GReTA: Hardware Optimized Graph Processing for GNNs

Kevin Kinningham, Phil Levis, Chris Ré
Stanford University
March 4th, 2020
Deep Neural Networks

Speech Recognition
Translation
Object Detection
Handwriting Recognition

Traditional DNN

Conv Layer → ... → Linear Layer → “Dog”
Deep Neural Networks + Graphs = ?

Speech Recognition
Object Detection
Handwriting Recognition
Translation

Traditional DNN

Conv Layer → … → Linear Layer → “Dog”

Social Networks
Citations
Protein Interactions
Road Networks
Deep Neural Networks + Graphs = GNNs

Traditional DNN

Graph Neural Network (GNN)
GNN Computation Is Irregular

- Computation pattern *changes* depending on input graph structure
- GNN layers follow message passing architecture
Existing DNN Representations Bad for GNNs

- Irregular computation is difficult to represent with static tensor network
  - E.g. Tensorflow

- Hard to handle large graphs
  - Must manually deal with partitioning variables
  - Hard to make efficient when graph shape can change
GReTA: Graph Framework for GNNs

- **Simple** to represent GNN layers
  - Computation defined on edges and vertices of input graph
  - Maps directly to message passing

- **Flexible** enough to allow a wide range of GNN models
  - Allows each execution phase to be customized

- **Efficient** execution on an accelerator
  - Partitioning: Limit accelerator memory usage without modifying user code
  - Tiling: Increase the reuse of layer weights
Talk Agenda

- Introduction
- GReTA Overview
- Execution Model
- Partitioning
- Experimental Results
- Conclusion
GReTA Overview

- GReTA represents computation using **graph framework**
  - Functions defined on edges and vertices
  - Can directly map message passing layer
- GNN layers implemented using four user-defined functions (UDFs)
  1. **G**ather: compute message for each edges
  2. **R**educe: reduce incoming messages per-vertex
  3. **T**ransform: combine reduced value with per-vertex accumulator
  4. **A**ctivate: perform non-linear function
Example: Graph Convolutional Network (GCN)

\[
h_v^{(\ell+1)} \leftarrow \text{ReLU} \left( W^{(\ell)} \cdot \left( \sum_{u \rightarrow v} h_u \right) + b^{(\ell)} \right)
\]

GCN layer update function
Example: Graph Convolutional Network (GCN)

\[ h_v^{(\ell+1)} \leftarrow \text{ReLU} \left( W^{(\ell)} \cdot \left( \sum_{u \rightarrow v} h_u \right) + b^{(\ell)} \right) \]

GCN layer update function

1. **Gather** messages using connected edges
Example: Graph Convolutional Network (GCN)

$$h_v^{(\ell+1)} \leftarrow \text{ReLU} \left( W^{(\ell)} \cdot \left( \sum_{u \rightarrow v} h_u^{(\ell)} \right) + b^{(\ell)} \right)$$

GCN layer update function

1. **Gather** messages using connected edges
2. **Reduce** to single vector by summation
Example: Graph Convolutional Network (GCN)

$$h_v^{(\ell+1)} \leftarrow \text{ReLU} \left( W^{(\ell)} \cdot \left( \sum_{u \rightarrow v} h_u^{(\ell)} \right) + b^{(\ell)} \right)$$

GCN layer update function

1. **Gather** messages using connected edges
2. **Reduce** to single vector by summation
3. **Transform** result using linear transformation
Example: Graph Convolutional Network (GCN)

$$h_v^{(\ell+1)} \leftarrow \text{ReLU} \left( W^{(\ell)} \cdot \left( \sum_{u \rightarrow v} h_u^{(\ell)} \right) + b^{(\ell)} \right)$$

GCN layer update function

1. **Gather** messages using connected edges
2. **Reduce** to single vector by summation
3. **Transform** result using linear transformation
4. **Activate** output using element-wise ReLU
class GCNLayer(GretaInterface):
    def gather(h_u, h_v, h_uv):
        return h_u
    def reduce(a_v, m_v):
        return a_v + m_v
    def transform(z_v, a_v, W, b):
        return z_v + W * a_v + b
    def activate(z_v):
        return relu(z_v)
Talk Agenda

