cuKE: An Efficient Code Generator for Score Function Computation in Knowledge Graph Embedding

Lihan Hu
University of Iowa
lihan-hu@uiowa.edu

Jing Li
Nvidia
tinli@nvidia.com

Peng Jiang
University of Iowa
peng-jiang@uiowa.edu

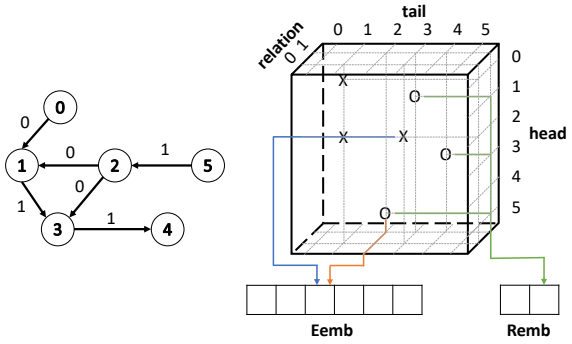
***Abstract*—Knowledge graph embedding (KGE) plays an important role in graph mining and learning applications by converting discrete graph structures to continuous vector representations. While previous systems have focused on scaling KGE onto multiple GPUs, the score function computation on each GPU can be a performance bottleneck. Existing KGE systems implement the score functions with separate tensor operations, leading to large memory consumption and poor memory access efficiency. To overcome the issues, we propose a code generator that automatically translates Python-like definitions of KGE score functions into efficient CUDA code. Our code generator exploits the unique feature of KGE score functions and performs an aggressive fusion of tensor operations. Additionally, our generated code performs a runtime inspection to reduce redundant memory access for edges with identical indices. Experiments show that our generated code uses much less memory than previous systems and achieves an average speedup of 14.9x over TorchScript and 7.8x over TVM.**

***Index Terms*—knowledge graph embedding, code generation, GPU**

I. Introduction

Knowledge graphs are used in many domains, such as social networks [1], knowledge bases [2], and bioinformatics [3], to represent entities and their relationships. Knowledge graph embedding (KGE) is a technique that converts discrete graph structures to continuous vector representations [4]. The learned vector representations are often used for clustering or link prediction tasks [5]. They can also serve as input to graph neural networks for downstream applications [6].

A KGE model assigns a vector (also known as an embedding) to each node and edge in the graph based on a score function. The score function measures the plausibility of edges. The objective is to maximize the plausibility of existing edges and minimize the plausibility of non-existing edges. Different KGE models use different score functions. For example, TransE [7] computes the score of an edge from head node h to tail node t with relation r as $||Eemb[h] - Eemb[t] + Remb[r]||_2$, where $Eemb$ represents the embeddings of all nodes and $Remb$ represents the embeddings of all relations. TransR [8] improves on TransE by multiplying the entity embeddings with a projection matrix before adding them to the relation embedding. Its score function can be written as $||(Eemb[h] - Eemb[t]) * Pm[r] + Remb[r]||_2$ where Pm represents the projection matrices. The score function is computed for a batch of sampled edges in each iteration of KGE training.



(a) Sampled edges (b) Redundant memory access

Fig. 1: Irregular memory access in KGE score function for a batch of sampled edges. Different edges may share the same head, tail, or relation, leading to redundant memory access. 'X' and 'O' represent sampled edges of relation types 0 and 1, respectively.

Several systems have been developed to facilitate KGE training, such as DGL-KE [9] and PyTorch-BigGraph [10]. These systems support large-scale KGE training on multiple GPUs and provide implementations of commonly used KGE models. However, the score functions implemented as sequences of batched matrix operations can be a performance bottleneck in these systems during KGE training. Our profiling of DGL-KE shows that computing the score functions can take up 50% to 80% of the total execution time. There are two inefficiencies with the naive implementation. First, they load data for each edge separately, leading to redundant data access and storage. As shown in Fig. 1, a batch of sampled edges may share the same heads, tails, or relations. Storing and accessing duplicate indices slows down the computation and prevents the training algorithm from using large batch sizes due to high memory consumption. Second, the computation requires multiple invocations of GPU kernels, which results in additional data movement across the GPU memory hierarchy

This work was supported by NSF award CNS-2310423.

for intermediate results.

To accelerate computation, one can write hand-optimized GPU kernels for different score functions. However, the variety of numerous existing functions and the constant emergence of new ones make manual implementation a daunting (if not impossible) task. Another approach is to use ML compilers such as TVM [11] and XLA [12] to automatically generate and optimize code for KGE functions. However, existing ML compilers are designed for DNNs and have limited support for sampled batch computations. First, they cannot optimize indirect memory access; their generated code has the redundant data access shown in Fig. 1b. Moreover, DNN compilers treat the batched vector multiplications in KGE functions as batched general matrix-matrix multiplications, and they miss many fusion opportunities for vector operations (explained in more detail in Sec. III).

In this paper, we propose `cuKE`, a domain-specific compiler that automatically generates and optimizes CUDA code for KGE score functions. Our system overcomes the limitations of existing ML compilers by leveraging two unique properties of KGE functions: 1) Many edges in a knowledge graph have the same relation types, and the redundant access to the same relation embeddings (or projection matrices) can be avoided by a runtime inspector; 2) The intermediate results of a KGE score function are always a batch of embedding vectors, and the operators can be efficiently fused by caching the intermediate data in GPU shared memory.

Our system allows the users to define a KGE score function as a sequence of Python-like tensor operations. It creates a computation graph based on the user's definition, analyzes the graph, and generates one single fused kernel for the function. To minimize the overhead of the runtime inspector, we developed an efficient GPU kernel that builds a mapping from the memory access indices of individual edges to the unique indices in each batch. During execution, our code accesses data from GPU global memory only for the unique indices. It caches the loaded data in GPU shared memory, which is reused by all edges in a batch.

We extensively evaluated our system by integrating it into DGL-KE to train different KGE models on real datasets. Compared to TorchScript [13] and TVM [11], our code allows training on larger batch sizes due to reduced memory consumption. In cases where TorchScript and TVM do not run out of memory, our code is up to 27.5x (14.9x on average) faster than TorchScript and up to 24.9x (7.8x on average) faster than TVM. We also compared our system with hand-optimized CUDA code that is manually fused based on our fusion strategy. Thanks to the runtime optimization, the code generated by `cuKE` is up to 3.6x (1.7x on average) faster than the hand-optimized code. The improved performance of score function computation leads to overall 1.3x to 2.3x speedups for KGE training over PyTorch.

II. Background

This section gives backgrounds on GPU architecture, knowledge graph embedding models, and current implementation of KGE score functions.

TABLE I: KGE terminology and notation: N is the number of nodes, M is the number of relations, Dim is the embedding length, and B is the batch size.

Notation	Description
$Eemb \in \mathbb{R}^{N \times Dim}$	embeddings of all entities
$Remb \in \mathbb{R}^{M \times Dim}$	embeddings of all relations
$Pv \in \mathbb{R}^{M \times Dim}$	projection vectors of all relations
$Pm \in \mathbb{R}^{M \times Dim \times Dim}$	projection matrices of all relations
$h \in [0..N)^B$	head indices of sampled edges
$t \in [0..N)^B$	tail indices of sampled edges
$r \in [0..M)^B$	relation indices of sampled edges

A. GPU Architecture

GPUs have been a major component of high-performance computing. They are specialized processors that feature massive parallelism. There are two types of parallelism on a GPU: single instruction multiple threads (SIMT) and multiple instruction multiple data (MIMD). GPU threads are organized into *warps*. Each warp has 32 threads that execute in SIMT fashion. Warps are further organized into *threadblocks* (also called cooperative thread arrays) and are launched together onto the streaming multiprocessors (SMs). Different warps can execute different instructions on different data. A typical GPU has tens of streaming multiprocessors, each of which has a number of registers and a programmable cache called *shared memory*. All threads in a threadblock can access the shared memory, which is much faster than the GPU's *global memory* but also much smaller. The typical size of shared memory on an SM is tens of KB. The NVIDIA RTX3090 GPU used in our experiment can have up to 100KB shared memory. When a threadblock uses more shared memory than what is available on an SM, it cannot be launched. A modern GPU can run more than 1K threads simultaneously on an SM, but the shared memory puts a limit on the number of active threads.

