Marius

Machine Learning over Billion-Edge Graphs 10x Faster and 5x Cheaper

Theo Rekatsinas | thodrek@cs.wisc.edu
The team

**Student Leads**

Jason Mohoney  
Roger Waleffe

**Project Leads**

Prof. Theo Rekatsinas  
Prof. Shivaram Venkataraman
The universality of semantic structure

Graphs are universal representations of rich semantics about entities (nodes) and their relationships (edges)
Harnessing the power of structure

Reasoning requires operating over relational structured data

- Node classification
- Link prediction
- Related-entities prediction
Modern Machine Learning over graphs

Learned vector representations of nodes and edges are key to deep graph learning
Graph learning is memory- and IO-bound

Graphs introduce irregular access patterns
Graph learning is memory- and IO-bound

Example: 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)
Graph learning is memory- and IO-bound

Example: 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)
Graph learning is memory- and IO-bound

Example: 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)
Graph learning is memory- and IO-bound

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
- 64 GB CPU Memory

Embedding tables do not fit in GPU memory
Moving embeddings to compute

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

- DGL
- PyTorch Big-Graph (PBG)
- PBG & DGL
The key bottleneck when training graph learning models is data movement.
Marius: Scalable graph learning

Learning Massive Graph Embeddings on a Single Machine, OSDI’2021
Find more at: marius-project.org

Pipelining and a novel data replacement policy allow Marius to maximize resource utilization of the entire memory hierarchy (including disk, CPU, and GPU memory)

Achieves graph learning over billion edge graphs in a single machine
Marius: Scalable graph learning

<table>
<thead>
<tr>
<th>System</th>
<th>Model</th>
<th>MRR</th>
<th>Hits @1</th>
<th>Hits @10</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBG</td>
<td>Dot</td>
<td>.313</td>
<td>.239</td>
<td>.451</td>
<td>5h15m</td>
</tr>
<tr>
<td>DGL-KE</td>
<td>Dot</td>
<td>.220</td>
<td>.153</td>
<td>.385</td>
<td>35h3m</td>
</tr>
<tr>
<td>Marius</td>
<td>Dot</td>
<td>.310</td>
<td>.236</td>
<td>.445</td>
<td>3h28m</td>
</tr>
</tbody>
</table>

Measuring time-to-reconstruction-accuracy for Dot-Product graph embeddings over the Twitter graph (41.6M nodes and 1.5B edges)

Marius can be **10x faster** than competing methods in a single box

*MRR: mean reciprocal rank (higher is better)*
Marius: Scalable graph learning

<table>
<thead>
<tr>
<th>System</th>
<th>Deployment</th>
<th>Epoch Time (s)</th>
<th>Per Epoch Cost ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Marius</td>
<td>1-GPU</td>
<td>288</td>
<td>.248</td>
</tr>
<tr>
<td>DGL-KE</td>
<td>2-GPUs</td>
<td>761</td>
<td>1.29</td>
</tr>
<tr>
<td>DGL-KE</td>
<td>4-GPUs</td>
<td>426</td>
<td>1.45</td>
</tr>
<tr>
<td>DGL-KE</td>
<td>8-GPUs</td>
<td>220</td>
<td>1.50</td>
</tr>
<tr>
<td>DGL-KE</td>
<td>Distributed</td>
<td>1237</td>
<td>1.69</td>
</tr>
<tr>
<td>PBG</td>
<td>1-GPU</td>
<td>1005</td>
<td>.85</td>
</tr>
<tr>
<td>PBG</td>
<td>2-GPUs</td>
<td>430</td>
<td>.73</td>
</tr>
<tr>
<td>PBG</td>
<td>4-GPUs</td>
<td>330</td>
<td>1.12</td>
</tr>
<tr>
<td>PBG</td>
<td>8-GPUs</td>
<td>273</td>
<td>1.86</td>
</tr>
<tr>
<td>PBG</td>
<td>Distributed</td>
<td>1199</td>
<td>1.64</td>
</tr>
</tbody>
</table>

Per-epoch runtime and monetary cost ($) for embedding the Freebase Knowledge Graph (86M nodes and 338M edges)

Marius can be 5x cheaper than competing methods; single-box (1GPU) Marius has comparable runtime with multi-GPU solutions
Open-source Marius

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==1.7.1cpu -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 Marius 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: example: docker build -t marius:example -f examples/docker/dockerfile examples/docker