- Introduction
- GReTA Overview
- Execution Model
- Partitioning
- Experimental Results
- Conclusion
GReTA Execution Model

Execution conceptually split into three phases
GReTA Execution Model

Execution conceptually split into three phases

1. **Accumulate Edges**
   - Gather/compute message for each edge
   - Reduce to single value per vertex
GReTA Execution Model

Execution conceptually split into three phases

1. **Accumulate Edges**
   - Gather/compute message for each edge
   - Reduce to single value per vertex

2. **Accumulate Vertices**
   - Combine reduced value with prior vertex accumulator state
GReTA Execution Model

Execution conceptually split into three phases

1. **Accumulate Edges**
   - Gather/compute message for each edge
   - Reduce to single value per vertex

2. **Accumulate Vertices**
   - Combine reduced value with prior vertex accumulator state

3. **Update Vertices**
   - Apply activate to accumulator
Talk Agenda

- Introduction
- GReTA Overview
- Execution Model
- Partitioning
- Experimental Results
- Conclusion
Optimizations for Hardware Implementation

Execution Partitioning

- Problem: Large graphs do not fit into limited accelerator memory
  - E.g. social media graphs with millions of users
- Solution: Partition graph and execute GReTA on each partition separately
- Results combined via vertex accumulators

Weight Tiling

- Problem: Bandwidth bottlenecks when layer weights are large
- Solution: Improve reuse by splitting weights into tiles
- Tiles can be reused across multiple vertices
Graph Partitioning Example

![Graph Diagram]

Nodes A, B, C, D, E, F are connected in a network, illustrating the concept of graph partitioning.
Graph Partitioning Example

Vertex Partition

V₁
A  B
F  C  D  E
V₂
V₃

Vertex Chunks
Graph Partitioning Example

Vertex Partition

Edge Partition

Source

Destination

A B C D E F
Graph Partitioning Example

**Vertex Partition**

**Edge Partition**

**Destination**

**Source**

E_{2,1}
Execution Partitioning Example
Execution Partitioning Example

Execution follows columns

Source

Destination

Edge Partition

Accelerator

Off-chip DRAM
Execution Partitioning Example

Execution follows columns

Source

Destination

Edge Partition

Accelerator

Off-chip DRAM

Accumulate Edges
Execution Partitioning Example

Source A B C D E F

Destination

Edge Partition

Execution follows columns

Accelerator

Off-chip DRAM

Accumulate Edges

Accumulate Edges
Execution Partitioning Example

Execution follows columns

Source

Destination

Edge Partition

Accelerator

Off-chip DRAM

Accumulate Edges
Execution Partitioning Example

Execution follows columns

Source
A
B
C
D
E
F

Destination
A
B
C
D
E
F

Edge Partition

Accelerator
Src Dst Edge
A B

Off-chip DRAM

Accumulate Edges

Accumulate Edges

Accumulate Edges

Accumulate Vertices

Update
Talk Agenda

● Introduction
● GReTA Overview
● Execution Model
● Partitioning
● Experimental Results
● Conclusion
Experimental Setup

- Implemented range of GNN models
  - GCN (simple, classic GNN model)
  - GraphSage (max-reduce instead of sum)
  - GIN (MLP in transform layer)
  - G-GCN (per-edge computation)

- Baseline
  - CPU: 2.6 GHz Intel Xeon E5-2690v4
  - GPU: Nvidia Tesla P100
  - Models implemented using Tensorflow

- Compared to custom 32nm GReTA accelerator

- Key performance metric: Total inference latency for batch size of 1

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Nodes</th>
<th>Edges</th>
<th>2-Hop</th>
</tr>
</thead>
<tbody>
<tr>
<td>YT</td>
<td>1.13M</td>
<td>2.98M</td>
<td>25</td>
</tr>
<tr>
<td>LJ</td>
<td>3.99M</td>
<td>34.6M</td>
<td>65</td>
</tr>
<tr>
<td>PO</td>
<td>1.63M</td>
<td>30.6M</td>
<td>167</td>
</tr>
<tr>
<td>RD</td>
<td>232K</td>
<td>47.4M</td>
<td>239</td>
</tr>
</tbody>
</table>
9-23x Latency Reduction vs CPU