B. Knowledge Graph Embedding

A knowledge graph is a directed, labeled, multi-edge graph where the nodes represent entities and edges represent relations in certain domain. Knowledge graph embedding (KGE) is a set of techniques that convert knowledge graphs into continuous, low-dimensional vector representations. KGE has gained significant attention in recent years due to their ability to capture semantic relationships between entities and improve the accuracy of knowledge graph completion and other related tasks. Several popular KGE models have been proposed, including TransE [7], TransR [8], TransH [14], TransF [15], and RESCAL [16]. These models differ in their underlying assumptions and have different score functions for measure the plausibility of edges. Table II lists the score functions of popular KGE models, based on the notation in Table I. For example, TransE assumes the entity embeddings and relation embeddings are in the same vector space and computes the score function as the distance between $Eemb[h] + Remb[r]$

TABLE II: Score functions of different KGE models: $||\cdot||_p$ represents p-norm.

Model	Score Function
TransE [7]	$\|Eemb[h] - Eemb[t] + Remb[r]\|_{1/2}$
TransH [14]	$\|Eemb[h] - Eemb[t] + Remb[r] - Pv[r] \cdot (Eemb[h] - Eemb[t])^T \cdot Pv[r]\|_2$
TransR [8]	$\|(Eemb[h] - Eemb[t]) \cdot Pm[r] + Remb[r]\|_2$
TransF [15]	$2 * Eemb[h] \cdot Eemb[t] + (Eemb[t] - Eemb[h]) \cdot Remb[r]$
RESCAL [16]	$Eemb[h] \cdot Pm[r] \cdot Eemb[t]$

and $Eemb[t]$. TransR assume entities and relations have different embedding spaces. It projects the entity embedding into relation embedding space by multiplying it with a projection matrix $Pm[r]$.

KGEs are trained based on the open world assumption, which states that a knowledge graph contains only true facts, and non-observed facts can be either false or missing [17]. In each training iteration, a small set of positive facts (i.e., existing edges) is sampled from the graph, and for each positive fact, one or more negative facts (i.e., non-existing edges) are sampled. The objective is to maximize the scores of positive edges and minimize the score of negative edges. A common method to sample negative edges is to replace (corrupt) either the head or tail entity of a positive edge with a set of random entities [18]. All the negative edges corrupted from the same positive edge have the same relation type.

C. Current Implementation of KGE Score Functions

The existing KGE systems [9], [10] implement the score functions based on batched matrix operations. List. 1 shows the implementation of TransR score function in DGL-KE [9]. Given a batch of edges with heads h, tails t, and relations r, it first reads the entity embeddings and the projection matrix of each edge from $Eemb$, $Remb$ and Pm (line 1 to 4). Then, it calls a batched matrix multiplication kernel (bmm) provided by libraries such as cuBLAS [19] to project the entity embeddings into relation embedding space (line 5). Finally, the projected embeddings are added with the relation embeddings, and a norm function is called to compute the scores (line 6). We can see that the naive implementation needs to materialize the entity embeddings and the projection matrix for each edge in the batch, which can take a lot of memory space. For example, when the batch size $B = 4096$ and the embedding length $Dim = 1024$, the projection matrices take $B * Dim * Dim * sizeof(float) = 16$GB. This prevents the current KGE systems from using larger batch sizes, which may lead to slow convergence of the training algorithm [20].

III. Limitations with Existing ML Compilers for KGE Functions

Many compiler tools have been developed to automatically generate and optimize code for tensor computations on different platforms [11]–[13], [21], [22]. Their basic idea is to represent the computation as a graph and perform code generation and optimization based on the graph. For example, Fig. 2 shows the computation graph for the TransR function

```
    # h = [...], t = [...], r = [...]
    # read entity and relation embeddings
1   head_embs = Eemb[h]
2   tail_embs = Eemb[t]
3   rel_embs = Remb[r]
    # read projection matrices
4   proj_mats = Pm[r]
    # batched matrix multiplication
5   tmp = bmm(head_embs - tail_embs, proj_mats)
6   scores = norm(tmp + rel_embs)
```

Listing 1: TransR implemented as batched matrix operations in DGL-KE.

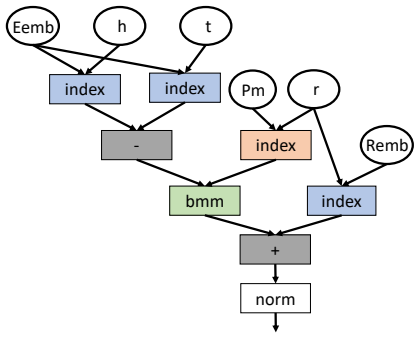


Fig. 2: Computation graph of TransR.

defined in Lst. 1. The compiler has a template implementation (usually in the form of intermediate representations) for each of the tensor operators, and it generates the code for the entire computation by taking a post-order traversal of the graph. The main power of these compilers comes from the automatic tuning of each operator to achieve the best parallelism or cache performance and the fusion between operators to avoid the materialization of intermediate data.

Oversimplified Batch Operator Disables Fusion: Since the ML compilers were designed for traditional DNNs, they have limited support for batched, irregular tensor computations. The first problem is that they treat all different types of batched matrix/vector multiplications as general matrix-matrix multiplication. For example, the multiplication of $Pv[r]$ and $(Eemb[h] - Eemb[t])$ in TransH is implemented as batched matrix-matrix multiplication even though the actual computation is batched vector inner-product. The oversimplification disables many operator fusion opportunities for KGE functions. For the TransH function, TVM generates two separate GPU kernels for $Pv[r] \cdot (Eemb[h] - Eemb[t])^T \cdot Pv[r]$: one for $Pv[r] \cdot (Eemb[h] - Eemb[t])^T$, and one for the multiplication of its result with $Pv[r]$. However, since we know that the result of the first multiplication is a batch of scalars, the two kernels can be easily fused together by caching the intermediate result in the GPU shared memory. Note that while recent ML compiler tools [22]–[24] can perform such fusion for a single edge, they do not support the fusion for batched computation. For instance, the RESCAL function in Table II has a matrix-vector multiplication ($Eemb[h] \cdot Pm[r]$) followed by a vector-vector multiplication. The first multiplication is a 'many-to-many' operator [22], and the second is an elementwise

multiplication followed by a reduction. For a single edge, the two operators can be fused together according to the fusion rules of these compilers [22], [24]. However, for a batch of edges, since both operators are represented as batched matrix-matrix multiplications and become 'many-to-many', they cannot be fused by these compilers.

Redundant Global Memory Access: Another limitation of existing ML compilers is that they focus on compile-time optimization and do not support runtime optimization for irregular computation. For KGE functions on a batch of edges, traditional ML compilers (e.g., TVM and XLA) need to load and store a copy of the data for each individual edge, even though many edges may use the same data as illustrated in Fig. 1. This leads to redundant data access and storage. While a recent compiler tool, FreeTensor [21], introduces loop constructs into tensor programs to express indirect tensor indexing and avoids the explicit storage of indexing results, it still needs to load data from GPU global memory for every loop iteration (i.e., for every edge for KGE functions), resulting in redundant global memory access.

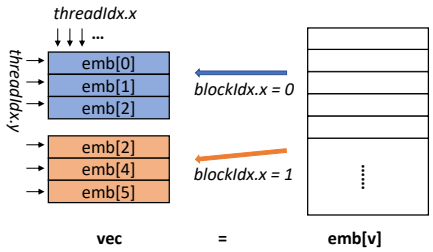
IV. Overview of cuKE

To overcome the limitations of existing solutions, we propose `cuKE`, a domain-specific compiler for generating efficient CUDA code for KGE functions.

