GReTA: Hardware Optimized Graph Processing for GNNs

Kevin Kiningham [1] **Philip Levis** [2] **Christopher Ré** [2]

Abstract

Graph Neural Networks (GNNs) are a type of deep neural network that can learn directly from irregular graph-structured data. However, GNN inference relies on sparse operations that are difficult to implement on accelerator architectures optimized for dense, fixed-sized computation. This paper proposes GReTA (Gather, Reduce, Transform, Activate), a graph processing abstraction designed to be efficient for an accelerator to execute and flexible enough to implement GNN inference. We demonstrate GReTA's advantages by designing and synthesizing a custom accelerator ASIC for GReTA and implementing several GNN models (GCN, GraphSage, G-GCN, and GIN.) Across several benchmark graphs, our implementation reduces 97th percentile latency by a geometric mean of 15× and 21× compared to a CPU and GPU baseline respectively.

1 Introduction

Traditional deep neural networks (DNNs) rely on regularly structured input (e.g. vectors, images, or sequences) making it difficult for them to learn from data with irregular structure (e.g. connections between users on social media.) Graph neural networks (GNNs) address this limitation by extending DNNs to operate on graph-valued data. During inference, vertices exchange messages along the edges of the input graph, combining per-vertex feature values with information from other vertices in their local neighborhood. GNNs have found success in a variety of domains, from recommending content on social media (Ying et al., 2018) to categorizing paper topics (Kipf & Welling, 2017).

A key challenge of GNNs is their reliance on sparse operations, such as aggregating data from an arbitrary set of neighbors. This makes them difficult to implement efficiently on widely deployed DNN accelerators, that are optimized for the dense matrix multiplications found in traditional DNNs (Balog et al., 2019). As a result, GNNs have seen limited use in real applications, even in domains where they are considered state of the art.

In this paper, we introduce GReTA, a graph processing abstraction designed to be straightforward to execute by a hardware accelerator while being flexible enough to implement the sparse operations used in GNNs. In GReTA, each layer of a GNN is decomposed into one or more GReTA programs. We show that by extending a traditional DNN accelerator with two new stages (gather and reduce), these programs can be executed efficiently. Finally, we give a partitioning technique to limit the amount of memory required to perform inference, a significant concern in practice for large GNN models.

We demonstrate the advantages of GReTA by designing and synthesizing a $32/28\,\mathrm{nm}$ ASIC capable of executing GReTA programs. Evaluated across four different GNN models, our implementation reduces 97th percentile latency by a geometric mean of 15× and 21× compared to a CPU and GPU baseline respectively.

2 Background

2.1 Graph Neural Networks

GNNs are a type of neural network that operate on graph-valued data. Unlike traditional DNNs, GNNs can directly use graph structure during learning. For example, consider the problem of classifying web pages by subject. A pure content-based approach (e.g. a recurrent neural network) considers only features derived directly from the content of the page. In contrast, a GNN can natively leverage both page content and graph structure (e.g. links between pages.) GNN models have achieved state of the art performance on a diverse set of tasks involving graphs (Ying et al., 2018).

Message Passing Architecture. While many GNN models have been proposed (Battaglia et al., 2018; Wu et al., 2019), modern variants typically follow an iterative message passing architecture (Gilmer et al., 2017). Algorithm 1 shows the forward pass of a single layer in this architecture. The layer takes as input a graph G, with each vertex assigned a feature vector $\boldsymbol{h}_v$. Computation is split into three operations:

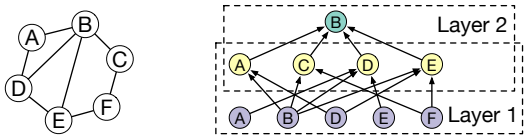
[1]Department of Electrical Engineering, Stanford University, Stanford, CA, USA [2]Department of Computer Science, Stanford University, Stanford, CA, USA. Correspondence to: Kevin Kiningham <kkiningh@stanford.edu>.

Proceedings of the 2^{nd} SysML Conference, Palo Alto, CA, USA, 2019.

