimensions	model	cora	ampt	amcp
32	Original model	1.350	3.823	6.783
	Proposed model	0.376	1.088	1.897
64	Original model	2.676	7.559	13.589
	Proposed model	0.735	2.017	3.600
96	Original model	3.999	11.295	20.303
	Proposed model	1.105	2.990	5.318

Table 9 Resource utilizations on XCZU7EV.

# graph embedding dimensions		BRAM	DSP	FF	LUT
32	Used	183	1,379	48,609	53,330
	%	58.65	79.80	10.55	23.15
64	Used	271	1,552	77,584	87,901
	%	86.86	89.81	16.84	38.15
96	Used	272	1,573	86,081	108,639
	%	87.18	91.03	18.68	47.15

4.4 Model Size

Here, we compare the original skip-gram model and our proposed sequential model for the FPGA in terms of the model size. Table 8 shows their model sizes. The results show that the proposed model is up to 3.82 times smaller than the original model, thanks to our simplified OS-ELM based model, where the output-side weights $\boldsymbol{\beta}$ are reused for the input-side weights (thus we do not have to retain $\boldsymbol{\alpha}$). This reduces the memory consumption compared to the original skip-gram model; thus, our proposed model is beneficial for resource-limited IoT devices.

4.5 FPGA Resource Utilization

Here, we evaluate the FPGA resource utilization of the proposed model. We use Zynq UltraScale+ XCZU7EV as a target FPGA device which has 11 Mb BRAM and 1,728 DSP slices. Table 9 shows the resource utilizations when the numbers of graph embedding dimensions are 32, 64, and 96, respectively. In our FPGA implementation, the computational parallelism is basically set to 32. However, when the number of graph embedding dimensions is 64 and 96, the parallelism is partially set to 48 and 64 so that execution times of pipeline stages are equalized for the dataflow optimization. As shown in the table, when the number of graph embedding dimensions is 32, 79.80% of DSP slices are consumed because fixed-point multiply-add operations are parallelized. When the number of graph embedding dimensions is 64, since the number of BRAM partitions is increased for further speedup, the BRAM and DSP utilizations are 86.86% and 89.81%, respectively.

4.6 Power Efficiency

Finally, we compare the power efficiency of our FPGA implementation with that of the CPU and GPU implementations to demonstrate the benefit of the FPGA implementation, which is also a key contribution of this paper. We measured the power consumption of the ZCU104 board using a current sensor, INA226. jetson-stats [24] and s-tui [25] are used to measure the power of Nvidia Jetson Xavier NX and Intel Core i7-11700, respectively. We measured the power consumption over one minute and averaged the measured values. In the cases of ZCU104 and Jetson Xavier NX, we subtracted the average power consumption in the idle state to obtain only the power consumption of our program. Table 10 shows the results.

Table 10 Comparison of power consumption (W).

Devices	Model	# graph embedding dimensions		
		32	64	96
ARM Cortex A-53 CPU	Original	0.230	0.303	0.314
	Proposed	0.214	0.265	0.312
Intel Core i7-11700	Original	35.4	36.8	35.6
	Proposed	35.1	36.4	35.5
GPU	Proposed	0.983	1.024	1.106
FPGA	Proposed	0.318	0.560	0.825

In each graph embedding dimension, the FPGA implementation consumes 110.38, 65.00, and 43.03 times less power than Intel Core i7-11700 CPU. It also achieves 3.09, 1.83, and 1.34 times less power consumption than that of Nvidia Jetson Xavie NX. Combined with the results from Tables 4 and 5, our FPGA implementation offers 111.81, 105.56, and 104.65 times power efficiency than Intel Core i7-11700 CPU in each graph embedding dimension. It also reaches 239.12, 135.76, and 88.87 times power efficiency than Nvidia Jetson Xavier NX. The FPGA implementation achieves 16.36 to 27.88 (32.91 to 78.12) times power efficiency when compared with the implementation of Proposed (Original) model on ARM Cortex-A53 CPU.

5. Conclusion

To improve the performance and power efficiency of the graph embedding, in this paper, we proposed an OS-ELM based sequentially-trainable model for graph embedding and implemented it on an FPGA device. Compared to the original skip-gram model, the proposed model achieved 1.89 to 2.77 times speedup. Furthermore, the FPGA implementation achieved 45.50 to 205.25 times speedup compared to the original model on ARM Cortex-A53 CPU. We also compared our proposed FPGA implementation with an embedded GPU implementation and achieved 88.87 to 239.12 times power efficiency than the Nvidia Jetson Xavier NX. In the proposed model, by replacing the input-side weights with trained output-side weights (i.e., $\boldsymbol{\beta}$), we achieved both the accuracy improvement and the memory size reduction. For the sequential training of dynamic graphs, we showed that although the original model decreases the accuracy, the proposed model can be trained without decreasing the accuracy. We also analyzed the impacts of the scale factor μ and the update frequency of sampling table of the proposed model on accuracy. As a result, we demonstrate that the FPGA implementation improves the performance and power efficiency

compared to the CPU and embedded GPU implementations. In our future work, our FPGA-based sequentially-trainable model will be combined with an FPGA-based random walk acceleration.

Acknowledgments

This work was partially supported by JST AIP Acceleration Research JPMJCR23U3, Japan.

