GraphVite: A High-Performance CPU-GPU Hybrid System for Node Embedding

Zhaocheng Zhu
Mila - Québec AI Institute
Université de Montréal
zhaocheng.zhu@umontreal.ca

Shizhen Xu
Tsinghua University
xsz12@mails.tsinghua.edu.cn

Meng Qu
Mila - Québec AI Institute
Université de Montréal
meng.qu@umontreal.ca

Jian Tang
Mila - Québec AI Institute
HEC Montréal
CIFAR AI Research Chair
jian.tang@hec.ca

ABSTRACT

Learning continuous representations of nodes is attracting growing interest in both academia and industry recently, due to their simplicity and effectiveness in a variety of applications. Most of existing node embedding algorithms and systems are capable of processing networks with hundreds of thousands or a few millions of nodes. However, how to scale them to networks that have tens of millions or even hundreds of millions of nodes remains a challenging problem. In this paper, we propose *GraphVite*, a high-performance CPU-GPU hybrid system for training node embeddings, by co-optimizing the algorithm and the system. On the CPU end, augmented edge samples are parallelly generated by random walks in an online fashion on the network, and serve as the training data. On the GPU end, a novel parallel negative sampling is proposed to leverage multiple GPUs to train node embeddings simultaneously, without much data transfer and synchronization. Moreover, an efficient collaboration strategy is proposed to further reduce the synchronization cost between CPUs and GPUs. Experiments on multiple real-world networks show that *GraphVite* is super efficient. It takes only about one minute for a network with 1 million nodes and 5 million edges on a single machine with 4 GPUs, and takes around 20 hours for a network with 66 million nodes and 1.8 billion edges. Compared to the current fastest system, *GraphVite* is about 50 times faster without any sacrifice on performance.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; *Parallel algorithms.*

KEYWORDS

Unsupervised node embedding, parallel processing, scalability, graphics processing unit

WWW '19, May 13–17, 2019, San Francisco, CA, USA

ACM ISBN 978-1-4503-6674-8/19/05.
https://doi.org/10.1145/3308558.3313508

ACM Reference Format:
Zhaocheng Zhu, Shizhen Xu, Meng Qu, and Jian Tang. 2019. GraphVite: A High-Performance CPU-GPU Hybrid System for Node Embedding. In *Proceedings of the 2019 World Wide Web Conference (WWW '19), May 13–17, 2019, San Francisco, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3308558.3313508

1 INTRODUCTION

Networks are ubiquitous in the real world. Examples like social networks [19], citation networks [27], protein-protein interaction networks [30] and many more cover a wide range of applications. In network analysis, it is critical to have effective representations for nodes, as these representations largely determine the performance of many downstream tasks. Recently, there is a growing interest in unsupervised learning of continuous node representations, which is aimed at preserving the structure of networks in a low-dimensional space. This kind of approaches has been proven successful in various applications, such as node classification [23], link prediction [14], and network visualization [31].

Many works have been proposed on this stream, including DeepWalk [23], LINE [32], and node2vec [8]. These methods learn effective node embeddings by predicting the neighbors of each node and can be efficiently optimized by asynchronous stochastic gradient descent (ASGD) [25]. On a single machine with multi-core CPUs, they are capable of processing networks with one or a few millions of nodes. Given that real-world networks easily go to tens of millions nodes and nearly billions of edges, how to adapt node embedding methods to networks of such large scales remains very challenging. One may think of exploiting computer clusters for training large-scale networks. However, it is a non-trivial task to extend existing methods to distributed settings. Even if distributed algorithms are available, the cost of large CPU clusters is still prohibitive for many users. Therefore, we are wondering whether it is possible to scale node embedding methods to very large networks on a single machine, which should be particularly valuable for common users.

Inspired by the recent success of training deep neural networks with GPUs [5, 13], we would like to utilize such highly parallel hardware to accelerate the training of node embeddings. However, directly adopting GPUs for node embedding could be inefficient, since the sampling procedure in node embedding requires excessive

random memory access on the network structure, which is at the disadvantage of GPUs. Compared to GPUs, CPUs are much more capable of performing random memory access. Therefore, it would be wise to use both CPUs and GPUs for training node embeddings. Along this direction, a straightforward solution is to follow the mini-batch stochastic gradient descent (mini-batch SGD) paradigm utilized in existing deep learning frameworks (e.g. TensorFlow [1] and PyTorch [22]). Different from deep neural networks, the training of node embeddings involves much more memory access per computation. As a result, mini-batch SGD would suffer from severe memory latency on the bus before it benefits from fast GPU computation. Therefore, other than mini-batch SGD, we need to design a system that leverages distinct advantages of CPUs and GPUs and uses them collaboratively to train node embeddings efficiently.

Overall, the main challenges of building an efficient node embedding system with GPUs are:

(1) **Limited GPU Memory** The parameter matrices of node embeddings are quite large while the memory of a single GPU is very small. Modern GPUs usually have a capacity of 12GB or 16GB.
(2) **Limited Bus Bandwidth** The bandwidth of the bus is much slower than the computation speed of GPUs. There will be severe latency if GPUs exchange data with the main memory frequently.
(3) **Large Synchronization Cost** A lot of data are transferred between CPUs and GPUs. Both the CPU-GPU or inter-GPU synchronizations are very costly.

In this paper, we propose a high-performance CPU-GPU hybrid system called *GraphVite* for training node embeddings on large-scale networks. *GraphVite* takes full advantages of CPUs and GPUs by co-optimizing the node embedding algorithm and the system. Specifically, we observe that existing node embedding methods typically consist of two stages, i.e. network augmentation and embedding training. In the first stage, an augmented network is constructed by random walks on the original network, and positive edges are sampled on the augmented network. In the second stage, node embeddings are trained according to the samples generated in the previous stage. Since the augmentation stage involves excessive random access, we resort to CPUs for this part. The training stage is assigned to GPUs as it is mainly composed of matrix computation.

In *GraphVite*, the above challenges are addressed by three components, namely parallel online augmentation, parallel negative sampling and collaboration strategy. In parallel online augmentation, CPUs augment the network with random walks and generate edge samples in an online fashion. In parallel negative sampling, the edge samples are organized into a grid sample pool, where each block corresponds to a subset of the network. Then GPUs iteratively fetch orthogonal blocks and their corresponding embeddings in each episode. Because GPUs do not share any embeddings, multiple GPUs can perform gradient updates with negative sampling in its own subset simultaneously. With such a design, the problem of limited GPU memory is solved as each GPU only stores the subset of node embeddings corresponding to the current sample block. The problem of limited bus bandwidth is mitigated since model parameters are transferred only when GPUs change their blocks. No inter-GPU synchronization is needed and CPU-GPU synchronization is only needed at the end of each episode. The collaboration strategy further reduces the synchronization cost between CPUs and GPUs on the sample pool.

