SparseTransX: Efficient Training of Translation-Based Knowledge Graph Embeddings Using Sparse Matrix Operations

Md Saidul Hoque Anik [1] **Ariful Azad** [1]

Abstract

Knowledge graph (KG) learning offers a powerful framework for generating new knowledge and making inferences. Training KG embedding can take a significantly long time, especially for larger datasets. Our analysis shows that the gradient computation of embedding is one of the dominant functions in the translation-based KG embedding training loop. We address this issue by replacing the core embedding computation with SpMM (Sparse-Dense Matrix Multiplication) kernels. This allows us to unify multiple scatter (and gather) operations as a single operation, reducing training time and memory usage. We create a general framework for training KG models using sparse kernels and implement four models, namely TransE, TransR, TransH, and TorusE. Our sparse implementations exhibit up to 5.3x speedup on the CPU and up to 4.2x speedup on the GPU with a significantly low GPU memory footprint. The speedups are consistent across large and small datasets for a given model. Our proposed sparse approach can be extended to accelerate other translation-based (such as TransC, TransM, etc.) and non-translational (such as DistMult, ComplEx, RotatE, etc.) models as well. An implementation of the SpTransX framework is publicly available as a Python package in https://github.com/HipGraph/SpTransX.

1 Introduction

Knowledge Graphs (KGs) are structured as directed graphs containing entities as nodes and relations as edges. Each edge in a KG is typically stored as a triplet (head, relation, tail)—abbreviated as (h, r, t)—where head and tail are entities connected by a relation that denotes the nature of their interaction. Knowledge graph embedding (KGE) techniques map these entities and relations into a continuous vector space, enabling efficient computation and manipulation while preserving the underlying structural properties of the KG. The entity and relation embeddings are widely used in many downstream tasks, such as KG completion (Bordes et al., 2013; Chen et al., 2020), entity classification (Nickel et al., 2012), and entity resolution (Bordes et al., 2014).

Translational models (Bordes et al., 2013; Lin et al., 2015) are a widely used and effective class of KGE methods. These models represent entities and relations in a continuous vector space, where relations are interpreted as translations applied to entity embeddings. However, training translational KGE models for large-scale KGs is computationally intensive and incurs high memory overhead, especially when large batches are used. These challenges are due in part to the fact that current KGE implementations represent triplets as dense matrices and rely heavily on fine-grained scatter-gather computations during training. This fine-grained computational model contributes to the following bottlenecks: (1) irregular memory access patterns from fine-grained operations on KGs and embeddings, which increase memory access costs, (2) increased backpropagation expenses due to more granular gradient computations, and (3) significant memory demands for dense matrices. In this paper, we address these issues by proposing a sparse-matrix representation of the KG and utilizing highly optimized sparse-matrix operations to streamline KGE training, thereby reducing both computational and memory bottlenecks.

Expressing graph operations through sparse linear algebra has been highly effective for developing efficient and scalable graph neural networks (GNNs). As a result, popular graph machine learning libraries, such as PyTorch Geometric (PyG) (Fey & Lenssen, 2019) and DGL (Wang, 2019), utilize optimized implementations of sparse-dense matrix multiplication (SpMM). Despite the widespread success of sparse operations in GNNs, existing KGE libraries have yet to adopt sparse operations for training KGE models. Even models utilizing sparse embeddings, such as TranSparse (Ji et al., 2016), store embeddings as dense matrices, limiting their ability to fully leverage sparse matrix operations.

One of this paper's main contributions is the development of sparse formulations for several popular translation-based

[1]Department of Computer Science and Engineering, Texas A&M University, USA. Correspondence to: Md Saidul Hoque Anik <anik@tamu.edu>.

Proceedings of the 8th MLSys Conference, Santa Clara, CA, USA, 2025. Copyright 2025 by the author(s).

Table 1. 200 epoch training time breakdown of a TransE model using the sparse approach compared to a non-sparse approach. The time shown is the average training time taken for 7 datasets listed in Table 3. The GPU is a single NVIDIA A100-SXM4 with 40 GB VRAM. The CPU is an AMD EPYC 7763 (Milan) CPU with 64 cores and 512GB DDR4 memory.

		Sparse	**Non-Sparse (TorchKGE)**
CPU	**Forward**	74.86	299.2
	Backward	166.59	919.17
	Step	15.4	15.95
GPU	**Forward**	18.2	48.8
	Backward	17.49	89.51
	Step	0.4	0.45

KGE models. Adapting different translation models to sparse operations presents unique challenges, as each model interprets translations differently. For instance, TransE (Bordes et al., 2013) uses a single embedding space for both entities and relations, while TransR (Lin et al., 2015) uses separate spaces. Despite these differences, we designed a unified framework that allows diverse translation models to be represented through sparse matrices and mapped to sparse matrix operations like SpMM. We collectively refer to these sparse variants of translation-based embedding methods as SpTransX.

We develop a comprehensive library based on our sparse formulation. This library consolidates most computations into several SpMM function calls, allowing optimized SpMM to directly accelerate the overall runtime of KGE training. We also discuss how to extend this concept to other non-translational models such as DistMult (Yang et al., 2014a) or ComplEx (Trouillon et al., 2016) in Appendix D. We observe that SpTransX models significantly outperform established knowledge graph frameworks, such as TorchKGE and DGL-KE, particularly in terms of training time and GPU memory usage. For example, the average improvement in training time for the TransE model is illustrated in Table 1.

Overall, this paper presents the following contributions:

1. **Sparse Formulations of Translation-Based KGE Models:** We introduce sparse formulations for translation-based KGE models, enabling the mapping of KGE computations to SpMM and leveraging well-established SpMM techniques in model training.

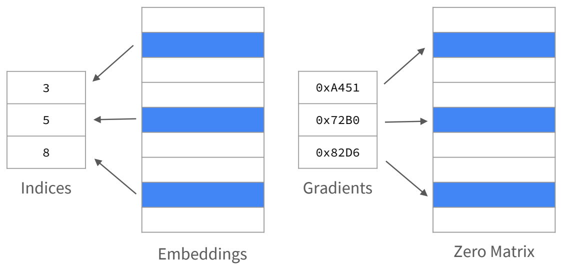
2. **Development of an Optimized Library:** Our library incorporates various optimization techniques, including SIMD vectorization, loop unrolling, cache blocking, tiling, and WARP-level GPU optimization, to enhance performance. As a result, SpTransX models significantly outperform established knowledge graph frameworks, such as TorchKGE and DGL-KE.

3. **Enhanced Large-Batch Training:** By reducing memory requirements, SpTransX facilitates large-batch training on memory-limited GPUs.

2 Background

Knowledge graph training is performed by learning the representations or embeddings of the entities and their corresponding relations on a set of training triplets or subgraphs. Each triplet or edge (in subgraph) contains a valid combination of subject (head), predicate (relation), and object (tail). Once trained, the embeddings can illustrate their semantic meaning and structure, enabling them to effectively perform reasoning-based tasks such as link prediction and entity classification. The training typically uses machine learning techniques and involves a gradient descent algorithm. The exact forward propagation process can vary depending on the model type. Translation-based models, such as TransE, TransR, etc, are widely used due to their simple yet effective way of capturing relations between entities. The training can also be done by using bilinear methods (DistMult (Yang et al., 2014b), RESCAL (Nickel et al., 2011b)), deep learning and convolution (ConvKB (Nguyen et al., 2017)), or Graph Neural Networks (R-GCN (Schlichtkrull et al., 2018)).

Training a translation-based knowledge graph embedding typically involves taking a list of triplets (head, tail, relation index) and optimizing their corresponding embeddings to minimize the distance between the $\vec{tail}$ and $\vec{head}+\vec{relation}$. The translational models vary based on (1) the linear transformation applied to the entities and relations and (2) the distance metric. The linear transformation can be applied to (a) individual entities/relations, (b) head - tail, or (c) the overall head - tail + relation. The measurement can be in a typical Euclidean space (L1 or L2) or a toroidal (wraparound) space distance (L1 torus or L2 torus) function. The training is typically done in batches, where a 'batch' of head, tail, and relations are fetched for training instead of single ones.



(a) Forward (b) Backward

Figure 1. Scatter and Gather operation in translational KG training

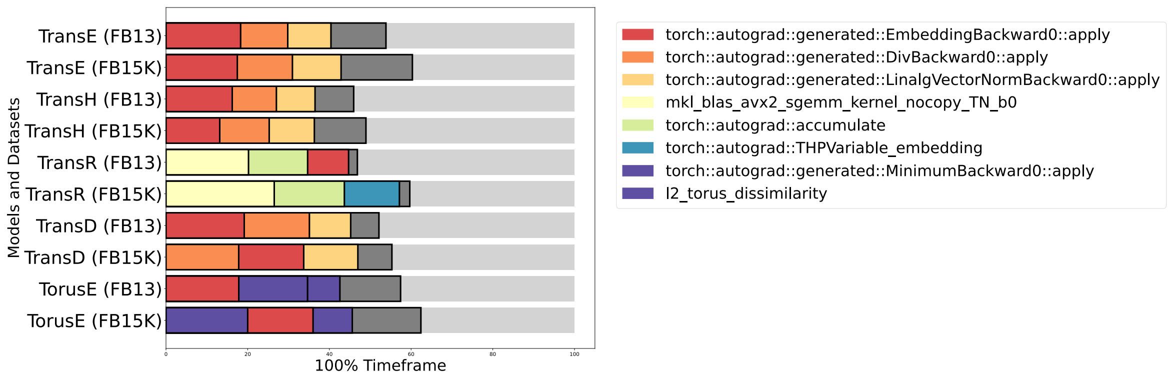


Figure 2. Top three CPU intensive functions for various translation-based KGE models and datasets (indicated in brackets). The redness represents the popularity of a function among models. The dark red box indicates that the corresponding function is used in several different models. Blue/Purple indicates that the function is typically exclusive to the current model. The dark gray box indicates the dataset loading time. The light gray box indicates the rest of the training time.