We first extend the batch operators in existing tensor compilers to support batched vector operations. Specifically, we add batched scalar-vector multiplication (`bsv`), batched vector-vector multiplication (`bvv`), batched matrix-vector multiplication (`bmv`), and batched vector outer-product (`bov`), to complement the batched matrix-matrix multiplication (`bmm`) in the current implementations of KGE functions. We also add an operator for gathering a batch of vectors (`gather_vec`) and an operator for gathering a batch of matrices (`gather_mat`), and distinguish them from the general indexing operation. This extension of batch operators (as listed in Table III) enables automatic fusion of all operations in a KGE function. Details of our code generation and operator fusion methods are described in Sec.V.

An important optimization of our system is that we avoid loading redundant data from GPU global memory for the `gather_vec` and `gather_mat` operations. This is achieved by obtaining the unique data accessing indices in a batch of edges and caching the data that are accessed multiple times in shared memory. The main challenge is how to obtain the unique indices efficiently so that the overhead does not outweigh the benefit of reduced global memory traffic. We developed an efficient parallel algorithm that inspects the data loading indices for a batch of sampled edges at runtime and returns the unique indices. Details of the runtime optimization are described in Sec. VI.

For a KGE function, our system generates the CUDA code and wraps it (together with the runtime inspection code) into a PyTorch C++ extension [25]. The extension can be used to replace the original score function implementation in any PyTorch-based KGE system to accelerate the computation.



(a) Thread mapping

```
1  __shared__ float vec[C][D];
2  for (jj=0; jj<Dim; jj+=D) {
   /* gather_vec.BODY(emb, v, jj, vec) */
3    for (j=threadIdx.y*blockDim.x+threadIdx.x;
     j<D*v.uniq_idx[blockIdx.x].size;
     j+=blockDim.x*blockDim.y) {
4      vec[i/D][i%D] = emb[v.uniq_idx[blockIdx.x][i/D]][i
           %D+jj]; }}
```

(b) CUDA code

Listing 2: Loading a batch of embedding vectors from global memory to shared memory (`gather_vec`).

V. Code Generation for Batched Vector Operations

This section describes how our system generates code for the batched vector operators listed in Table III and how it fuses them together.

A. Code for Individual Operators

Since the computation of the score function for different edges is independent, a simple way to map the computation onto a GPU is to divide the edges into chunks and assign each chunk to a threadblock. To reduce the materialization of intermediate results, we can cache the data in shared memory. However, shared memory may not be sufficient to hold the entire intermediate data in most cases. For instance, for a chunk of 32 edges with an embedding length of 1024, a `BVec` takes $32 * 1024 * sizeof(float) = 128$KB, which exceeds the shared memory size of most GPUs. To fit the data in shared memory, we divide the embedding vector into blocks of D elements, where D is set to a small number (usually 32 or 64), and store one block in shared memory at a time.

Based on the above discussion, we implement the `gather_vec` operator as shown in Lst. 2. Suppose a threadblock processes a chunk of C edges. We allocate a buffer of size $C * D$ in shared memory. The kernel function has an outer loop that iterates over the output vector block by block (line 3). In each iteration, a warp of threads (with different $threadIdx.x$) loads D elements from an embedding vector (emb) to the shared memory buffer (vec) in a coalesced way (line 5 and 6). Different warps (which have different $threadIdx.y$) read data for different edges. As multiple edges may share the same index to emb, only the data of unique indices ($uniq\_idx$) are loaded. The loaded data are stored in the shared memory buffer, and a mapping (named buf_idx) from the actual indices to the positions of cached data in vec

TABLE III: Batched operators in cuKE. `BInt`, `BFloat`, `BVec`, and `BMat` represent a batch of integers, floating points, vectors, and matrices. The `bov` operator is only used in the backward pass and cannot be followed by other operators.

cuKE Operator Prototype	Overloading	Description
`gather_vec(BVec, BInt) -> BVec`	[]	Gather a batch of vectors or matrices based on the given indices.
`gather_mat(BMat, BInt) -> BMat`		
`scal_mul_vec(BFloat, BVec) -> BVec`	`bsv`	Multiply a batch of scalars with a batch of vectors.
`vec_mul_vec(BVec, BVec) -> BFloat`	`bvv`	Compute the inner-product of two batches of vectors.
`vec_mul_mat(BVec, BMat) -> BVec`	`bmv`	Multiply a batch of matrices with a batch of vectors.
`vec_outer_vec(BVec, BVec) -> BMat`	`bov`	Compute the outer-product of two batches of vectors.

```
1  __shared__ float vec3[C][D];
2  i = threadIdx.y;
3  for (jj=0; jj<Dim; jj+=D) {
     /* elementwise.BODY(vec1, vec2, vec3) */
     /* vec1 and vec2 are loaded or computed by jj */
4    for (j=threadIdx.x; j<D; j+=blockDim.x) {
5      vec3[i][j] = vec1[buf_idx[blockIdx.x][i]][j] +
           vec2[i][j]; }
     /* vec3 is consumed */ }
```

Listing 3: Element-wise addition for two batches of vectors.

```
1  float t = 0;
2  i = threadIdx.y;
3  for (jj=0; jj<Dim; jj+=D) {
     /* vec_mul_vec.BODY(vec1, vec2, t) */
     /* vec1 and vec2 are loaded or computed by jj */
4    for (j=threadIdx.x; j<D; j+=blockDim.x) {
5      t += vec1[buf_idx[blockIdx.x][i]][j] * vec2[i][
           j]; }}
6  t = warp_reduce(t);
```

Listing 4: Computing the inner-product of two batches of vectors (`vec_mul_vec`) .

is constructed. We will explain how $uniq\_idx$ and buf_idx can be efficiently obtained at runtime in Sec. VI.

The implementation of `gather_mat` is similar to `gather_vec`. We allocate a buffer of size $C1 * D * D$ in shared memory and load a $D * D$ submatrix from global memory into the buffer at a time for each edge. To avoid shared memory overflow, we set $C1$ to 2 in our implementation and store the two most frequently accessed matrices in shared memory for each threadblock. If a matrix is not cached in shared memory, its buf_idx is set to C plus its actual index, so that the program knows to load data from global memory.

Lst. 3 shows the code of element-wise operation on two batches of vectors. It processes the vectors block by block (line 3). The thread mapping is the same as the gather operations: different threads in a warp process different elements of a vector, and different warps process different vectors (line 4 and 5). The code assumes that the input vectors are available in shared memory buffers ($vec1$ and $vec2$) at the beginning of each iteration of the outer loop, and the output is consumed at the end of the iteration. If an input vector ($vec1$ in this case) is from a `gather_vec` operator, we need to access the data for an edge i using its index in the buffer ($buf\_idx[i]$). If an input vector ($vec2$ in this case) is the result of another element-wise operator, the data is directly accessed using index i.

For the `vec_mul_vec` operator, the inputs are two batches

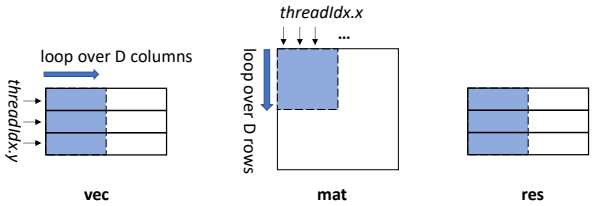
```
1  __shared__ float res[C][D];
2  i = threadIdx.y;
3  for (jj=0; jj<Dim; jj+=D) {
   /* scal_mul_vec.BODY(t, vec, res) */
   /* t is computed by jj*/
   /* vec is loaded or computed by jj*/
4  for (j=threadIdx.x; j<D; j+=blockDim.x) {
5    res[i][j] = t * vec[i][j]; }
   /* res is consumed */ }
```

Listing 5: Multiplying a batch of scalars with a batch of vectors (`scal_mul_vec`).

of vectors, and the kernel computes the inner product for each pair of input vectors. As shown in Lst. 4, it reads in D elements from the input vectors in each iteration, multiplies the corresponding elements, and accumulates the partial result in a register t. Here, the reading of the input vectors is the same as in the element-wise operations: if an input vector is from a `gather_vec` operator, the data should be loaded with $buf\_idx[i]$; if an input vector is from an element-wise operator, the data should be loaded with i directly. At the end of the outer loop, the values of t in different threads of a warp are added, and the final result is broadcast back to t. This step can done efficiently using CUDA warp-level primitives [26].

