Improve the performance of MariusGNN

Saanidhi Arora   Rahul Chunduru   Sunaina Krishnamoorthy   Chaithanya Naik Mude
{sarora45, chunduru2, skrishnamoo5, cmude}@wisc.edu

Abstract

Graph Neural Networks (GNNs) are widely used in knowledge graphs, social networks, and protein structures. Real-world applications use graphs consisting of millions of nodes and correspondingly millions to billions of edges, requiring hundreds of GBs of data and significant compute, which can become expensive and challenging to use GPU accelerators. This makes machine training of GNN on large-scale graphs even more challenging. In this project, we wish to optimize the MariusGNN pipeline which has a single GPU, full memory stack asynchronous training architecture. In particular, we show that using CUDA streams to interleave GPU compute with data transfer significantly improves device utilization. We have also made other storage and CPU computation improvements that improve upon memory utilization. Finally, we hope that our work results in MariusGNN to be taken as a competitive and efficient training framework for massively large GNNs.

1 Introduction

Graphs are commonly used to encode relationships between entities in many domains ranging from social media and knowledge bases. Real-world applications use graphs consisting of millions of nodes and correspondingly millions to billions of edges, for example, Facebook's social media graph or DNA molecule chain.

There are several learning tasks of interest on large-scale graphs. Broadly, we can classify them into two categories. (1) Node classification, which involves categorizing a node (possibly with features) with a pre-defined label. An example use case is that of classifying a social media profile as a bot/malicious user or classifying paper in a citation graph etc, (2) Link prediction, which involves determining the likelihood of having a compatible edge between nodes. An example use case is that of the friendship recommendation feature on Facebook.

Graph Neural Networks has emerged as the state of the art systems for performing classification tasks on graphs. A training example for GNN consists of the node (edge) label and its neighborhood in the case of Node (link prediction) classification tasks. However, training GNNs is considerably more difficult than conventional neural networks. The first challenge that arises while training GNNs is that, unlike image/text models, the training examples are mutually dependent (as per the graph structure). Thus, the creation of a mini-batch needs to take the graph structure into account. Sampling is generally used to limit the number of nodes required for multi-hop training. Additionally, the storage and compute requirement to process large embed dings of massively huge graphs make training even more difficult. Effectively utilizing compute and storage resources for graph training tasks is therefore a non-trivial problem.

MariusGNN [10] is a relatively new GNN framework that performs graph training on a single GPU machine while utilizing the full memory stack. To perform efficient training with large graph data sets on disk, it uses a staleness-bound asynchronous batch pipe-lining architecture - where CPU sample batches, are transferred to GPU memory and operated upon for model updates.

While pipelining helps cost I/O cost, there is still GPU underutilization even for small graphs. The peak GPU utilization is 15-20% less than similar systems which are surprising. There are several areas of improvement (both storage and compute-related) that we believe are resulting in the under-performance of Marius. These result in a significant slowdown when Marius is trained for large graphs.

In this project, we wish to exploit more areas where communication can be overlapped with computation in the Marius training pipeline, thereby increasing its efficiency. We hope to achieve this by employing NVIDIA Cuda streams in the pipeline for better GPU utilization, and likewise, improve upon the pre-fetch of active edges by making doing it in an asynchronous way, along with changing int64 to int32 at necessary places for efficient data representation.

We hope that with our work, Marius becomes a more effective, cheaper, and competitive option for training large-scale graph machine learning tasks.

2 Background

In this section, we discuss the necessary background on GNNs and CUDA streams necessary to better understand the motivation behind this work.

2.1 Graph Neural Networks

Graph Neural Networks (GNNs) are widely used in knowledge graphs, social networks, and protein structures. GNNs are neural networks that may be applied directly to graphs to perform node-level, edge-level, and graph-level prediction tasks. By encoding data dependencies in the graphs, they achieve state-of-the-art performance in node classification and link prediction. The ability of GNN to model the dependencies between nodes in a network offers a breakthrough in graph analysis research. The basic purpose of a GNN architecture is to learn an embedding that incorporates information about its surroundings.