We evaluate *GraphVite* on 4 real-world networks of different scales. On a single machine with 4 Tesla P100 GPUs, our system only takes one minute to train a network with 1 million nodes and 5 million edges. Compared to the current fastest system [32], *GraphVite* is 51 times faster and does not sacrifice any performance. On a network with 66 million nodes and 1.8 billion edges, *GraphVite* takes only around 20 hours to finish training. We also investigate the speed of *GraphVite* under different hardware configurations. Even on economic GPUs like GeForce GTX 1080, *GraphVite* is able to achieve a speedup of 29 times compared to the current fastest system.

Organization Section 2 reviews existing state-of-the-art node embedding methods and points out the challenges of extending these methods to GPUs. Section 3 introduces our proposed system in details. We present our experiments in Section 4, followed by extensive ablation studies in Section 5. Section 6 summarizes the related work, and we conclude this paper in Section 7.

2 PRELIMINARIES

In this section, some preliminary knowledge is introduced. We first review existing state-of-the-art node embedding methods, followed by a discussion on the main challenges of extending these methods to GPUs.

2.1 Node Embedding Methods Review

Given a network $G = (V, E)$, the goal of node embedding is to learn a low-dimensional representation for each node. The learned embeddings are expected to capture the structure of the network. Towards this goal, most existing methods train node embeddings to distinguish the edges in E (i.e. positive edges) from some randomly sampled node pairs (i.e. negative edges). In other words, edges are essentially utilized as training data. Since many real-world networks are extremely sparse, most existing embedding methods conduct random walks on the original network to introduce more connectivity. Specifically, they connect nodes within a specified distance on a random walk path as additional positive edges. For example, LINE [32] uses a breadth-first search strategy on low-degree nodes, while DeepWalk [23] uses a depth-first search strategy for all nodes. Node2vec [8] developed a mixture of the above two strategies.

Once the network is augmented, node embeddings are trained on samples from the augmented network. Typically, the embeddings are encoded in two sets, namely **vertex** embedding matrix and **context** embedding matrix. For an edge sample (u, v), the dot product of $\mathbf{vertex}[u]$ and $\mathbf{context}[v]$ is computed to predict whether the sample is a positive edge. This encourages neighbor nodes to have close embeddings, whereas distant nodes will have very different embeddings.

Overall, the computation procedures of these node embedding methods can be divided into two stages: **network augmentation** and **embedding training**. Algorithm 1 summarizes the general framework of existing node embedding methods. Note that the first

stage can be easily parallelized, and the second stage can be parallelized via asynchronous SGD. In most existing node embedding systems, these two stages are executed in a sequential order, with each stage parallelized by a bunch of CPU threads.

Algorithm 1 General framework of node embedding

```
E' ← E
for v ∈ V do                                          ▷ parallelizable
    for u ∈ Walk(v) do
        E' ← E' ∪ {(v, u, Weight(v, u))}
    end for
end for

for each iteration do                                 ▷ parallelizable
    v, u ← EdgeSampling(E')
    Train(vertex[v], context[u], label = 1)
    for u' ∈ NegativeSampling(V) do
        Train(vertex[v], context[u'], label = 0)
    end for
end for
```

2.2 Challenges for Hybrid Node Embedding System

Inspired by the recent success of training neural networks with GPUs, we are interested in building a node embedding system by leveraging the power of GPUs, which can benefit the embedding training stage. Since the first stage of network augmentation involves extensive memory random access, CPUs are more suitable for this stage. As a result, we desire to develop a hybrid CPU-GPU system for training node embeddings. A common approach for a hybrid machine learning system is the mini-batch SGD paradigm, which is widely adopted in existing deep learning frameworks such as TensorFlow [1] and PyTorch [22]. In mini-batch SGD, model parameters are stored on GPUs and training data is iteratively passed to GPUs in batches.

However, mini-batch SGD cannot be applied directly to node embedding on large networks. Take a scale-free network with 50 million nodes and 1 billion edges as an example. (1) The size of the augmented network goes to 373 GB large, which may overwhelm the memory of most servers. We need to figure out a way to generate the augmented network and edge samples on the fly. (2) The embedding matrices are much larger than the parameter matrices in deep neural networks. Either **vertex** or **context** matrix consumes 23.8 GB memory, which is beyond the memory limit of any single GPU. As a result, both the **vertex** and **context** embedding matrices have to be stored in the main memory, and transferred to the GPUs in small parts during training. See Table 1 for a detailed analysis of the memory cost.

For the second problem, while transferring parameters from CPUs and GPUs sounds feasible, it will become a bottleneck if we consider the bus bandwidth between CPUs and GPUs. Consider training d-dimensional embeddings with n edge samples. There is $O(nd)$ computation workload and also $O(nd)$ memory access if the samples are not overlapped with each other. Since the computation speed of GPUs is way faster than the speed of bus transfer, the entire system will be bounded by the speed of parameter transfer from CPUs to GPUs severely. Indeed, such a system is even worse than its CPU parallel counterpart, which is verified in our experiments (see Table 3).

	Size	Example	Memory cost
nodes	$\|V\|$	$5 * 10^7$	191 MB
edges	$\|E\|$	$1 * 10^9$	7.45 GB
augmented edges	$\|E'\|$	$5 * 10^{10}$	373 GB
vertex	$\|V\| \times d$	$5 * 10^7 \times 128$	23.8 GB
context	$\|V\| \times d$	$5 * 10^7 \times 128$	23.8 GB

Table 1: Memory cost of node embedding on a scale-free network with 50 million nodes and 1 billion edges.

Another challenge in a hybrid system is the large synchronization cost. Since the system is distributed on multiple CPUs and GPUs, there is necessary data (e.g. parameters and edge samples) shared across sub tasks. A trivial but safe solution is to synchronize the shared data frequently, which will result in huge synchronization cost. To achieve high speed performance, the system should reduce shared data as much as possible, and use a collaboration strategy to minimize synchronization cost between devices.

3 *GRAPHVITE*: A HYBRID CPU-GPU NODE EMBEDDING SYSTEM

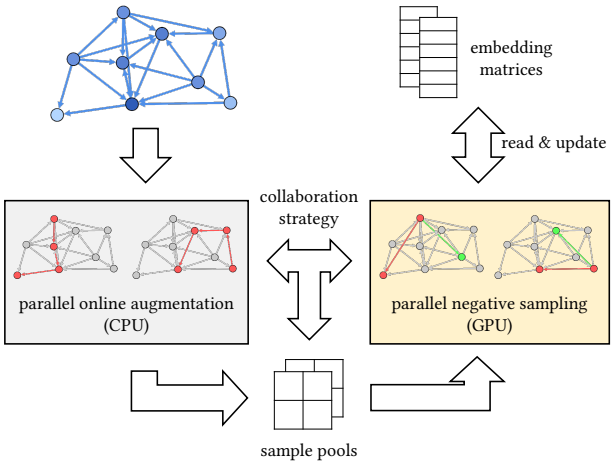


Figure 1: Overview of our hybrid system. The gray and yellow boxes correspond to the stages of network augmentation and embedding training respectively. The former is performed by parallel online augmentation on CPUs, while the latter is performed by parallel negative sampling on GPUs. The two stages are executed asynchronously with our collaboration strategy.

