DGL-KE: Training Knowledge Graph Embeddings at Scale

Da Zheng
dzzhen@amazon.com
AWS AI

Xiang Song
xiangsx@amazon.com
AWS Shanghai AI Lab

Chao Ma
manchao@amazon.com
AWS Shanghai AI Lab

Zeyuan Tan
zeyut@amazon.com
AWS Shanghai AI Lab

Jin Dong
jindon@amazon.com
AWS Shanghai AI Lab

Hao Xiong
xiongha@amazon.com
AWS Shanghai AI Lab

Zheng Zhang
zhaz@amazon.com
AWS Shanghai AI Lab

George Karypis
gkarypis@amazon.com
AWS AI

ABSTRACT

Knowledge graphs have emerged as a key abstraction for organizing information in diverse domains and their embeddings are increasingly used to harness their information in various information retrieval and machine learning tasks. However, the ever growing size of knowledge graphs requires computationally efficient algorithms capable of scaling to graphs with millions of nodes and billions of edges. This paper presents DGL-KE, an open-source package to efficiently compute knowledge graph embeddings. DGL-KE introduces various novel optimizations that accelerate training on knowledge graphs with millions of nodes and billions of edges using multi-processing, multi-GPU, and distributed parallelism. These optimizations are designed to increase data locality, reduce communication overhead, overlap computations with memory accesses, and achieve high operation efficiency. Experiments on knowledge graphs consisting of over 86M nodes and 338M edges show that DGL-KE can compute embeddings in 100 minutes on an EC2 instance with 8 GPUs and 30 minutes on an EC2 cluster with 4 machines with 48 cores/machine. These results represent a 2× ∼ 5× speedup over the best competing approaches. DGL-KE is available on https://github.com/awslabs/dgl-ke.

CCS CONCEPTS

• Information systems Data mining: Retrieval models and ranking;
Web searching and information discovery.

KEYWORDS

knowledge graph, large scale, distributed training

1 INTRODUCTION

Knowledge graphs (KGs) are data structures that store information about different entities (nodes) and their relations (edges). They are used to organize information in many domains such as music, movies, (e-)commerce, and sciences. A common approach of using KGs in various information retrieval and machine learning tasks is to compute knowledge graph embeddings (KGE) [4, 18]. These approaches embed a KG’s entities and relation types into a d-dimensional space such that the embedding vectors associated with the same entity and relation type are close to each other. Numerous models for computing knowledge graph embeddings have been developed, such as TransE [2], TransR [10] and DistMult [20].

As the size of KGs has grown, so has the time required to compute their embeddings. As a result, a number of approaches and software packages have been developed that exploit concurrency in order to accelerate the computations. Among them are GraphVite [21], which parallelizes the computations using multi-GPU training; BigGraph (PBG) [9], which uses distributed training to split the computations across a cluster of machines. However, these approaches suffer from high data-transfer overheads and low computational efficiency. As a result, the time required to compute embeddings for large KGs is high.

In this paper we present various optimizations that accelerate KGE training on knowledge graphs with millions of nodes and billions of edges using multi-processing, multi-GPU, and distributed parallelism. These optimizations are designed to increase data locality, reduce communication overhead, overlap computations with memory accesses, and achieve high operation efficiency.

We introduce novel approaches of decomposing the computations across different computing units (cores, GPUs, machines) that enable massive parallelization while reducing write conflicts and communication overhead. The write conflicts are reduced by partitioning the processing associated with different relation types across the computing units as well as reducing data communication on multi-GPU...
training. The communication overhead is reduced by using a min-
cut-based graph partitioning algorithm (METIS [6]) to distribute
the knowledge graph across the machines. For entity embeddings,
we introduce massive asynchronicity by having separate processes
to compute the gradients of embeddings independently as well as
allowing entity embedding updates overlapped with mini-batch com-
putation. Finally, we use various negative sampling strategies to
construct mini-batches with a small number of embeddings involved
in a batch, which reduces data movement from memory to computing
units (e.g., CPUs and GPUs).

We implement an open-source KGE package called DGL-KE
that incorporates all of the optimization strategies to train KG em-
bettions on large KGs efficiently. The package is implemented
with Python on top of Deep Graph Library (DGL) [19] along with
a C++-based distributed key-value store specifically designed for
DGL-KE. We rely on DGL to perform graph-related computation,
such as sampling, and rely on existing deep learning frameworks,
such as Pytorch [13] and MXNet [3], to perform tensor computation.
DGL-KE is available at https://github.com/awslabs/dgl-ke.

We experimentally evaluate the performance of DGL-KE on
different knowledge graphs and compare its performance against
GraphVite and Pytorch-BigGraph. Our experiments show that DGL-
KE is able to compute embeddings whose quality is comparable to
that of competing approaches at a fraction of their time. In particular,
on knowledge graph containing over 86M nodes and 338M edges
DGL-KE can compute the embeddings in 100 minutes on an EC2
instance with 8 GPUs and 30 minutes on an EC2 instance with 4 ma-
achines with 48 cores/machine. These results represent a 5× and 2× speedup over the time required by GraphVite and Pytorch-BigGraph,
respectively.

2 BACKGROUND

Definitions & Notation. A graph is composed of vertices and
edges $G = (V, E)$, where $V$ is the set of vertices and $E$ is the set of
edges. A knowledge graph (KG) is a special type of graph whose
vertices and edges have types. It is a flexible data structure that repre-
sests entities and their relations in a dataset. A vertex in a knowledge
graph represents an entity and an edge represents a relation between
two entities. The edges are usually in the form of triplets $(h, r, t)$,
each of which indicates that a pair of entities $h$ (head) and $t$ (tail)
are coupled via a relation $r$.