Prediction problems on graphs can be divided into three categories: graph-level, node-level, and edge-level. In a task at the graph level, we predict a single property for the entire graph. For a node-level task, we make a prediction about a node's property. We want to estimate the characteristics or existence of edges in a graph for an edge-level job.

Most graphs in the production level setting consist of millions of nodes and correspondingly millions to billions of edges, requiring hundreds of GBs of data, which can become expensive and challenging to use GPU accelerators. This makes training such large-scale GNNs even more challenging.

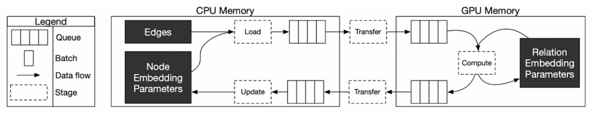

Figure 1: Marius system design

Graph neural networks have evolved into effective and useful tools for machine learning tasks in the graph domain in recent years. This advancement can be attributed to advancements in expressive power, model flexibility, and training methods.

2.2 MariusGNN

MariusGNN [10] [11] is an efficient open-source massively large-scale graph neural network training procedure. It utilizes the entire storage hierarchy including a disk with a single GPU node and is shown to perform better than multi-GPU state-of-the-art (SoTA) solutions. The efficient performance of training with large graphs on differently sized datasets can be attributed to its asynchronous batch pipelining approach. It involves two steps, firstly, the graph is partitioned into subgraphs, that can easily fit into CPU and GPU memory. Then, CPU samples batch in a pipelined fashion.

MariusGNN design 7 choice is in contrast with several existing state-of-the-art (SoTA) GNN training solutions, such as Deep Graph Library (DGL) and PyTorch Geometric (PyG) which use CPU memory for graph storage and distributed mini-batch training over multiple GPUs. It has been shown that MariusGNN's disk-based, single GPU training can be 8× faster than eight-GPU deployments of SoTA Systems. The procedure tries to minimize the total time required along with maximizing disk utilization for training without compromising accuracy. This will also reduce the monetary cost reduction because of the increased utilization of resources that are available.

2.3 CUDA Streams

A simple CUDA program consists of three steps 2, including copying the data from host to device, kernel execution, and copying data back from the de-

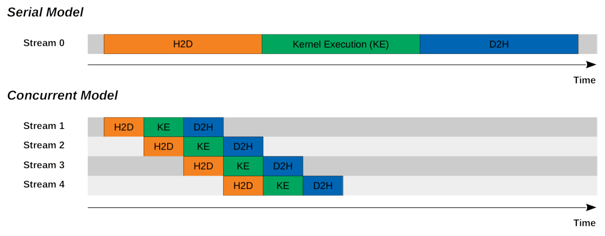

Figure 2: CUDA workflow

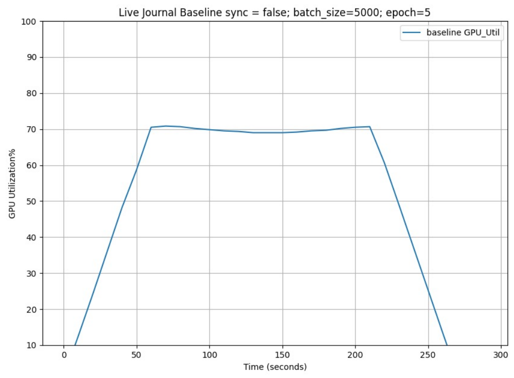

Figure 3: Marius baseline GPU utilization for live journal

vice to host. In the serial execution model, data is first copied from the input memory of the host to the device, then the kernel computes the output, and lastly, the output is copied from the device back to the host. In the concurrent model, these three tasks are made asynchronous.

In order to make these steps execute concurrently, we can use CUDA streams. A CUDA stream is a sequence of operations that execute in issue order on the GPU. CUDA operations in different streams may run concurrently and may be interleaved. The operations including memory copy from host to device, kernel execution, and memory copy from device to host can be broken down into parts, overlapped, and interleaved.

3 Why do we need a better Marius?

The current implementation of Marius doesn't fully utilize GPU for large compute-bound graphs. This is evident from the GPU utilization plot (Fig 3) for training link prediction on livejournal dataset.