In this section, we introduce a high-performance hybrid CPU-GPU system called *GraphVite* for training node embeddings. Our

system leverages distinct advantages of CPUs and GPUs and addresses the above three challenges. Specifically, we propose a parallel online augmentation for efficient network augmentation on CPUs. We introduce a parallel negative sampling to cooperate multiple GPUs for embedding training. A collaboration strategy is also proposed to reduce the synchronization cost between CPUs and GPUs.

3.1 Parallel Online Augmentation

As discussed in Section 2.1, the first stage of node embedding methods is to augment the original network with random walks. Since the augmented network is usually one or two magnitude larger than the original one, it is impossible to load it into the main memory if the original network is already very large. Therefore, we introduce a parallel online augmentation, which generates augmented edge samples on the fly without explicit network augmentation. Our method can be viewed as an online extension of the augmentation and edge sampling method used in LINE[32]. First, we draw a departure node with the probability proportional to the degree of each node. Then we perform a random walk from the departure node, and pick node pairs within a specific augmentation distance s as edge samples. Note that edge samples generated in the same random walk are correlated and may degrade the performance of optimization. Inspired by the experience replay technique widely used in reinforcement learning [15, 20], we collect edge samples into a sample pool, and shuffle the sample pool before transferring it to GPUs for embedding training. The proposed edge sampling method can be parallelized when each thread is allocated with an independent sample pool in advance. Algorithm 2 gives the process of parallel online augmentation in details.

Algorithm 2 Parallel Online Augmentation

```
1:  function ParallelOnlineAugmentation(num_CPU)
2:      for i ← 0 to num_CPU − 1 do                    ▷ paralleled
3:          pool[i] ← ∅
4:          while pool is not full do
5:              x ← DepartureSampling(G)
6:              for u, v ∈ RandomWalkSampling(x) do
7:                  if Distance(u, v) <= s then
8:                      pool.append((u, v))
9:                  end if
10:             end for
11:         end while
12:         pool[i] ← Shuffle(pool[i])
13:     end for
14:     return Concatenate(pool[·])
15: end function
```

Pseudo Shuffle While shuffling the sample pool is important to optimization, it slows down the network augmentation stage (see Table 7). The reason is that a general shuffle consists of lots of random memory access and cannot be accelerated by the CPU cache. The loss in speed will be even worse if the server has more than one CPU socket. To mitigate this issue, we propose a pseudo shuffle technique that shuffles correlated samples in a much more cache-friendly way and improves the speed of the system significantly. Note that most correlation comes from edge samples that share the source node or the target node in the same random walk. As such correlation occurs in a group of s samples for an augmentation distance s, we divide the sample pool into s continuous blocks, and scatter correlated samples into different blocks. For each block, we always append samples sequentially at the end, which can benefit a lot from CPU cache. The s blocks are concatenated to form the final sample pool.

3.2 Parallel Negative Sampling

In the embedding training stage, we divide the training task into small fragments and distribute them to multiple GPUs. The sub tasks are necessarily designed with little shared data to minimize the synchronization cost among GPUs. To see how model parameters can be distributed to multiple GPUs without overlap, we first introduce a definition of *ϵ-gradient exchangeable*.

Definition 1. ***ϵ-gradient exchangeable.*** *A loss function $L(X;\theta)$ is ϵ-gradient exchangeable on two sets of training data X_1, X_2 if for small $\epsilon \geq 0$, $\forall\theta_0 \in \Theta$ and $\forall\alpha \in \mathbb{R}^+$, exchanging the order of two gradient descent steps results in a vector difference with norm no more than ϵ.*

$$\begin{cases} \theta_1 \leftarrow \theta_0 - \alpha\nabla L(X_1;\theta_0) \\ \theta_2 \leftarrow \theta_1 - \alpha\nabla L(X_2;\theta_1) \end{cases} \tag{1}$$

$$\begin{cases} \theta_1' \leftarrow \theta_0 - \alpha\nabla L(X_2;\theta_0) \\ \theta_2' \leftarrow \theta_1' - \alpha\nabla L(X_1;\theta_1') \end{cases} \tag{2}$$

i.e. $\|\theta_2 - \theta_2'\| \leq \epsilon$ is true for the above equations.

Particularly, we abbreviate *0-gradient exchangeable* to *gradient exchangeable*. Due to the sparse nature of node embedding training, there are many sets that form *gradient exchangeable* pairs in the network. For example, for two edge sample sets $X_1, X_2 \subseteq E$, if they do not share any source nodes or target nodes, X_1 and X_2 are *gradient exchangeable*. Even if X_1 and X_2 share some nodes, they can still be *ϵ-gradient exchangeable* if the learning rate α and the number of iterations are bounded.

Based on the gradient exchangeability observed in node embedding, we propose a parallel negative sampling algorithm for the embedding training stage. For n GPUs, we partition rows of **vertex** and **context** into n partitions respectively (see the top-left corner of Figure 2). This results in an $n \times n$ partition grid for the sample pool, where each edge belongs to one of the blocks. In this way, any pair of blocks that does not share row or column is *gradient exchangeable*. Blocks in the same row or column are *ϵ-gradient exchangeable*, as long as we restrict the number of iterations on each block.

