marius
Learning Massive Graph Embeddings on a Single Machine

Jason Mohoney, Roger Waleffe, Yiheng Xu,
Theodoros Rekatsinas, Shivaram Venkataraman

University of Wisconsin-Madison
Graphs are **universal representations** of rich semantics about entities (nodes) and their relationships (edges).
Graph Embeddings

Objective: Apply modern ML on graphs

Transform node and edge-types into embeddings (vectors)

Example Tasks:
- Link Prediction (focus of this work)
- Node classification
- Graph classification

Subgraph of the Freebase knowledge graph
Graph Embeddings

Objective: Apply modern ML on graphs

Transform node and edge-types into embeddings (vectors)

Example Tasks:
- Link Prediction (focus of this work)
- Node classification
- Graph classification

Subgraph of the Freebase knowledge graph
Learning Graph Embeddings

Training requires iterating over all edges and retrieving/updating embedding vectors

Training Process

// E ordered randomly
for (s, r, d) in E:

// compute loss of model for an edge
computeLoss(s, r, d)

// apply updates to embeddings of edge
update(s, r, d)

\[ G = (V, R, E) \]
Learning Graph Embeddings

Training requires iterating over all edges and retrieving/updating embedding vectors

**Batched Training**

```cpp
// E randomly grouped into batches
for batch in E:
    // compute loss of model for a batch
    computeLoss(batch)
    // apply updates to embeddings in a batch
    update(batch)
```

\[ G = (V, R, E) \]
Learning Graph Embeddings

Training requires iterating over all edges and retrieving/updating embedding vectors

**Batched Training:** single iteration
batch = [(JB, Born, M), (M, Capital, W)]

// load embeddings
computeLoss(batch)

// update embeddings
update(batch)

\[ G = (V, R, E) \]
Learning Graph Embeddings

Training requires iterating over all edges and retrieving/updating embedding vectors

Batched Training: single iteration

\[
\text{batch} = [(\text{JB, Born, M}), (\text{M, Capital, W})]
\]

// load embeddings and compute loss
computeLoss(batch)

// update embeddings
update(batch)
Batched Training: single iteration

```plaintext
batch = [(JB, Born, M), (M, Capital, W)]
```

// load embeddings and compute loss
```
computeLoss(batch)
```

// update embeddings
```
update(batch)
```
Learning Graph Embeddings

Training requires iterating over all edges and retrieving/updating embedding vectors

Batched Training: single iteration

batch = [(JB, Born, M), (M, Capital, W)]

// load embeddings and compute loss
computeLoss(batch)

// update embeddings
update(batch)

Training requires efficient access to embedding parameters

Graph with batch highlighted

Irregular Access
Key Challenge: Data Movement

Large Datasets

Freebase86m:
- 338 million edges, 86 million nodes, 15,000 edge types
- Size of node embedding table for d = 400:
  \[ 86 \text{ million} \times 400 \times 4 \text{ bytes} = 138 \text{ GB} \]

AWS P3.2xLarge instance:
- 16 GB GPU Memory
- 61 GB CPU Memory

Embedding table unable to fit in GPU memory!

Moving embeddings to compute

How to scale?

1. Store embeddings in CPU memory and transfer to GPU(s)
   - Bottlenecked by transfer overheads
   - Limited scalability

2. Partition node embeddings and store on disk
   - Limited by disk throughput

3. Distribute embeddings across multiple machines
   - Bottlenecked by transfer overheads
   - Expensive

Can the data movement bottlenecks be mitigated?
Scaling to Large Graphs: marius

Design Goal: Eliminate data movement overheads inherent in graph embedding training

Method
- Use pipelining and async IO to hide data movement
- Utilize the full memory hierarchy with a partition buffer
- Minimize IO with Buffer-aware Edge Traversal Algorithm (BETA)

Results
- 10x reduction in runtime vs. DGL-KE on Twitter
- 3.7x runtime reduction vs. PBG on Freebase86m
- 2x higher utilization than PBG, 6-8x higher utilization than DGL-KE
Method
- Use pipelining and async IO to hide data movement
- Utilize the full memory hierarchy with a partition buffer
- Minimize IO with the Buffer-aware Edge Traversal Algorithm (BETA)
Scaling to Large Graphs: 

Method
- Use pipelining and async IO hide data movement
- Utilize the full memory hierarchy with a partition buffer
- Minimize IO with Buffer-aware Edge Traversal Algorithm (BETA)

Maximize GPU utilization
Scaling to Large Graphs: Marius

Method
- Use pipelining and async IO to hide data movement
- Utilize the full memory hierarchy with a partition buffer
- Minimize IO with a Buffer-aware Edge Traversal Algorithm (BETA)

Minimize IO through partition caching
Scaling to Large Graphs: Marius

Method
- Use pipelining and async IO to hide data movement
- Utilize the full memory hierarchy with a partition buffer
- Minimize IO with Buffer-aware Edge Traversal Algorithm (BETA)

Minimize IO to lower bound

Partitioned Embeddings

Marius Architecture
Processing Partitions

Node Embedding Partitions

Node embeddings are partitioned uniformly into $p$ disjoint partitions.