- **15x** g.mean across all datasets/models
- Best results on models where message passing dominates (GCN, G-GCN)

![GReTA Latency Reduction vs CPU](image)
6-67x Latency Reduction vs GPU

- 21x g.mean across all datasets/models
- Best speedup on models with low overall latency (GCN, GIN)
- Small batch size means data transfer latency often dominates
Conclusion

Key features of GReTA:

1. **Simple** representation using a graph framework
2. **Expressive** enough to allow for a wide range of GNNs
3. **Efficient** execution on an accelerator

**Future work:** Apply GReTA beyond GNNs? Integration with existing frameworks?
Conclusion

Key features of GReTA:

1. **Simple** representation using a graph framework
2. **Expressive** enough to allow for a wide range of GNNs
3. **Efficient** execution on an accelerator

**Future work:** Apply GReTA beyond GNNs? Integration with existing frameworks?

Q&A? 😐❓
GReTA Accelerator

- Replace setup with unit for **Gather-ing** edge/vertex values
  - Uses graph adjacency info stored in Unified Buffer
- New accumulator unit for **Reduce**
- Note: Existing NN ops can still run on new architecture!
  - Gather unit just performs single load
  - Reduce unit performs no-op
Compiling GReTA to a TPU-like Architecture

Execution in four stages

1. **Load**: Move data from unified buffer into setup unit
Traditional DNN Accelerator Model

Execution in four stages

1. **Load:** Move data from unified buffer into setup unit
2. **Compute:** Multiply setup data by pre-loaded weight values
Traditional DNN Accelerator Model

Execution in four stages

1. **Load**: Move data from unified buffer into setup unit
2. **Compute**: Multiply setup data by pre-loaded weight values
3. **Accumulate**: Collect output from compute over $N$ cycles
Traditional DNN Accelerator Model

Execution in four stages

1. **Load**: Move data from unified buffer into setup unit
2. **Compute**: Multiply setup data by pre-loaded weight values
3. **Accumulate**: Collect output from compute over $N$ cycles
4. **Activate**: Execute required activation/normalization and store result
Traditional DNN Accelerator Model

Execution in four stages

1. **Load**: Move data from unified buffer into setup unit
2. **Compute**: Multiply setup data by pre-loaded weight values
3. **Accumulate**: Collect output from compute over $N$ cycles
4. **Activate**: Execute required activation/normalization and store result

- Key insight: Stages 2-4 can already execute GReTA's **Transform** and **Activate** UDFs
- Only need to add hardware for **Gather** and **Reduce**
Graph Partitioning

- Problem: Data for full graph may be too large to fit entirely on accelerator
- Solution: Partition graph and execute phases for each partition separately

![Graph Partitioning Diagram]

|------|--------|--------|------|

Vertex Partition

Edge Partition
Interleaving Execution

- Multiple GReTA programs in a layer may reuse data
  - Read identical edge/vertex data
  - Reuse accumulator values

- Interleaving execution improves data locality

\[
\begin{align*}
    h_{v,1} & \leftarrow W_1 \sum_{u \rightarrow v} h_u \\
    h'_v & \leftarrow h_{v,1} + W_2 \sum_{u \rightarrow v} h_u
\end{align*}
\]

Identical vertex data read twice

Vertex accumulator must be unloaded and reloaded

Execution
Interleaving Execution

- Multiple GReTA programs in a layer may reuse data
  - Read identical edge/vertex data
  - Reuse accumulator values

- Interleaving execution improves data locality

\[
\begin{align*}
    h_{v,1} & \leftarrow W_1 \sum_{u \rightarrow v} h_u \\
    h'_v & \leftarrow h_{v,1} + W_2 \sum_{u \rightarrow v} h_u
\end{align*}
\]

Vertex data read once, reused

Vertex accumulator stays loaded

Execution

Optimizations
Weight Tiling

- Problem: Layer weights can be too large to fully load into GEMM unit
- Existing solution: Slice weights into tiles and reloading for each new vertex
  - Unfortunately, gives worst case reuse of each tile
  - Accelerator often bottlenecked on loading/reload weight tiles