We define *episode* as the block-level step used in parallel negative sampling. During each episode, we send n orthogonal blocks and their corresponding **vertex** and **context** partitions to n GPUs respectively. Each GPU then updates its own embedding partitions with ASGD. Because these blocks are mutually *gradient exchangeable* and do not share any row in the parameter matrices, multiple GPUs can perform ASGD concurrently without any synchronization. At the end of each episode, we gather the updated parameters from all GPUs and assign another n orthogonal blocks. Here *ϵ-gradient exchangeable* is controlled by the number of total samples

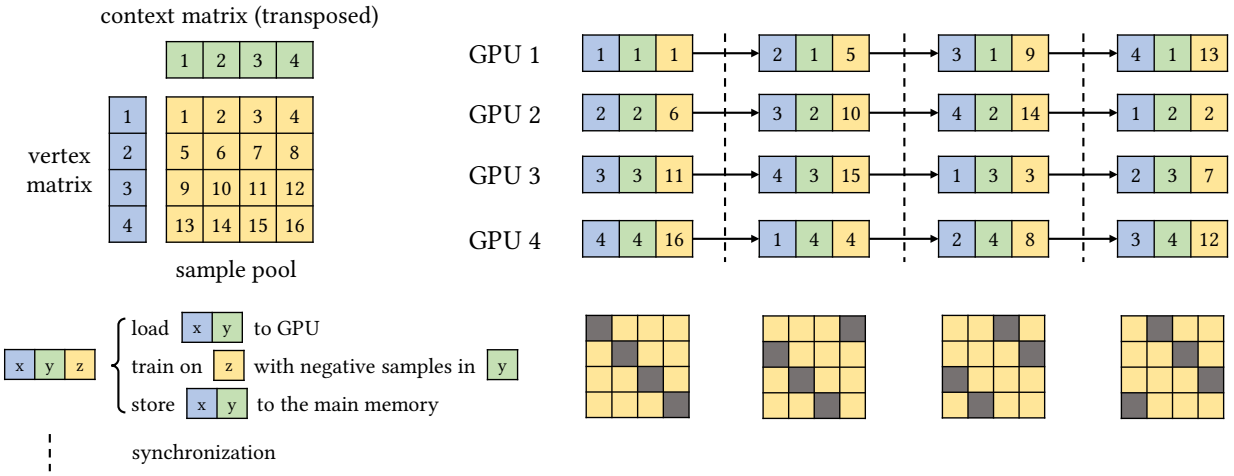


Figure 2: Illustration of parallel negative sampling on 4 GPUs. During each episode, GPUs take orthogonal blocks from the sample pool. Each GPU trains embeddings with negative samples drawn from its own context nodes. Synchronization is only needed between episodes.

in n orthogonal blocks, which we define as *episode size*. The smaller episode size, the better *ϵ-gradient exchangeable* we will have for embedding training. However, smaller episode size will also induce more frequent synchronization. Hence the episode size is tuned so that there is a good trade off between the speed and *ϵ-gradient exchangeable* (see Section 5.3). Figure 2 gives an example of parallel negative sampling with 4 partitions.

Typically, node embedding methods sample negative edges from all possible nodes. However, it could be very time-consuming if GPUs have to communicate with each other to get the embeddings of their negative samples. To avoid this cost, we restrict that negative samples can only be drawn from the **context** rows on the current GPU. Though this seems a little problematic, we find it works well in practice. An intuitive explanation is that with parallel online augmentation, every node is likely to have positive samples with nodes from all context partitions. As a result, every node can potentially form negative samples with all possible nodes.

Note that although we demonstrate with the number of partitions equal to n, the parallel negative sampling can be easily generalized to cases with any number of partitions greater than n, simply by processing the orthogonal blocks in subgroups of n during each episode. Algorithm 3 illustrates the hybrid system for multiple GPUs.

Algorithm 3 Parallel Negative Sampling

```
function ParallelNegativeSampling(num_GPU)
    vertex_partitions ← Partition(vertex)
    context_partitions ← Partition(context)
    while not converge do
        pool ← ParallelOnlineAugmentation(num_CPU)
        block[·][·] ← Redistribute(pool)
        for offset ← 0 to num_GPU − 1 do
            for i ← 0 to num_GPU − 1 do                ▷ paralleled
                vid ← i
                cid ← (i + offset)  mod num_GPU
                send vertex_partitions[vid] to GPU i
                send context_partitions[cid] to GPU i
                train block[vid][cid] on GPU i
                receive vertex_partitions[vid] from GPU i
                receive context_partitions[cid] from GPU i
            end for
        end for
    end while
end function
```

3.3 Collaboration Strategy

Our parallel negative sampling enables different GPUs to train node embeddings concurrently, with only synchronization required between episodes. However, it should be noticed that the sample pool is also shared between CPUs and GPUs. If they synchronize on the sample pool, then only workers of the same stage can access the pool at the same time, which means hardware is idle for half of the time. To eliminate this problem, we propose a collaboration strategy to reduce the synchronization cost. We allocate two sample pools in the main memory, and let CPUs and GPUs always work on different pools. CPUs first fill up a sample pool and pass it to GPUs. After that, parallel online augmentation and parallel negative sampling are performed concurrently on CPUs and GPUs respectively. The two pools are swapped when CPUs fill up a new pool. Figure 1 illustrates this procedure. With the collaboration strategy, the synchronization cost between CPUs and GPUs is reduced and the speed of our hybrid system is almost doubled.

3.4 Discussion

Here we further discuss some practical details of our hybrid system.

Batched Transfer In parallel negative sampling, the sample pool is assigned to GPUs by block, which is sometimes very large for the

Dataset	YOUTUBE	FRIENDSTER-SMALL	HYPERLINK-PLD	FRIENDSTER
$\|V\|$	1,138,499	7,944,949	39,497,204	65,608,376
$\|E\|$	4,945,382	447,219,610	623,056,313	1,806,067,142
Evaluation Task	47-class node classification	100-class node classification	link prediction	100-class node classification

Table 2: Statistics of the datasets used in experiments

memory of a GPU. Instead of copying the whole sample block to a GPU, we transfer the sample block by a small granularity. In this way, the memory cost of edge samples on GPUs becomes negligible.

Bus Usage Optimization When the number of partitions equals the number of GPUs, we can further optimize the bus usage by fixing the context partition for each GPU. In this way, we save the transfer of *context* matrix and further reduce the synchronization cost between CPUs and GPUs.

Single GPU Case Although parallel negative sampling is proposed for multiple GPUs, our hybrid system is compatible with a single GPU. Typically a GPU can hold at most 12 million node embeddings. So a single GPU is sufficient for training node embeddings on networks that contain no more than 12 million nodes.

4 EXPERIMENTS

In this section, we verify the effectiveness and efficiency of *GraphVite*. We first evaluate our system on YOUTUBE, which is a large network widely used in the literature of node embeddings. Then we evaluate *GraphVite* on three larger datasets.

4.1 Datasets

We use the following datasets in our experiments. Statistics of these networks are summarized in Table 2.

- YOUTUBE [19] is a large-scale social network in the Youtube website. It contains 1 million nodes and 5 million edges. For some of the nodes, they have labels that represent the type of videos users enjoy.
- FRIENDSTER-SMALL [37] is a sub-graph induced by all the labeled nodes in FRIENDSTER. It has 8 million nodes and 447 million edges. The node labels in this network are the same as those in FRIENDSTER.
- HYPERLINK-PLD [16] is a hyperlink network extracted from the Web corpus [1]. We use the pay-level-domain aggregated version of the network. It has 43 million nodes and 623 million edges. This dataset does not contain any label.
- FRIENDSTER [37] is a very large social network in an online gaming site. It has 65 million nodes and 1.8 billion edges. Some nodes have labels that represent the group users join.