Knowledge graph embeddings are low-dimensional representa-
tion of entities and relations. These embeddings carry the infor-
mation of the entities and relations in the knowledge graph and
are widely used in tasks, such as knowledge graph completion and
recommendation. Throughput the paper, we denote the embedding
vector of head entity, tail entity and relation with $h$, $t$ and $r$, respec-
tively; all the embedding have the same dimension size of $d$.

Knowledge Graph Embedding (KGE) Models. KGE models train
entity embeddings and relation embeddings in a knowledge graph.
They define a score function on the triplets and optimize the function
to maximize the scores on triplets that exist in the knowledge graph
and minimize the scores on triplets that do not exist.

Many score functions have been defined to train knowledge graph
embeddings [17] and Table 1 lists the ones used by the KGE models
supported by DGL-KE. TransE and TransR are two representative

| Table 1: Knowledge graph models. $M_r$ is a relation-specific pro-
|jection matrix. TransE uses L1 or L2 norm in its score function. |
| Models       | score function $f(h, r, t)$          |
| TransE [2]   | $-||h + r - t||_2$                 |
| TransR [10]  | $-||M_r h + r - M_t t||_2^2$       |
| DistMult [20] | $h' \text{diag}(r) t$              |
| ComplEx [16] | $\text{Re} \left(h' \text{diag}(r) t\right)$ |
| RESCAL [12]  | $h' M_t t$                       |
| RotatE [15]  | $-||h \circ r - t||_2^2$           |

translational distance models, where we use L1 or L2 to define the
distance. DistMult, ComplEx, and RESCAL are semantic matching
models that exploit similarity-based scoring functions. Some of the
models are much more computationally expensive than other models.
For example, TransR is $d$ times more computationally expensive
than TransE because TransR has additional matrix multiplications on
both head and tail entity embeddings, instead of just element-wise
operations on embeddings in TransE.

To train a KGE model, we define a loss functions on a set of
positive and negative samples from the knowledge graph. Two loss
functions are commonly used. The first is the the loss function given by

$$
\text{minimize} \sum_{h, r, t \in D^+} \log (1 + \exp(-y \times f(h, r, t))),
$$

where $D^+$ and $D^-$ are the positive and negative sets of triplets,
respectively and $y$ is the the label of a triplet, +1 for positive and −1
for negative. The second is the pairwise ranking loss given by

$$
\text{minimize} \sum_{h, r, t \in D^+, h', r', t' \in D^=} \max(0, y - f(h, r, t) + f(h', r', t')).
$$

A common strategy of generating negative samples is to corrupt a
triplet by replacing its head entity or tail entity with entities sampled
from the graph with some heuristics to form a negative sample
$(h, r, t')$ or $(h', r, t)$, where $h'$ and $t'$ denote the randomly sampled
entities. Potentially, we can corrupt the relation in a triplet. In this
work, we only corrupt entities to generate negative samples.

**Mini-batch training and Asynchronous updates.** A KGE model
is typically trained in a mini-batch fashion. We first sample a mini-
batch of $b$ triplets $(h, r, t)$ that exist in the knowledge graph. The
mini-batch training is sparse because a batch only involves in a small
number of entity embeddings and relation embeddings. We can take
advantage of the sparsity and train KGE models asynchronously
with sparse gradient updates [14]. That is, we sample multiple mini-
batches independently, perform asynchronous stochastic gradient
descent (SGD) on these mini-batches in parallel and only update
the embeddings involved in the mini-batches. This training strategy
maximizes parallelization in mini-batch training but may lead to
conflicts in updating gradients. When two mini-batches run simulta-
neously, they may use the same entity or relation embeddings. In this
case, the gradient of the embeddings is computed based on the stale
information, which results in a slower convergence or not converging
to the same local minimum.
3 METHODS

A naive implementation of KGE training results in low computation-to-memory density for many KGE models, which prevents us from using computation resources efficiently. When performing computation on a batch, we need to move a set of entity and relation embeddings to computation resources (e.g., CPUs and GPUs) from local CPU memory or remote machines. For example, for a mini-batch with $b$ positive triplets, $k$ negative triplets, and $d$-dimensional embeddings, both the computational and data movement complexity of TransE is $O(bd(k + 1))$, resulting in a computational density of $O(1)$. Given that computations are faster than memory accesses, reducing data movement is key to achieving efficient KGE training.

In addition, we need to take advantage of parallel computing resources. This includes multi-core CPUs, GPUs and a cluster of machines. Our training algorithm needs to allow massive parallelization while still minimizing conflicts when updating embeddings in parallel.

In this work, we implement DGL-KE on top of DGL [19], completely with Python. It relies on DGL for graph computation, such as sampling, and relies on deep learning frameworks, such as Pytorch and MXNet, for tensor operations.

3.1 Overview

DGL-KE provides a unified implementation for efficient KGE training on different hardware. It optimizes for three types of hardware configurations: (i) many-core CPU machines, (ii) multi-GPU machines, and (iii) a cluster of CPU/GPU machines. In each type of the hardware, DGL-KE parallelizes the training with multiprocessing to fully utilize the parallel computation power of the hardware.

For all different hardware configurations, the training process starts with a preprocessing step to partition a knowledge graph and follows with mini-batch training. The partitioning step assigns a disjoint set of triplets in a knowledge graph to a process so that the process performs mini-batch training independently.