The implementation of `scal_mul_vec` is similar to element-wise multiplication, except that one of the inputs is scalar and is read directly from the register t produced by its preceding `vec_mul_vec` operator. The code is shown in Lst. 5.

For the `vec_mul_mat` operator, since only a small block of the input matrix can be stored in shared memory, we read D elements from the input vector and multiply them with the $D * D$ elements of the input matrix at a time. As shown in Lst. 6a, the threads in a warp (with different $threadIdx.x$) fetch multiple columns of the input matrix and multiply them with one element in the input vector. Different warps (with different $threadIdx.y$) conduct the multiplication for different vectors. The CUDA code is shown in Lst. 6b. It computes the output vectors block by block. When computing the first block (i.e., $jj == 0$), it assumes the input vector is available in vec from the preceding operator. Since the entire input vectors are repeatedly accessed for computing each block of the output vectors, we store the input vectors in global memory to avoid redundant computation (unless the preceding operator is a $gather\_vec$). When accessing the input matrix, we need to check the value of buf_idx (line 5-9). If $buf\_idx[blockIdx.x][i] < C$, which means



(a) Thread mapping

```
1  __shared__ float vec[C][D],mat[C1][D][D],res[C][D];
2  i = threadIdx.y;
3  for (jj=0; jj<Dim; jj+=D) {
     /* vec_mul_mat.BODY(vec, mat, res) */
4    for (kk=0; kk<Dim; kk+=D) {
       /* mat is loaded by jj*/
       /* vec is computed and stored when jj==0 and is
            loaded otherwise */
5      t = buf_idx[blockIdx.x][i];
6      pmat = t < C ? mat : Pm;
7      idx = t < C ? t : t - C;
8      ofs_k = t < C ? 0 : kk;
9      ofs_j = t < C ? 0 : jj;
10     for (j=threadIdx.x; j<D; j+=blockDim.x) {
11       for (k=0; k<D; k++) {
12         res[i][j] += vec[i][k] * pmat[idx][k+ofs_k][j+
               ofs_j]; }}}
     /* res is consumed */ }
```

(b) CUDA code

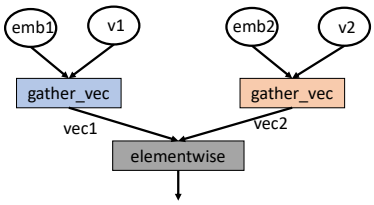
Listing 6: Multiplying a batch of vectors with a batch of matrices (`vec_mul_mat`).

the matrix is stored in the shared memory buffer, we fetch the data from mat. If $buf\_idx[i] \geq C$, which means the matrix is not cached, we load the data from the projection matrices (Pm) in global memory with the actual index $buf\_idx[blockIdx.x][i]-C$. In our actual implementation, we also cache res and vec in registers to exploit the data reuse in loop k and loop j.

The implementation of `vec_outer_vec` is similar to `vec_mul_vec`, with the difference that the results are accumulated in an output matrix.

B. Operator Fusion

As the astute readers may have noticed, the code in Lst. 2 to 6 shares the same outermost loop. This means that each operator consumes the input vectors in the same order that its preceding operators generate the outputs. Thanks to this property, our batched vector operators can be easily fused by combining their loop bodies. This allows us to generate a single CUDA kernel for any KGE function. For example, the subexpression $Eemb[h] - Eemb[t]$ in TransE and TransR is an element-wise '-' operator that takes the output of two `gather_vec` operators as input. As shown in Lst. 7, we put the loop bodies of the three operators in the same loop. The first `gather_vec` operator reads D elements of vectors from $Eemb$ into $vec1$. The second `gather_vec` operator reads D elements of vectors from $Eemb$ into $vec2$. Both $vec1$ and $vec2$ are in shared memory. The `elementwise` operator reads corresponding vectors from $vec1$ and $vec2$ based on the buf_idx of $v1$ and $v2$. It stores the output in $vec3$ in shared memory, which can be consumed by subsequent operators.

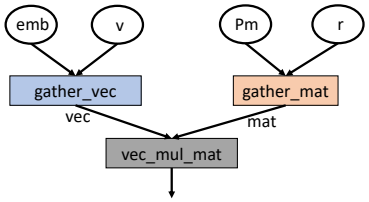


(a) Computation graph

```
1  __shared__ float vec1[C][D], vec2[C][D], vec3[C][D];
2  i = threadIdx.y;
3  for(jj=0; jj<Dim; jj+=D){
4    gather_vec.BODY(emb1, v1, jj, vec1);
5    gather_vec.BODY(emb2, v2, jj, vec2);
6    for(j=threadIdx.x; j<D; j+=blockDim.x){
7      vec3[i][j] = vec1[v1.buf_idx[blockIdx.x][i]][j]
                  - vec2[v2.buf_idx[blockIdx.x][i]][j];}}
```

(b) CUDA code

Listing 7: Fusing two gather operators with an elementwise operator. Different colors represent different operators in the computation graph.



(a) Computation graph

```
1  __shared__ float vec[C][D],mat[C1][D][D],res[C][D];
2  i = threadIdx.y;
3  for (jj=0; jj<Dim; jj+=D) {
4    for (kk=0; kk<Dim; kk+=D) {
5      gather_vec.BODY(emb, v, kk, vec);
6      gather_mat.BODY(Pm, r, jj, kk, mat);
7      for (j=threadIdx.x; j<D; j+=blockDim.x) {
8        for (k=0; k<D; k++) {
9          res[i][j] += vec[v.buf_idx[blockIdx.x][i]][k]
             * mat[r.buf_idx[blockIdx.x][i]][k][j];}}}}
```

(b) CUDA code

Listing 8: Fusing two gather operators with batched vector-matrix multiplication.

Next, we consider the fusion of `vec_mul_mat` with its preceding operators. The input vectors of `vec_mul_mat` can be generated by a `gather_vec` operator or an element-wise operator, whereas the input matrix must come from a `gather_mat`. The fused code is shown in Lst. 8. By embedding the code of `gather_vec` (or `elementwise`) and `gather_mat` into the loop body of `vec_mul_mat` (line 5 and 6), we obtain a block of D vector elements in vec and a block of $D * D$ matrix elements in mat. The inner loops (j and k) multiply them together and accumulate the results to the output vectors res. Here, we omit the case where the input matrix is not cached in shared memory for simplicity of illustration, but the code is similar to lines 5-9 in Lst. 6. As the output vectors are computed block by block, the loop body of the code snippet can be further embedded into subsequent

operators. The fusion of `vec_outer_vec` with its preceding operators can be conducted in the same way.

While shared memory is fast on a GPU, we can further improve performance and reduce shared memory usage by keeping intermediate data in registers in some cases. When two `elementwise` operators are fused, we need not store the output of the first operator in shared memory; it can be directly consumed by the subsequent `elementwise` operator in registers. For example, the element-wise '-' operator in Lst. 7 can be combined with an element-wise '+' operator to obtain the code for the TransE score function: $Eemb[h] - Eemb[t] + Remb[r]$. Since the two operators have the same inner and outer loops, we can keep the result of $vec1[h.buf\_idx[i]][j] - vec2[t.buf\_idx[i]][j]$ in registers when adding it to the corresponding element in $Remb[r]$. Another scenario is when an `elementwise` operator is combined with a `vec_mul_mat` operator. Each thread holds one element of the input vectors produced by `elementwise`. Since the vector-matrix multiplication for an edge is computed within a warp, the vector elements can be shuffled to other threads in the warp using CUDA warp-level primitives [26].

