High-Performance, Scalable, and Accurate GNN Inference Serving System

Yiwei Zhang and Salman Munaf
MadData
Group: p25

1 Introduction

Recently, Graph Neural Network (GNN) has gained increasing popularity and drew the attention of many researchers because of its expressive power in modeling the relationships (edges) between different entities (nodes) in a graph. It has achieved state-of-the-art inference performances in plenty of graph-related tasks, including node classification[15], graph classification[21], and link prediction[18].

Similar to Deep Neural Networks (DNNs), a GNN model's life cycle is made up by two stages: the training stage and inference stage. During the training stage, the GNN model takes graphs in the training data set as inputs and iteratively update the weights of the model using the back-propagation algorithm. It poses high computation cost because each node need to gather and aggregate feature vectors from all of its neighbor nodes and sometimes edges to compute a forward pass. Many modern GNN models are utilizing stochastic node sampling techniques to train on a small portion of supporting neighbors in each epoch. During the inference stage, the trained GNN model would perform the forward pass on the whole input graph. We can notice the computation patterns are quite different between the training and inference stage. Each graph only needs a few iterations without the backward pass in the inference stage. During the training stage, people would care about the accuracy or loss of the GNN model. While during the inference stage, people would care more about latency, throughput and acceptable performance degradation under heavy load.

In this work, we focus on the inference performance of the GNN models for three reasons. First, inference stage takes up most of the life cycle of a GNN model. After a GNN model has been deployed, tons of application would rely on the inference performance of the GNN model. For example, Facebook processes tens of trillions of inference queries per day[20, 5]. Second, although the computation cost of the inference stage is lower because it lacks backward pass and repeated updating of weights, it faces more strict requirements on latency and throughput [22], considering the customer-facing nature [2]. What's more, although training stage is always carried on powerful hardwares such as GPUs and TPUs, the inference stage is usually carried on the edge devices with much less computation resources and parallelism. Last but not the least, GraphBERT [24] has shown that we can directly transfer (or wirh light fine-tuning) the pre-trained GNN models to address new tasks and scenarios. It makes inference performance of the GNN models even more important.

Existing deep learning frameworks do not easily support high-performance GNN inference at scale. Tensorflow [1] and PyTorch [14] typically utilize data or model parallelism by splitting the input data samples or DNN models across multiple machine or GPUs. However, GNN models usually apply small DNN models on very large and irregular input data samples, which makes it hard to fit in a single device. The irregular data access pattern of GNN models is also quite different from that of DNN models, making it harder to deploy GNN models on such frameworks. Some specialized prediction serving systems [10, 3] mainly focus on DNN models, which makes it hard to deploy GNN models on such systems. Some recent GNN systems [12, 17, 9] have made advancements towards GNN training at scale. However, they either focus solely on GNN training or consider GNN training and inference as a whole, caring little about latency, throughput, and performance degradation under heavy load of GNN inference.

We try to build a GNN inference system that supports affordable, low-latency, high-throughput GNN inference at scale, while also preserving the accuracy of the model. To achieve this goal, we address three challenges for the system.

Computing Resources. Recent works on GNN systems are using different kinds of computing resources, such as single machine [12], distributed resources [9], and severless threads [17]. We try to build our system in the single machine setting because it's the general setting during GNN inference when you have limited computing

power.

Graph Sizes. Different graphs have different sizes and latency requirements. For example, spatial-temporal graphs in traffic prediction normally have 10^3 to 10^6 nodes and require latency of seconds[6]. However, social network graphs in recommendation system normally have 10^6 to 10^9 nodes and require latency of milliseconds[4]. We would focus on small graphs first and then extend to large graphs.

Application Numbers. DNN serving systems such as Clipper[3] and Tensorflow Serving[13] provides single-application solutions, while Nexus[16] coordinates the serving of multiple DNN applications. We try to provide a multiple-application solution with our system.

Overall, we try to solve these challenges and coordinate the serving of multiple GNN application on a single machine with various graphs size inputs.

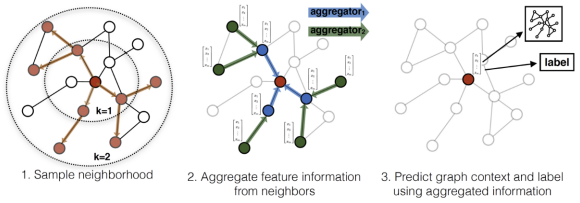

Figure 1: GraphSage

2 Background

The algorithm we are using is GraphSage[7], and inductive deep learning method for graphs, which can be used to generate nodes' low-dimensional representations. It can generalize to more unseen nodes compared to transductive algorithms. Some previously graph learning algorithms such as Graph Convolutional Networks and DeepWalk are all inherently transductive, which means they can only generate nodes' embeddings present in the fixed graph in the training phase.