The training process starts with triplets with the index position of the head, tail, and relation entities. In each epoch, embeddings are fetched from the indices, and linear transformation is applied to them to compute the final loss. This means the forward propagation involves several (typically three or more) 'gather' operations (see Figure 1(a)) that collect the index batch's head, tail, and relation embeddings. Some models also require one or more transform matrices (may be based on the relation), which are also gathered in sthis step. Consequently, the backward propagation performs the opposite, the 'scatter' operations that distribute gradients across the corresponding indices (see Figure 1(b))

These individual operations, especially the gradient computations in the backward step, can take up around 40% of the CPU's training time (see Figure 2). In particular, we observe that embedding gradient computation is among the top three CPU-intensive functions for most translational models.

3 Related Work

3.1 Translational Models for KGE

Translation-based models represent entities and relations in a continuous vector space, interpreting relations as translations operating on entity embeddings. Several well-known models follow this approach, including TransE (Bordes et al., 2013), TransR (Lin et al., 2015), TransH (Wang et al., 2014), TransD (Ji et al., 2015), TransA (Xiao et al., 2015), TransG (Xiao et al., 2016), TransC (Lv et al., 2018), TransM (Fan et al., 2014), TorusE (Ebisu & Ichise, 2018), and KG2E (He et al., 2015). Each model varies in how it represents the head, relation, and tail embeddings to capture relational semantics effectively. For instance, TransE embeds entities and relations in the same vector space $\mathbb{R}^d$, assuming that relations can be modeled as a simple addition between the head and tail entities. In contrast, TransR utilizes distinct vector spaces for entities and relations, allowing it to better capture heterogeneous relation types, while TransE struggles with symmetric and one-to-many relations. Some models, like TransH, introduce translations on hyperplanes to address the limitations of basic Euclidean embeddings. More recently, models such as rotatE (Sun et al., 2019) have enabled translations within hyperbolic space instead of Euclidean space, allowing for better representation of hierarchical structures commonly found in some knowledge graphs. It has been observed that translation-based models are typically more computationally efficient compared to semantic matching models that use a bilinear score function, such as DistMult (Yang et al., 2014a), RESCAL (Nickel et al., 2011a), and ComplEx (Trouillon et al., 2016). This efficiency, along with their adaptability across different KG structures, makes translation-based models a popular choice for large-scale knowledge graph applications.

3.2 KGE frameworks

Several frameworks are available for training knowledge graphs (KGs). Some, like TorchKGE (Boschin, 2020) and DGL-KE (Zheng et al., 2020), are specifically designed for this purpose. Others, such as PyTorch Geometric (Fey & Lenssen, 2019) and GraphStorm (Zheng et al., 2024), offer facilities for training KG models in addition to modules for training graph neural networks.

Many frameworks are built on top of the PyTorch Framework, including TorchKGE, PyKeen (Ali et al., 2021), PyTorch Geometric, etc. AmpliGraph (Costabello et al., 2019) has Tensorflow 2.0 backend. Some frameworks support hybrid backends, such as DGL-KE or OpenKE (Han et al., 2018). DGL-KE supports PyTorch and MXNet as the backend. OpenKE supports PyTorch, Tensorflow, and C++ as the backend. Most frameworks have support for Python.

Some frameworks, such as Pykg2vec (Yu et al., 2019) or DGL-KE, choose not to use the autograd feature of the backend ML, such as PyTorch, and implement their custom gradient update mechanism. PyKeen is designed to be highly extensible and uses a modular code base. It features automatic memory optimization support that generates sub-batches when the user-defined batch does not fit in the memory.

Most frameworks, such as TorchKGE, PyG, and PyKeen, use PyTorch's embedding module directly to store entity and relation embeddings. Others, such as DGL-KE, convert the training triplets into DGL graphs before training. DGL-KE, PyTorch BigGraph (Lerer et al., 2019), PyKeen, and several other frameworks allow multi-CPU and multi-GPU training using Python and distributed frameworks such as DGL or PyTorch Lightning.

3.3 Sparse Operations in Graph ML

Expressing graph operations through sparse linear algebra has proven highly effective for developing efficient and scalable graph learning algorithms. For example, the forward and backward propagation in graph convolutional networks (GCNs) and graph attention networks (GATs) can be optimized with sampled dense-dense matrix multiplication (SDDMM), sparse-dense matrix multiplication (SpMM), or their combination, known as FusedMM (Fey & Lenssen, 2019; Wang, 2019; Rahman et al., 2021). Similarly, SpMM and sparse-sparse matrix multiplication (SpGEMM) are widely used in algorithms for graph embedding (Ranawaka et al., 2024; Rahman et al., 2020b), clustering (Azad et al., 2018), and visualization (Rahman et al., 2020a). Consequently, popular graph machine learning libraries, such as PyTorch Geometric (PyG) (Fey & Lenssen, 2019) and DGL (Wang, 2019), rely on optimized implementations of SpMM, SpGEMM, and SDDMM available in vendor-provided libraries like cuSparse, MKL, or open-source libraries such as iSpLib (Hoque Anik et al., 2024), FeatGraph (Hu et al., 2020), and SparseTIR (Ye et al., 2023). Despite the wide success of sparse operations in GNNs and graph embeddings, to the best of our knowledge, existing knowledge graph embedding libraries do not leverage sparse operations for training KGE models.

4 METHODOLOGY

4.1 Sparse Approach

We observe that the embedding extraction operation and its gradient computation is a bottleneck in the training of many translational models (Figure 2). We tackle this by replacing the typical embedding extraction process with Sparse-Dense Matrix Multiplication (SpMM). We form a sparse incidence matrix out of the training triplets so that multiplying it with the embedding matrix would directly generate at least a portion of the scores for each triplet.

The sparse approach unifies the embedding gather operations for entities in forward propagation and scatter operations for gradients in backward propagation. This unified framework enables us to leverage high-performance matrix multiplication techniques, such as loop unrolling, cache blocking, tiling, and WARP-level GPU primitives. Additionally, we can apply advanced parallelization methods, including dynamic load balancing across threads and code generation, as well as more efficient SIMD (Single Instruction, Multiple Data) vectorization.

In the following subsections, we briefly discuss how to perform training in a sparse approach using SpMM instead of regular embedding extraction for several translation-based models. Both forward and backward propagation of our approach benefit from the efficiency of a high-performance SpMM (proof shown in Appendix G). This concept also extends broadly to various other knowledge graph embedding (KGE) methods as well, including DistMult, ComplEx, and RotateE (detailed formulations are provided in Appendix D). The sparsity of our formulation and related computational complexity are discussed in Appendix B and C.

4.2 Adjacency Matrix Formulation

We analyze the score function of several translation-based models and observe that many models such as TransE, TransR, and TransH take head, tail, and relation - and compute either (a) (head - tail) or (b) (head - tail + relation) expression before applying additional linear projections as needed. For simplicity, we refer to these as 'ht' and 'hrt' expressions, respectively. Table 2 lists a few of such models and their corresponding score functions. For some models, the expressions mentioned earlier are apparent, and for others, we need to perform minor algebraic rearrangements. These formulations are listed from subsection 4.3 to 4.6.

Table 2. Translational models with common expressions in score function $f_r(h, t)$

MODEL	SCORING FUNCTION
TRANSE (BORDES ET AL., 2013)	$\|\|\vec{h} + \vec{r} - \vec{t}\|\|$
TRANSH (WANG ET AL., 2014)	$\|\|\vec{h_\perp} + \vec{d_r} - \vec{t_\perp}\|\|$
TRANSR (LIN ET AL., 2015)	$\|\|M_r\vec{h} + \vec{r} - M_r\vec{t}\|\|$
TORUSE (EBISU & ICHISE, 2018)	$\|\|\vec{h} + \vec{r} - \vec{t}\|\|$
TRANSA (XIAO ET AL., 2015)	$\|\vec{h} + \vec{r} - \vec{t}\|^T W_r \|\vec{h} + \vec{r} - \vec{t}\|$
TRANSC (LV ET AL., 2018)	$\|\|\vec{h} + \vec{r} - \vec{t}\|\|_2^2$
TRANSM (FAN ET AL., 2014)	$w_r\|\|\vec{h} + \vec{r} - \vec{t}\|\|$

Instead of gathering head, tail, and relations individually from the indices and then computing the *ht* and *hrt* expressions, we can directly get this result by forming an incidence matrix. The following subsection describes how we can compute *ht* and *hrt* expressions using sparse-dense matrix multiplication.

4.2.1 *ht or (head - tail) computation*

Let the knowledge graph contain N entities and M triples in the training data, with an embedding size (dimension) denoted by d. To compute the *ht* expression, we store entity embedding in a dense matrix $\mathbf{E} \in \mathbb{R}^{N\times d}$, where each row stores the embedding of an entity. We store the training triples in a sparse incidence matrix $\mathbf{A} \in \{-1, 0, 1\}^{M\times N}$, where the rows represent the training triplets and the columns represent entities. For a triplet, the corresponding column of a head or tail index is filled with the coefficient of the head or tail. In the expression head - tail, the coefficient of the head is $+1$ and -1 for the tail. This implies that each row of the incidence matrix contains exactly two nonzero entries. Once we multiply this incident sparse matrix $\mathbf{A}$ with the embedding matrix $\mathbf{e}$, we get the array of (head - tail) for the corresponding training triplets. Figure 3(a) shows an example of this calculation. This computed expression can be used to complete the score calculation.