Algorithm 1 Message Passing Layer Forward Pass

Input: Graph $G = (V, E)$, Vertex and edge features $\boldsymbol{h}_v$ and $\boldsymbol{h}_{(u,v)}$, Neighborhood function $N(v)$
Output: Updated vertex features $\boldsymbol{z}_v$

for (u, v) **in** E **do**
  $\boldsymbol{m}_{u,v} \leftarrow \text{Send}(\boldsymbol{h}_v, \boldsymbol{h}_u, \boldsymbol{h}_{(u,v)})$
end for
for v **in** V **do**
  $\boldsymbol{a}_v \leftarrow \text{Aggregate}(\{\boldsymbol{m}_{u,v} \mid u \in N(v)\})$
  $\boldsymbol{z}_v \leftarrow \text{Update}(\boldsymbol{h}_v, \boldsymbol{a}_v)$
end for



(a) Input graph (b) Nodeflow

Figure 1. An example of the nodeflow when performing inference for vertex B using a two layer GNN.

- *Send* computes a message vector $\boldsymbol{m}_{u,v}$ for each edge.
- *Aggregate* reduces the set of incoming messages for each vertex to a single vector $\boldsymbol{a}_v$.
- *Update* combines each vertex's current value with the output of aggregation to produce an updated vector $\boldsymbol{z}_v$.

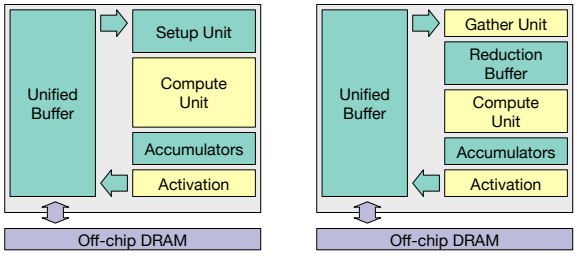
By iteratively applying K of these layers, the final state for each vertex captures information about the structure of its K-hop neighborhood.

Nodeflow. When performing inference on a subset of vertices, a layered graph structure called the *nodeflow* (Huang et al., 2019) is commonly used to describe how feature vectors are propagated during a model's forward pass. In this paper, we denote the nodeflow for a particular layer as the three-tuple (U, V, E), where U is the set of vertices read, V is the set of vertices updated, and E is the subset of edges connecting vertices in U and V. Figure 1 shows an example of the nodeflow for a two layer GNN when computing inference for a single vertex.

2.2 DNN Accelerator Model

Figure 2a shows an abstract, TPU-like (Jouppi et al., 2017), DNN accelerator model we use as a architectural baseline. The model executes operations in four stages:

1. *Load*: Data is loaded from the unified buffer into the setup unit, which applies any preprocessing required for the compute unit.
2. *Compute*: The core computation of the accelerator is performed, typically a dense, fixed-sized matrix multiplication. Note that weight values must be loaded into the compute unit separately, before computation begins.
3. *Accumulate*: The output of the compute unit is collected in the accumulator buffer.
4. *Activate*: Activation and normalization operations are applied to the values in the accumulation buffer and the result is stored back into the unified buffer.



(a) Baseline (b) GReTA

Figure 2. Comparison between a traditional DNN accelerator (2a) and an extended architecture designed to execute GReTA programs (2b). The main difference is the replacement of the setup unit with the gather unit and reduction buffers.

3 GRETA

GReTA is a graph processing abstraction designed for efficient execution on an accelerator. Similar to other graph processing frameworks, such as PowerGraph (Gonzalez et al., 2012), computation in GReTA is expressed by implementing stateless user-defined functions (UDFs).

GReTA's interface consists of four such UDFs (*Gather*, *Reduce*, *Transform*, and *Activate*) which are executed in a series of phases. First, in the aggregation phases, *Gather* loads data associated with each edge and vertex in the nodeflow. The result is accumulated into a single value per vertex using *Reduce*. Then, *Transform* is applied, combining the reduced value with the previous vertex state. It also typically performs the most significant computation in the GReTA program (e.g. a matrix multiplication.) Finally, during the update phase, *Activate* performs any final computation and stores the result. Algorithm 2 shows GReTA's full execution semantics.