4.2 Compared Systems

We compare *GraphVite* with the following node embedding systems.

- LINE [32] [2] is a CPU parallel system based on C++. We parallel its network augmentation stage for fair comparison with other methods.
- DeepWalk [23] [3] is a CPU parallel system based on Python and gensim [26].
- node2vec [8] [4] is another CPU parallel system based on Python and gensim [26].
- LINE in OpenNE [2] [5] is a GPU system based on the Python and TensorFlow [1].

Although DeepWalk and node2vec are implemented in Python, their computation is fully paralleled by Cython code without GIL, which is comparable to C++ implementations.

4.3 Implementation Details

Our implementation generally follows the open source codes of LINE [2] and DeepWalk [3]. We adopt the asynchronous SGD [25] in GPU training, and leverage the on-chip shared memory of GPU for fast forward and backward propagation. We also utilize the alias table trick [8, 32] to boost parallel online augmentation and parallel negative sampling.

Our hyperparameters are set according to the settings in LINE [32] and DeepWalk [23]. We treat networks as undirected graphs. During the network augmentation stage, we sample random walks with a length of 40 edges. We use a degree-guided strategy to partition both **vertex** and **context** matrices. More specifically, we first sort nodes by their degrees and then assign them into different partitions in a zig-zag fashion, as illustrated in Figure 3. We tune the episode size to maximize the speed of our hybrid system. During the embedding training stage, negative samples are sampled with a probability proportional to the 3/4 power of the node degrees. For each positive sample, we draw 1 negative sample and scale the gradient of the negative sample by 5 to match the gradient scale in LINE. We follow the initial learning rate of 0.025 and the linear learning rate decay mechanism in LINE and DeepWalk. We only adopt the *O3* optimization in *g++* and *nvcc*. We do not use any non-standard optimizations or low precision training [17, 39], though they may further improve the speed of our system.

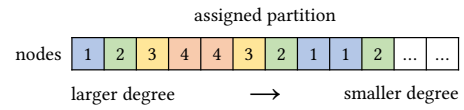


Figure 3: Degree-guided node and context partition strategy in the case of 4 partitions.

We define a training epoch as training $|E|$ positive edge samples. For YOUTUBE, the length of random walk is set to 5, and the total

[1] http://commoncrawl.org/
[2] https://github.com/tangjianpku/LINE
[3] https://github.com/phanein/deepwalk
[4] https://github.com/aditya-grover/node2vec
[5] https://github.com/thunlp/OpenNE

Method	CPU threads	GPU	Training time	Preprocessing time
LINE [32]	20	-	1.24 hrs	17.4 mins
DeepWalk [23]	20	-	1.56 hrs	14.2 mins
node2vec [8]	20	-	47.7 mins	25.9 hrs
LINE in OpenNE [2]	1	1	> 1 day	2.14 mins
GraphVite	6	1	**3.98 mins**(18.7×)	**7.37 s**
GraphVite	24	4	**1.46 mins**(50.9×)	**16.0 s**

Table 3: Results of time of different systems on Youtube. The preprocessing time refers to all the overhead before training, including network input and offline network augmentation. Note the preprocessing time of OpenNE is not comparable since it does not have the network augmentation stage. The speedup ratio of *GraphVite* is computed with regard to LINE, which is the current fastest system.

	% Labeled Nodes	1%	2%	3%	4%	5%	6%	7%	8%	9%	10%
Micro-F1(%)	LINE[32]	32.98	36.70	38.93	40.26	41.08	41.79	42.28	42.70	43.04	43.34
	LINE[32]+augmentation	36.78	40.37	42.10	43.25	43.90	44.44	44.83	45.18	45.50	45.67
	DeepWalk[32]	**39.68**	41.78	42.78	43.55	43.96	44.31	44.61	44.89	45.06	45.23
	GraphVite	39.19	**41.89**	**43.06**	**43.96**	**44.53**	**44.93**	**45.26**	**45.54**	**45.70**	**45.86**
Macro-F1(%)	LINE[32]	17.06	21.73	25.28	27.36	28.50	29.59	30.43	31.14	31.81	32.32
	LINE[32]+augmentation	22.18	27.25	29.87	31.88	32.86	33.73	34.50	35.15	35.76	36.19
	DeepWalk[32]	**28.39**	**30.96**	**32.28**	**33.43**	**33.92**	34.32	34.83	35.27	35.54	35.86
	GraphVite	25.61	29.46	31.32	32.70	33.81	**34.59**	**35.27**	**35.82**	**36.14**	**36.49**

Table 4: Results of node classification on Youtube

number of training epochs is set to 4,000. For the other 3 datasets, the length of random walks is set to 2, and the total number of training epochs is set to 2,000 since they are denser. The dimension of node embeddings is set to 128 except on Friendster, where we use 96. For other hyperparameters, we follow their default values in previous works. For fair comparison, we report the training time of all methods with the same number of training epochs. We parallel the network augmentation in LINE. For DeepWalk, we store the random walks in memory, which is the fastest setting.

4.4 Results on Youtube

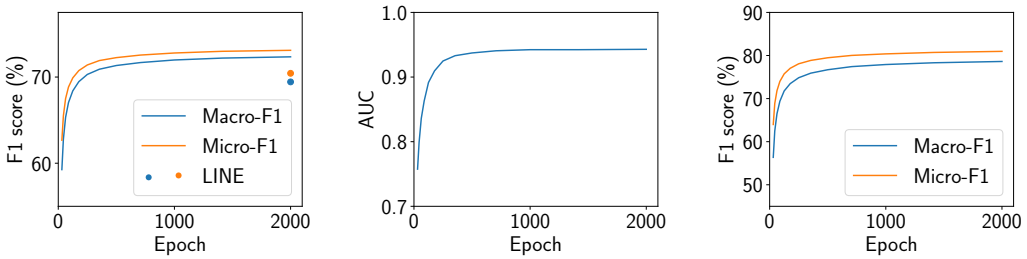
We first evaluate our hybrid system on the widely-used Youtube dataset. We compare the speed and performance of *GraphVite* with existing systems of node embedding. For existing systems, we replicate their parallel implementations and report their training time under the same number of training epochs. Table 3 presents the speed of different systems. Among all existing systems, LINE [32] takes the minimal total time to run. However, the GPU implementation of LINE in OpenNE is even worse than its CPU counterpart, possibly due to the mini-batch SGD paradigm it uses. Compared to the current fastest system, LINE, *GraphVite* is much more efficient. With 4 GPUs, our system finishes training node emebdding on a million-scale network in only one and a half minutes. Even on a single GPU, *GraphVite* takes no more than 4 minutes and is still 19 times faster than LINE.