Edge Buckets

Edge bucket $(i,j)$ contains all edges with a source in partition $i$ and a destination in partition $j$.

To iterate over all edges, we need to iterate over all edge buckets.
Edge bucket orderings and IO

The order in which edge buckets are processed has an impact on IO.

Size of partitions: \( \frac{138 \text{ GB}}{6} = 23 \text{ GB} \)

\( 23 \text{ GB} / 400 \text{ MBps} = \sim 57 \text{ seconds} \)

Costly swaps!

Example: After processing edge bucket (3, 2)

Processing (2, 3): Requires no extra swaps
Processing (2, 4): Requires one swap
Processing (4, 5): Requires two swaps
Edge bucket orderings and IO

A Lower Bound

Can never process more than $2c - 1$ edge buckets per swap

$$\left\lceil \frac{p^2 - c^2}{2c - 1} \right\rceil = \left\lceil \frac{6^2 - 3^2}{2 \times 3 - 1} \right\rceil = 6$$

6 swaps

Random Ordering  ~23 swaps

Hilbert Curve Ordering  12 swaps

BETA Ordering  7 swaps

Partitions in Buffer

$c = 3$

Partitions on disk  \[ p = 6 \]
Buffer-aware Edge Traversal Algorithm (BETA)

**BETA Ordering**

1. Randomly initialize buffer

2. Use the last spot in the buffer to cycle through the rest of the partitions, processing their corresponding edge buckets

3. Fix a new $c - 1$ partitions and repeat until all edge buckets have been processed

<table>
<thead>
<tr>
<th>Source Partition</th>
<th>Destination Partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 1 2 3 4 5</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

$\Theta_0 \Theta_1 \Theta_2 \Theta_3 \Theta_4 \Theta_5$

Partitions in Buffer

Partitions on disk

$c = 3$

$p = 6$
Buffer-aware Edge Traversal Algorithm (BETA)

**BETA Ordering**

1. Randomly initialize buffer

2. Use the last spot in the buffer to cycle through the rest of the partitions, processing their corresponding edge buckets

3. Fix a new $c - 1$ partitions and repeat until all edge buckets have been processed
Buffer-aware Edge Traversal Algorithm (BETA)

**BETA Ordering**

1. Randomly initialize buffer

2. Use the last spot in the buffer to cycle through the rest of the partitions, processing their corresponding edge buckets

3. Fix a new $c - 1$ partitions and repeat until all edge buckets have been processed

0 swaps*

* Not counting initialized buffer, as with the previous orderings
Buffer-aware Edge Traversal Algorithm (BETA)

**BETA Ordering**

1. Randomly initialize buffer

2. Use the last spot in the buffer to cycle through the rest of the partitions, processing their corresponding edge buckets

3. Fix a new $c - 1$ partitions and repeat until all edge buckets have been processed

0 swaps*

* Not counting initialized buffer, as with the previous orderings
Buffer-aware Edge Traversal Algorithm (BETA)

**BETA Ordering**

1. Randomly initialize buffer

2. Use the last spot in the buffer to cycle through the rest of the partitions, processing their corresponding edge buckets

3. Fix a new $c - 1$ partitions and repeat until all edge buckets have been processed

2 swaps
Buffer-aware Edge Traversal Algorithm (BETA)

**BETA Ordering**

1. Randomly initialize buffer

2. Use the last spot in the buffer to cycle through the rest of the partitions, processing their corresponding edge buckets

3. Fix a new $c - 1$ partitions and repeat until all edge buckets have been processed

<table>
<thead>
<tr>
<th>Partitions in Buffer</th>
<th>$c = 3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta_0$</td>
<td>$\Theta_1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Partitions on disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta_0$</td>
</tr>
</tbody>
</table>

$p = 6$
Buffer-aware Edge Traversal Algorithm (BETA)

**BETA Ordering**

1. Randomly initialize buffer
2. Use the last spot in the buffer to cycle through the rest of the partitions, processing their corresponding edge buckets
3. Fix a new c - 1 partitions and repeat until all edge buckets have been processed

5 swaps
Buffer-aware Edge Traversal Algorithm (BETA)

**BETA Ordering**

1. Randomly initialize buffer

2. Use the last spot in the buffer to cycle through the rest of the partitions, processing their corresponding edge buckets

3. Fix a new c - 1 partitions and repeat until all edge buckets have been processed

5 swaps
Buffer-aware Edge Traversal Algorithm (BETA)

**BETA Ordering**

1. Randomly initialize buffer

2. Use the last spot in the buffer to cycle through the rest of the partitions, processing their corresponding edge buckets

3. Fix a new $c - 1$ partitions and repeat until all edge buckets have been processed

5 swaps
Buffer-aware Edge Traversal Algorithm (BETA)