The specific steps performed during each mini-batch are:

1. Samples triplets from the local partition that belongs to a process to form a mini-batch and constructs negative samples in the mini-batch.
2. Fetches entity and relation embeddings that are involved in the mini-batch from the global entity and relation embedding tensors.
3. Performs forward computation and back-propagation on the embeddings fetched in the previous step in order to compute the gradients of the embeddings.
4. Applies the gradients to update the embeddings involved in the mini-batch. This step requires to apply an optimization algorithm to adjust the gradients and write the gradients back to the global entity and relation embedding tensors.

KGE training on a knowledge graph involves two types of data: the knowledge graph structure and the entity and relation embeddings. As illustrated in Figure 1, we deploy different data placement for different hardware configurations. In many-core CPU machines, DGL-KE keeps the knowledge graph structure as well as entity and relation embeddings in shared CPU memory accessible to all processes. A trainer process reads the entity and relation embeddings from the global embeddings directly through shared memory. In multi-GPU machines, DGL-KE keeps the knowledge graph structure and entity embeddings in shared CPU memory because entity embeddings are too large to fit in GPU memory. It may place relation embeddings in GPU memory to reduce data communication. As such, a trainer process reads entity embeddings from CPU shared memory and reads relation embeddings directly from GPU memory. In a cluster of machines, DGL-KE implements a C++-based distributed key-value store (KVStore) to store both entities and relation embeddings. The KVStore partitions the entity embeddings and relation embeddings automatically and stripes them across all KVStore servers. A trainer process accesses embeddings from distributed KVStore with the pull and push API. We partition the knowledge graph structure and each trainer machine stores a partition of the graph. The graph structure of the partition is shared among all trainer processes in the machine.

The rest of this section describes various optimization techniques that we developed in DGL-KE: graph partitioning in the preprocessing step (Section 3.2), negative sampling (Section 3.3), data access to relation embeddings (Section 3.4), and finally applying gradients to the global embeddings (Section 3.5).

3.2 Graph partitioning

In distributed training, we partition the graph structure and embeddings and store them across the machines of the cluster. During the training, each machine may need to read entity and relation embeddings from other machines to construct mini-batches. The key of optimizing distributed training is to reduce communication required to retrieve and update entity and relation embeddings.

To reduce the communication caused by entity embeddings in a batch, we deploy METIS partitioning [6] on the knowledge graph in the preprocessing step. For a cluster of $P$ machines, we split the graph into $P$ partitions so that we assign a METIS partition (all entities and triplets incident to the entities) to a machine as shown in Figure 2. With METIS partitioning, the majority of the triplets are in the diagonal blocks. We co-locate the embeddings of the entities with the triplets in the diagonal block by specifying a proper data partitioning in the distributed KVStore. When a trainer process samples triplets in the local partition, most of the entity embeddings accessed by the batch fall in the local partition and, thus, there is little network communication to access entity embeddings from other machines.

3.3 Negative sampling

KGE training samples triplets to form a batch and construct a large number of negative samples for each triplet in the batch. For all different hardware, DGL-KE performs sampling on CPUs and offloads the entire sampling computation to DGL for efficiency. If we construct negative samples independently for each triplet, a mini-batch will contain many entity embeddings, which results in accessing many embeddings.

We deploy a joint negative sampling to reduce the number of entities involved in a mini-batch. In this approach, instead of independently corrupting every triplet $k$ times, we group the triplets into sets of size $g$ and corrupt them together. For example, when corrupting the tail entities of a set, we uniformly sample $k$ entities to replace the tail entities of that set. We corrupt the head entities in a
Figure 1: The optimized data placement of DGL-KE in three different parallel hardware.

Figure 2: Adjacent matrix of a large graph after applying METIS partitioning, indexed by machine partition. Note that majority of the edges fall within a partition. As a result, the adjacency matrix has majority of non-zeros lying on the diagonal blocks.

This negative sampling strategy introduces two benefits. First, it reduces the number of entities involved in a mini-batch, resulting in a smaller amount of data access. For a $d$-dimensional embedding, each mini-batch of size $b$ now only needs to access $O(bd + bkd/g)$ instead of $O(bd(k + 1))$ words of memory. When $g$ grows as large as $b$, the amount of data accessed by this negative sampling is about $b$ times smaller ($b$ is usually in the order of 1000). This benefit is more significant in multi-GPU training because we store entity embeddings in CPU memory and send the entity embeddings to the GPUs in every mini-batch. Second, it allows us to replace the original computation with more efficient tensor operations. Inside a group of negative samples, head entities and tail entities are densely connected. We now divide the computation of a score function on a negative sample into two parts. For example, the score function of TransE, $-||h + r - t||_2$, is divided into $o = h + r$ and $-||o - t'||_2$. The vector $o$ is computed as before because there are only $b$ pairs of $h$ and $r$. The computation of $-||o - t'||_2$ is converted into a generalized matrix multiplication, which can be performed using highly optimized math libraries. There are $bk$ pairs of $o$ and $t'$.

We also deploy non-uniform negative sampling with a probability proportional to the degree of each entity (PBG uses a similar strategy). On a large knowledge graph, uniform negative sampling results in easy negative samples [8]. One way of constructing harder negative samples is to corrupt a triplet with entities sampled proportional to the entity degree. In order to do this efficiently, instead of sampling entities from the entire graph, we construct negative samples with the entities that are already in the mini-batch. This is done by uniformly sampling some of the mini-batch’s triplets and connecting the sampled head (tail) entities with the tail (head) entities of the mini-batch’s triplets to construct the negative samples. Note that this uniform triplet sampling approach leads to an entity sampling approach that is proportional to the entity degree in the mini-batch. In practice, we combine these negative samples with uniformly negative samples to form the full set of negative samples for a mini-batch.