One may be curious about the performance of node embeddings learned by *GraphVite*. Therefore, we compare the performance of *GraphVite* with existing systems on the standard task of multi-label node classification. Note that normalizing the embeddings or not yields different trade off between Micro-F1 and Macro-F1 metrics. For fair comparison, we follow the practice in [32] and train one-vs-rest linear classifiers over the normalized node embeddings. Table 4 summarizes the performance over different percentages of training data. It is observed that *GraphVite* achieves the best or competitive results in most settings, showing that *GraphVite* does not sacrifice any performance. In some small percentage cases, *GraphVite* falls a little behind DeepWalk. This is because *GraphVite* uses negative sampling for optimization, while DeepWalk uses both hierarchical softmax and negative sampling, which could be more robust to few labeled data.

4.5 Results on Larger Datasets

	Friendster-small	Hyperlink-PLD	Friendster
1 GPU	8.78 hrs	-	-
4 GPU	2.79 hrs	5.36 hrs	20.3 hrs

Table 5: Results of time on larger datasets. We only evaluate Hyperlink-PLD and Friendster with 4 GPUs since their embedding matrices cannot fit into the memory of a single GPU.



(a) Friendster-small (b) Hyperlink-PLD (c) Friendster

Figure 4: Performance curves of *GraphVite* on larger datasets. For Friendster, we plot the results of LINE for reference. The other systems cannot solve any of these datasets within a week.

	Parallel Online Augmentation	Parallel Negative Sampling (4 GPUs)	Collaboration Strategy	Micro-F1	Macro-F1	Training time
Single GPU baseline				35.26	20.38	8.61 mins
	✓			41.48	29.80	6.35 mins
		✓		34.38	19.81	2.66 mins
	✓	✓		41.75	29.30	2.24 mins
GraphVite	✓	✓	✓	41.89	29.46	**1.46 mins**

Table 6: Ablation of main components in *GraphVite*. Note that the baseline has the same GPU implementation with *GraphVite* and parallel edge sampling on CPU. The baseline should be regarded as a very strong one.

To demonstrate the scalability of *GraphVite*, we further test *GraphVite* on three larger networks. We learn the node embeddings of Friendster-small with 1 GPU and 4 GPUs. For Hyperlink-PLD and Friendster, since their embedding matrices cannot fit into the memory of a single GPU, we only evaluate them with 4 GPUs. Table 5 gives the training time of *GraphVite* on these datasets. The training time of baseline systems is not reported here, as all existing systems cannot solve such large networks in a week, except LINE [32] on Friendster-small. Compared to them, *GraphVite* takes less than 1 day to train node embeddings on the largest dataset Friendster with 1.8 billion edges, showing that *GraphVite* can be an efficient tool for analyzing billion-scale networks.

We also evaluate the performance of the node embeddings on these datasets. For Friendster-small and Friendster, we test their node embeddings on multi-label node classification. The test set is built on the top-100 communities of Friendster and has a total of 39,679 nodes. We do not normalize the learned embeddings during evaluation, and report Macro-F1 and Micro-F1 based on 2% labeled data. For Hyperlink-PLD, we adopt link prediction as the evaluation task since node labels are not available. We randomly exclude 0.01% edges from the training set, and combine them with the same number of uniformly sampled negative edges to form a test set. Each edge sample is scored by the cosine similarity of two node embeddings. We report the AUC metric for link prediction. Figure 4 presents the performance of *GraphVite* over different training epochs on these datasets. On Friendster-small, we also plot the performance of LINE for reference. Due to the long training time, we only report the performance of LINE by the end of all training epochs. It is observed that *GraphVite* converges on all these datasets. On the Friendster-small dataset, *GraphVite* significantly outperforms LINE. On the Hyperlink-PLD, we get an AUC of 0.943. On Friendster, the Micro-F1 reaches about 81.0%. All the above observations verify the performance of our system.

5 ABLATION STUDY

To have a more comprehensive understanding of different components in *GraphVite*, we conduct several ablation experiments. For intuitive comparison, we evaluate these experiments on the standard Youtube dataset. We only report performance results based on 2% labeled data due to space limitation. All the speedup ratios are computed with respect to LINE [32].

5.1 What is the contribution of each main component?

In the *GraphVite*, parallel online augmentation, parallel negative sampling, and the collaboration strategy are the main components in the sytem. Here we study how these components contribute to the performance of our system. We compare *GraphVite* with a strong baseline system with single GPU. Specifically, the baseline has the same GPU implementation as *GraphVite*, while it uses the standard parallel edge sampling instead of parallel online augmentation, and executes two stages sequentially.

Table 6 shows the results of this ablation. Compared to the baseline, we notice that parallel online augmentation helps improve the quality of node embeddings, since it introduces more connectivity

to the sparse network. Besides, parallel online augmentation also accelerates the system a little, as it reuses nodes and reduces the amortized cost of each sample. With parallel negative sampling, we are able to employ multiple GPUs for training, and the speed is boosted by about 3 times. Moreover, the collaboration strategy even improves the speed and does not impact the performance.

5.2 Is it necessary to perform pseudo shuffle?

In parallel online augmentation, *GraphVite* performs pseudo shuffle to decorrelate the augmented edge samples, while some existing systems [8, 23] do not shuffle their samples. We compare the proposed pseudo shuffle with three baselines, including no shuffle, a full random shuffle and an index mapping algorithm. The index mapping algorithm preprocesses a random mapping on the indexes of samples and saves the time of computing random variables.

Table 7 gives the results of different shuffle algorithms on a single GPU. It is observed that all shuffle algorithms are about 1 percent better than the no shuffle baseline. However, different shuffle algorithms vary largely in their speed. Compared to the no shuffle baseline, the random shuffle and index mapping algorithms slow down the system by several times, while our pseudo shuffle has only a little overhead. Therefore, we conclude that pseudo shuffle is the best practice considering both speed and performance.

Shuffle algorithm	Micro-F1(%)	Training time
None	40.41	3.60 mins
Random shuffle	41.61	17.1 mins
Index mapping	41.21	12.1 mins
Pseudo shuffle	**41.52**	**3.98 mins**

Table 7: Results of performance and speed by different shuffle algorithms. The proposed pseudo shuffle algorithm achieves the best trade off between performance and speed.

5.3 What is a practical choice for episode size?

In parallel negative sampling, *GraphVite* relies on the property of gradient exchangeability to ensure its approximation to standard SGD. While the smaller episode size provides better exchangebility, it will increase the frequency of synchronization over rows of the embedding matrices, and thus slows down embedding training. To quantify such influence in speed and performance, we examine our system on 4 GPUs with different episode sizes.