There are several opportunities to improve the performance of MariusGNN further. Currently, MariusGNN uses a DENSE data structure followed by in-CPU sampling for better multi-hop GNN training performance over SoTA systems. This can be enhanced by using CUDA streams for the entire data pipeline similar to the SALIENT system [7] for higher GPU utilization.

Another area of improvement is storage/buffer management. MariusGNN represents a graph as an edge list. For large-scale graphs, storing the entire graph structure and the base representations of the nodes in memory is not possible. In such situations, mini-batches of training data are constructed over subgraphs (partitions) of the original graph which can be loaded into the main memory. The current MariusGNN implementation of functions that initialize and update these in-memory subgraphs are storage-inefficient and at times blocking. Also, for a decoder-only GNN model, they perform huge amounts of redundant I/O and compute tasks that can be skipped for efficiency.

Since the graph storage sizes are extremely large, and there are many GNN neighborhood dependencies, mini-batch training coupled with multi-hop neighbor sampling is necessary for learning GNNs over large-scale inputs. Creating mini-batches during training requires much computation and data movement due to the exponential growth of multi-hop graph neighborhoods along network layers. Transferring data from CPU to GPU also creates a bottleneck, and the transfer time grows longer as the neighborhoods get larger. To mitigate these bottlenecks, the SALIENT paper [7] uses pipelining of batch transfer with GPU computation. SALIENT also increases GPU utilization even further by overlapping data transfers with GPU training computations by using separate GPU streams for computation and data transfer. We believe that this same technique can be applied to MariusGNN to increase GPU utilization.

4 Our Contributions

We present our contributions to improving the MariusGNN training pipeline. Briefly, we have made changes to 2 compute optimizations and 1 storage

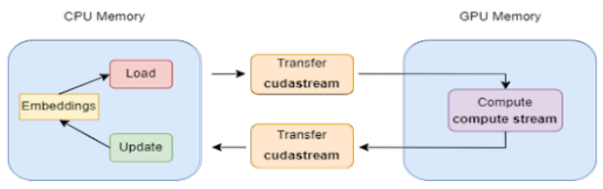

Figure 4: Represents CUDA stream in training pipeline

optimization.

4.1 CUDA streams

We used CUDA streams to overlap GPU computation with data transfer to/from the host and device. The PyTorch C++ API [1] supports CUDA streams with the CUDAStream class (implemented in CUDAStream.h) and other useful helper functions to make streaming operations easy. To execute a GPU operation on CUDA streams, we use the very simple yet powerful CUDA stream guards API. The following is a code snippet indicating our change.

```
using namespace at::cuda;

// get a new stream from CUDA pool
CUDAStream newStream =
            getStreamFromPool();

// set the current CUDA stream to
// `myStream` within the scope using
// CUDA stream guard
{
  CUDAStreamGuard guard(newStream);

  // data transfer or GPU compute
  ...
}
```

By placing data transfer and GPU computing by workers in separate CUDA streams, we expect the interleaving of tasks which should lead to higher GPU utilization.

Present-day GPU chips can support up to 32 concurrent stream operation execution. To exploit this level of parallelism as much as possible, We have employed CUDA streams described above in two different modes.

- **Common data stream**. In this mode, we use a common stream for all toDevice (similarly, toHost) data transfer workers. This is achieved by creating a data stream as a *static* object accessible by all instances of the same worker class.
- **Separate per worker streams**. In this mode, we use separate data streams for each individual worker. Since the number of workers performing data transfer is 4, this leads to a total of 4 * 2 (equal number to and for) + 1 (for compute operations) = 9. As this number is below max concurrent CUDA stream execution possible, we expect significant speedup in this mode.

4.2 Async Prefetch of active nodes/edges

The active edges are currently being prefetched synchronously, we added *setActiveEdgesAsync* (and *setActiveNodesAsync*) to prefetch active edges asynchronously and update the information we store for the graph in the memory, so as to speed up the computation. This decreases the time taken to train when there are large numbers of swaps, which requires prefetch of active edges, and performing in asynchronously will enhance the performance.

This modification especially has visible effect when the training graph consists of many partitions and therefore a significant portion of the training time is spent in performing swaps. By performing asynchronous fetch of nodes/edges, we effectively overlap data communication with computation.