In the distributed training, we sample entities from the local METIS partition to corrupt triplets in a mini-batch to minimize the communication caused by negative samples. This ensures that negative samples do not increase network communication. This strategy in general results in harder negative samples. The corrupted head/tail entities sampled from the local METIS partition are topologically closer to the tail/head entities of the triplets in the batch.

3.4 Relation partitioning

Both GraphVite and PBG treat relation embeddings as dense model weights. As a result, for each mini-batch they incur the cost of retrieving them and updating them. If the number of relations in the knowledge graph is small, this is close to optimal and does not impact the performance. However, when the knowledge graph has a large number of relations (greater than the mini-batch size; $\approx 1000$), the number of distinct relations in each mini-batch will be a subset of them and as such, treating them as dense model weights will result in unnecessary data access/transfer overheads. To address this limitation, DGL-KE performs sparse relation embedding reads and
sparse gradient updates on relation embeddings. This significantly reduces the amount of data transferred in multi-processing, multi-GPU, and distributed training.

To further reduce the amount of access to relation embeddings in a mini-batch, DGL-KE decomposes the computations among the computing units by introducing a novel relation partitioning approach. This relation partitioning tries (i) to equally distribute the triplets and the relations among the partitions and (ii) to minimize the number of distinct relations that are assigned to each partition as a result of (i). The first goal ensures that the computational and memory requirements are balanced across the computing units, whereas the second goal ensures that the relation-related data that needs to be transferred is minimized. In order to derive such a relation partitioning, we use the following fast greedy algorithm. We sort the relations based on their frequency in non-increasing order. We iterate over the sorted relations and greedily assign a relation to the partition with the smallest number of triplets so far. This strategy usually results in balanced partitioning while ensuring that each relation belongs to only one partition. However, the above algorithm will fail to produce a balance partitioning when the knowledge graph contains relations that are very frequent. In such cases, the number of triplets for those relations may exceed the partition size. To avoid load imbalance, we equally split the most common relations across all partitions. After relation partitioning, we assign a relation partition to a computing unit. This ensures that the majority of relation embeddings are updated by only one process at a time. This optimization applies to many-CPU-core training and multi-GPU training.

A potential drawback of relation partitioning is that it restricts the relations that may appear inside a mini-batch. This reduces the randomization of stochastic gradient descent, which can impact the quality of the embeddings. To tackle this problem, we introduce randomization in the partitioning algorithm and at the start of each epoch we compute a somewhat different relation partitioning.

When we use relation partitioning in multi-GPU training, we store all relation embeddings on GPUs and update relation embeddings in GPUs locally. This is particularly important for KGE models with large model weights on relations, such as TransR and RESCAL. Take TransR for an example. It has an entity projection matrix on each relation, which is much larger than a relation embedding. Moving them to CPU is the bottleneck of the entire computation. If we keep all of these projection matrices in GPUs, the communication overhead drops from $O(bd^2)$ to $O(bd)$, which is significantly smaller than the naive solution, usually in the order of 100 times smaller.

### 3.5 Overlap gradient update with batch processing

In multi-GPU training, some of the steps in a mini-batch computation run on CPUs while the others run on GPUs. When we run them in serial in a process, the GPU remains idle when the CPU writes the gradients. To avoid GPU idling, we overlap entity embedding update with the batch computation in the next mini-batch. This allows us to overlap the computation in CPUs and GPUs. Note that even though this approach can potentially increase the staleness of the embeddings used in a mini-batch, the likelihood of that happening is small for knowledge graphs with a sufficiently large number of entities relative to the number of training processes.

To perform this optimization, we split the gradient updates into two parts: one involving relation embeddings, which are updated by the trainer process, and the other involving the entity embeddings, which are off-loaded to a dedicated gradient update process for each trainer process. Once the trainer process finishes writing the relation gradients, it proceeds to the next mini-batch, without having to wait for the writing of the entity gradients to finish. Our experiments show that overlapping gradient updates provide 40% speedup for most of the KGE models on Freebase.

### 3.6 Other optimizations

**Periodic synchronization among processes.** When training KGE models with multiprocessing completely independently, different processes may run at a different rate, which results in inconsistent model accuracy. We observe that the trained embeddings sometimes have much worse accuracy at some runs. As such, we add a synchronization barrier among all training processes after a certain number of batches to ensure that all processes train roughly at the same rate. Our observation is that the model can be trained stably if processes synchronize after every few thousand batches.

**Distributed Key-Value store.** In DGL-KE, we implement a KVStore for model synchronization with efficient C++ back-end. It uses three optimizations that are designed specifically for distributed KGE training. First, because the relations in some knowledge graphs have a long-tail distribution, it reshuffles the relation embeddings in order to avoid single hot-point of KVStore. Second, DGL-KE uses local shared-memory access instead of network communication if the worker processes and KVStore processes are on the same machine. This optimization can significantly reduce networking overhead especially on METIS graph partition. Third, it launches multiple KVStore servers in a single machine to parallelize the computation in KVStore. All KVStore servers inside a machine share embeddings via local shared-memory. Finally, similar to the optimization we used in multi-GPU training, the gradient communication and local gradient computation will be overlapped in KVStore.