Therefore, if the graph would evolve in the future and new nodes which are unseen in the training phase would be incorporated into the graph, then we would need to retrain the whole graph so that the new nodes' embeddings could be computed. This is a hugh limitation which makes the mentioned transductive ethods not efficient to be used on the evolving graphs such as protein-protein networks and social networks. It's because they are unable to generalize on the unseen nodes. Another major limitation of these transductive methods such as DeepWalk and Node2Vec is that they are not able to leverage the features of nodes such as node degrees, node profile information and text attributes. On the other hand, the mentioned GraphSage algorithm both exploits the rich features of nodes and each node's neighbourhood topological structure at the same time, in order to effectively generate new nodes' representations without the retraining process.

The whole working process of the GraphSage algorithm can be diveded into two major phases. First, the neighbourhood sampling of the input graph is performed. Second, learning aggregation functions are performed at each depth of search.

In this paper, we plan to use a popular package PyTorch Geometric and a popular graph learning benchmark Open-Graph-Benchmark libraries[8]. Among all the graph datasets in OGB, we choose the ogbn-products dataset. It's an unweighted and undirected graph, which is a common and simple setting. It represents an Amazon product co-purchasing network which can be used to make shopping recommendations or predict individual shopping preferences.

3 Design

3.1 Nvidia's Triton Inference Server

After our thorough investigation, we found out although inference serving system isn't quite a popular research area, it has attracted a lot of investments and startups in business scenario. This is why we choose Nvidia's Triton Inference Server as our base framework, which simplifies AI models deployment in production at scale. It supports many important features that we need. Multiple popular deep learning framework backends are natively supported such as TensorFlow, PyTorch, Python, ONNX Runtime and even some custom backends. It also supports many different kinds of inference requests with advanced dynamic batching and scheduling algorithms. It also supports multiple models or multiple instances of the same model be run on CPUs and GPUs using the same or different deep learning or machine learning backends. Live model updates is also supported. We don't need to reinvent the wheel.

The Triton Inference Server also supports maximising hardware utilization rate by design through dynamic batching and concurrent model execution in order to increase inference performance. Concurrent execution means you can run multiple different models or multiple copies of the same model on the same GPU. The Triton Inference Server also supports dynamically grouping inference queries on the server-side in order to maximize performance.

2 shows the architecture of the Triton Inference server together with its different connected components and it's functionalities. First, we create an AI model repository where we put out deep learning models together

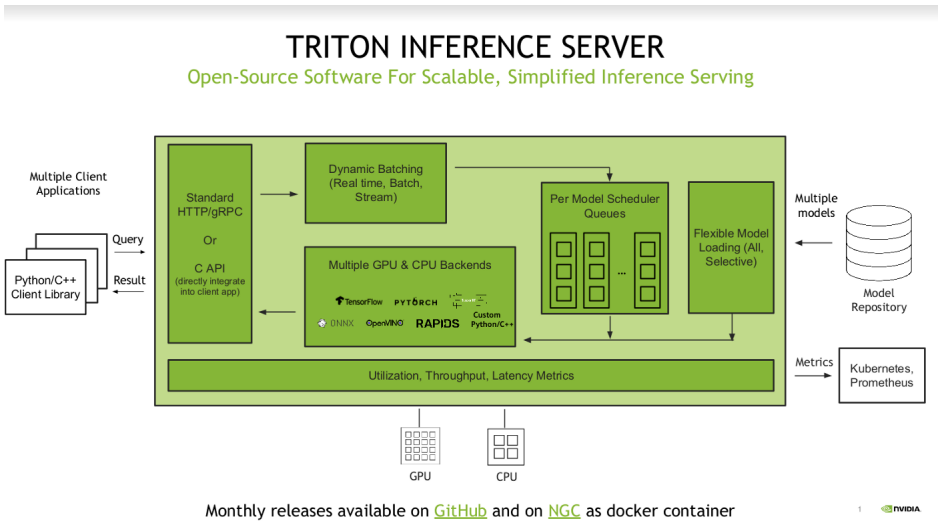

Figure 2: Triton Inference Server Architecture

with their individual configuration files. Config.pbtxt, the model configuration file, contains very detailed permissible input/output dimensions and data types, desired batch sizes, platforms/frameworks and versions because the server have no access to the details about these configurations. So we put them into a separate configuration file. After that, we would retrieve the AI model repository in the Triton Inference server to begin serving all those models on a single CPU, GPU or even on multiple GPUs.