4.2.2 *hrt or (head + relation - tail) computation*

Evaluating this expression requires accessing two separate dense matrices when entity and relation embeddings are stored individually. We can still compute this expression in a single sparse-dense matrix multiplication if we stack the entity and relations horizontally in the incidence sparse matrix and vertically as an embedding dense matrix.

Let the knowledge graph contain R relations. To compute the *hrt* expression with a single SpMM operation, we store the entity and relation embeddings in the same dense matrix $\mathbf{E} \in \mathbb{R}^{(N+R)\times d}$, where the first N row stores the embeddings of entities and the last R rows store the embeddings of relations. For this computation, we store the training triples in a sparse incidence matrix $\mathbf{A} \in \{-1, 0, 1\}^{M\times (N+R)}$, where the rows represent the training triplets and the columns represent entities and relations. As before, we place the expressions' coefficients in the corresponding columns ($+1$ for head and relation, -1 for tail). The relation associated with each triple is represented by placing a $+1$ in the corresponding column for that relation. Note that we offset the relation index by the total number of entities in the incidence matrix $\mathbf{A}$. This ensures that, when multiplied, the relation index aligns correctly with the corresponding relation embedding located just below the entity embeddings. Finally, we multiply the sparse matrix with the combined dense embedding matrix to get the hrt expression result. Figure 3(b) shows an example of this computation.

The following subsections contain the implementation of four translational models using the sparse approach. Throughout the rest of the paper, we refer to these four implementations collectively as **SparseTransX**, or **SpTransX** in short.

4.3 TransE Formulation

For triplets ($\vec{h}$, $\vec{r}$, $\vec{t}$), where $\vec{h}$ is the head entity vector, $\vec{r}$ is the relation entity vector, and $\vec{t}$ is the tail entity vector, TransE tries to enforce the following for a training set U:

$$\begin{aligned} &\forall(\vec{h}, \vec{t}, \vec{r}) \in U, \\ &\qquad \vec{h} + \vec{r} \approx \vec{t} \\ &\implies \vec{h} + \vec{r} - \vec{t} \approx \vec{0} \end{aligned} \tag{1}$$

For TransE, a normalization function (L1 or L2) is typically applied to this expression to get the final score. We can directly obtain this expression using the hrt computation method discussed in subsection 4.2.2.

4.4 TransR Formulation

The TransR model applies a linear projection to the head and tail before computing the score. For a projection matrix M_r corresponding to relation $\vec{r}$, TransR tries to enforce the following translation:

$$\begin{aligned} &\forall(\vec{h}, \vec{t}, \vec{r}) \in U, \\ &\qquad M_r\vec{h} + \vec{r} \approx M_r\vec{t} \\ &\implies M_r\vec{h} + \vec{r} - M_r\vec{t} \approx \vec{0} \\ &\implies M_r(\vec{h} - \vec{t}) + \vec{r} \approx \vec{0} \end{aligned} \tag{2}$$

After rearrangement, we see that it contains the (head-tail) expression. This can be computed using the ht computation method discussed in subsection 4.2.1.

4.5 TransH Formulation

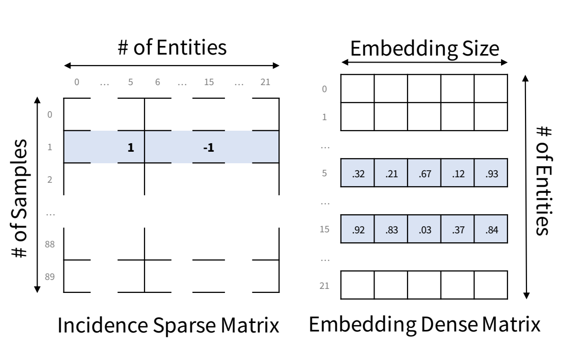
TransH (Translating Embeddings on Hyperplanes) extends TransE by allowing each relation to have its hyperplane, addressing the limitation that a single translation vector cannot handle 1-to-N, N-to-1, and N-to-N relations effectively. In TransH, each relation $\vec{r}$ is associated with a hyperplane characterized by a normal vector $\vec{w_r}$ and a translation vector $\vec{d_r}$. The projection of entities onto the hyperplane is then used in the translation. It tries to enforce the following:

$$\begin{aligned} &\forall(\vec{h}, \vec{t}, \vec{r}) \in U, \\ &\qquad \vec{h_\perp} + \vec{d_r} \approx \vec{t_\perp} \end{aligned} \tag{3}$$

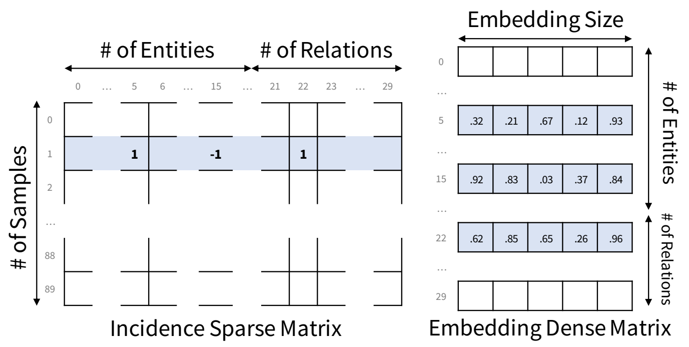
Where,

$$\begin{aligned} \vec{h_\perp} &= \vec{h} - (\vec{w_r}^T \cdot \vec{h})\vec{w_r} \\ \vec{t_\perp} &= \vec{t} - (\vec{w_r}^T \cdot \vec{t})\vec{w_r} \end{aligned}$$

Substituting these values in Equation 3, we find that for every triplet, TransH is trying to enforce:



(a) $(h - t)$ Computation. All cells in the highlighted row are zero except h-idx and t-idx. For the highlighted row, h-idx = 5, t-idx = 15, entity-count = 22.



(b) $(h + r - t)$ Computation. All cells in the highlighted row are zero except h-idx, t-idx, and r-idx + entity-count. For the highlighted row, h-idx = 5, t-idx = 15, entity-count = 20, r-idx = 2.

Figure 3. Computing common expressions using SpMM. Only the highlighted row is populated for demonstration.

$$\begin{aligned}
&\vec{h} - (\vec{w_r}^T \cdot \vec{h})\vec{w_r} + \vec{d_r} \approx \vec{t} - (\vec{w_r}^T \cdot \vec{t})\vec{w_r} \\
\implies &\vec{h} - (\vec{w_r}^T \cdot \vec{h})\vec{w_r} + \vec{d_r} - \vec{t} + (\vec{w_r}^T \cdot \vec{t})\vec{w_r} \approx \vec{0} \\
\implies &\vec{h} - \vec{t} + \vec{d_r} - (\vec{w_r}^T \cdot \vec{h})\vec{w_r} + (\vec{w_r}^T \cdot \vec{t})\vec{w_r} \approx \vec{0} \\
\implies &(\vec{h} - \vec{t}) + \vec{d_r} - \vec{w_r}^T \cdot (\vec{h} - \vec{t})\vec{w_r} \approx \vec{0}
\end{aligned}$$

We observe that the final arrangement contains two expressions of ht. This can be computed using the ht computation method discussed in subsection 4.2.1.

4.6 TorusE Formulation

The TorusE model is very similar to TransE regarding the score function. It typically uses L1/L2 torus distance instead of regular L1/L2 norm and only works with the fractional components of the embeddings.

Just like TransE, it also tries to enforce the following:

$$\begin{aligned}
&\forall(\vec{h}, \vec{t}, \vec{r}) \in U, \\
&\qquad \vec{h} + \vec{r} \approx \vec{t} \\
&\implies \vec{h} + \vec{r} - \vec{t} \approx \vec{0}
\end{aligned} \tag{4}$$

We can directly obtain this expression using the hrt computation method discussed in subsection 4.2.2.

4.7 SparseTransX Framework

We develop a general framework for SpTransX model training to enable efficient translation-based model training for large KG datasets. The framework is implemented using PyTorch 2.3 and consists of several modules, which are briefly described below.

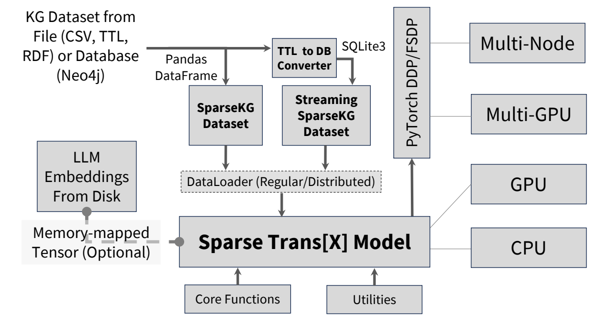


Figure 4. SparseTransX Framework

4.7.1 *SparseTransX Models*

This module contains the sparse implementations of the translational models. These implementations are agnostic to the sparse matrix library used underneath. The models have built-in support for streaming embeddings from disc storage when the embeddings are too large to fit in CPU memory. This streaming model support is implemented using PyTorch memory-mapped tensors. Researchers often use Large Language Model (LLM) embeddings such as BERT (Devlin et al., 2019), T5 (Colin, 2020), or GPT (Radford, 2018) to perform knowledge graph completion (Wang et al., 2022; Kim et al., 2020) and want to start with pre-trained embeddings that are typically too large to fit on CPU memory. Such training can be performed using this feature of the framework. Finally, this module also has functionalities for calculating scores, predicting links, and classifying entities in addition to the training loop.