The remaining case for fusion is between `vec_mul_vec` and `scal_mul_vec` where the result of the first operator (a batch of scalars) is given to the second operator as input. The fusion can be easily achieved by inserting the code of Lst. 4 (with a changed loop iterate name) into the outer loop of Lst. 5.

VI. Runtime Inspection to Avoid Redundant Data Access

As shown in Fig. 1b, different edges in a sampled batch may share the same head, tail, or relation during KGE training. To avoid redundant data access, our `gather_vec` and `gather_mat` operators only load the embedding vectors or projection matrices for edges with unique indices ($uniq\_idx$ in Lst. 2) into shared memory. When subsequent operators use this data, they access it from shared memory based on the locations in the buffer (buf_idx in Lst. 3). We now explain how the $uniq\_idx$ and buf_idx can be efficiently constructed at runtime.

A. Building Unique and Buffer Index

Fig. 7 illustrates the index-building procedure for the tail indices of the six edges from Fig. 1a. The indices are $(1, 3, 1, 3, 4, 2)$. We first load the indices into shared memory and sort them. The mapping between the unsorted and sorted indices is returned in an array ord where $ord[i] = j$ means $unsorted[i]$ is stored in $sorted[j]$. Then, each thread $i > 0$ checks if $sorted[i]$ is the same as $sorted[i-1]$. If not, the thread writes 1 to $is\_uniq[i]$, indicating that $sorted[i]$ is a unique index. In this example, thread-2,3,5 find that their corresponding values are larger than the previous ones, so they write 1 to $is\_uniq[2]$, $is\_uniq[3]$, and $is\_uniq[5]$. After all the threads finish writing into is_uniq, a prefix sum is computed on is_uniq, which gives us the positions of unique indices in the buffer (buf_idx).

Since it is only beneficial to cache the repeatedly accessed data in shared memory, we need to select the edges based on their access counts. To obtain the access counts of different indices, each thread $i > 0$ checks again if $sorted[i]$ is the same as $sorted[i-1]$; if not, thread-i writes i to $pcount[buf\_idx[i]]$. The values in $pcount$ are the prefix sum of the access counts of different indices. The access count of each index can then be obtained by $pcount[i+1] - pcount[i]$ for all but the last unique index. The access count of the last unique index is $C - pcount[buf\_idx[last]]$.

Next, we select the most frequent indices based on the $count$. This is achieved by writing $sorted[i]$ to $cache[buf\_idx[i]]$ for all $i > 0$ that have $sorted[i] \neq sorted[i-1]$ and $i = 0$, and sorting $cache$ based on $count$. The positions of $cache[i]$ in the $sorted\_cache$ are stored in $cache\_ord[i]$. For projection matrices, $uniq\_idx$ is simply the first $C1$ elements from $sorted\_cache$. For embedding vectors, $uniq\_idx$ contains the first few elements from $sorted\_cache$ with counts larger than a threshold.

Last, according to the $uniq\_idx$, we update the buf_idx. Each thread checks if $sorted[i]$ is in $uniq\_idx$ (by checking if $uniq\_idx[cache\_ord[buf\_idx[i]]]$ is valid or not). If it is, the thread writes $cache\_ord[buf\_idx[i]]$ to $buf\_idx[i]$. If a thread finds $sorted[i]$ is not in $uniq\_idx$, it writes $C + sorted[i]$ to $buf\_idx[i]$. The values in buf_idx are finally shuffled based on ord to recover the order of the original unsorted edges.

The overhead of this runtime inspection is small compared to the KGE function as each step has $O(C)$ or $O(C \log(C))$ complexity and can be done in parallel within the shared memory.

B. Improving Data Reuse with On-the-Fly Edge Reordering

In real knowledge graphs, the number of edges is typically much larger than the number of relation types, meaning that there are many duplicate relations in each batch. If we reorder the edges and group together edges of the same relation in each batch, there will be fewer unique indices in each chunk of C edges. This leads to better data reuse with relation embeddings and projection matrices in each threadblock. This reordering can be accomplished by sorting the edges based on their relation indices. However, since the reordering needs to be performed in each training iteration, sorting the entire batch could be expensive in comparison to its benefits. To reduce the overhead of reordering, we can divide the edges into smaller groups and only sort the edges within each group. The group size is set to pC where p is a configurable parameter. When $p = 1$, only the duplicate indices within each threadblock are considered, resulting in the same data reuse as unsorted edges. As p increases, the sorting becomes more expensive but also brings more edges of the same relation together, thus achieving better data reuse.

Fig. 8 shows the number of unique relation indices processed by all threadblocks with and without edge reordering. We test with a batch size of 8192, $C = 16$, and different values of p, on different input graphs (listed in Table IV). The batches are randomly sampled from the input graphs and the

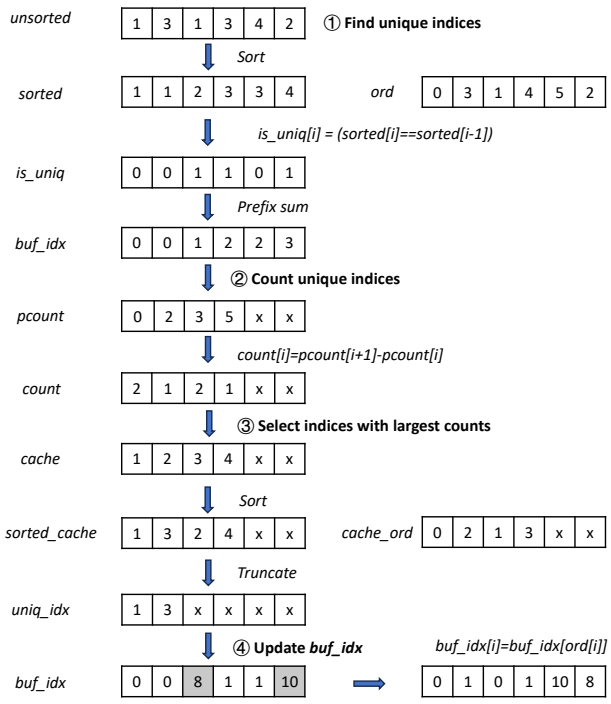


Fig. 7: A running example of parallel index-building. 'x' represents undefined values.

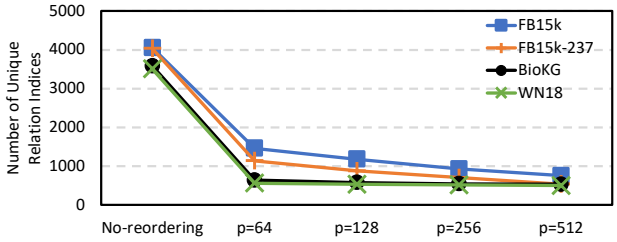


Fig. 8: Number of unique relation indices processed by all threadblocks with and without edge reordering.

numbers reported are the average of 1000 runs. We can see that the number of unique indices is significantly reduced (3.4x to 7x) by sorting or partially sorting the edges. The results validate the effectiveness of our edge-reordering technique. As p increases, the number of unique indices decreases, but the sorting overhead increases. In our experiments, we set $p = 128$ as it achieves good data reuses while maintaining a small sorting overhead.

VII. Experiment Evaluations

This section presents an evaluation of our system and compares it with existing tools for implementing KGE functions.

A. Experiment Setup

Datasets: We run our experiments on four knowledge graphs: FB15k, FB15k-237, BioKG and WN18, as listed in Table IV. The four graphs are commonly used for evaluating KGE systems [9], [10], [20]. FB15k and FB15k-237 are derived from the Freebased Knowledge Graph [27], WN18 is derived from WordNet [28], and BioKG is derived from BioDBLinker [29].