Batch

class Batch
Contains metadata, edges and embeddings for a single batch.
Subclassed by PartitionBatch

Public Functions

constructor

-Batch()

void localSample()

destructor Construct additional negative samples and neighborhood information from the batch

void accumulateUniqueIndices()

Populates the unique_**_indices tensors

void _embeddingToDevice() in device_id

Transfers embeddings, optimizer state, and indices to specified device

void prepareBatch()

Populates the src_pos_embeddings, dst_pos_embeddings, relation_embeddings, src_reg_embeddings, and dst_reg_embeddings tensors for model computation

void accumulateGradients()

Accumulates gradients into the unique_node_gradients and unique_relaion_gradients tensors, and applies optimizer update rule to create the unique_node_gradients2 and unique_relaion_gradients2 tensors

void _embeddingToHost()

Transfers gradients and embedding updates to host

Released at: marius-project.org

Apache-2.0 License
Using Marius

Config-based development

• No-code paradigm: running Marius only requires a simple configuration file

• Customize parameters, defaults provided if not specified

• Easy to run from command line
Using Marius

- Features a Python API
- Write custom models
- High-degree of control and customization
Using Marius

Interoperability

• Multiple data converters to transform raw data into the Marius input format

• Support for conversion of TSV, CSV, Parquet file formats

• Output embeddings can be converted to commonly used types such as PyTorch tensors
Key innovation in Marius

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)**

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
Partition-based processing

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

**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 ordering and IO

Random Ordering  ~23 swaps

Locality-aware Ordering  12 swaps

We show a Lower Bound

Can never process more than \(2c - 1\) edge buckets per swap

\[
\left\lceil \frac{p^2 - c^2}{2c - 1} \right\rceil = \left\lceil \frac{6^2 - 3^2}{2 \cdot 3 - 1} \right\rceil = 6
\]

We propose an ordering which is close to this bound
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
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
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
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

BETA ordering gives 7 swaps (6 is the lower bound)
BETA ordering enables high GPU utilization

**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)**

**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
Use-case: Construction of Scientific Knowledge Graphs

Simplified Knowledge Base

Textual Mentions

Fossils from an extinct toothed (Archaeocete) whale, *Basilosauras cetoides*, were found in a surface exposure of the Pachuta Marl Member of the late Eocene Yazoo Clay near the Matherville community in Wayne County, Mississippi.

The Yazoo Clay Formation makes up the upper half of the Jackson Group in the central Gulf Coastal Plain, representing deposition during the TACG43 marine transgression.

Analagical Reasoning Example

Joint embeddings of text and existing knowledge graphs to enable analogical reasoning and knowledge completion in any domain
Marius: Scalable graph learning

Learning Massive Graph Embeddings on a Single Machine, OSDI’2021

Marius achieves graph learning over billion-edge graphs 10x faster and 5x cheaper than competing solutions

Find more at: marius-project.org
Marius: Scalable graph learning

Learning Massive Graph Embeddings on a Single Machine, OSDI’2021

Marius achieves graph learning over billion-edge graphs
10x faster and 5x cheaper than competing solutions

Find more at: marius-project.org

Thank you!
@thodrek
The case of exploiting the full memory stack

<table>
<thead>
<tr>
<th>d</th>
<th>Size</th>
<th>Partitions</th>
<th>MRR</th>
<th>Runtime (Epoch)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>13.6 GB</td>
<td>-</td>
<td>.698</td>
<td>4m</td>
</tr>
<tr>
<td>50</td>
<td>34.4 GB</td>
<td>-</td>
<td>.722</td>
<td>4.8m</td>
</tr>
<tr>
<td>100</td>
<td>68.8 GB</td>
<td>32</td>
<td>.726</td>
<td>12.1m</td>
</tr>
<tr>
<td>400</td>
<td>275.2 GB</td>
<td>32</td>
<td>.731</td>
<td>92.4m</td>
</tr>
<tr>
<td>800</td>
<td>550.4 GB</td>
<td>64</td>
<td>.731</td>
<td>396m</td>
</tr>
</tbody>
</table>

*MRR: mean reciprocal rank (higher is better)

Per-epoch runtime and reconstruction-accuracy as we increase the embedding size for Freebase (86M nodes and 338M edges)

Higher-dimensional embeddings can lead to higher accuracy