4.7.2 *Dataloaders*

Our framework contains various dataloaders for shared and distributed training. It supports several standard knowledge graph formats, such as TTL, RDF, and CSV. Additionally, it contains a streaming dataset module for datasets that are too large to fit in memory. When invoked, it creates an SQLite representation of the knowledge graph and stores the entity-index mapping in the database along with the triplets. All dataloaders connect to the sparse model input using a common interface.

4.7.3 *Utilities and Core Functions*

This module consists of the sparse adjacency matrix builder described in subsection 4.2.1 and 4.2.2, an efficient sparse negative sampler, and the matrix multiplier interface.

5 Experimental Setting

We implement the SpTransX models using PyTorch Framework and compare their total training time and GPU memory allocation with other well-known KG frameworks. We run these experiments for 7 datasets consisting of various sizes on a single CPU and a single GPU system separately. We provide a guideline to reproduce the experiment for one of the datasets in Appendix A. Although our framework supports distributed training using standard PyTorch data-parallel libraries, this paper focuses on sparse techniques and presents experiments conducted on a single GPU only. A preliminary result on distributed training of a large knowledge graph dataset, COVID-19 (Tabassum et al., 2024), is shown in Appendix F.

5.1 Datasets

Below are the 7 datasets used in the experiments.

Table 3. Knowledge graph datasets

Dataset	Entity	Relations	Training Triplets
FB15k	14951	1345	483142
FB15k237	14541	237	272115
WN18	40943	18	141442
WN18RR	40943	11	86835
FB13	67399	15342	316232
YAGO3-10	123182	37	1079040
BioKG	93773	51	4762678

5.2 Frameworks and Models

For comparison, we pick three popular KG frameworks: TorchKGE, PyTorch Geometric, and DGL-KE. PyTorch Geometric (or PyG) supports the TransE model, while DGL-KE supports the TransE and TransR models. TorchKGE supports all four models: TransE, TransR, TransH, TorusE.

5.3 Training Loop

We prepare 11 separate scripts (SpTransE, SpTransR, SpTransH, SpTorusE, transe-torchkge, transr-torchkge, transh-torchkge, toruse-torchkge, transe-dglke, transr-dglke, and transe-pyg) to train the models on various datasets. Each script receives the dataset name as a command-line argument. The dataset is loaded from a shared repository. All frameworks use the same training configuration (learning rate: 0.0004, margin: 0.5), dissimilarity function (L2 or L2 torus), and loss function (MarginRankingLoss), and run for 200 epochs. Batch size and embedding dimensions are selected to maximize accuracy while utilizing available GPU memory (see subsection 6.1). The following table lists the dimensions and batch sizes used for different models.

Table 4. Training configuration for CPU and GPU. Reduced embedding and batch size for TransR and TransH due to memory limitation.

Model	Embedding	Batch
TransE	1024	12×32768
TorusE	1024	12×32768
TransR	128	2×32768
TransH	Ent=128,Rel=128	32768

The negative samples are generated once per positive sample and are pre-generated outside the training loop.

5.4 System and Profiler Details

All experiments are run on the NERSC Perlmutter system. The GPU experiments run on a single NVIDIA A100-SXM4 GPU with 40 GB VRAM. The CPU experiments run on an AMD EPYC 7763 (Milan) CPU with 64 cores and 512GB DDR4 memory.

We use Python's $time$ module to measure training time and its breakdown. PyTorch's CUDA module is used to measure peak memory usage in GPU experiments. Finally, Linux's $perf$ tool is used to measure the cache miss rate and FLOPs count for CPU experiments.

5.5 SparseTransX Configuration

In the SparseTransX framework, users can choose to use any high-performance SpMM to perform model training. We select iSpLib SpMM (Hoque Anik et al., 2024) for CPU training and DGL g-SpMM (Wang, 2019) for GPU training. CSR (Compressed Sparse Row) format is used for iSpLib, and COO (Coordinate) format is used for DGL as per the library requirement. The framework automatically generates sparse minibatches in the correct format when the user specifies the underneath SpMM library.

6 Results

6.1 Hyperparameter Selection

Knowledge graphs are primarily used for link prediction tasks (Gregucci et al., 2023). Hits@10 is a popular measurement of link prediction accuracy. We train the KG models on various embedding sizes and plot the corresponding Hits@10 accuracy in Figure-5. We observe that accuracy increases as the entity embedding size grows. We keep the number of positive and negative edges equal within a batch for each model.

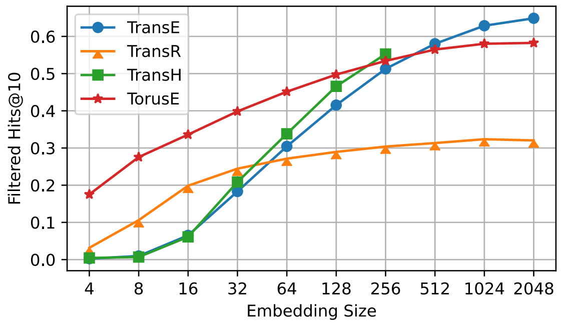


Figure 5. Hits@10 accuracy w.r.t. embedding size for FB15K dataset. 100 epoch training with a batch size of 32768 and relation entity dimension as 8 (for TransH model). The TransH model encounters out-of-memory issues when the embedding size exceeds 256. Other models converge at an embedding size of approximately 2048 and show no improvement in Hits@10 accuracy for larger embeddings.

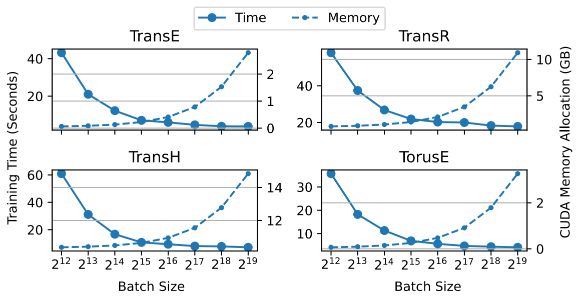


Figure 6. Training time and GPU memory allocation w.r.t. Batch Size. 100 epoch training with entity dimension as 128 and relation dimension as 8 (for TransH model).

Another significant hyperparameter is the batch size. We plot model training time and GPU memory allocation for various batch sizes in Figure 6. We observe that maximum CUDA memory utilization is possible when the largest batch size is used. It also corresponds to the fastest training time.

6.2 Training Performance

We measure the total training time, GPU memory allocation, CPU Cache miss, and FLOPs count for various datasets on the available models of the frameworks mentioned in subsection 5.2.

6.2.1 Training Time

The total training time for various datasets on CPU and GPU are shown in Figure 7. Our implementation outperforms all frameworks for both CPU and GPU. The speedup is consistent for both small and large datasets.

SpTransX models exhibit good speedup on CPU and GPU systems. The speedups are consistent across datasets for the same model. We observe the most speedup in the TransE model. This is because, for this model, the computational bottleneck is the embedding gradient computation (see Figure 2). We eliminate this bottleneck by replacing fine-grained embedding scatter-gather with SpMM, which results in faster training time and efficient GPU memory usage (due to lower intermediate variable usage).

Although TorusE uses the same scoring function, we do not observe the same amount of speedup in this model compared to TransE. This is because the primary computational bottleneck in this model is not always the embedding computation but the torus L2 dissimilarity function (marked as yellow boxes in Figure 2).

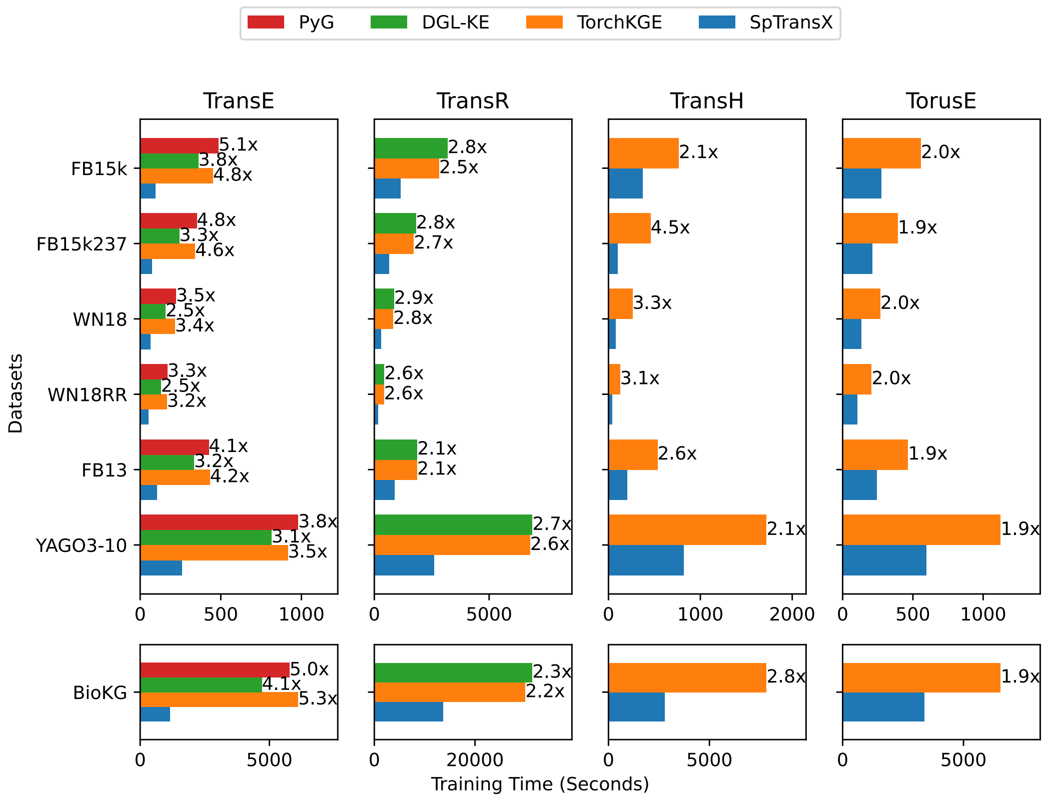
Among TransR and TransH, TransR is computationally more demanding. However, we still manage to perform better in TransR compared to TransH because the computational graph of TransH is much larger than TransR, and the embedding computation (or SpMM in our case) accounts for a lower percentage of system time compared to TransR. This means SpTransX has less impact on TransH compared to TransR.