TABLE IV: Knowledge graph datasets.

Model	# nodes	# edges	# relations
FB15k [27]	14951	592213	1345
FB15k-237 [27]	14541	310116	237
BioKG [29]	93773	5088434	51
WN18 [28]	40943	151442	18

Baselines: We compare the CUDA code generated by our system with three baselines:

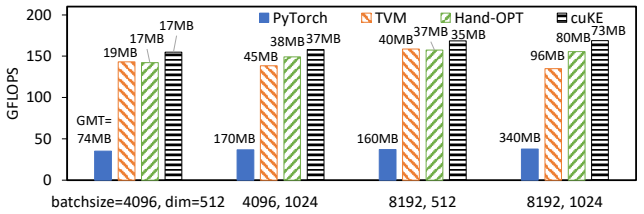
1) PyTorch-based implementations in DGL-KE [9]. The score functions are implemented with basic tensor operations supported by cuBLAS [19]. We also compile the functions with TorchScript [13]. The compiled code does not always outperform the original Python implementation. We report the performance of the better one between the compiled and uncompiled code.
2) TVM [11] is an open-source machine learning compiler for different platforms. In the latest version, TVM provides a DSL called Relay to represent the computation in neural networks [30]. We use Relay to implement the KGE functions and test the performance of its generated CUDA code.
3) We also write hand-optimized CUDA code for the KGE functions in forward computation. The code is manually fused in the same way as described in Sec. V. However, it loads data for every edge in a batch. We compare this hand-optimized version with the code generated by cuKE to demonstrate the benefits of our runtime optimization technique.

Platform and Settings: Our experiments are conducted on an Nvidia RTX3090 GPU, which has 82 SMs running at 1.4 GHz, and 24 GB global memory with a bandwidth of 936 GB/s. We use NVCC 11.7 for CUDA code compilation. The execution time was measured as the average of 3000 training iterations. We set $C = 16$, $D = 64$, $C1 = 2$ in our code for different operators in Lst. 2∼ 6.

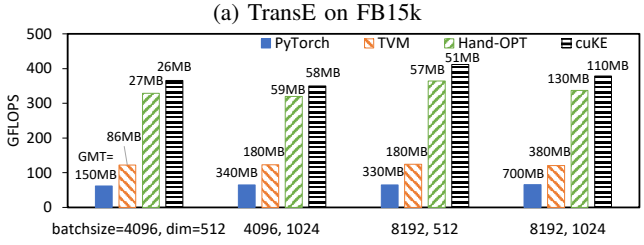
B. Performance of Score Function Computation

Fig. 9a shows the throughput of TransE positive score computation with different batch sizes and embedding dimension lengths on FB15k graph. The global memory traffic sizes are labeled above the bars. PyTorch/TorchScript has the lowest throughput due to the largest memory traffic. This is because TorchScript has the lowest fusion ability. Our profiling with Nvidia Nsight [31] shows that TorchScript cannot even fuse index operators with an element-wise operator. All the other three versions can fuse index operations with element-wise, avoiding the explicit storage of indexing results in GPU global memory. The code generated by our system is slightly faster than TVM and the hand-optimized code because it only loads data for unique indices.

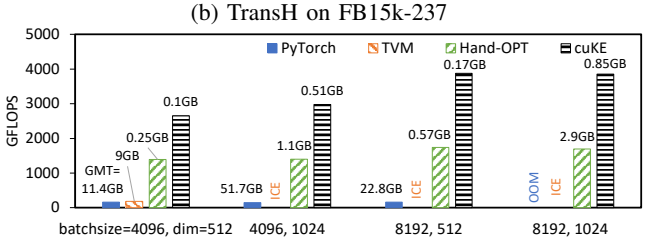
The advantage of our system's fusion ability is more evident in the performance results of the TransH function.



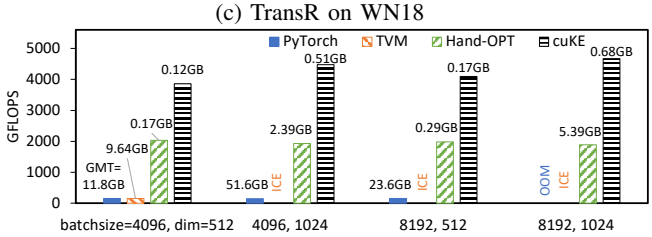
(a) TransE on FB15k



(b) TransH on FB15k-237



(c) TransR on WN18

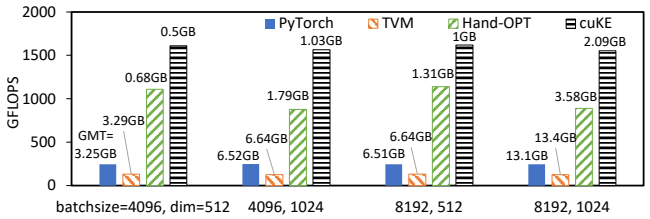


(d) RESCAL on BIOKG

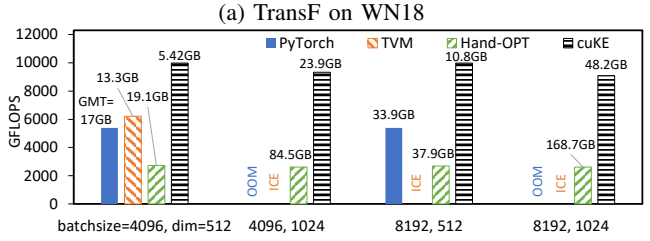
Fig. 9: Throughput of positive score computation. ICE means internal compilation error. OOM means out-of-memory. Global memory traffic (GMT) is labeled on each bar.

Figure 9b shows that TorchScript and TVM have low throughput because of high global memory traffic. According to the CUDA code generated by TVM, TVM does not fuse index operators with an element-wise operator when the indexing results are used by multiple terms in the function. TVM does fuse the inner-product operator with element-wise (i.e., $Pv[r] \cdot (Eemb[h] - Eemb[t])^T$), but it misses further fusion for the multiplication of the intermediate results with $Pv[r]$ again. Our hand-optimized code has similar global memory traffic as cuKE; however, cuKE achieves better performance due to the runtime optimization that avoids redundant memory access. We achieve an average speedup of 5.9x compared to TorchScript, 3.1x compared to TVM, and 1.2x compared to hand-optimized code.

While operator fusion generally leads to increased register and shared memory usage, it does not affect GPU utilization in our experiments. In all our test cases, each thread uses at most 160 bytes of registers and 36 bytes of shared memory. Since the default setting of each SM has 256KB register and



(a) TransF on WN18



(b) TransR on FB15k-237

Fig. 10: Throughput of negative score computation.

48KB shared memory on our test platform, all threads on an SM can be launched at the same time. Our profiling results with Nvidia Nsight show that the warp occupancy of our fused kernels ranges from 75% to 93%.

The benefit of our runtime optimization might seem marginal for TransE and TransH when compared with the hand-optimized code. This is because the two functions only use embedding vectors. When the score function involves loading projection matrices, our runtime optimization can save much more global memory traffic. This can be seen from the performance results of TransR and RESCAL in Fig. 9c and 9d. TorchScript and TVM have low throughput and large memory traffic due to their limited fusion ability. Since they need to explicitly store the copy of the projection matrices for the given indices, TorchScript runs out of memory during execution and TVM reports internal compilation errors (due to an overflow of threadblock numbers) when the batch size and dimension length are large. Although the global memory traffic is significantly reduced by the manual fusion in the hand-optimized code, it still needs to access the projection matrices multiple times for duplicate indices. Our runtime optimization further reduces the global memory traffic by 1.4x to 7.9x. For TransR, our code achieves an average speedup of 20.6x compared to TorchScript, 14.2x compared to TVM, and 2.1x compared to the hand-optimized code. For RESCAL, we achieve an average speedup of 27.5x compared to TorchScript, 24.9x compared to TVM, and 2.2x compared to hand-optimized code.