### 4 RELATED WORK

There are a few packages that have been developed to compute embeddings of knowledge graphs efficiently and scale to large knowledge graphs.

OpenKE [5] is one of the first packages for training knowledge graph embeddings and provides a large list of models. However, it is implemented entirely in Python and cannot scale to very large graphs.

Pytorch-BigGraph (PBG) [9] is developed with an emphasis on scalability to large graphs and distributed training on a cluster of machines. The package does not support GPU training. Although PBG and DGL-KE share similar negative sampling strategies, PBG applies different strategies for distributed training. It randomly divides the adjacency matrix of the graph into 2D blocks and assigns blocks to each machine based on a schedule that avoids conflicts with respect to the entity embeddings. It treats entity embeddings as sparse model weights and relation embeddings as dense model weights. The random 2D partitioning along with the use of dense model weights for relation embeddings results in a large amount of
We evaluated the performance of the different KGE models and used three datasets to evaluate and compare the performance of the models. We conduct our evaluation on EC2 CPU and GPU instances, including GraphVite and PBG; see Table 2 for machine configurations.

## 5 EXPERIMENTAL METHODOLOGY

DGL-KE is implemented in Python and relies on PyTorch for tensor operations, as is the case in PBG, whereas GraphVite focuses on multi-GPU training and does not support distributed training. When it trains a large knowledge graph, it keeps embeddings on CPU memory. It constructs a subgraph, moves all data in the subgraph to the GPU memory and performs many mini-batch training steps on the subgraph. This method reduces data movement between CPUs and GPUs at the cost of increasing the staleness of the embeddings, which usually results in slower convergence.

### 5.1 Hardware platform

We performed the link-prediction task using two different protocols. The first, which was used for FB15k and WN18, works as follows. For each triplet $(h, r, t)$ in the validation/test set, referred to as a positive triplet, we generated all possible triplets of the form $(h', r, t)$ and $(h, r, t')$ by corrupting the head and tail entities. We then removed from them any triplets that already exist in the dataset. The set of triplets that remained formed the negative triplets associated with the initial positive triplet. We then used the score function of the model in question (Table 1) to score the triplets. The second protocol, which was used for Freebase, is similar to the first one with the following two differences: (i) we use only 2000 negative triplets; 1000 sampled uniformly from the entire set of negative samples and 1000 sampled proportionally to the degree of the corrupted entities; and (ii) we did not remove from the 2000 negative triplets any triplets that are in the dataset. Note that the reason for the second protocol was due to the size of Freebase, which made the first protocol computationally expensive.

We assessed the performance by using the standard metrics [9] of $\text{Hit}_k$ (for $k \in \{1, 3, 10\}$), Mean Rank (MR), and Mean Reciprocal Rank (MRR). All these metrics are derived by comparing how the score of the positive triplet relates to the scores of its associated negative instances. For a positive triplet $t$, let $S_t$ be the list of triplets containing $t$ and its associated negative triplets ordered in a non-increasing score order, and let $\text{rank}_i$ be the position of $t_i$ in $S_t$. Given that, $\text{Hit}_k$ is the average number of times the positive triplet is among the $k$ highest ranked triplets; $\text{MR}$ is the average rank of the positive instances, whereas $\text{MRR}$ is the average reciprocal rank of the positive instances. Mathematically, they are defined as

\[
\text{Hit}_k = \frac{1}{Q} \sum_{i=1}^{Q} \chi_{\text{rank}_i \leq k},
\]

\[
\text{MR} = \frac{1}{Q} \sum_{i=1}^{Q} \text{rank}_i,
\]

and

\[
\text{MRR} = \frac{1}{Q} \sum_{i=1}^{Q} \frac{1}{\text{rank}_i},
\]

where $Q$ is the total number of positive triplets and $\chi_{\text{rank}_i \leq k}$ is 1 if $\text{rank}_i \leq k$, otherwise it is 0. Note that $\text{Hit}_k$ and $\text{MRR}$ are between 0 and 1, whereas $\text{MR}$ ranges from 1 to the $\sum_i^Q |S_i|$.

### 5.2 Datasets

We used three datasets to evaluate and compare the performance of DGL-KE against that of GraphVite and PBG. Table 3 shows various statistics for these datasets. FB15k and Freebase were derived from the Freebase Knowledge Graph [1], whereas WN18 was derived from WordNet [11]. The FB15k and WN18 datasets are standard benchmarks for evaluating KGE methods. The Freebase dataset corresponds to the Freebase Knowledge Graph. All datasets are downloaded from [7].

### 5.3 Evaluation methodology

We evaluated the performance of the different KGE models and methods using a link (relation)-prediction task. In order to train and evaluate the models, we split each dataset into training, validation, and test subsets. For FB15k and WN18, we used the same splits that were used in previous evaluations [15] (available in [7]). Freebase is split with 5% of the triplets for validation, 5% for test, and the remaining 90% for training (also available in [7]).

