ARCHITECTURAL IMPLICATIONS OF EMBEDDING DIMENSION DURING GCN ON CPU AND GPU

Matthew Adiletta, David Brooks, Gu-Yeon Wei
Harvard University

ABSTRACT

Graph Neural Networks (GNNs) are a class of neural networks designed to extract information from the graphical structure of data. Graph Convolutional Networks (GCNs) are a widely used type of GNN for transductive graph learning problems which apply convolution to learn information from graphs. GCN is a challenging algorithm from an architecture perspective due to inherent sparsity, low data reuse, and massive memory capacity requirements. Traditional neural algorithms exploit the high compute capacity of GPUs to achieve high performance for both inference and training. The architectural decision to use a GPU for GCN inference is a question explored in this work. GCN on both CPU and GPU was characterized in order to better understand the implications of graph size, embedding dimension, and sampling on performance.

1 INTRODUCTION

A Graph Convolutional Network (GCN) is a type of Graph Neural Network (GNN) which applies convolution to graphically structured data [1]. A graph convolution layer can be disaggregated into two steps; neighborhood aggregation and update. During neighborhood aggregation, the sparse graphical structure of the data is exploited and each vertex aggregates the embedding vectors of its neighbors. During update, a convolution kernel is applied to transform the embeddings, similarly to a conventional Convolutional Neural Network (CNN). After the graph convolution layer, a non-linear activation function is applied, resulting in the input to the next convolution layer.

Both CPUs and GPUs have been widely used for training and inference of GCNs [2–11]. CPUs offer terabytes of memory capacity and a scalable computing platform while GPUs offer high memory bandwidth, latency tolerance, and massive compute parallelism. One challenge with using GPUs for Graph Convolution is memory capacity. Today's largest GPUs have memory capacity on the orders of 10s of GBs while large scale GCNs have a memory footprint on the orders of 100s of GBs to terabytes - if a graph cannot fit in a GPU's memory, alternative techniques must be explored for offloading the GCN to GPU.

GCN mini-batching with vertex aware clustering is one technique used to overcome memory capacity limitations when using a GPU. Mini-batching involves creating a subgraph from a set of vertices and their N-hop neighbors [12]. Minibatch sampling can be accomplished using batch-wise sampling or layer-wise sampling.

Batch-wise sampling starts by sampling all the vertices needed for a mini-batch computation. This can be accomplished by clustering vertices together and sampling vertices from a cluster, or randomly sampling vertices across the entire graph. Sampling within a cluster helps preserve data locality by increasing the reuse of embeddings; although clustering adds additional overhead which may not be tolerated in some cases.

After a set of vertices V_0 are sampled, all neighboring vertices must be discovered. Using a neighborhood sampling algorithm, the set of neighbors V_1 are found. The computation of the 1-hop intermediate representation for vertices V_0 can be found using neighborhood aggregation. For the next layer, all neighbors of set V_1 are added to the set of vertices $V = V_0 \cup V_1 \cup V_2$. This allows the computation for 2-hop intermediate representation of vertices V_0. For a GCN of depth D, sampling must occur D times leading to a set of vertices $V = V_0 \cup V_1 \cup ... \cup V_{D-1} \cup V_D$.

One problem with this batch-wise full-neighborhood algorithm is that as the number of layers increases, the number of vertices needed for the next layer's computation increases exponentially until a saturation point where almost the full graph is being used. Layer-wise sampling attempts to solve this problem by sampling within a layer. Rather than computing an output for a vertex in a single-shot, layer-wise sampling compute intermediate representations for each vertex, requiring a larger memory capacity than batch-wise sampling. The use case for layer-wise sampling and batch-wise sampling depends heavily on the sparsity of the graph and the total graph size. This relationship is not investigated thoroughly in this work.