3.1 Expressing Inference

GReTA is expressive enough to allow implementing inference for a wide variety of GNNs. To implement a particular model, each layer of the GNN is mapped to one or more GReTA programs. Mapping is typically straightforward since the overall semantics of GReTA are similar to the

Algorithm 2 GReTA Execution Semantics

Input: Layer nodeflow (U, V, E); Vertex and edge data h_v^t and $h_{(u,v)}^t$; Weights W^t; Number of tiles T
Output: Updated vertex data h'_v

for $t = 1$ **to** T **do**
  /* Accumulate Edges Phase */
  for (u, v) **in** E **do**
    a_v^t = Reduce(a_v^t, Gather(h_u^t, h_v^t, $h_{(u,v)}^t$))
  end for
  /* Accumulate Vertices Phase */
  for v **in** V **do**
    z_v = Transform(z_v, a_v^t, W^t)
  end for
end for
/* Update Phase */
for v **in** V **do**
  h'_v = Activate(z_v)
end for

```
def Gather(h_u^t, h_v^t, h_(u,v)^t): return h_u^t
def Reduce(a_v^t, m_v^t):
    a_v^t.value += m_v^t ; a_v^t.count += 1
    return a_v^t
def Transform(z_v, a_v^t, W^t):
    return z_v + W^t a_v^t.value / a_v^t.count
def Activate(z_v): return ReLU(z_v)
```

Figure 3. GReTA pesudocode implementation for GCN inference.

message passing architecture introduced in Section 2.1

Here, we demonstrate mapping inference for a common model, the Graph Convolutional Network (GCN) (Kipf & Welling, 2017). GCN follows the message passing architecture with the following send, aggregate, and update operations:

$$\boldsymbol{m}_{u,v} \leftarrow \boldsymbol{h}_u \tag{1}$$

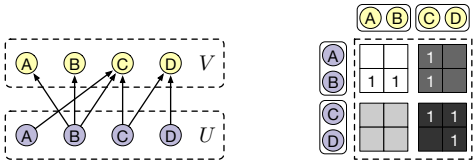
$$\boldsymbol{a}_v \leftarrow \text{mean}(\{\boldsymbol{m}_{u,v} \mid u \in N(v)\}) \tag{2}$$

$$\boldsymbol{z}_v \leftarrow \text{ReLU}(W\boldsymbol{a}_v) \tag{3}$$

The corresponding GReTA implementation is given in Figure 3. The only subtlety is mapping the mean function, which we implement by tracking separate value and count fields in the accumulator and performing an element-wise division operation in *Transform*.

3.2 Hardware Acceleration

GReTA has three major features that make execution on an accelerator easier than existing graph processing frameworks. First, data values are split into tiles (e.g. $h_v = [h_v^1, \ldots, h_v^T]$), the size of which can be tailored to match the native width of the accelerator datapath. Second, the



(a) Original (b) Partitioned

Figure 4. An example of the nodeflow for a layer (left) and its possible partitioning (right). Rows and columns correspond to chunks of input and output vertices respectively.

outer loop of the accumulate phase iterates over tiles rather than edges or vertices. This allows each weight tile to be reused across all vertices, improving performance. Finally, the *Transform* UDF can accumulate values using z_v, rather than simply performing an update operation. This makes it much easier to implement layers that sum together multiple values (e.g. $\sigma(W_1 h + W_2 h)$) without having to store and reload intermediate results.

Figure 2b shows a modified version of the DNN accelerator architecture designed to execute GReTA programs. The most significant deviation from the original architecture is the replacement of the setup unit with two new components: the gather unit and the reduction buffer. The gather unit iterates over the edges defined by the nodeflow, loads the associated vertex and edge data from the unified buffer, and then executes the operation specified by the *Gather* UDF. The result is passed to the reduction buffer, which executes *Reduce* as well as the preprocessing originally performed by the setup buffer. The final two UDFs in GReTA (*Transform* and *Activate*) are mapped onto the compute and activation units respectively, which operate essentially unmodified.