Figure 5 plots the curves of speed and performance with respect to different episode sizes. On the performance side, we notice that the performance of *GraphVite* is insensitive to the choice of the episode size. Compared to the single GPU baseline, parallel negative sampling achieves competitive or slightly better results, probably due to the regularization effect introduced by partition. On the speed side, larger episode size achieves more speedup since it reduces the amortized burden of the bus. The speed drops at very large episode size, as there becomes only a few episodes in training. Therefore, we choose an episode size of $2 * 10^8$ edge samples for YOUTUBE. Generally, the best episode size is proportional to $|V|$, so one can set the episode size for other networks accordingly.

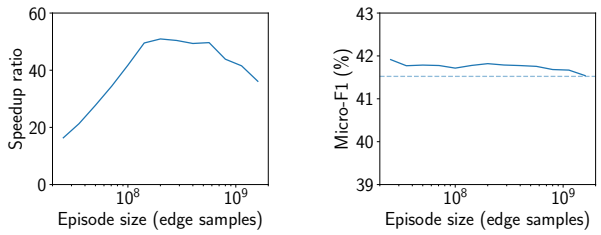


Figure 5: Speed and performance of *GraphVite* with respect to different episode sizes. The dashed line represents the single GPU baseline without parallel negative sampling.

5.4 What is the speedup w.r.t the numebr of CPUs and GPUs?

In *GraphVite*, both online augmentation and negative sampling can be parallelized on multiple CPUs or GPUs, and synchronization is only required between episodes. Therefore, our system should have great scalability. To verify that point, we investigate our system with different number of CPU and GPU. We change the number of GPU from 1 to 4, and vary the number of sampler per GPU from 1 to 5. The effective number of CPU threads is $\#GPU * (\#sampler\ per\ GPU + 1)$ as there is one scheduler thread for each GPU.

Figure 6 plots the speedup ratio with respect to different number of CPUs and GPUs. The speedup ratio almost forms a plane over both variables, showing that our system scales almost linearly to the hardware. Quantitatively, *GraphVite* achieves a relative speedup of 11× when the hardware is scaled to 20×. The speedup is about half of its theoretical maximum. We believe this is mainly due to the increased synchronization cost, as well as increased load on shared main memory and bus when we use more CPUs and GPUs.

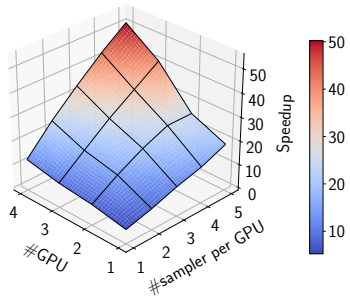


Figure 6: Results of speedup under different number of hardware. It is observed that the speedup is almost linear to the number of CPUs and GPUs.

5.5 Does the hardware configuration matter?

Up to now, all experiments are conducted on a server with Xeon E5 CPUs and Tesla P100 GPUs. One might wonder whether such a high performance depends on the specific hardware configuration. Therefore, we further test our system on an economic server with Core i7 CPUs and GTX 1080 GPUs.

Table 8 compares the results from two configurations. Different hardware does have difference in speed, but the gap is marginal. The time only increases to 1.6× when we move to the economic server. Note that this two configurations are almost the best and the worst in current machine learning servers, so one could expect a running time between these two configurations on his own hardware.

Hardware	CPU threads	GPU	Training time
Tesla P100 server	6	1	3.98 mins
	24	4	1.46 mins
GTX 1080 server	3	1	6.28 mins
	12	4	2.48 mins

Table 8: Training time of *GraphVite* under different hardware configurations. Generally *GraphVite* may take a time between these two configurations on most hardware.

6 RELATED WORK

Node embedding has been proven effective in a wide range of applications, such as node classification [10], link prediction [14], and network visualization [31]. Many different methods [4, 8, 23, 24, 32, 33, 35, 36] have been proposed to learn node embeddings that preserve the structure of networks from different aspects. Among them, DeepWalk [23], LINE [32] node2vec [8] and VERSE [33] are built on either edge or path samples of networks, which makes them the most scalable methods of all. Our work follows this stream and is related to these methods.

Node Embedding Algorithm Generally, node embedding algorithms consist of two stages, namely network augmentation and embedding training. The network augmentation stage is widely adopted in existing methods [4, 8, 23, 24, 32] to improve the performance of learned embeddings on sparse networks. DeepWalk [23] and node2vec [8] augment networks by generating random paths according to different distributions. The edge samples derived by path are correlated in those methods. LINE [32] directly adds edges to networks and generates independent edge samples. GraRep [4] and NetMF [24] take different powers of the adjacency matrix as augmentation. Our parallel online augmentation generates decorrelated edge samples using pseudo shuffle, and thus is close to the augmentation in LINE. However, our augmentation does not need to store the whole augmented network, which saves a lot of disk and memory usage compared to existing methods.

In the embedding training stage, most existing node embedding algorithms [8, 23, 32] train node embedding with standard negative sampling [18] in a shared memory space. While there is a parallel word embedding algorithm [29] that restricts negative sampling within the context partition of each worker, it still needs to transfer rows of embedding matrices between each worker for positive samples. By contrast, our parallel negative sampling trains on orthogonal sample blocks and does not need any transfer between worker during an episode. The most related method is the distributed SGD used in large-scale matrix factorization algorithms [3, 6, 38, 40]. These methods divide the input matrix into $n \times n$ blocks and factorize orthogonal blocks simultaneously. Different from these methods, our system mainly focuses on the negative sampling technique and is designed for the node embedding task.

Node Embedding System From the perspective of system, our work belongs to the parallel implementation of node embedding. There are many CPU parallel systems, including DeepWalk [3], LINE [2], node2vec [4] and VERSE [6]. These systems use asynchronous SGD [25] in embedding training and exploit multiple CPU threads for acceleration. Due to the limited computation speed of CPUs, such systems cannot scale to ten-million-scale networks without a large CPU cluster. Recently, there are some GPU parallel systems [2] built on deep learning frameworks like TensorFlow [1] or PyTorch [22]. Since existing frameworks are based on mini-batch SGD paradigm, these systems severely suffer the problem of limited bus bandwidth, and are even worse than their CPU counterparts. Compared to them, *GraphVite* is a hybrid CPU-GPU system that leverages distinct advantages of CPUs and GPUs, and uses them collaboratively to train node embedding, which makes it much faster than either pure CPU or mini-batch-SGD based systems.

In addition, parallel word embedding systems [9, 11, 18, 28] are also very related to our work, since they share similar embedding training and negative sampling steps with node embedding. Among these methods, Wombat [28] and BlazingText [9] accelerate training with GPUs. Wombat only supports single GPU. BlazingText can scale to multiple GPUs, but it simply makes a copy of the parameter matrices on each GPU. Our system is more efficient than BlazingText in two aspects. First, our system partition the parameter matrices and consumes less memory on each GPU. Second, our system requires less synchronization cost, as GPUs do not share any rows in the parameter matrices.