**BETA Ordering**

1. Randomly initialize buffer

2. Use the last spot in the buffer to cycle through the rest of the partitions, processing their corresponding edge buckets

3. Fix a new $c - 1$ partitions and repeat until all edge buckets have been processed

$\emptyset$ swaps

Close to the 6 swap lower bound!
Open sourced system: marius-project.org

Built on PyTorch

~15,000 lines of C++ and growing

Python API

Installation from source with Pip

1. Install latest version of PyTorch for your CUDA version:
   - Linux:
     - CUDA 10.1: python3 -m pip install torch==1.7.1+cu101 -f https://download.pytorch.org/whl/torch_stable.html
     - CUDA 10.2: python3 -m pip install torch==1.7.1
     - CPU Only: python3 -m pip install torch==1.7.1+cpu -f https://download.pytorch.org/whl/torch_stable.html
   - MacOS:
     - CPU Only: python3 -m pip install torch==1.7.1

2. Clone the repository git clone https://github.com/marius-team/marius.git

3. Build and Install Markus cd marius; python3 -m pip install ...

Marius in Docker

Marius can be deployed within a docker container. Here is a sample ubuntu dockerfile (local examples/docker/dockerfile) which contains the necessary dependencies preinstalled:

Building and running the container

Build an image with the name marius and the tag example:

```
docker build -t marius:example -f examples/docker/dockerfile examples/docker
```
Experimental evaluation

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Models</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Freebase86m knowledge graph</td>
<td>Dot</td>
<td>Amazon EC2 p3.2xlarge</td>
</tr>
<tr>
<td>Twitter social graph</td>
<td>ComplEx</td>
<td>V100 GPU, 61GB DRAM</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>DistMult</td>
<td></td>
</tr>
<tr>
<td>Freebase15k</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Presented here
- Large scale single-GPU comparison with PBG (Facebook) and DGL-KE (Amazon)
- BETA ordering runtime and IO reduction vs. existing orderings and lower bound

More in the paper
- System comparisons on two small/medium sized benchmark datasets
- Cost comparisons with multi-GPU and distributed configurations of DGL-KE and PBG
- The impact of asynchronous training and IO
- Scaling to configurations that are order(s) of magnitude larger than GPU and CPU capacity
### Accuracy and Runtime Comparisons

#### Twitter

<table>
<thead>
<tr>
<th>System</th>
<th>Model</th>
<th>MRR</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBG</td>
<td>Dot Product</td>
<td>0.313</td>
<td>5h15m</td>
</tr>
<tr>
<td>DGL-KE</td>
<td>Dot Product</td>
<td>0.220</td>
<td>35h3m</td>
</tr>
<tr>
<td>Marius</td>
<td>Dot Product</td>
<td>0.310</td>
<td><strong>3h28m</strong></td>
</tr>
</tbody>
</table>

Marius up to **10x** faster than DGL-KE on large social graphs

#### Freebase86m

<table>
<thead>
<tr>
<th>System</th>
<th>Model</th>
<th>MRR</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBG</td>
<td>ComplEx</td>
<td>0.725</td>
<td>7h27m</td>
</tr>
<tr>
<td>Marius</td>
<td>ComplEx</td>
<td>0.726</td>
<td><strong>2h1m</strong></td>
</tr>
</tbody>
</table>

Marius up to **3.7x** faster than PBG on large knowledge graphs

---

All systems are trained to 10 epochs, reaching convergence at near the same time.

**Twitter**
- 1.46 billion edges
- 41.6 million nodes
- 1 edge-type
- \(d = 50\)

**Freebase86m**
- 338 million edges
- 86 million nodes
- 15,000 edge-types
- \(d = 100\)
Compared Orderings

Lower bound
- Minimum number of swaps possible for a configuration

Hilbert
- Uses a Hilbert space filling curve to generate an ordering of the edge buckets

Hilbert Symmetric
- Modified Hilbert ordering which reduces swaps by 2x
- Processes edge buckets \((j,i)\) and \((i,j)\) together

Random
- Not evaluated, impractical to run as swaps scale quadratically with increasing partitions

BETA
- Our approach
Buffer-aware Edge Traversal Algorithm (BETA)

BETA ordering leads to 33% reduction in IO over locality based orderings

Near the lower bound

Reduction in IO corresponds directly with ~33% reduction in runtime

\[ c = \frac{p}{4} \quad d = 100 \]

\[ c: \text{buffer capacity}, \quad p: \text{num partitions}, \quad d: \text{embedding size} \]
Conclusion & Future Work

Existing systems bottlenecked by data movement

Marius alleviates data movement bottlenecks
- Pipelining/Async IO
- Partition Buffer
- BETA Ordering

Future work

High Energy Physics

Paleobiology (VLDB Demo 2021)
Learning Massive Graph Embeddings on a Single Machine

Jason Mohoney*, Roger Waleffe, Yiheng Xu, Theodoros Rekatsinas, Shivaram Venkataraman

* Email: mohoney2@wisc.edu

Open-source at marius-project.org

Thank you!