Table 2: Evaluation hardware configuration.

| EC2 Type     | Hardware Config                        | Eval Section |
|--------------|----------------------------------------|--------------|
| c5d.24xlarge | 2x24 cores, 700GB RAM, 100Gbps network | sec 6.2, 6.3 |
| p3.16xlarge  | 2x16 cores, 500GB RAM, 8 V100 GPUs     | sec 6.1      |

Table 3: Knowledge graph datasets.

| Dataset   | # Vertices | # Edges | #Relations | Systems          |
|-----------|------------|---------|------------|------------------|
| FB15k [2] | 14,951     | 592,213 | 1,345      | DGL-KE, GraphVite|
| WN18 [2]  | 40,943     | 151,442 | 18         | DGL-KE, GraphVite|
| Freebase [1] | 86,054,151 | 338,586,276 | 14,824 | DGL-KE, PBG |

We run Ubuntu 18.04 on all EC2 instances, where the Python version is 3.6.8 and Pytorch version is 1.3.1. On GPU instances, the CUDA version is 10.0. When comparing the performance of DGL-KE against that of GraphVite and PBG, we use GraphVite v0.2.1 downloaded from Github on November 12 2019 and PBG downloaded from their Github repository on October 15 2019. All frameworks use the same Pytorch version.

### 5.4 Software environment

We performed the link-prediction task using two different protocols. The first, which was used for FB15k and WN18, works as follows. For each triplet $(h, r, t)$ in the validation/test set, referred to as a positive triplet, we generated all possible triplets of the form $(h', r, t)$ and $(h, r, t')$ by corrupting the head and tail entities. We then removed from them any triplets that already exist in the dataset. The set of triplets that remained formed the negative triplets associated with the initial positive triplet. We then used the score function of the model in question (Table 1) to score the triplets. The second protocol, which was used for Freebase, is similar to the first one with the following two differences: (i) we use only 2000 negative triplets; 1000 sampled uniformly from the entire set of negative samples and 1000 sampled proportionally to the degree of the corrupted entities; and (ii) we did not remove from the 2000 negative triplets any triplets that are in the dataset. Note that the reason for the second protocol was due to the size of Freebase, which made the first protocol computationally expensive.

We assessed the performance by using the standard metrics [9] of $\text{Hit}_k$ (for $k \in \{1, 3, 10\}$), Mean Rank (MR), and Mean Reciprocal Rank (MRR). All these metrics are derived by comparing how the score of the positive triplet relates to the scores of its associated negative instances. For a positive triplet $t$, let $S_t$ be the list of triplets containing $t$ and its associated negative triplets ordered in a non-increasing score order, and let $\text{rank}_i$ be the position of $t_i$ in $S_t$. Given that, $\text{Hit}_k$ is the average number of times the positive triplet is among the $k$ highest ranked triplets; $\text{MR}$ is the average rank of the positive instances, whereas $\text{MRR}$ is the average reciprocal rank of the positive instances. Mathematically, they are defined as

\[
\text{Hit}_k = \frac{1}{Q} \sum_{i=1}^{Q} \chi_{\text{rank}_i \leq k},
\]

\[
\text{MR} = \frac{1}{Q} \sum_{i=1}^{Q} \text{rank}_i,
\]

and

\[
\text{MRR} = \frac{1}{Q} \sum_{i=1}^{Q} \frac{1}{\text{rank}_i},
\]

where $Q$ is the total number of positive triplets and $\chi_{\text{rank}_i \leq k}$ is 1 if $\text{rank}_i \leq k$, otherwise it is 0. Note that $\text{Hit}_k$ and $\text{MRR}$ are between 0 and 1, whereas $\text{MR}$ ranges from 1 to the $\sum_i^Q |S_i|$.

### 5.5 Hyperparameters

For the FB15k and WN18 and all methods (DGL-KE and GraphVite) we performed an extensive hyper-parameter search and report the results that achieve the best performance in terms of MRR, as we believe it is a good measure to assess the overall performance of the
methods. Due to the size of Freebase, we only report results for a single set of hyper-parameter values. We use the hyperparameters that perform the best on FB15k for Freebase.

To ensure that the accuracy results are comparable, all methods used exactly the same test set and evaluation protocols described in the previous section.

6 RESULTS

6.1 Multi-GPU training

Both memory and computing capacity on a multi-GPU machine have a diverse set of characteristics, which make the various optimizations described in Sections 3.3–3.6 relevant. A detailed evaluation of these optimizations follows.

6.1.1 Negative sampling. Joint negative sampling shown in Section 3.3 has two effects: (i) enable more efficient tensor operators and (ii) reduce data movement in multi-GPU training. Figure 3 shows the result. To illustrate the speedup of using more efficient tensor operators, we run the TransE model on FB15k with all data in a single GPU. Joint negative sampling gives about $4 \times$ speedup. To illustrate the speedup of reducing data movement, we run the TransE model on FB15k in 8 GPUs, where the entity embeddings are stored in CPU memory. Joint negative sampling gets much larger speedup, e.g., about $40 \times$, because naive sampling requires swapping many more entity embeddings between CPU and GPU than joint negative sampling and data communication becomes the bottleneck.

6.1.2 Degree-based negative sampling. Although degree-based negative sampling does not speed up training, it improves the model accuracy (Table 4) on Freebase. This suggests that non-uniform negative sampling to generate “hard” negative samples is effective, especially on large knowledge graphs.

6.1.3 Overlap gradient update with batch computation. This technique overlaps the computation of GPUs and CPUs to speed up the training. Figure 4 shows the speedup of using this technique (comparing sync and async) on FB15k and Freebase. It has limited speedup on small knowledge graphs for some models, but it has roughly 40% speedup on Freebase for almost all models. The effectiveness of this optimization depends on the computation time in CPUs and GPUs. Large knowledge graphs, such as Freebase, requires hundreds of GBytes to store the entire entity embeddings and suffers from slow random memory access during entity embedding update. In this case, overlapping the CPU/GPU computation benefits a lot.