The performance results of negative score computation follow a similar pattern as positive computation. Since the negative edges corrupted from each positive edge share the same head (or tail) and relation, our CUDA kernel can easily benefit from this property and load the entity embedding and projection matrix only once for a group of negative edges. As the memory access is more regular, negative score computation in general achieves higher throughputs than positive computation. As shown in Fig. 10, for the TransF function,

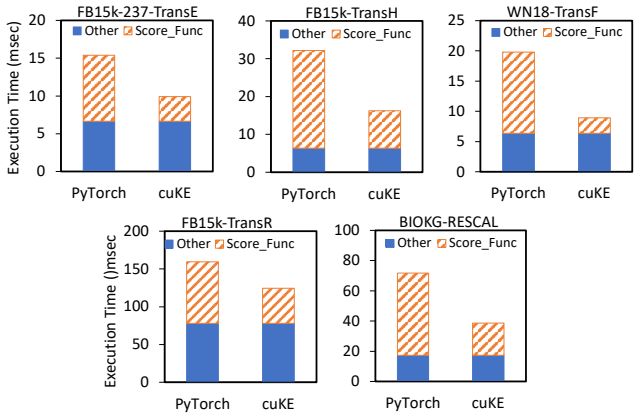


Fig. 11: Execution time per iteration for different score functions and input graphs (batch_size=4096 and dim=512).

our system achieves an average speedup of 6.4x, 12.2x, and 1.6x against TorchScript, TVM, and hand-optimized code, due to the reduced global memory traffic. For TransR, the average speedups against TorchScript, TVM, and hand-optimized code are 1.9x, 1.6x, and 3.6x. For RESCAL, the average speedups against the other three versions are 3.8x, 4.4x, and 1.2x, respectively. We skip the figures for TransE and RESCAL due to the space limit.

C. Overall Performance

To demonstrate the performance gains of our method for end-to-end KGE training, we compare the overall training time with PyTorch. Fig. 11 shows the breakdown execution time in each iteration for different score functions and input graphs with batch size 4096 and dimension 512.

The training process includes both forward and backward passes. We have explained the forward performance in the previous subsection. Our system does not support automatic differentiation, but we can easily generate code for backward computation by explicitly writing the formulas for computing the gradients. For example, suppose $dout$ is the gradient of loss w.r.t $((Eemb[h] - Eemb[t]) \cdot Pm[r] + Remb[r])$ in the TransR function. The gradient w.r.t $Remb[r]$ is simply $dout$. The gradient w.r.t $Eemb[h]$ and $Eemb[t]$ are $\pm\texttt{bmv}(dout, Pm[r])$. The gradient w.r.t $Pm[r]$ is $\texttt{bov}(dout, Eemb[h] - Eemb[t])$. Since $\texttt{bov}$ is always used for computing gradients w.r.t projection matrices and is not followed by any other operators, our fusion strategy described in Sec. V-B still applies. The backward computation also benefits from the runtime optimization in the same way as the forward computation. The results show that our system improves the end-to-end performance of KGE training by 1.3x to 2.3x over PyTorch.

D. Memory Consumption

We now compare the memory consumption of our code with PyTorch implementation. TVM has a similar memory consumption to PyTorch as they both store the embedding vectors and projection matrices for all edges in a batch. The only slight advantage is that TVM can fuse some computation kernels to reduce the storage of intermediate data. Since the entity embeddings and projection matrices of the entire graph are usually distributed on multiple devices for large-scale KGE training [9], both our and PyTorch implementations need to load and store a copy of the data for the unique entities and relations in a batch on the current device. In general, this data is small since there are few relation types in a batch. Table V shows the memory consumption of our code for computing TransR score functions on four different graphs with different batch sizes. In comparison, PyTorch needs to store the projection matrix for each individual edge. When the dimension length is set to 512, each embedding vector takes $512 * sizeof(float) = 2$KB, and each projection matrix takes $512 * 512 * sizeof(float) = 1$MB. The lower bound of memory usage for a batch can be calculated as $(3 * 2KB + 1MB) * batch\_size$, which are 4GB, 8GB, 16GB for a batch size of 4096, 8192, or 16384. According to Table V, our code requires less than 10% of the memory used by PyTorch.

TABLE V: Memory usage (in GB) of our code for computing the TransR positive score function with different batch sizes and dimension 512.

	4096	8192	16384
FB15k	0.77	0.99	1.37
FB15k-237	0.43	0.56	0.72
BioKG	0.16	0.23	0.33
WN18	0.15	0.27	0.45

VIII. Related Work

KGE systems. Several packages have been developed for knowledge graph embedding and scaling the computation to large knowledge graphs. OpenKE [32] is an early package for KGE training and provides implementations for a large list of KGE models. DGL-KE [9] is an open-source library for KGE that is implemented using the Deep Graph Library (DGL). It features distributed storage of embeddings and projection matrices and support for large-scale KGE training. Pytorch-BigGraph (PBG) [10] is developed with an emphasis on scalability to large graphs with millions of nodes and edges. However, it did not support GPU training. These systems implement KGE score functions based on basic tensor operations provided by PyTorch or other operator-based ML libraries.

Operator-based ML libraries. There are multiple operator-based machine learning libraries that can be used to build graph embedding models. PyTorch [33] is a popular deep-learning framework known for its ease of use, dynamic computation graph, and good support for GPU acceleration. MXNet [34] and TensorFlow [35] also represent a program with a computation graph, where each node represents a tensor operation. JAX [36] allows the use of Python's dynamic control flow while providing efficient vectorized operations and automatic differentiation.

ML compilers. PyTorch and TensorFlow support code optimization and generation based on their computation graphs. PyTorch code can be compiled to TorchScript [13], which is a high-performance and portable Python execution environment that can be used for deploying PyTorch models. TensorFlow code can be optimized by XLA [12], which is a domain-specific compiler for linear algebra operations. TVM [11] supports highly customized operators by introducing a compute-and-schedule programming model, where users first specify the mathematical definition of computation and then optimize it with explicit or machine-learning-guided transformations. FreeTensor [21] is a DSL designed for expressing irregular tensor computations. It supports free-form tensor expressions, which allow users to express computations in a natural and intuitive way. Compared to existing ML compilers, our work leverages the specific features of KGE score functions and achieves more efficient operator fusion and irregular memory access optimization.

IX. Conclusion

In this paper, we propose cuKE, a domain-specific compiler for generating KGE score functions on GPU. The main idea is to extend the tensor operations in traditional ML compilers with batched tensor operators. Based on the batch representation, our system automatically fuses the operators to avoid storage of intermediate data in GPU global memory. We also avoid redundant data access by using a runtime inspector. The experiments show that our system achieves significant speedups against TorchScript and TVM.

References

[1] R. Zafarani, M. A. Abbasi, and H. Liu, *Social media mining: an introduction*. Cambridge University Press, 2014.

[2] K. M. Fairchild, S. E. Poltrock, and G. W. Furnas, "Semnet: Three-dimensional graphic representations of large knowledge bases," in *Cognitive science and its applications for human-computer interaction*. Psychology Press, 2013, pp. 215–248.

[3] S. Brohee and J. Van Helden, "Evaluation of clustering algorithms for protein-protein interaction networks," *BMC bioinformatics*, vol. 7, pp. 1–19, 2006.

[4] 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*, vol. 29, no. 12, pp. 2724–2743, 2017.

[5] A. Rossi, D. Barbosa, D. Firmani, A. Matinata, and P. Merialdo, "Knowledge graph embedding for link prediction: A comparative analysis," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 15, no. 2, pp. 1–49, 2021.

[6] "Www-18: Tutorial representation learning on networks," http://snap.stanford.edu/proj/embeddings-www/, 2018.