6.2.2 GPU Memory Usage

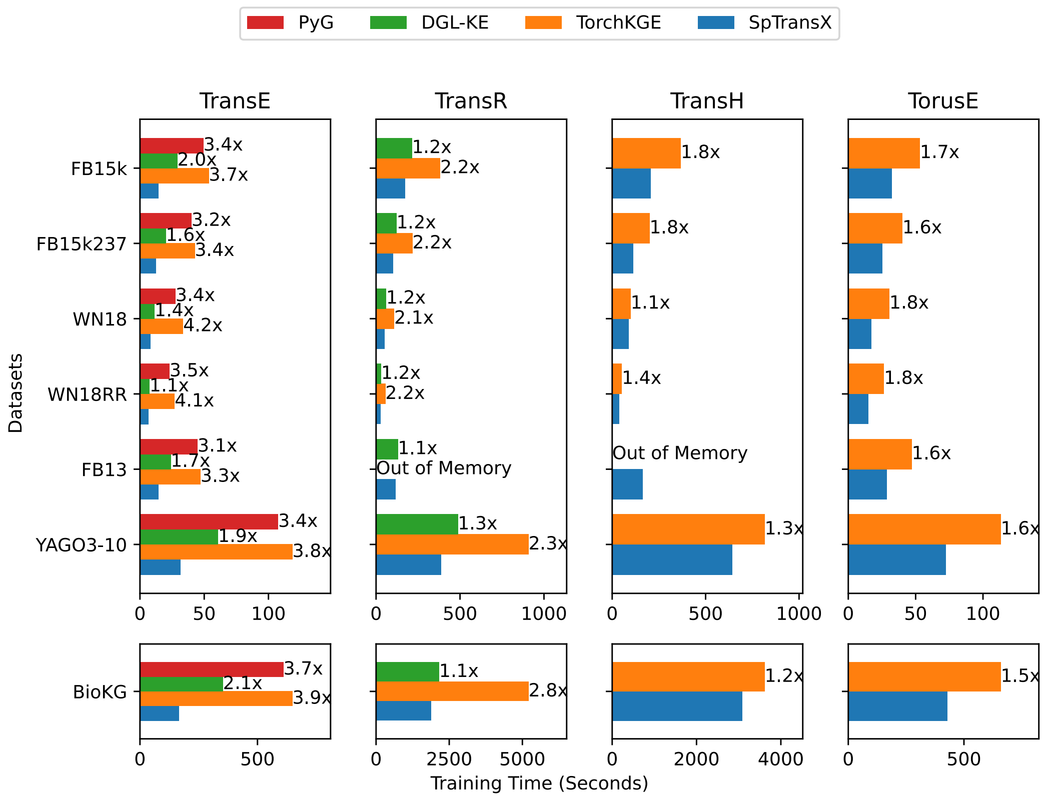
Our implementations of the models take up significantly less CUDA memory than other frameworks. Table 5 demonstrates the average CUDA memory allocation for various frameworks and our implementations.

Table 5. Average CUDA memory allocation for various models (in GB)

Model	SpTransX	TorchKGE	DGL-KE	PyG
TransE	**5.61**	13.55	11.37	13.54
TransR	**13.65**	20.42	30.73	-
TransH	**0.28**	3.1	-	-
TorusE	**12.03**	15.87	-	-

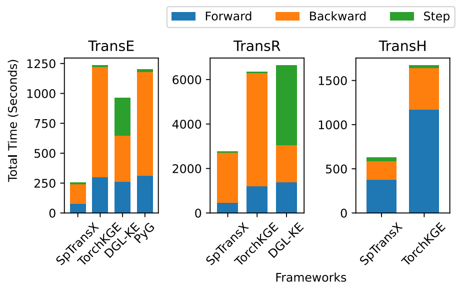


(a) CPU

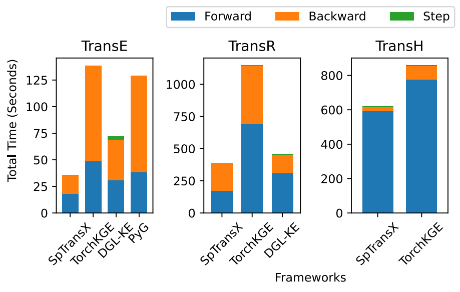


(b) GPU

Figure 7. Total training time for CPU and GPU for various datasets. The slowdown factors of each framework compared to SpTransX are shown along the bars.



(a) CPU



(b) GPU

Figure 8. Breakdown of total training time for CPU and GPU on average of 7 datasets

SpTransX is optimized for GPU memory usage by limiting the model size to tensors only necessary during the training time. Furthermore, the SpMM accounts for fewer intermediate variables, reducing the memory footprint. We observe the highest GPU memory efficiency in the TransH model, around 11× more efficient than TorchKGE on average. This is because the training loop uses linear algebraic implementation (discussed in subsection 4.5) and reuses several expressions to reduce unnecessary GPU memory allocation.

6.2.3 *Breakdown of Training Time*

Each model training epoch consists of loss calculation (forward propagation), gradient computation (backward call), and parameter update (optimizer step). The chart in Figure 8 shows the average breakdown of the three steps for the frameworks.

We observe that SparsTransX improves the average forward propagation time for both CPU and GPU. It also outperforms backward computation for all cases except in TransR with DGL-KE for both GPU and CPU. DGL-KE uses the heterograph data structure instead of a regular triplet array and updates the backward gradients manually through the DGL graph API. This results in an unusually long parameter update time for DGL-KE in the CPU. This issue is not present in GPU since DGL-KE has a separate GPU implementation. Despite the slower backward time, SpTransX outperforms DGL-KE in terms of overall training time.

Table 6. Average FLOPs count of 7 datasets (factor of $\times 10^{10}$)

MODEL	SPTRANSX	TORCHKGE	DGL-KE	PYG
TRANSE	**220**	483.87	293.06	483.82
TRANSR	**567.37**	1157.94	874.67	-
TRANSH	**9.66**	19.58	-	-
TORUSE	**289.99**	387.93	-	-

6.2.4 *FLOPs count and Cache Miss Rate*

We measure the FLOPs count for our CPU implementation and the cache miss rate. SpTransX exhibits a lower FLOP count than other frameworks for all models on average, as shown in Table 6. It uses high-performance SpMM that typically uses fewer floating-point operations than regular non-sparse implementations. This results in the lowest average FLOP count for SpTransX compared to all frameworks for all models.

Table 7. Average cache miss rate of 7 datasets (in %)

MODEL	SPTRANSX	TORCHKGE	DGL-KE	PYG
TRANSE	**26.54**	29.37	29.99	29.04
TRANSR	**17.02**	19.20	29.54	-
TRANSH	10.43	**9.75**	-	-
TORUSE	**21.53**	22.94	-	-

Table 7 lists the average cache miss rates. We observe that SpTransX performs better in all cases except for the TransH model. In this case, SparseTransX has a slightly higher cache miss rate than its peer, TorchKGE. This is because the impact of SpMM is small in the TransH model, and other operations overshadow the improved cache miss rate obtained by the efficient SpMM.

6.2.5 *Model Accuracies*

The sparse approach does not change the computational steps and thus does not affect the model accuracy. The accuracies of our implementations are consistent with that of other models, such as TorchKGE. For 100 epochs training on WN18 datasets with a fixed learning rate of 0.0004, SpTransX's TransE, TorusE, and TransH models receive 0.72, 0.63, and 0.59 Hits@10 scores, whereas TorchKG's models receive 0.74, 0.63, and 0.60. A more detailed evaluation (discussed in Appendix E) reveals that SpTransX achieves similar or better Hits@10 accuracy compared to TorchKGE when the training loop is equipped with a learning rate scheduler.

7 Conclusion

Despite the inherent sparsity of knowledge graphs and their embedding algorithms, existing frameworks often do not leverage sparse matrix operations to accelerate the training of KGE models. We develop sparse formulations of translation-based KGE models that significantly outperform established knowledge graph frameworks, such as TorchKGE and DGL-KE, particularly regarding training time and GPU memory usage. Our findings demonstrate that the proposed approach consistently achieves improved performance across a range of both small and large datasets. We design a PyTorch-based library named SparseTransX that incorporates the sparse formulation methods for Knowledge Graph models and can demonstrate the aforementioned performance gains. The design of the library is flexible, and it can potentially perform distributed training once coupled with PyTorch Distributed Data Parallel (DDP) and Fully Sharded Data Parallel (FSDP) wrappers.