4.3 Int32 for data representation

We changed the int64 data representations to int32 to effectively represent the graph data. The MariusGNN codebase used int64 representations to store the number of nodes and edges for the graphs used for training. The graph data, on the other hand, might be efficiently stored in int32 value types.

The number of nodes and edges in ogbn_arxiv dataset is nearly 1.1 million $\approx 2^{24}$ which is less than 2^{31}. Hence, int32 data representation can easily accommodate the graph datasets used in our project. These adjustments resulted in a slight decrease in CPU memory utilization.

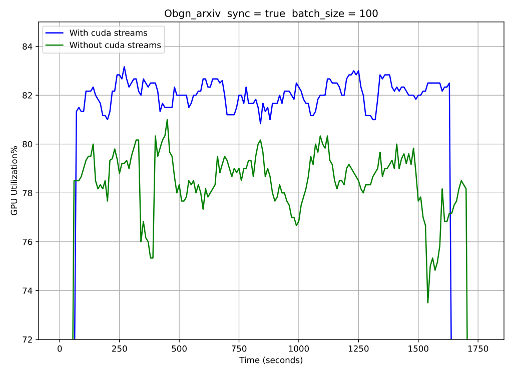

Figure 6: Results for Ogbn-arxiv

5 Evaluation

In this section, we present our experimental setup and methods used to obtain results and also mention a few difficulties we faced during the evaluation.

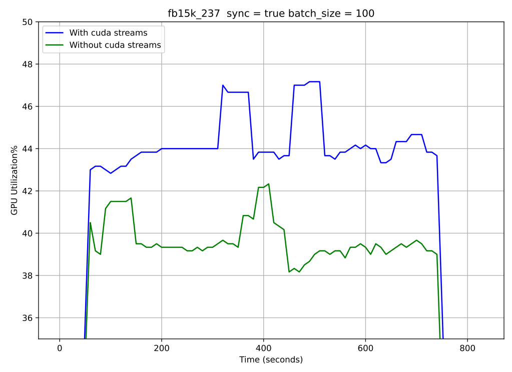

Figure 5: Results for fb15k-237

5.1 Baselines

We used the performance of MariusGNN as our baseline for the comparison. We ran GNN training using the datasets and the hardware setup described below to obtain our baseline.

5.2 Workloads

We ran training on three different graph datasets. We chose datasets of three different sizes so that we could do a comprehensive comparison across different-sized graphs, and observe whether our additions provided significant improvement in the training performance of the system. We used the **fb15k_237** dataset (contains around 35k edges) as our small-scale link prediction task, **ogbn_arxiv** graph (contains around 90k edges) for medium-sized node classification task, and **live_journal** dataset (contains around 69M edges) for large sized link classification task.

5.3 Hardware Setup

We used Google Cloud Platform for evaluating our implementation. We used several GPU instances for our training based on availability (GPU instances are so hard to reserve on GCP!)

- single NVIDIA Tesla P4 GPU attached to a VM instance, with 100 GB storage.
- single NVIDIA A100 40GB attached to a VM instance, with 100 GB storage.
- single NVIDIA V100 attached to a VM instance, with 500 GB storage.

To collect metrics from the GPU, we installed a **GPU monitoring agent** provided by Google Cloud Platform on the VM instance. We followed a tutorial to install the agent on Linux VMs, by downloading the necessary packages and creating a Python virtual environment and starting the monitoring agent daemon. Once this was set up, the metrics and logs get sent to the Google Cloud Console from where we were able to see the graphs for GPU utilization, GPU memory utilization, temperature, and so on.

5.4 Metrics Tracked

The following are metrics of interest to us.

- **GPU utilization**: We used Google cloud's Linux GPU Monitoring agent for collecting metrics from the GPU. An increase in GPU Utilization was our main goal, this shows us how much adding CUDA streams helps in better utilizing the resource.
- **Total training time**: Training time should ideally decrease after adding our improvements since more work should happen in parallel.