Note that although GReTA allows any operation to be defined for each UDF, only a small number of operations are needed for most GNN models in practice. Specifically, we allow *Gather* to be identity (no-op), element-wise sum, product, and a lookup-table operation. For *Reduce*, we allow element-wise max, sum, and sum with count (the operation used by GCN in Section 3.1.) While these covered all GNN models we investigated, expanding the set of supported operations may be required for other GNNs. We leave exploring other possible implementations for future work.

3.3 Nodeflow Partitioning

The total memory required for GReTA is proportional to the number of vertices in the nodeflow. Here, we introduce a technique to partition the nodeflow into blocks that can be executed separately. This allows reducing the amount of memory required to execute a GReTA program by loading only a subset of vertices at a time, making it feasible to execute GReTA on an accelerator with limited memory.

First, the input and output vertices are partitioned offline

into chunks of size n and m respectively. Likewise, the edges of the nodeflow are partitioned into blocks of size $n \times m$, where block $NF_{i,j}$ stores the edges connecting input vertices in chunk U_i to output vertices in chunk V_j. During execution, the accumulate phases for each block in a column is executed sequentially, skipping blocks that are empty. Then, the update phase executes once, writing the update values for the vertices in the corresponding output chunk back to unified memory. Note that by processing an entire column we ensure every edge associated with a given chunk of output vertices will be processed, regardless of the input chunk the edge originates from. Figure 4 shows an example of partitioning the nodeflow using this method.

4 Evaluation

To evaluate the performance of GReTA on an accelerator, we implemented a custom accelerator in SystemVerilog and performed synthesis and place and route, targeting a generic $32/28\,\mathrm{nm}$ CMOS process and a $1\,\mathrm{GHz}$ operating frequency. Our final design achieved an area of $11.27\,\mathrm{mm}^2$, a peak operating power of $6.73\,\mathrm{W}$, and a peek performance of $2\,\mathrm{TOPS/s}$. To measure the performance, we also developed a cycle accurate C++ model, using Ramulator (Kim et al., 2016) to estimate DRAM timings.

To demonstrate flexibility, we mapped inference for four different GNN models to our accelerator as GReTA programs: GCN (Kipf & Welling, 2017), GraphSage (Hamilton et al., 2017), G-GCN (Bresson & Laurent, 2017; Marcheggiani & Titov, 2017), and GIN (Xu et al., 2019). Each model was evaluated on datasets chosen from previous evaluations of GNNs, the SNAP project (Leskovec & Krevl, 2014), and the UF sparse matrix collection (Davis & Hu, 2011). In all experiments, we use a feature size of 602 (the feature size of the Reddit dataset), a hidden dimension of 512, a pooling dimension of 256, and a batch size of 1.

Finally, we implemented each model in Tensorflow to use as a baseline. To discount Tensorflow's overhead, we measured the time to execute an equivalent model with all tensor dimensions set to zero, and subtract this from the latency measurement. We used an Intel Xeon E5-2690v4 for the CPU baseline, restricted to a single socket to avoid latency variation from NUMA. For the GPU baseline we used a Nvidia Tesla P100.

Table 1 shows the GReTA ASIC's total execution time (latency) to compute inference for each model and the speedup versus our CPU and GPU implementation at the 97th percentile. Compared to the CPU implementation, GReTA achieved a latency improvement of between 23× (GCN, LiveJournal) and 9× (GIN, Youtube) with a geometric mean of 15.1× across all datasets and models. GReTA tends to give a smaller speedup on models that perform a larger portion of their computation during the update operation. For example, GIN uses a two-layer MLP in its update phase and performs significantly more computation than GCN which uses a single matrix multiply. This is because the CPU implementation is mostly bottlenecked by cache bandwidth during the aggregation operation, and models with more computationally intensive update operations spend a smaller portion of their time performing aggregation.

Table 1. 97th percentile inference latency for GReTA ASIC versus CPU and GPU baselines.