Our sparse approach has multiple benefits. By using sparse representations, we reduce memory usage during training, which allows us to work with larger knowledge graphs without exhausting GPU resources. The efficiency improvements in training time come from optimizing matrix operations. Given the extensive research in parallel sparse matrix operations and the availability of highly optimized libraries, our approach paves the way for faster computations and enhanced scalability for larger knowledge graphs. We believe this work will inspire further advancements in the development of robust and scalable knowledge graph frameworks.

Acknowledgements

This research is supported by the NSF OAC-2112606 and OAC-2339607 grants and DOE DE-SC0022098 and DE-SC0023349 awards.

References

Ali, M., Berrendorf, M., Hoyt, C. T., Vermue, L., Sharifzadeh, S., Tresp, V., and Lehmann, J. Pykeen 1.0: a python library for training and evaluating knowledge graph embeddings. *Journal of Machine Learning Research*, 22(82):1–6, 2021.

Azad, A., Pavlopoulos, G. A., Ouzounis, C. A., Kyrpides, N. C., and Buluç, A. Hipmcl: a high-performance parallel implementation of the markov clustering algorithm for large-scale networks. *Nucleic Acids Research*, 46(6): e33–e33, 2018.

Bordes, A., Usunier, N., Garcia-Duran, A., Weston, J., and Yakhnenko, O. Translating embeddings for modeling multi-relational data. In Burges, C., Bottou, L., Welling, M., Ghahramani, Z., and Weinberger, K. (eds.), *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013.

Bordes, A., Glorot, X., Weston, J., and Bengio, Y. A semantic matching energy function for learning with multi-relational data: Application to word-sense disambiguation. *Machine Learning*, 94:233–259, 2014.

Boschin, A. TorchKGE: knowledge graph embedding in python and pytorch. *arXiv preprint arXiv:2009.02963*, 2020.

Chen, Z., Wang, Y., Zhao, B., Cheng, J., Zhao, X., and Duan, Z. Knowledge graph completion: A review. *IEEE Access*, 8:192435–192456, 2020.

Colin, R. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21: 140–1, 2020.

Costabello, L., Bernardi, A., Janik, A., Creo, A., Pai, S., Van, C. L., McGrath, R., McCarthy, N., and Tabacof, P. AmpliGraph: a Library for Representation Learning on Knowledge Graphs, March 2019. URL https://doi.org/10.5281/zenodo.2595043.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. In Burstein, J., Doran, C., and Solorio, T. (eds.), *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423.

Ebisu, T. and Ichise, R. Toruse: Knowledge graph embedding on a lie group. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

Fan, M., Zhou, Q., Chang, E., and Zheng, F. Transition-based knowledge graph embedding with relational mapping properties. In *Proceedings of the 28th Pacific Asia conference on language, information and computing*, pp. 328–337, 2014.

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

Gregucci, C., Nayyeri, M., Hernández, D., and Staab, S. Link prediction with attention applied on multiple knowledge graph embedding models. In *Proceedings of the ACM Web Conference 2023*, pp. 2600–2610, 2023.

Han, X., Cao, S., Xin, L., Lin, Y., Liu, Z., Sun, M., and Li, J. Openke: An open toolkit for knowledge embedding. In *Proceedings of EMNLP*, 2018.

He, S., Liu, K., Ji, G., and Zhao, J. Learning to represent knowledge graphs with gaussian embedding. In *Proceedings of the 24th ACM international on conference on information and knowledge management*, pp. 623–632, 2015.

Hoque Anik, M. S., Badhe, P., Gampa, R., and Azad, A. iSpLib: a library for accelerating graph neural networks using auto-tuned sparse operations. In *Companion Proceedings of the ACM on Web Conference 2024*, pp. 778–781, 2024.

Hu, Y., Ye, Z., Wang, M., Yu, J., Zheng, D., Li, M., Zhang, Z., Zhang, Z., and Wang, Y. FeatGraph: a flexible and efficient backend for graph neural network systems. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–13. IEEE, 2020.

Ji, G., He, S., Xu, L., Liu, K., and Zhao, J. Knowledge graph embedding via dynamic mapping matrix. In *Proceedings of the 53rd annual meeting of the association for computational linguistics*, pp. 687–696, 2015.

Ji, G., Liu, K., He, S., and Zhao, J. Knowledge graph completion with adaptive sparse transfer matrix. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

Kim, B., Hong, T., Ko, Y., and Seo, J. Multi-task learning for knowledge graph completion with pre-trained language models. In *Proceedings of the 28th international conference on computational linguistics*, pp. 1737–1743, 2020.

Lerer, A., Wu, L., Shen, J., Lacroix, T., Wehrstedt, L., Bose, A., and Peysakhovich, A. Pytorch-biggraph: A large scale graph embedding system. *Proceedings of Machine Learning and Systems*, 1:120–131, 2019.

Lin, Y., Liu, Z., Sun, M., Liu, Y., and Zhu, X. Learning entity and relation embeddings for knowledge graph completion. In *Proceedings of the AAAI conference on artificial intelligence*, volume 29, 2015.

Lv, X., Hou, L., Li, J., and Liu, Z. Differentiating concepts and instances for knowledge graph embedding. *arXiv preprint arXiv:1811.04588*, 2018.

Nguyen, D. Q., Nguyen, T. D., Nguyen, D. Q., and Phung, D. A novel embedding model for knowledge base completion based on convolutional neural network. *arXiv preprint arXiv:1712.02121*, 2017.

Nickel, M., Tresp, V., Kriegel, H.-P., et al. A three-way model for collective learning on multi-relational data. In *International conference on machine learning*, volume 11, pp. 3104482–3104584, 2011a.

Nickel, M., Tresp, V., Kriegel, H.-P., et al. A three-way model for collective learning on multi-relational data. In *ICML*, volume 11, pp. 3104482–3104584, 2011b.

Nickel, M., Tresp, V., and Kriegel, H.-P. Factorizing yago: scalable machine learning for linked data. In *Proceedings of the 21st international conference on World Wide Web*, pp. 271–280, 2012.

Radford, A. Improving language understanding by generative pre-training. *https://openai.com/index/language-unsupervised/*, 2018.

Rahman, M. K., Sujon, M. H., and Azad, A. Batchlayout: A batch-parallel force-directed graph layout algorithm in shared memory. In *2020 IEEE Pacific Visualization Symposium (PacificVis)*, pp. 16–25. IEEE, 2020a.

Rahman, M. K., Sujon, M. H., and Azad, A. Force2vec: Parallel force-directed graph embedding. In *2020 IEEE International Conference on Data Mining (ICDM)*, pp. 442–451. IEEE, 2020b.

Rahman, M. K., Sujon, M. H., and Azad, A. FusedMM: A unified SDDMM-SpMM kernel for graph embedding and graph neural networks. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 256–266. IEEE, 2021.

Ranawaka, I., Hussain, M. T., Block, C., Gerogiannis, G., Torrellas, J., and Azad, A. Distributed-memory parallel algorithms for sparse matrix and sparse tall-and-skinny matrix multiplication. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–17. IEEE, 2024.

Schlichtkrull, M., Kipf, T. N., Bloem, P., Van Den Berg, R., Titov, I., and Welling, M. Modeling relational data with graph convolutional networks. In *The semantic web: 15th international conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, proceedings 15*, pp. 593–607. Springer, 2018.

Sun, Z., Deng, Z.-H., Nie, J.-Y., and Tang, J. Rotate: Knowledge graph embedding by relational rotation in complex space. *arXiv preprint arXiv:1902.10197*, 2019.

Tabassum, A., Kannan, R., Yin, J., Lim, S.-H., Cong, G., Hasan, S., Patton, R., and Potok, T. E. Knowledge graph embedding using large language models for covid-19. Technical report, Oak Ridge National Laboratory (ORNL), Oak Ridge, TN, USA, 2024.

Trouillon, T., Welbl, J., Riedel, S., Gaussier, É., and Bouchard, G. Complex embeddings for simple link prediction. In *International conference on machine learning*, pp. 2071–2080. PMLR, 2016.

Wang, L., Zhao, W., Wei, Z., and Liu, J. SimKGC: simple contrastive knowledge graph completion with pre-trained language models. *arXiv preprint arXiv:2203.02167*, 2022.

Wang, M. Y. Deep graph library: Towards efficient and scalable deep learning on graphs. In *ICLR workshop on representation learning on graphs and manifolds*, 2019.

Wang, Z., Zhang, J., Feng, J., and Chen, Z. Knowledge graph embedding by translating on hyperplanes. In *Proceedings of the AAAI conference on artificial intelligence*, volume 28, 2014.

Xiao, H., Huang, M., Hao, Y., and Zhu, X. Transa: An adaptive approach for knowledge graph embedding. *arXiv preprint arXiv:1509.05490*, 2015.

Xiao, H., Huang, M., and Zhu, X. TransG: a generative model for knowledge graph embedding. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pp. 2316–2325, 2016.

Yang, B., Yih, W.-t., He, X., Gao, J., and Deng, L. Embedding entities and relations for learning and inference in knowledge bases. *arXiv preprint arXiv:1412.6575*, 2014a.

Yang, B., Yih, W.-t., He, X., Gao, J., and Deng, L. Embedding entities and relations for learning and inference in knowledge bases. *arXiv preprint arXiv:1412.6575*, 2014b.