- **Model Accuracy**: We compare accuracy with our baseline to ensure that we are still able to achieve similar levels of accuracy to the original and that our implementation has not hampered training.
- **Memory utilization**: We made some storage optimizations, so we used memory utilization as a metric to check if the total memory used has decreased.
- **Disk I/O**: The amount of disk I/O should remain at a similar level to the original with just the addition of CUDA streams since this optimization only overlaps communication with computation between CPU and GPU.

Hyper parameters: We ran our experiments with varying batch sizes 1000-10000, for 10-100 epochs. For small training graphs, we chose a larger batch size and more number of epochs to better demonstrate the impact of the CUDA streams on the GPU utilization.

The mode of training can be either **Sync** or **Async**. The synchronous trainer waits for a mini-batch model update to be applied before the next batch's processing. Whereas, Async mode uses thread workers to transfer multiple batches from host to device while training is still being executed. It is to be noted that async mode adds staleness to the pipeline.

6 Results

In this section, we discuss about the effect of CUDA streams and how it varies when synchronous trainer is employed or pipeline trainer is employed. The effect of batch size when CUDA streams are used. We also discuss about the effect of usage of CUDA streams per worker vs CUDA streams per task, as in one for each of compute, transfer of data from CPU to GPU memory and from GPU to CPU memory. Finally we also discuss about the disk utilization across the experiments.

6.1 Synchronous trainer vs Pipeline trainer

The synchronous trainer performs all the computations one-by-one, so the total time taken by it would be more than pipeline trainer which can compute more due to its asynchronous way. The total time taken by pipeline trainer should be lesser than that of synchronous trainer.

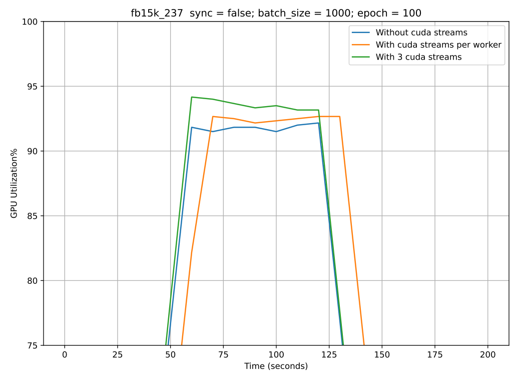

Figure 7: Results for Asynchronous pipeline across three variants for fb15k-237 dataset; a. Without CUDA streams; b. With CUDA stream for each worker (with 4 workers for async); c. With one CUDA stream for each compute, data transfer from CPU to GPU and from GPU to CPU

6.2 Effect of CUDA streams per worker

We present our results of employing CUDA streams for training. Usage of CUDA streams enhances the GPU utilization and also decreases the total time taken for the computation. We also observed that using separate data streams per worker marginally improves both the utilisation and training time.

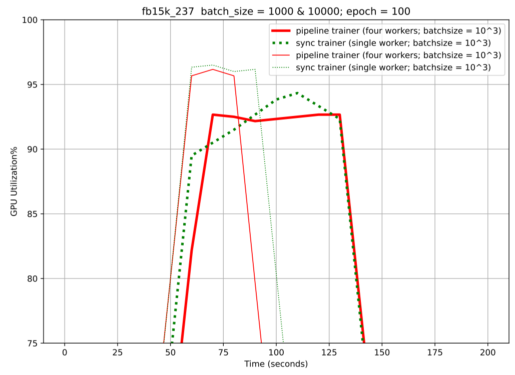

Figure 8: Comparison between Synchronous trainer and Pipeline trainer with variation across batch size

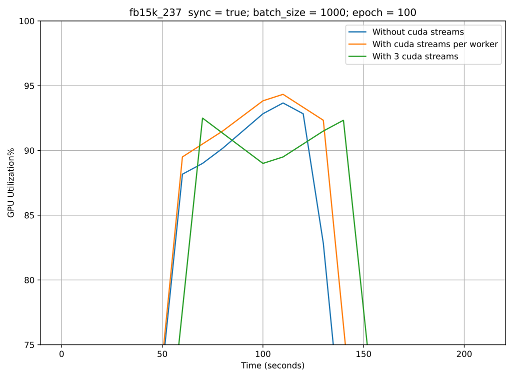

Figure 9: Results for Synchronous pipeline across three variants for fb15k-237 dataset; a. Without CUDA streams; b. With CUDA stream for each worker (with a single worker for sync); c. With one CUDA stream for each compute, data transfer from CPU to GPU and from GPU to CPU