When models are retrained with incoming new data, we don't need to restart the Triton Inference server or even interrupt the application. We can just use different polling modes to easily make model updates. After the Triton Inference server is loaded with all the models, then we can begin sending the inferences queries for the input data to the served/hosted model on the Inference server. Then we can receive the predictions or corresponding embedding that we need. All these inference requests/scripts could be written in a client application in C++/Python.

3.2 Deployment of GraphSage Model on Triton

After we complete the the Triton Inference server setup, we can start hosting/serving/deploying the trained GraphSage model on the Inference server. We need to follow the steps to deploy the GraphSage model on the Inference server:

1. Make the trained GraphSage model converted into a format which can be located on the server such as the JIT traced version.
2. Create an individual config.pbtxt configuration file for the model.
3. Start the Triton Inference server using the model generated above.

The first step needs us to convert the trained GraphSage model into its JIT traced evrsion. Torchscript can be used to create optimizable and serialisable models directly from PyToch code according to the PyTorch' official documentation. We are allowed to export our models and re-use them in other programs such as some efficiency-orienting C++ programs.

To export a model, we would need standard length and dummy inputs and to execute one model's forward pass. With this one model's forward pass fed with dummy inputs, PyTorch will keep track of the various operations of each tensor in the model and also record the operations so that the "trace" of the model could be created. Because the trace it created is relative to the dimension of the dummy inputs, the inputs of the model would always be limited by the dummy input dimensions in the future and won't be effective for other batch sizes or sequence lengths. Therefore, it's suggested that we could trace our model with the largest dimension of the dummy inputs we can think of that can be fed into the model. We could also use truncation or padding on input sequences besides it.

There are totally seven inputs required by us to create a trace of PyTorch Geometric's GraphSage model.

1. Padded feature matrix including nodes in the computation graph
2. adjacency edge list for all the edges involved in layer 3

3. the bipartite graph shape at Hop 3
4. adjacency edge list for all the edges involved in layer 2
5. the bipartite graph shape at Hop 2
6. adjacency edge list for all the edges involved in layer 1
7. the bipartite graph shape at Hop 1

We can create the configuration file (config.pbtxt) for PyTorch Geometric GraphSage model according to the inputs and outputs by specifying their types and dimensions.

Finally, we could instantiate the Triron Inference server with this added model above in the model repository.

3 shows the structure of our framework. We built an Amazon Product Recommendation app with the help of Graph Learning Algorithms, Graph Database ArangoDB and Nvidia's Triton Inference server.

After we get the trained PyTorch Geometric's GraphSage model, we can host/serve/deploy the model on Nvidia's Triton Inference server. After that, we can update the Amazon Product Graph that we stored inside the graph database ArangoDB. We do this by sending inference requests to the Inference server with the GraphSage model and updating the predicted node embeddings. After the graph we stored inside the graph database ArangoDB is completely updated with the predicted node embeddings, we could make recommendations for amazon products by using ArangoDB's AQL query language similar to SQL.

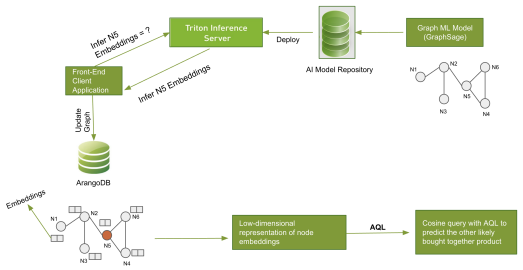

Figure 3: Framework

4 Evaluation

4.1 Experiment Setup

We deployed this framework on Google Cloud. And we are using the deep-learning VMs provided by Google Cloud with a single NVIDIA Tesla T4 GPU which has 16GB GPU memory.

4.2 Single Client

For demonstration purpose, we calculated the cosine similarity between predicted node embeddings in ArangoDB and showed recommendation results in the visualization Figure 4.

Each node represent an Amazon product. If two nodes are connected together, it means theses two products have high cosine similarity in their node embeddings and are likely to be recommended by each other.