Ye, Z., Lai, R., Shao, J., Chen, T., and Ceze, L. Sparsetir: Composable abstractions for sparse compilation in deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pp. 660–678, 2023.

Yu, S. Y., Rokka Chhetri, S., Canedo, A., Goyal, P., and Faruque, M. A. A. Pykg2vec: A python library for knowledge graph embedding. *arXiv preprint arXiv:1906.04239*, 2019.

Zheng, D., Song, X., Ma, C., Tan, Z., Ye, Z., Dong, J., Xiong, H., Zhang, Z., and Karypis, G. DGL-KE: training knowledge graph embeddings at scale. In *Proceedings of the 43rd international ACM SIGIR conference on research and development in information retrieval*, pp. 739–748, 2020.

Zheng, D., Song, X., Zhu, Q., Zhang, J., Vasiloudis, T., Ma, R., Zhang, H., Wang, Z., Adeshina, S., Nisa, I., et al. GraphStorm: all-in-one graph machine learning framework for industry applications. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 6356–6367, 2024.

A Artifact Evaluation

A.1 Abstract

We provide a guideline to reproduce the training times for one of the datasets reported in this paper. We share a publicly available GitHub repository (https://github.com/OnixHoque/sptransx-mlsys2025-reproduce) containing two bash scripts and a Jupyter Notebook that generate the workflow. The first bash script installs the frameworks needed to compare the performance with SpTransX and generates the environments required to run them. Another bash script runs the actual CPU and GPU experiments and stores the training times into text files. We include a validation Jupyter Notebook that performs processing on these text files and generates tables that depict Figure 7 of the paper (for FB15K dataset). The experiments are run on one of the seven datasets (FB15K). The training time is computed for a single minibatch to keep the workflow short. The validation notebook approximates the total training time by multiplying the number of batches with the calculated training time. We further discuss how to run the workflow and perform validation from the generated results (also available in the Readme file of the GitHub repository). Please note that SpTransX is referred to as FastKG (former name) in the workflow.

A.2 Artifact check-list (meta-information)

- **Algorithm:** TransE, TransR, TransH, TorusE (Knowledge Graph Embedding Training Models)
- **Program:** PyTorch
- **Compilation:** Conda, PIP, Python 3.7 and Python 3.8
- **Data set:** FB15K Knowledge Graph Dataset
- **Run-time environment:** Conda, GCC
- **Hardware:** A CPU (preferably AMD) with 64 cores and 512GB DDR4 memory. An NVIDIA GPU with 40GB VRAM.
- **Run-time state:** Training Time
- **Execution:** Bash Script and Jupyter Notebook (requires CUDA for GPU experiments)
- **Metrics:** Training Time (in seconds)
- **Output:** Table showing total training time and slowdown factors of other frameworks
- **Experiments:** Performs a single minibatch training on FB15K dataset for 200 epochs for all models available in all four frameworks in CPU and GPU
- **How much disk space required (approximately)?:** 50 GB
- **How much time is needed to prepare workflow (approximately)?:** 3 hours
- **How much time is needed to complete experiments (approximately)?:** 3 hours
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT
- **Workflow framework used?:** Bash Scripts and Jupyter Notebook
- **Archived (provide DOI)?:** Not generated yet

A.3 Description

A.3.1 How delivered

The workflow is available in the following GitHub repository: https://github.com/OnixHoque/sptransx-mlsys2025-reproduce.

A.3.2 Hardware dependencies

The CPU and GPU experiments were run on dedicated CPU/GPU (single) nodes of NERSC Perlmutter. Their configurations are given below. The parameters are set to maximize CPU/GPU utilization. Similar configurations are recommended to reproduce the results. The CPU configuration is AMD EPYC 7763 (Milan) CPU with 64 cores and 512GB DDR4 memory. The GPU configuration is a single NVIDIA A100-SXM4 GPU with 40 GB VRAM.

A.3.3 Software dependencies

The experiments were tested on the following configuration.

- GCC 12.2
- Conda 24.9.1
- Python 3.9 (3.8 for DGLKE)
- PyTorch 2.3.1 (1.7.1 for DGLKE)
- CUDAToolKit 12.1 (11.0 for DGLKE)

A.3.4 Datasets

FB15K dataset is used in the workflow. It is one of the seven datasets used in the paper. It is included in the repository.

A.4 Environment Installation

To set up the environments, clone the GitHub repository and run the following command. It will create two virtual environments. One specific to DGLKE, and another for the rest.

```
./0.setup_environments.sh
```

A.5 Experiment Workflow

To run the experiments, execute the following command.

```
./1.run_experiments.sh
```

It will generate the training time of a single minibatch training for various models and frameworks of the FB15K dataset. The outputs will be saved in cpu.txt and gpu.txt.

A.6 Evaluation and Expected Result

To generate the table of Figure 7 (for FB15K) in the paper, execute the Jupyter Notebook 2.validation.ipynb. It will parse the generated text files and produce the tables for CPU and GPU for the FB15k dataset. The table includes the total training time and the slowdown factors of each framework compared to SpTransX.

It is expected that SpTransX is up to $5\times$ and $4\times$ faster in CPU and GPU, respectively, compared to other frameworks. The Jupyter Notebook contains the expected tables for both CPU and GPU from past experiments.

A.7 Notes

Please note that **SpTransX** is referred to as **FastKG** (former name) in the workflow.

B Applicability of SparseTransX for dense graphs

Even for fully dense graphs, our KGE computations remain highly sparse. This is because our SpMM leverages the incidence matrix for triplets, rather than the graph's adjacency matrix. In the paper, the sparse matrix $A \in \{-1, 0, 1\}^{M\times(N+R)}$ represents the triplets, where N is the number of entities, R is the number of relations, and M is the number of triplets. This representation remains extremely sparse, as each row contains exactly three non-zero values (or two in the case of the "ht" representation). Hence, the sparsity of this formulation is independent of the graph's structure, ensuring computational efficiency even for dense graphs.

C Computational Complexity

For a sparse matrix A with $m \times k$ having $nnz(A)$ = number of non zeros and dense matrix X with $k \times n$ dimension, the computational complexity of the SpMM is $O(nnz(A) \cdot n)$ since there are a total of $nnz(A)$ number of dot products each involving n components. Since our sparse matrix contains exactly three non-zeros in each row, $nnz(A) = 3m$. Therefore, the complexity of SpMM is $O(3m \cdot n)$ or $O(m \cdot n)$, meaning the complexity increases when triplet counts or embedding dimension is increased. Memory access pattern will change when the number of entities is increased and it will affect the runtime, but the algorithmic complexity will not be affected by the number of entities/relations.

D Applicability to Non-translational Models

Our paper focused on translational models using sparse operations, but the concept extends broadly to various other knowledge graph embedding (KGE) methods. Neural network-based models, which are inherently matrix-multiplication-based, can be seamlessly integrated into this framework. Additionally, models such as DistMult, ComplEx, and RotatE can be implemented with simple modifications to the SpMM operations. Implementing these KGE models requires modifying the addition and multiplication operators in SpMM, effectively changing the semiring that governs the multiplication.

In the paper, the sparse matrix $A \in \{-1, 0, 1\}^{M\times(N+R)}$ represents the triplets, and the dense matrix $E \in \mathbb{R}^{(N+R)\times d}$ represents the embedding matrix, where N is the number of entities, R is the number of relations, and M is the number of triplets. TransE's score function, defined as $h + r - t$, is computed by multiplying A and E using an SpMM followed by the L2 norm. This operation can be generalized using a semiring-based SpMM model: $Z_{ij} = \bigoplus_{k=1}^{n}(A_{ik} \otimes E_{kj})$

Here, $\oplus$ represents the semiring addition operator, and $\otimes$ represents the semiring multiplication operator. For TransE, these operators correspond to standard arithmetic addition and multiplication, respectively.

DistMult

DistMult's score function has the expression $h \odot r \odot t$. To adapt SpMM for this model, two key adjustments are required: The sparse matrix A stores $+1$ at the positions corresponding to h_{idx}, t_{idx}, and r_{idx}. Both the semiring addition and multiplication operators are set to arithmetic multiplication. These changes enable the use of SpMM for the DistMult score function.

ComplEx

ComplEx's score function has $h \odot r \odot \bar{t}$, where embeddings are stored as complex numbers (e.g., using PyTorch). In this case, the semiring operations are similar to DistMult, but with complex number multiplication replacing real number multiplication.

RotatE

RotatE's score function has $h \odot r - t$. For this model, the semiring requires both arithmetic multiplication and subtraction for $\oplus$. With minor modifications to our SpMM implementation, the semiring addition operator can be adapted to compute $h \odot r - t$.

Support from other libraries

Many existing libraries, such as GraphBLAS (Kimmerer, Raye, et al., 2024), Ginkgo (Anzt, Hartwig, et al., 2022), and Gunrock (Wang, Yangzihao, et al., 2017), already support custom semirings in SpMM. We can leverage C++ templates to extend support for KGE models with minimal effort.