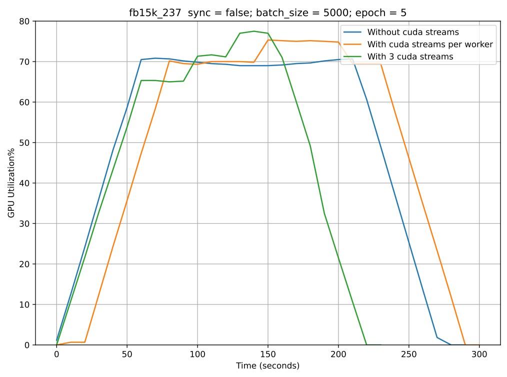

Figure 10: Results for Asynchronous pipeline across three variants for liveJournal dataset; a. Without CUDA streams; b. With CUDA stream for each worker (with a single worker for sync); c. With one CUDA stream for each compute, data transfer from CPU to GPU and from GPU to CPU

6.3 Effect of batch size

As we increase the batch size, the GPU utilization goes up and the total time required to finish the computation decreases as expected. The results are shown in Figure 11 and 12. We also observe that smaller batch size has considerably longer total training epoch time. We conjecture this is because of the constant fixed overhead incurred during data transfer with large number of smaller batches.

6.4 Device choice

The choice of GPU device greatly affects these plots. In particular, the gpu utilization is low when we used NVIDIA A100 as compared to NVIDIA P4, as the former is a much powerful compute resource.

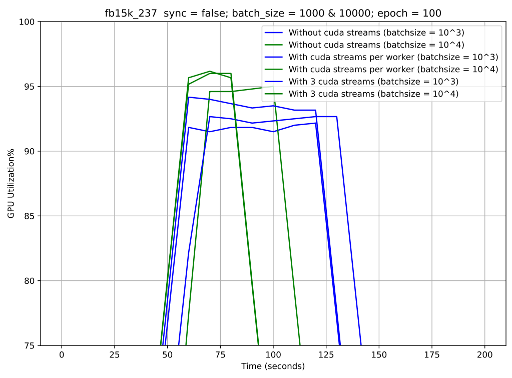

Figure 11: Results for Asynchronous pipeline varying batch size;

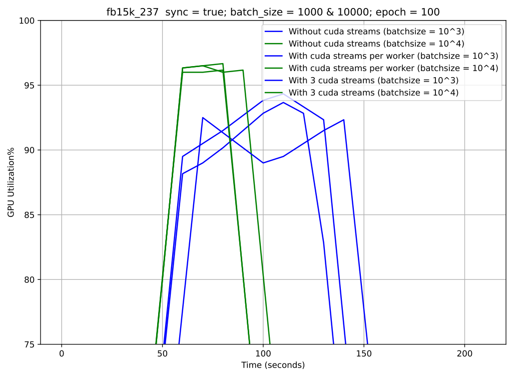

Figure 12: Results for Asynchronous pipeline varying batch size;

6.5 Disk I/O

The Disk utilization stays almost constant throughout varied experiments discussed above is shown in the below figure 13 as expected.

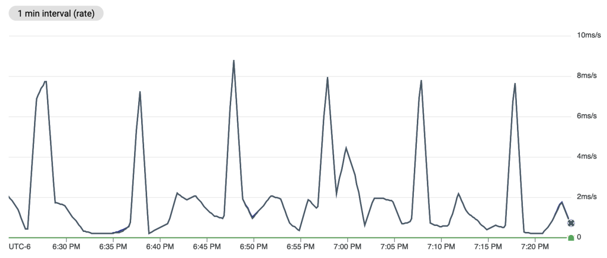

Figure 13: Variation of Disk I/O utilization throughout various experiments

6.6 Accuracy

As expected, we observe that our cuda stream and int32 changes don't affect model accuracy. This is because our modifications don't change the pipeline semantics, rather only optimises resource utilization. We present accuracy results of training **ogbn_arxiv** graph in different modes.