			CPU		GPU	
Model	Dataset	GReTA	µs	×	µs	×
gcn	youtube	14.1	253.2	(17.9)	791.3	(56.0)
gcn	livejournal	15.4	353.1	(23.0)	1040.0	(67.7)
gcn	pokec	15.9	327.6	(20.7)	790.3	(49.8)
gcn	reddit	16.2	310.8	(19.2)	698.4	(43.1)
g-gcn	youtube	120.6	1975.9	(16.4)	1191.0	(9.9)
g-gcn	livejournal	134.1	2303.9	(17.2)	1685.7	(12.6)
g-gcn	pokec	146.4	2467.3	(16.9)	1168.8	(8.0)
g-gcn	reddit	146.9	2766.0	(18.8)	872.6	(5.9)
gs-max	youtube	101.1	1244.1	(12.3)	1150.1	(11.4)
gs-max	livejournal	113.7	1650.5	(14.5)	2006.6	(17.7)
gs-max	pokec	124.5	1931.2	(15.5)	1494.2	(12.0)
gs-max	reddit	125.2	2021.1	(16.1)	1064.6	(8.5)
gin	youtube	29.2	285.3	(9.8)	1180.6	(40.4)
gin	livejournal	30.5	306.2	(10.0)	960.5	(31.5)
gin	pokec	31.0	300.5	(9.7)	676.6	(21.8)
gin	reddit	31.3	312.1	(10.0)	818.8	(26.1)

Compared to the GPU implementation, GReTA achieved speedups ranging from 68× (Livejournal, GCN) to 6× (Reddit, G-GCN), with a geometric mean of 21×. Models with relatively low overall latency (GCN, GIN) have a significantly higher speedup than with our CPU implementation. This is largely due to the overhead of transferring data from host to GPU memory (roughly 200-500 µs, depending on the nodeflow size), which comprises a large portion of the overall latency for models like GCN (25-50% of total latency.) On models with a higher latency (e.g. G-GCN), we still achieve significant speedup due to low GPU utilization. With a batch size of 1, there is insufficient computation during each layer to fully utilize the resources of the GPU and the kernel launch overhead tends to dominate.

5 Conclusion

GNNs represent a promising new method in machine learning to learn directly from graph-structured data. However, the mismatch between the sparse operations required for GNNs and accelerators optimized for dense matrix multiplication represents a significant barrier for deploying GNNs more widely. This paper presents GReTA, a graph processing abstraction designed to both be an easy target for mapping GNN operations and be efficiently executed by an accelerator. Our evaluation of an ASIC implementation of GReTA on a range of real graphs shows a 97th percentile latency improvement of between 9× to 23× and 6× to 21× compared to a CPU and GPU baseline respectively.

REFERENCES

Balog, M., van Merriënboer, B., Moitra, S., Li, Y., and Tarlow, D. Fast training of sparse graph neural networks on dense hardware. *arXiv preprint arXiv:1906.11786*, 2019.

Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

Bresson, X. and Laurent, T. Residual gated graph convnets. *arXiv preprint arXiv:1711.07553*, 2017.

Davis, T. A. and Hu, Y. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 1263–1272. JMLR.org, 2017.

Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pp. 17–30, 2012.

Hamilton, W., Ying, Z., and Leskovec, J. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pp. 1024–1034, 2017.

Huang, Z., Zheng, D., Gan, Q., Zhou, J., and Zhang, Z. Nodeflow and sampling, 2019. https://doc.dgl.ai/tutorials/models/5_giant_graph/1_sampling_mx.html#nodeflow, Accessed 2020-01-01.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pp. 1–12, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4892-8. doi: 10.1145/3079856.3080246. URL http://doi.acm.org/10.1145/3079856.3080246.

Kim, Y., Yang, W., and Mutlu, O. Ramulator: A fast and extensible dram simulator. *IEEE Computer architecture letters*, 15(1):45–49, 2016.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.

Leskovec, J. and Krevl, A. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

Marcheggiani, D. and Titov, I. Encoding sentences with graph convolutional networks for semantic role labeling. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pp. 1507–1516, Copenhagen, Denmark, September 2017. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/D17-1159.

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., and Yu, P. S. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, 2019.

Xu, K., Hu, W., Leskovec, J., and Jegelka, S. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=ryGs6iA5Km.

Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W. L., and Leskovec, J. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 974–983. ACM, 2018.