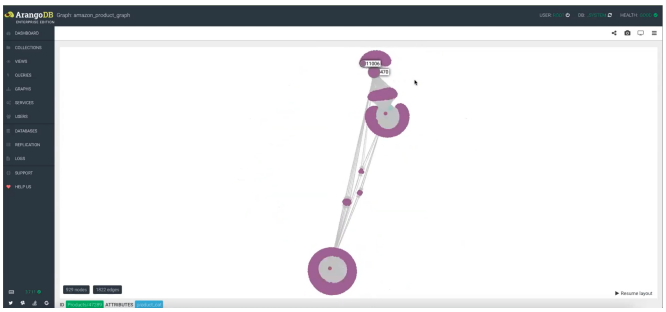

Figure 4: Recommendation Visualization

4.3 Multiple Clients

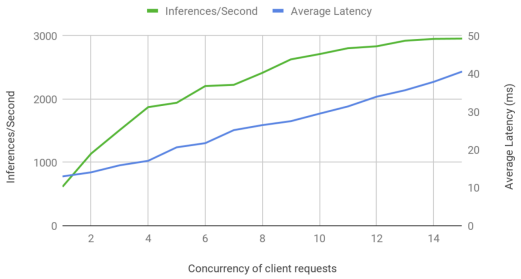

Figure 5: Scaling performance

We also investigate the scaling performance of the system by recording the Inference per Second and Average Latency when the server is handling multiple inference request clients shown in Figure 5.

We hold 12 copies of the same GraphSage model on the same GPU. Each model only take up 1.33GB GPU Memory with the help of Triron Inference server. When we are having more than 12 concurrent inference request clients such as 14 clients, each model copy/instance would fulfill one request simultaneously. The rest 2 clients are queued in Triron Inference server's scheudler queue to be executed after the 12 clients are finished. This approach would increase the utilization rate of the hardware.

We can see from the graph that Inferences per Second would grow rapidly at the beginning when we are

increasing the concurrency of client requests. Then it will plateau at around 12 clients. This meets our expectation that performance would not increase when further inference request clients are queued in server's scheduler.

We can also that average latency is steadily increasing when we are increasing the concurrency of client requests. To investigate this, we breakdown the latency into five parts and record the time spent on different parts with increasing concurrency of client requests. The result is shown in Figure 6.

These five parts in latency are:

1. Client Send,
2. Network + Server Send/Recv,
3. Server Wait,
4. Server Compute,
5. Client Recv.

We can see from the graph that time spent on Client Send and Client Recv stay the same with differnet concurrency of Client Requests. This is straightforward because client side would not degrade when there are more concurrent inference clients.

Time spent on Server Computer steadily grows with increasing concurrency of client requests. This is because we are grouping inference requests on the server side with dynamic batching. The inference requests are not computed at the same time. With more concurrent client requests, we will need more batches to complete the computing.

Time spent on Network+Server Send/Recv and Server Wait altogether is increasing with more concurrency of client requests. When we are having multiple inference client requests, the communication overhead over network would be significant since we are not sending simple control information over the network but the node embeddings. Server Wait time would only appear when we are having 15 or more concurrent Inference clients. This meets our expectation because we only have 12 model copies on the single GPU. If we have more concurrent client requests, we would definitely need some client requests to wait in the scheduler queue.

So we can see the reason behind the increasing average latency is that server computing time and communication overhead would grow with more concurrent client requests, some client requests may even need to wait for execution.

5 Related Work

We identify there are three areas close to our work: DNN Serving Systems, GNN Training (and Serving) systems,

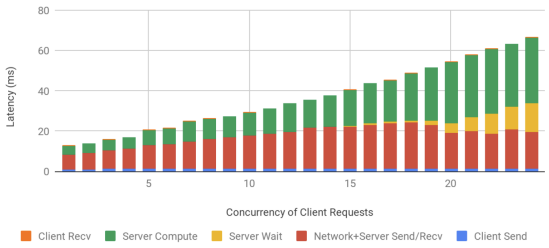

Figure 6: Latency breakdown

and GNN Inference Accelerators. We also evaluate their shortcoming in the previous section.

DNN Serving Systems TensorFlow-serving[13] serves DNN models in production settings and offers a high-performance prediction API to simplify deploying new algorithms. Clipper[3] provides a general-purpose low-latency prediction serving system, which simplifies model deployment across frameworks and applications. Pretzel [10] is a prediction serving system introducing a novel white box architecture enabling both end-to-end and multi-model optimizations.