6.1.4 Relation partitioning. After relation partitioning, we pin relation embeddings (and projection matrices) in each partition inside certain GPU, which reduces the data movements between CPUs and GPUs. The speedup is highly related to the model size and the number of relations in the dataset. Figure 4 shows the speedup of using relation partitioning in multi-GPU training (comparing async and async + rel_part bar) on FB15k and Freebase. For example, relation partitioning has significant speedup on TransR because the relation-specific projection matrices result in a large amount of data communication between CPU and GPU. Even for models with only relation embeddings, relation partitioning in general gets over 10% speedup.

6.1.5 Overall speed and accuracy. After deploying all of the optimizations evaluated above, we measure the speedup of DGL-KE with multiple GPUs on both FB15k and Freebase. Figure 5 shows that DGL-KE accelerates training almost linearly with multiple GPUs. On Freebase, DGL-KE further speeds up by running 16 processes on 8 GPUs. By running two processes on each GPU, we better utilize the computation in GPUs and PCIe buses by overlapping computation and data communication between CPUs and GPUs.
With all these techniques, we train KGE models efficiently. For small knowledge graphs, such as FB15k, DGL-KE trains most of KGE models, even as complex as RotatE and TransR, within a few minutes. For large knowledge graphs, such as Freebase, DGL-KE trains many of KGE models around one or two hours and trains more complex models within a reasonable time, for example we train TransR in about 8 hours using 8 GPUs.

With a maximum speedup of $11 \times$ with single-GPU training, we sacrifice little on accuracy. Table 5 and Table 6 shows the accuracy of DGL-KE with 1 and 8 GPUs on FB15k and Freebase. The $1\text{GPU}$ columns shows the baseline accuracy and the $Fastest$ shows the accuracy with the fastest configuration on 8 GPUs. For FB15k, we achieve the fastest training speed with 8 processes on 8 GPUs, while for Freebase, we use 8 GPUs and 16 concurrent processes. In all experiments, the total number of epochs we run is the same for both the $1\text{GPU}$ and $Fastest$ settings. Here, we only show TransR with 8 GPUs on Freebase because training TransR on one GPU takes very long time.

### 6.2 Many-core training

Many of the techniques illustrated in multi-GPU training can also be applied to multi-core training. Figure 6 shows that DGL-KE scales well on an r5dn instance with 48 CPU cores. The training accuracy result of TransE and DistMult with 48 CPU cores is shown in Table 7 (column labeled “Single”).

### 6.3 Distributed training

In distributed training, we use 4 r5dn.24xlarge EC2 instances as our cluster environment. In this section, we compare the baseline single-machine training with distributed training using both random partitioning and METIS partitioning on Freebase.

METIS partitioning on distributed training gets nearly $3.5 \times$ speedup compared with the single-machine baseline (Figure 7) without sacrificing any model accuracy (Table 7). The training speed of using METIS partitioning get about 20% speedup over random partitioning because METIS partitioning leads to much lower overhead than random partitioning.

### 6.4 Overall performance

We evaluate DGL-KE on the datasets in Table 3 and compare with two existing packages: GraphVite and PBG on both CPUs and GPUs. Because GraphVite and PBG only provide a subset of the models in
Table 5: The overall performance of DGL-KE after various optimizations with 8 GPUs on FB15k.

| Model     | 1GPU Fastest | 1GPU Fastest | 1GPU Fastest | 1GPU Fastest | 1GPU Fastest | 1GPU Fastest |
|-----------|--------------|--------------|--------------|--------------|--------------|--------------|
| TransE_l1 | 0.860        | 0.857        | 0.848        | 0.879        | 0.892        | 0.884        |
| DistMult  | 0.857        | 0.806        | 0.796        | 0.838        | 0.823        | 0.819        |
| ComplEx   | 0.848        | 0.636        | 0.614        | 0.724        | 0.698        | 0.665        |
| RotatE    | 0.879        | 0.664        | 0.732        | 0.716        | 0.789        | 0.769        |
| TransR    | 0.892        | 0.789        | 0.716        | 0.789        | 0.752        | 0.736        |

Table 6: The overall performance of DGL-KE after various optimizations with 8 GPUs on Freebase.

| Model     | 1GPU Fastest | 1GPU Fastest | 1GPU Fastest | 1GPU Fastest | 1GPU Fastest | 1GPU Fastest |
|-----------|--------------|--------------|--------------|--------------|--------------|--------------|
| TransE_l2 | 0.865        | 0.822        | 0.839        | 0.837        | 0.837        | 0.830        |
| DistMult  | 0.822        | 0.785        | 0.780        | 0.785        | 0.773        | 0.668        |
| ComplEx   | 0.839        | 0.785        | 0.780        | 0.785        | 0.668        | 0.653        |
| RotatE    | 0.837        | 0.773        | 0.780        | 0.785        | 0.773        | 0.668        |
| TransR    | 0.837        | 0.773        | 0.780        | 0.785        | 0.773        | 0.668        |

Table 7: The accuracy of random graph partition and METIS graph partition for distributed training.

| Model     | Single Random | METIS Single Random | METIS Single Random | METIS Single Random |
|-----------|--------------|---------------------|---------------------|---------------------|
| TransE    | 0.796        | 0.790               | 0.751               | 0.739               |
| DistMult  | 0.734        | 0.735               | 0.726               | 0.712               |
| ComplEx   | 0.634        | 0.689               | 0.634               | 0.696               |
| MR        | 54.51        | 64.05               | 34.59               | 123.1               |
| MRR       | 0.696        | 0.726               | 0.692               | 0.68                |

Figure 8: The runtime of PBG and DGL-KE on Freebase.