7 CONCLUSION

In this paper, we present a high-performance CPU-GPU hybrid system for node embedding. Our system extends existing node embedding methods to GPUs and significantly accelerates training node embeddings on a single machine. With parallel online augmentation, *GraphVite* efficiently utilizes CPU threads to generate augmented edge samples for node embedding training. With parallel negative sampling, *GraphVite* enables training node embeddings on multiple GPUs without much synchronization. A collaboration strategy is also developed to reduce the synchronization cost between CPUs and GPUs. Experiments on 4 large networks prove that *GraphVite* significantly outperforms existing systems in speed without sacrifice on performance. In the future, we plan to generalize our system to semi-supervised settings and graph neural networks, such as graph convolutional networks [12], graph attention networks [34], and neural message passing networks [7].

ACKNOWLEDGEMENTS

We would like to thank Compute Canada [7] for supporting GPU servers. Jian Tang is supported by the Natural Sciences and Engineering Research Council of Canada and the Canada CIFAR AI Chair Program. We specially thank Wenbin Hou for useful discussions on C++ and GPU programming techniques, and Sahith Dambekodi for proofreading this paper.

[6] https://github.com/xgfs/verse
[7] https://www.computecanada.ca

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: a system for large-scale machine learning.. In *OSDI*, Vol. 16. 265–283.

[2] Natural Language Processing Lab at Tsinghua University. 2017. OpenNE: An open source toolkit for Network Embedding. https://github.com/thunlp/OpenNE.

[3] Prasad G Bhavana and Vineet C Nair. 2019. BMF: Block matrix approach to factorization of large scale data. *arXiv preprint arXiv:1901.00444* (2019).

[4] Shaosheng Cao, Wei Lu, and Qiongkai Xu. 2015. Grarep: Learning graph representations with global structural information. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. ACM, 891–900.

[5] Dan C Ciresan, Ueli Meier, Jonathan Masci, Luca Maria Gambardella, and Jürgen Schmidhuber. 2011. Flexible, high performance convolutional neural networks for image classification. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, Vol. 22. Barcelona, Spain, 1237.

[6] Rainer Gemulla, Erik Nijkamp, Peter J Haas, and Yannis Sismanis. 2011. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 69–77.

[7] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. 2017. Neural message passing for quantum chemistry. *arXiv preprint arXiv:1704.01212* (2017).

[8] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 855–864.

[9] Saurabh Gupta and Vineet Khare. 2017. BlazingText: Scaling and Accelerating Word2Vec using Multiple GPUs. In *Proceedings of the Machine Learning on HPC Environments*. ACM, 6.

[10] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584* (2017).

[11] Shihao Ji, Nadathur Satish, Sheng Li, and Pradeep Dubey. 2016. Parallelizing word2vec in multi-core and many-core architectures. *arXiv preprint arXiv:1611.06172* (2016).

[12] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[14] David Liben-Nowell and Jon Kleinberg. 2007. The link-prediction problem for social networks. *Journal of the American society for information science and technology* 58, 7 (2007), 1019–1031.

[15] Long-Ji Lin. 1993. *Reinforcement learning for robots using neural networks*. Technical Report. Carnegie-Mellon Univ Pittsburgh PA School of Computer Science.

[16] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. 2015. The graph structure in the web: Analyzed on different aggregation levels. *The Journal of Web Science* 1, 1 (2015), 33–47.

[17] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaev, Ganesh Venkatesh, et al. 2017. Mixed precision training. *arXiv preprint arXiv:1710.03740* (2017).

[18] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

[19] Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. ACM, 29–42.

[20] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[21] David Newman, Padhraic Smyth, Max Welling, and Arthur U Asuncion. 2008. Distributed inference for latent dirichlet allocation. In *Advances in neural information processing systems*. 1081–1088.

[22] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. (2017).

[23] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 701–710.

[24] Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Kuansan Wang, and Jie Tang. 2018. Network embedding as matrix factorization: Unifying deepwalk, line, pte, and node2vec. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. ACM, 459–467.

[25] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*. 693–701.

[26] Radim Rehurek and Petr Sojka. 2010. Software framework for topic modelling with large corpora. In *In Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Citeseer.

[27] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. 2008. Collective classification in network data. *AI magazine* 29, 3 (2008), 93–93.

[28] Trevor M Simonton and Gita Alaghband. 2017. Efficient and accurate Word2Vec implementations in GPU and shared-memory multicore architectures. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE, 1–7.

[29] Stergios Stergiou, Zygimantas Straznickas, Rolina Wu, and Kostas Tsioutsiouliklis. 2017. Distributed Negative Sampling for Word Embeddings.. In *AAAI*. 2569–2575.

[30] Damian Szklarczyk, John H Morris, Helen Cook, Michael Kuhn, Stefan Wyder, Milan Simonovic, Alberto Santos, Nadezhda T Doncheva, Alexander Roth, Peer Bork, et al. 2016. The STRING database in 2017: quality-controlled protein–protein association networks, made broadly accessible. *Nucleic acids research* (2016), gkw937.

[31] Jian Tang, Jingzhou Liu, Ming Zhang, and Qiaozhu Mei. 2016. Visualizing large-scale and high-dimensional data. In *Proceedings of the 25th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 287–297.

[32] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. 2015. Line: Large-scale information network embedding. In *Proceedings of the 24th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 1067–1077.

[33] Anton Tsitsulin, Davide Mottin, Panagiotis Karras, and Emmanuel Müller. 2018. VERSE: Versatile Graph Embeddings from Similarity Measures. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 539–548.

[34] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* 1, 2 (2017).

[35] Petar Veličković, William Fedus, William L Hamilton, Pietro Liò, Yoshua Bengio, and R Devon Hjelm. 2018. Deep graph infomax. *arXiv preprint arXiv:1809.10341* (2018).

[36] Daixin Wang, Peng Cui, and Wenwu Zhu. 2016. Structural deep network embedding. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 1225–1234.

[37] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems* 42, 1 (2015), 181–213.

[38] Hyokun Yun, Hsiang-Fu Yu, Cho-Jui Hsieh, SVN Vishwanathan, and Inderjit Dhillon. 2014. NOMAD: Non-locking, stOchastic Multi-machine algorithm for Asynchronous and Decentralized matrix completion. *Proceedings of the VLDB Endowment* 7, 11 (2014), 975–986.

[39] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. 2016. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160* (2016).

[40] Yong Zhuang, Wei-Sheng Chin, Yu-Chin Juan, and Chih-Jen Lin. 2013. A fast parallel SGD for matrix factorization in shared memory systems. In *Proceedings of the 7th ACM conference on Recommender systems*. ACM, 249–256.