References

[1] A. Grover and L. Jure, "Node2vec: Scalable feature learning for networks," Proc. ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD), pp.855–864, Aug. 2016.
[2] K. Sunaga, K. Sugiura, and H. Matsutani, "An FPGA-based accelerator for graph embedding using sequential training algorithm," Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS) Workshops, pp.148–154, May 2024.
[3] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," arXiv preprint arXiv:1301.3781, Sept. 2013.
[4] G.H. Nguyen, J.B. Lee, R.A. Rossi, N.K. Ahmed, E. Koh, and S. Kim, "Continuous-time dynamic network embeddings," Proc. International Workshop on Learning Representations for Big Networks (BigNet), pp.969–976, April 2018.
[5] S. De Winter, T. Decuypere, S. Mitrović, B. Baesens, and J.D. Weerdt, "Combining temporal aspects of dynamic networks with Node2Vec for a more efficient dynamic link prediction," IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM), pp.1234–1241, Aug. 2018.
[6] S. Mahdavi, S. Khoshraftar, and A. An, "dynnode2vec: Scalable dynamic network embedding," IEEE International Conference on Big Data (Big Data), pp.3762–3765, Dec. 2018.
[7] N.-Y. Liang, G.-b. Huang, P. Saratchandran, and N. Sundararajan, "A fast and accurate online sequential learning algorithm for feedforward networks," IEEE Trans. Neural Netw., vol.17, no.6, pp.1411–1423, Nov. 2006.
[8] M. Tsukada, M. Kondo, and H. Matsutani, "A neural network-based on-device learning anomaly detector for edge devices," IEEE Trans. Comput., vol.69, no.7, pp.1027–1044, 2020.
[9] Y. Li, S. Xie, Z. Wan, H. Lv, H. Song, and Z. Lv, "Graph-powered learning methods in the Internet of Things: A survey," Machine Learning with Applications, vol.11, 100441, 2023.
[10] G. Dong, M. Tang, Z. Wang, J. Gao, S. Guo, L. Cai, R. Gutierrez, B. Campbell, L.E. Barnes, and M. Boukhechba, "Graph neural networks in IoT: A survey," ACM Trans. Sensor Networks, vol.19, no.2, pp.1–50, 2023.
[11] S. Zhang, A. Sohrabizadeh, C. Wan, Z. Huang, Z. Hu, Y. Wang, Y. Lin, J. Cong, and Y. Sun, "A survey on graph neural network acceleration: Algorithms, systems, and customized hardware," arXiv preprint arXiv:2306.14052, June 2023.
[12] Y. Qiao, W. Zhang, X. Du, and M. Guizani, "Malware classification based on multilayer perception and Word2Vec for IoT security," ACM Trans. Internet Technology (TOIT), vol.22, no.1, pp.1–22, 2021.
[13] S. Kim, Y. Suh, and H. Lee, "What IoT devices and applications should be connected? Predicting user behaviors of IoT services with node2vec embedding," Information Processing and Management, vol.59, no.2, 102869, 2022.
[14] H. Tan, X. Chen, Y. Chen, B. He, and W.-F. Wong, "LightRW: FPGA accelerated graph dynamic random walks," Proc. ACM Management of Data, pp.1–27, 2023.
[15] T. Ono, T. Shoji, M.H. Waidyasooriya, M. Hariyama, Y. Aoki, Y. Kondoh, and Y. Nagasawa, "FPGA-based acceleration of Word2vec using OpenCL," Proc. IEEE International Symposium on Circuits and System (ISCAS), pp.1–5, 2019.
[16] O. Press and L. Wolf, "Using the output embedding to improve language models." arXiv preprint arXiv:1608.05859, Feb. 2017.
[17] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed Representations of words and phrases and their compositionality," Proc. International Conference on Neural Information Processing Systems (NeurIPS), pp.3111–3119, Dec. 2013.
[18] A.J. Walker, "An efficient method for generating discrete random variables with general distributions," ACM Trans. Mathematical Software, vol.3, no.3, pp.253–256, Sept. 1977.
[19] S. Ji, N. Satish, S. Li, and P. Dubey, "Parallelizing word2vec in shared and distributed memory," IEEE Trans. Parallel Distrib. Syst., vol.30, no.9, pp.2090–2100, 2019.
[20] W. Jakob, "pybind11 — Seamless operability between C++11 and Python," https://github.com/pybind/pybind11, 2016.
[21] A.K. McCallum, K. Nigam, J. Rennie, and K. Seymore, "Automating the construction of internet portals with machine learning," Information Retrieval, vol.3, pp.127–163, 2000.
[22] O. Shchur, M. Mumme, A. Bojchevski, and S. Günnemman, "Pitfalls of graph neural network evaluation." arXiv preprint arXiv:1811.05868, Nov. 2018.
[23] J. McAuley, C. Targett, Q. Shi, and A. van den Hengel, "Image-based recommendations on styles and substitutes," Proc. International ACM Conference on Research and Development in Information Retrieval (SIGIR), pp.43–52, 2015.
[24] R. Bonghi, "jetson-stats," https://github.com/rbonghi/jetson_stats, 2007.
[25] A. Manuskin, "The stress terminal UI: s-tui," https://github.com/amanusk/s-tui, 2017.

Kazuki Sunaga received the B.E. and M.E. degrees from Keio University, Yokohama, Japan in 2023 and 2025, respectively.

Keisuke Sugiura received the B.E., M.E., and Ph.D. degrees from Keio University in 2020, 2022, and 2024, respectively. He is currently an assistant professor at the Institute of Systems and Information Engineering, University of Tsukuba, Tsukuba, Japan. His research interests include the computer architecture and robotics.

Hiroki Matsutani received the B.A., M.E., and Ph.D. degrees from Keio University in 2004, 2006, and 2008, respectively. He is currently a professor at the Department of Information and Computer Science, Keio University. His research interests include the areas of computer architecture and machine learning.