Mode	Batch #	epochs	accuracy
Baseline	1000	10	67.17
Cuda streams	1000	10	68.36

6.7 Difficulties faced

Some of the difficulties faced during evaluation are the low availability of GPU in both cloud lab and Google Cloud. We also troubleshoot docker-related issues, repeated ssh connection failures to Google cloud VM, cmake creating CMakeCache.txt at different locations, and installation setup issues because of a mismatch in versions of CUDA between the GPU toolkit and hardware.

7 Related Work

7.1 Improving GPU Utilization

The main bottlenecks that exist in GNN systems are the process of preparing data for GPUs and the transfer of data from CPUs to GPUs, rendering them inefficient for training large graphs with billions of edges. Much work focuses on optimizing these processes and eliminating these bottlenecks. One such system is SALIENT, which aims to improve GPU utilization by creating a performance-engineered neighborhood sampler, a shared-memory parallelization strategy, and the pipelining of batch transfer with GPU computation. SALIENT achieves a 3.3 to 4x speedup on its datasets using the Cuda streams.

An additional work that focuses on optimizing the preparation of data for GPUs is the BGL system [8] which uses a dynamic cache engine to minimize feature retrieving traffic, improves the graph partition algorithm to reduce cross-partition communication during subgraph sampling, and utilizes careful resource isolation to reduce contention between different data preprocessing stages.

7.2 Deep Graph Library

Deep Graph Library enables users to quickly adapt and leverage existing deep learning framework components across several frameworks. They provide cutting-edge advancements for training large graphs with GPU-accelerated efficiency.

7.3 Pytorch Geometric

PyTorch Geometric is a deep learning library built on PyTorch for learning irregularly structured objects such as graphs and manifolds. They achieve high performance by leveraging dedicated CUDA kernels and in-memory dataset processing

8 Future work

CUDA streams provide a different approach to enhancing the performance of training procedures. Currently, we are using it for a single GPU and with one CUDA stream per worker and one CUDA stream per task, as in, one for each of the compute and data transfer from GPU to CPU and back. Currently, most Cloud Providers offer the capability to attach multiple GPUs to a single VM instance, and if we are just restricting to a single GPU, we might not be fully utilizing the resources we get. For being more monetary cost-efficient, one of the future approaches can be to extend the idea to utilize multiple GPUs. This can enhance GPU utilization furthermore, and might as well make Marius even more cost-effective.

[1]

References

[1] PyTorch CUDA API. https://pytorch.org/docs/stable/cuda.html.

[2] Web data commons - hyperlink graphs. http://webdatacommons.org/hyperlinkgraph/.

[3] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[4] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[5] Google. Freebase data dumps. https://developers.google.com/freebase. 2018.

[6] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. Ogb-lsc: A large-scale challenge for machine learning on graphs. *arXiv preprint arXiv:2103.09430*, 2021.

[7] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems*, 33:22118–22133, 2020.

[8] Tim Kaler, Nickolas Stathas, Anne Ouyang, Alexandros-Stavros Iliopoulos, Tao Schardl, Charles E Leiserson, and Jie Chen. Accelerating training and inference of graph neural networks with fast sampling and pipelining. *Proceedings of Machine Learning and Systems*, 4:172–189, 2022.

[9] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo. Bgl: Gpu-efficient gnn training by optimizing graph data i/o and preprocessing. *arXiv preprint arXiv:2112.08541*, 2021.

[10] Jason Mohoney, Roger Waleffe, Henry Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. Marius: Learning massive graph embeddings on a single machine. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 533–549. USENIX Association, July 2021.

[11] Roger Waleffe, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman. Marius++: Large-scale training of graph neural networks on a single machine. 02 2022.

[12] Minjie Yu Wang. Deep graph library: Towards efficient and scalable deep learning on graphs. In *ICLR workshop on representation learning on graphs and manifolds*, 2019.

[1]If not mentioned specifically, dataset used for evaluations is fb15k-237 dataset. Thanks a lot for Roger Waleffe, Jason Mohoney for helping in understanding the Marius codebase and their constant support.