[7] A. Bordes, N. Usunier, A. García-Durán, J. Weston, and O. Yakhnenko, "Translating embeddings for modeling multi-relational data," in *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, C. J. C. Burges, L. Bottou, Z. Ghahramani, and K. Q. Weinberger, Eds., 2013, pp. 2787–2795. [Online]. Available: https://proceedings.neurips.cc/paper/2013/hash/1cecc7a77928ca8133fa24680a88d2f9-Abstract.html

[8] Y. Lin, Z. Liu, M. Sun, Y. Liu, and X. Zhu, "Learning entity and relation embeddings for knowledge graph completion," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, B. Bonet and S. Koenig, Eds. AAAI Press, 2015, pp. 2181–2187. [Online]. Available: http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9571

[9] D. Zheng, X. Song, C. Ma, Z. Tan, Z. Ye, J. Dong, H. Xiong, Z. Zhang, and G. Karypis, "DGL-KE: training knowledge graph embeddings at scale," in *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval, SIGIR 2020, Virtual Event, China, July 25-30, 2020*, J. X. Huang, Y. Chang, X. Cheng, J. Kamps, V. Murdock, J. Wen, and Y. Liu, Eds. ACM, 2020, pp. 739–748. [Online]. Available: https://doi.org/10.1145/3397271.3401172

[10] A. Lerer, L. Wu, J. Shen, T. Lacroix, L. Wehrstedt, A. Bose, and A. Peysakhovich, "Pytorch-biggraph: A large scale graph embedding system," in *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*, A. Talwalkar, V. Smith, and M. Zaharia, Eds. mlsys.org, 2019. [Online]. Available: https://proceedings.mlsys.org/book/282.pdf

[11] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Q. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: an automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, A. C. Arpaci-Dusseau and G. Voelker, Eds. USENIX Association, 2018, pp. 578–594. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/chen

[12] A. Sabne, "Xla : Compiling machine learning for peak performance," 2020.

[13] "Torchscript," https://pytorch.org/docs/stable/jit.html, 2023.

[14] Z. Wang, J. Zhang, J. Feng, and Z. Chen, "Knowledge graph embedding by translating on hyperplanes," in *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, C. E. Brodley and P. Stone, Eds. AAAI Press, 2014, pp. 1112–1119. [Online]. Available: http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8531

[15] J. Feng, M. Huang, M. Wang, M. Zhou, Y. Hao, and X. Zhu, "Knowledge graph embedding by flexible translation," in *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifteenth International Conference, KR 2016, Cape Town, South Africa, April 25-29, 2016*, C. Baral, J. P. Delgrande, and F. Wolter, Eds. AAAI Press, 2016, pp. 557–560. [Online]. Available: http://www.aaai.org/ocs/index.php/KR/KR16/paper/view/12887

[16] M. Nickel, V. Tresp, and H. Kriegel, "A three-way model for collective learning on multi-relational data," in *Proceedings of the 28th International Conference on Machine Learning, ICML 2011, Bellevue, Washington, USA, June 28 - July 2, 2011*, L. Getoor and T. Scheffer, Eds. Omnipress, 2011, pp. 809–816. [Online]. Available: https://icml.cc/2011/papers/438_icmlpaper.pdf

[17] L. Drumond, S. Rendle, and L. Schmidt-Thieme, "Predicting RDF triples in incomplete knowledge bases with tensor factorization," in *Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26-30, 2012*, S. Ossowski and P. Lecca, Eds. ACM, 2012, pp. 326–331. [Online]. Available: https://doi.org/10.1145/2245276.2245341

[18] D. Krompaß, S. Baier, and V. Tresp, "Type-constrained representation learning in knowledge graphs," in *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*, ser. Lecture Notes in Computer Science, M. Arenas, Ó. Corcho, E. Simperl, M. Strohmaier, M. d'Aquin, K. Srinivas, P. Groth, M. Dumontier, J. Heflin, K. Thirunarayan, and S. Staab, Eds., vol. 9366. Springer, 2015, pp. 640–655. [Online]. Available: https://doi.org/10.1007/978-3-319-25007-6_37

[19] "The cuda basic linear algebra subroutine library," https://docs.nvidia.com/cuda/cublas/, 2023.

[20] J. Mohoney, R. Waleffe, H. Xu, T. Rekatsinas, and S. Venkataraman, "Marius: Learning massive graph embeddings on a single machine," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI) 21*, 2021.

[21] S. Tang, J. Zhai, H. Wang, L. Jiang, L. Zheng, Z. Yuan, and C. Zhang, "Freetensor: a free-form DSL with holistic optimizations for irregular tensor programs," in *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, R. Jhala and I. Dillig, Eds. ACM, 2022, pp. 872–887. [Online]. Available: https://doi.org/10.1145/3519939.3523448

[22] W. Niu, J. Guan, Y. Wang, G. Agrawal, and B. Ren, "Dnnfusion: accelerating deep neural networks execution with advanced operator fusion," in *Proceedings of the 42nd ACM SIGPLAN International*

Conference on Programming Language Design and Implementation, 2021, pp. 883–898.

[23] Z. Zheng, X. Yang, P. Zhao, G. Long, K. Zhu, F. Zhu, W. Zhao, X. Liu, J. Yang, J. Zhai *et al.*, "Astitch: enabling a new multi-dimensional optimization space for memory-intensive ml training and inference on modern simt architectures," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 359–373.

[24] J. Zhao, X. Gao, R. Xia, Z. Zhang, D. Chen, L. Chen, R. Zhang, Z. Geng, B. Cheng, and X. Jin, "Apollo: Automatic partition-based operator fusion through layer by layer optimization," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 1–19, 2022.

[25] "Pytorch c++ extension," https://pytorch.org/tutorials/advanced/cpp_extension.html, 2023.

[26] "Using cuda warp-level primitives," https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/, 2018.

[27] K. D. Bollacker, C. Evans, P. K. Paritosh, T. Sturge, and J. Taylor, "Freebase: a collaboratively created graph database for structuring human knowledge," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, J. T. Wang, Ed. ACM, 2008, pp. 1247–1250. [Online]. Available: https://doi.org/10.1145/1376616.1376746

[28] G. A. Miller, "WORDNET: A lexical database for english," in *Human Language Technology, Proceedings of a Workshop held at Plainsboro, New Jerey, USA, March 8-11, 1994*. Morgan Kaufmann, 1994. [Online]. Available: https://aclanthology.org/H94-1111/

[29] B. Walsh, S. K. Mohamed, and V. Novácek, "Biokg: A knowledge graph for relational learning on biological data," in *CIKM '20: The 29th ACM International Conference on Information and Knowledge Management, Virtual Event, Ireland, October 19-23, 2020*, M. d'Aquin, S. Dietze, C. Hauff, E. Curry, and P. Cudré-Mauroux, Eds. ACM, 2020, pp. 3173–3180. [Online]. Available: https://doi.org/10.1145/3340531.3412776

[30] "Introduction to relay ir," https://tvm.apache.org/docs/arch/relay_intro.html.

[31] "Nvidia nsight systems," https://developer.nvidia.com/nsight-systems.

[32] X. Han, S. Cao, X. Lv, Y. Lin, Z. Liu, M. Sun, and J. Li, "Openke: An open toolkit for knowledge embedding," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, EMNLP 2018: System Demonstrations, Brussels, Belgium, October 31 - November 4, 2018*, E. Blanco and W. Lu, Eds. Association for Computational Linguistics, 2018, pp. 139–144. [Online]. Available: https://doi.org/10.18653/v1/d18-2024

[33] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[34] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *CoRR*, vol. abs/1512.01274, 2015. [Online]. Available: http://arxiv.org/abs/1512.01274

[35] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[36] R. Frostig, M. J. Johnson, and C. Leary, "Compiling machine learning programs via high-level tracing," *Systems for Machine Learning*, vol. 4, no. 9, 2018.