DGL-KE, we only compare with them with the models available in these two packages.

6.4.1 Comparison with GraphVite. DGL-KE is consistently faster than GraphVite on both FB15k and WN18 (Figure 9 and Figure 10) when training all KGE models to reach similar accuracy (Table 8 and Table 9). For most of the models, DGL-KE is 5x faster than GraphVite. This is mainly due to DGL-KE converges faster than GraphVite. In all cases, DGL-KE only needs less than 100 epochs to converge but GraphVite needs thousands of epochs. When evaluating GraphVite, we use the recommended configuration by the package for each algorithm when running on 1 GPU and 4 GPUs, while having some hyperparameter tuning for 8 GPUs to get compatible results with 1 GPU runs. When evaluating DGL-KE, we use the same dimension size of entity and relation embedding as GraphVite, but tune hyper-parameters such as learning rate, negative sample size and batch size, for better accuracy.

6.4.2 Comparison with PBG. DGL-KE runs twice as fast as PBG when training KGE models on Freebase (Figure 8). There are many factors that contribute to the slower training speed in PBG. One of the major factors is that PBG handles relation embeddings as dense model weights. As such, the computation in a batch involves in all relation embeddings in the graph, which is 10 times more than necessary on Freebase. In contrast, DGL-KE reduces the number of
We develop an efficient package called DGL-KE to train knowledge graph embeddings on a variety of hardware, including many-core CPU, multi-GPU as well as cluster of machines. Our experiments show that DGL-KE scales well with machine resources almost linearly while still achieving very high model accuracy. DGL-KE significantly outperforms the state-of-the-art packages for optimization techniques to improve locality, reduce data communication, while harnessing parallel computing capacity. As a result, DGL-KE GraphVite almost linearly while still achieving very high model accuracy. DGL-KE GraphVite

### Table 8: The accuracy of DGL-KE and GraphVite on FB15k with 1, 4 and 8 GPUs.

|          | DGL-KE | GraphVite |
|----------|--------|-----------|
|          | 1 GPU  | 4 GPU     | 8 GPU    |
| Hit@10   | 0.873  | 0.866     | 0.869    |
| Hit@3    | 0.801  | 0.791     | 0.793    |
| MR       | 0.844  | 0.815     | 0.812    |
| MRR      | 0.717  | 0.712     | 0.711    |

|          | DGL-KE | GraphVite |
|----------|--------|-----------|
|          | 1 GPU  | 4 GPU     | 8 GPU    |
| Hit@10   | 0.895  | 0.890     | 0.887    |
| Hit@3    | 0.835  | 0.825     | 0.816    |
| MR       | 0.702  | 0.680     | 0.715    |
| MRR      | 0.777  | 0.762     | 0.736    |

### Table 9: The accuracy of DGL-KE and GraphVite on wn18 with 1, 4 and 8 GPUs.

|          | DGL-KE | GraphVite |
|----------|--------|-----------|
|          | 1 GPU  | 4 GPU     | 8 GPU    |
| Hit@10   | 0.950  | 0.950     | 0.948    |
| Hit@3    | 0.930  | 0.928     | 0.926    |
| MR       | 0.345  | 0.355     | 0.329    |
| MRR      | 0.763  | 0.759     | 0.726    |

7 CONCLUSIONS

We develop an efficient package called DGL-KE to train knowledge graph embeddings at large scale. It implements a number of optimization techniques to improve locality, reduce data communication, while harnessing parallel computing capacity. As a result, DGL-KE significantly outperforms the state-of-the-art packages for knowledge graph embeddings on a variety of hardware, including many-core CPU, multi-GPU as well as cluster of machines. Our experiments show that DGL-KE scales well with machine resources almost linearly while still achieving very high model accuracy. DGL-KE is available at https://github.com/awslabs/dgl-ke.

8 ACKNOWLEDGMENTS

We thank the RotatE authors for making their knowledge graph embedding package KnowledgeGraphEmbedding open-source. DGL-KE was built based on their package.

REFERENCES

[1] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: A collaboratively created graph database for structuring human knowledge.
[13] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[14] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems 24*, 2011.

[15] Zhiqing Sun, Zhu-Hong Deng, Jian-Yun Nie, and Jian Tang. RotatE: Knowledge graph embedding by relational rotation in complex space. *CoRR*, abs/1902.10197, 2019.

[16] Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. Complex embeddings for simple link prediction. *CoRR*, abs/1606.06357, 2016.

[17] Q. Wang, Z. Mao, B. Wang, and L. Guo. Knowledge graph embedding: A survey of approaches and applications. *IEEE Transactions on Knowledge and Data Engineering*, 29(12), Dec 2017.

[18] Quan Wang, Zhendong Mao, Bin Wang, and Li Guo. Knowledge graph embedding: A survey of approaches and applications. *IEEE Transactions on Knowledge and Data Engineering*, 29(12):2724–2743, 2017.

[19] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander Smola, and Zheng Zhang. Deep graph library: Towards efficient and scalable deep learning on graphs. 2019.

[20] Bishan Yang, Scott Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. Embedding entities and relations for learning and inference in knowledge bases. In *Proceedings of the International Conference on Learning Representations (ICLR)* 2015, May 2015.

[21] Zhaocheng Zhu, Shizhen Xu, Meng Qu, and Jian Tang. Graphvite: A high-performance CPU-GPU hybrid system for node embedding. *CoRR*, abs/1903.00757, 2019.