Table 8. Average of 9 Hits@10 Accuracy for WN18 dataset

Model	TorchKGE	SpTransX
TransE	0.79 ± 0.001700	0.79 ± 0.002667
TransR	0.29 ± 0.005735	0.33 ± 0.006154
TransH	0.76 ± 0.012285	0.79 ± 0.001832
TorusE	0.73 ± 0.003258	0.73 ± 0.002780

E Model Performance Evaluation and Convergence

SpTransX follows a slightly different loss curve (see Figure 9) and eventually converges with the same loss as other non-sparse implementations such as TorchKGE. We test SpTransX with the WN18 dataset having embedding size 512 (128 for TransR and TransH due to memory limitation) and run 200-1000 epochs. We compute average Hits@10 of 9 runs with different initial seeds and a learning rate scheduler. The results are shown below. We find that Hits@10 is generally comparable to or better than the

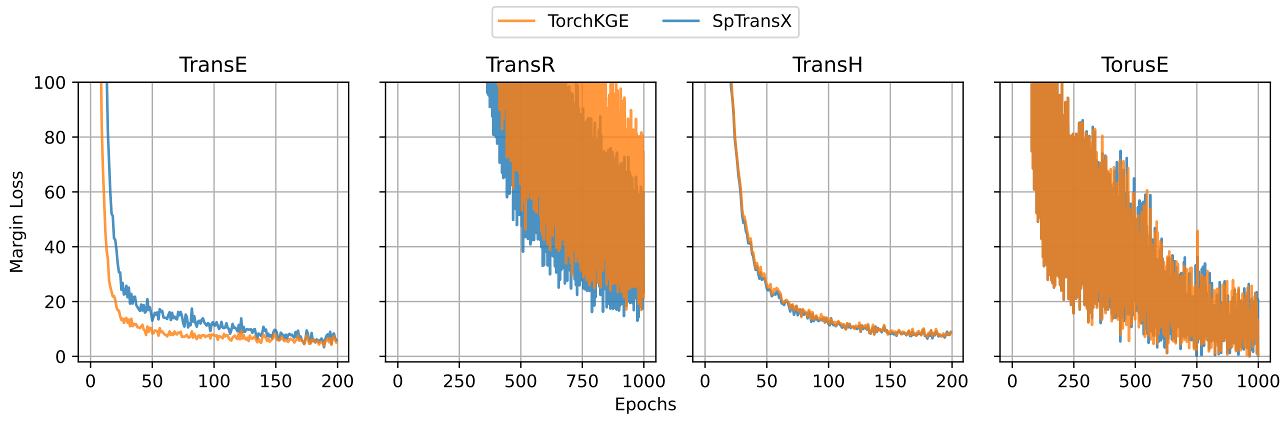


Figure 9. Loss curve for sparse and non-sparse approach. The sparse approach eventually reaches the same loss value with similar Hits@10 test accuracy.

Hits@10 achieved by TorchKGE.

F Preliminarily Experiment on Scaling

Communication can be a significant bottleneck in distributed KGE training when using sparse-dense matrix multiplication (SpMM) kernels. We perform a preliminary experiment to observe the scaling capacity of SparseTransX for the TransE model using a large knowledge graph dataset, COVID-19 (Tabassum et al., 2024). We wrap the model in the Data Distributed Parallel (DDP) wrapper provided by PyTorch and train the model to 64 NVIDIA A100 GPUs to observe scaling behaviour. DDP (Distributed Data Parallel) trains a model by replicating it across multiple GPUs, synchronizing gradients during backpropagation to ensure consistent updates. The training times for various GPUs are reported in Table 9.

Table 9. Scaling TransE model on COVID-19 dataset with 60,820 entities, 62 relations, and 1,032,939 triplets (Tabassum et al., 2024)

Number of GPUs	500 epoch time (seconds)
4	706.38
8	586.03
16	340.00
32	246.02
64	179.95

It indicates that communication is not a bottleneck up to 64 GPUs. If communication becomes a performance bottleneck at larger scales, we plan to explore alternative communication-reducing algorithms, including 2D and 3D matrix distribution techniques, which are known to minimize communication overhead at extreme scales. Additionally, we will incorporate model parallelism alongside data parallelism for large-scale knowledge graphs.

G Backpropagation of SpMM

Our main computational kernel is the sparse-dense matrix multiplication (SpMM). The computation of backpropagation of an SpMM w.r.t. the dense matrix is also another SpMM. To see how, let's consider the sparse-dense matrix multiplication $AX = C$ which is part of the training process. As long as the computational graph reduces to a single scaler loss $\mathfrak{L}$, it can be shown that $\frac{\partial C}{\partial X} = A^T$. Here, X is the learnable parameter (embeddings), and A is the sparse matrix. Since A^T is also a sparse matrix and $\frac{\partial \mathfrak{L}}{\partial C}$ is a dense matrix, the computation $\frac{\partial \mathfrak{L}}{\partial X} = \frac{\partial C}{\partial X} \times \frac{\partial \mathfrak{L}}{\partial C} = A^T \times \frac{\partial \mathfrak{L}}{\partial C}$ is an SpMM. This means that both forward and backward propagation of our approach benefit from the efficiency of a high-performance SpMM.

Proof that $\frac{\partial C}{\partial X} = A^T$

To see why $\frac{\partial C}{\partial X} = A^T$ is used in the gradient calculation, we can consider the following small matrix multiplication without loss of generality.

$$A = \begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix}$$
$$X = \begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix}$$
$$C = \begin{bmatrix} c_1 & c_2 \\ c_3 & c_4 \end{bmatrix}$$

Where $C = AX$, thus-

$$c_1 = f(x_1, x_3)$$
$$c_2 = f(x_2, x_4)$$
$$c_3 = f(x_1, x_3)$$
$$c_4 = f(x_2, x_4)$$

Therefore-

$$\frac{\partial \mathfrak{L}}{\partial x_1} = \frac{\partial \mathfrak{L}}{\partial c_1} \times \frac{\partial c_1}{\partial x_1} + \frac{\partial \mathfrak{L}}{\partial c_2} \times \frac{\partial c_2}{\partial x_1} + \frac{\partial \mathfrak{L}}{\partial c_3} \times \frac{\partial c_3}{\partial x_1} + \frac{\partial \mathfrak{L}}{\partial c_4} \times \frac{\partial c_4}{\partial x_1}$$
$$= \frac{\partial \mathfrak{L}}{\partial c_1} \times \frac{\partial \mathfrak{c}_1}{\partial x_1} + 0 + \frac{\partial \mathfrak{L}}{\partial c_3} \times \frac{\partial \mathfrak{c}_3}{\partial x_1} + 0$$
$$= a_1 \times \frac{\partial \mathfrak{L}}{\partial c_1} + a_3 \times \frac{\partial \mathfrak{L}}{\partial c_3}$$

Similarly-

$$\frac{\partial \mathfrak{L}}{\partial x_2} = a_1 \times \frac{\partial \mathfrak{L}}{\partial c_2} + a_3 \times \frac{\partial \mathfrak{L}}{\partial c_4}$$
$$\frac{\partial \mathfrak{L}}{\partial x_3} = a_2 \times \frac{\partial \mathfrak{L}}{\partial c_1} + a_4 \times \frac{\partial \mathfrak{L}}{\partial c_3}$$
$$\frac{\partial \mathfrak{L}}{\partial x_4} = a_2 \times \frac{\partial \mathfrak{L}}{\partial c_2} + a_4 \times \frac{\partial \mathfrak{L}}{\partial c_4}$$

This can be expressed as a matrix equation in the following manner-

$$\frac{\partial \mathfrak{L}}{\partial X} = \frac{\partial C}{\partial X} \times \frac{\partial \mathfrak{L}}{\partial C}$$
$$\implies \begin{bmatrix} \frac{\partial \mathfrak{L}}{\partial x_1} & \frac{\partial \mathfrak{L}}{\partial x_2} \\ \frac{\partial \mathfrak{L}}{\partial x_3} & \frac{\partial \mathfrak{L}}{\partial x_4} \end{bmatrix} = \frac{\partial C}{\partial X} \times \begin{bmatrix} \frac{\partial \mathfrak{L}}{\partial c_1} & \frac{\partial \mathfrak{L}}{\partial c_2} \\ \frac{\partial \mathfrak{L}}{\partial c_3} & \frac{\partial \mathfrak{L}}{\partial c_4} \end{bmatrix}$$

By comparing the individual partial derivatives computed earlier, we can say-

$$\begin{bmatrix} \frac{\partial \mathfrak{L}}{\partial x_1} & \frac{\partial \mathfrak{L}}{\partial x_2} \\ \frac{\partial \mathfrak{L}}{\partial x_3} & \frac{\partial \mathfrak{L}}{\partial x_4} \end{bmatrix} = \begin{bmatrix} a_1 & a_3 \\ a_2 & a_4 \end{bmatrix} \times \begin{bmatrix} \frac{\partial \mathfrak{L}}{\partial c_1} & \frac{\partial \mathfrak{L}}{\partial c_2} \\ \frac{\partial \mathfrak{L}}{\partial c_3} & \frac{\partial \mathfrak{L}}{\partial c_4} \end{bmatrix}$$
$$\implies \begin{bmatrix} \frac{\partial \mathfrak{L}}{\partial x_1} & \frac{\partial \mathfrak{L}}{\partial x_2} \\ \frac{\partial \mathfrak{L}}{\partial x_3} & \frac{\partial \mathfrak{L}}{\partial x_4} \end{bmatrix} = A^T \times \begin{bmatrix} \frac{\partial \mathfrak{L}}{\partial c_1} & \frac{\partial \mathfrak{L}}{\partial c_2} \\ \frac{\partial \mathfrak{L}}{\partial c_3} & \frac{\partial \mathfrak{L}}{\partial c_4} \end{bmatrix}$$
$$\implies \frac{\partial \mathfrak{L}}{\partial X} = A^T \times \frac{\partial \mathfrak{L}}{\partial C}$$
$$\therefore \frac{\partial C}{\partial X} = A^T \quad \square$$