GNN Training (and Inference) systems Dorylus [17] is is a distributed GNN training system that scales to large billion-edge graphs with low-cost cloud resources. GNNAdvisor [19] is an adaptive and efficient runtime system to accelerate various GNN workloads on GPU platforms, providing support for both training and inference stage. ROC [9] is a distributed multi-GPU framework for fast GNN training and inference on graphs with graph partitioning and memory management optimizations.

GNN Inference Accelerators Lots of hardware accelerators have been proposed for GNN inference acceleration[11, 23]. Another work [25] accelerates GNN inference by pruning the dimensions in each layer with negligible accuracy loss. GIN [5] generates high-efficient inference kernels of GNNs without requiring users writing low-level codes.

6 Future work

We will try to investigate how different embedding sizes would influence the inference performance. It would also be interesting to deploy different models or multiple copies of the same model on multiple GPUs and investigate the scaling performance. Another interesting idea would be to compare the inference performance on CPU and GPU and further compare the cost against Google Cloud deployment.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] L. Bernardi, T. Mavridis, and P. . Estevez. 150 successful machine learning models: 6 lessons learned at booking.com. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019.

[3] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 613–627, 2017.

[4] S. Fan, J. Zhu, X. Han, C. Shi, L. Hu, B. Ma, and Y. Li. Metapath-guided heterogeneous graph neural network for intent recommendation. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2478–2486, 2019.

[5] Q. Fu and H. H. Huang. Gin: High-performance, scalable inference for graph neural networks.

[6] S. Guo, Y. Lin, N. Feng, C. Song, and H. Wan. Attention based spatial-temporal graph convolutional networks for traffic flow forecasting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 922–929, 2019.

[7] W. L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 1025–1035, 2017.

[8] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687*, 2020.

[9] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems*, 2:187–198, 2020.

[10] Y. Lee, A. Scolari, B.-G. Chun, M. D. Santambrogio, M. Weimer, and M. Interlandi. {PRETZEL}: Opening the black box of machine learning prediction serving systems. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 611–626, 2018.

[11] S. Liang, Y. Wang, C. Liu, L. He, L. Huawei, D. Xu, and X. Li. Engn: A high-throughput and energy-efficient accelerator for large graph neural networks. *IEEE Transactions on Computers*, 2020.

[12] 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). USENIX Association*, pages 533–549, 2021.

[13] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139*, 2017.

[14] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.

[15] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Gallagher, and T. Eliassi-Rad. Collective classification in network data. *AI Mag.*, 29:93–106, 2008.

[16] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram. Nexus: a gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 322–337, 2019.

[17] J. Thorpe, Y. Qiao, J. Eyolfson, S. Teng, G. Hu, Z. Jia, J. Wei, K. Vora, R. Netravali, M. Kim, et al. Dorylus: affordable, scalable, and accurate gnn training with distributed cpu servers and serverless threads. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 495–514, 2021.

[18] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio', and Y. Bengio. Graph attention networks. *ArXiv*, abs/1710.10903, 2018.

[19] Y. Wang, B. Feng, G. Li, S. Li, L. Deng, Y. Xie, and Y. Ding. Gnnadvisor: An adaptive and efficient runtime system for {GNN} acceleration on

gpus. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 515–531, 2021.

[20] C.-J. Wu, D. M. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. M. Hazelwood, E. Isaac, Y. Jia, B. Jia, T. Leyvand, H. Lu, Y. Lu, L. Qiao, B. Reagen, J. Spisak, F. Sun, A. Tulloch, P. Vajda, X. Wang, Y. Wang, B. Wasti, Y. Wu, R. Xian, S. Yoo, and P. Zhang. Machine learning at facebook: Understanding inference at the edge. *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344, 2019.

[21] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? *ArXiv*, abs/1810.00826, 2019.

[22] N. J. Yadwadkar, F. Romero, Q. Li, and C. Kozyrakis. A case for managed and model-less inference serving. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 184–191, 2019.

[23] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie. Hygcn: A gcn accelerator with hybrid architecture. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 15–29. IEEE, 2020.

[24] J. Zhang, H. Zhang, C. Xia, and L. Sun. Graph-bert: Only attention is needed for learning graph representations. *arXiv preprint arXiv:2001.05140*, 2020.

[25] H. Zhou, A. Srivastava, H. Zeng, R. Kannan, and V. Prasanna. Accelerating large scale real-time gnn inference using channel pruning. *arXiv preprint arXiv:2105.04528*, 2021.