Another solution to minimize exploding neighborhoods is to sample using a fixed number of neighbors per vertex during aggregation. This provides a predictable run time and fixed memory capacity. The GraphSage paper demonstrated that using fixed-size, uniform sampling could achieve high performance with as few as two samples per vertex [12]. Other works such as in Cluster Sampling [13] use a sampling fanout of (15, 10, 5). Fixed-size sampling changes the accuracy of the model; however, some cases show that this normalization strategy can actually improve accuracy. This work does not explore fixed-sampling, as the primary investigation is a performance characterization of the vanilla GCN algorithm. Advanced sampling and clustering techniques was left to future work.

Many works which implement vertex-sampling techniques are for training on GPUs. The reason they may not implement sampling for inference on GPU is because sampling on CPU is a large bottleneck. During sampling, the CPU must traverse a graph and touch all vertices which are required in the GPU computation. The CPU must also construct a sub-graph, aggregate embeddings, and move the data between the CPU and GPU, which is expensive. This is true especially for small graphs.

In a production GCN setting, the model weights may be located on the GPU, while the graph and feature vectors arrive through a query. Thus, for each inference, the entire graph and feature matrix must be offloaded. Furthermore, each embedding vector in the GCN likely has low reuse (depending on the sparsity of a graph), therefore, the FLOPs/bytes ratio of GCN is also very low. GPUs excel at problems with high FLOPs/bytes ratio, which means GPUs may not be the best solution for GCNs. Finally, for mini-batching cases, each time a mini-batch is sampled, the CPU must execute the sampling, which is slow. Then a large feature vector matrix and adjacency matrix must be offloaded to GPU. Both these operations bottleneck the GPU performance and cause severe under utilization of the GPU.

1.1 Contributions

The performance of GCN was evaluated for a sweep of embedding dimensions. From an application perspective, the embedding dimension is a hyperparameter useful for improving model accuracy. From an architecture perspective, the embedding dimension influences the ratio of sparse to dense compute, memory capacity, memory bandwidth utilization, and end-to-end latency. The impact of embedding dimension was studied from the architecture perspective, offering the following contributions:

1. ***Performance analysis of GCN on CPU***: The execution of GCN inference on CPU is a baseline for this work. GCN inference was characterized, and the contribution of key kernels on execution time was highlighted. This is important for the research community because it reveals the importance of accelerating sparse computation while keeping memory-capacity as a first-order requirement. It also confirms that sparse computation is a performance driver for GCN across all embedding dimensions. Finally, it highlights the architectural implications of the GCN embedding dimension hyperparameter.

2. ***Performance analysis of GCN on GPU***: The execution time breakdown for running full-graph convolution on GPU is shown. Memory capacity restrictions on the A100-40GB GPU prevented the characterization of some large-scale graphs; however, many graphs with commonly used embedding dimensions were profiled. Bottlenecks which appear when using a GPU for full graph convolution were highlighted. Finally, GCN inference performance on GPU was compared against CPU, leading to a discussion about when GPUs are better than CPUs for GCN inference.

3. ***Performance analysis of Full-Neighborhood-Sampling Graph Convolution on GPU***: The execution time breakdown for running neighborhood sampling with full neighborhoods using both batch-wise and layer-wise techniques during graph convolution when a graph does not fit in GPU memory was shown. One concern was the unbounded nature of using full-neighborhoods; however, the batch size was carefully selected as to near-maximally utilize, but not exceed, the memory capacity of the A100. Future systems with increased memory capacity may see improvements in run time due to reduced data movement, as fewer batches would be required.

1.2 Related Work

GraphSage evolved neighborhood sampling to use a fixed-neighborhood [12]. They noted that although the number of vertices in each layer still grows exponentially, the run-time and memory-capacity demands were more predictable than full-neighborhood algorithms.

Other works proposed methods for optimizing sampling and offload times between CPU and GPU. LazyGCN [14] was proposed to pipeline sampling and offload time using a two step process. First, a large batch of vertices were sampled on CPU using full-neighborhoods and offloaded to GPU. Then the GPU applied fixed-neighborhood mini-batch sampling on the batch of vertices. Trade-offs in the CPU cluster-batch size and GPU mini-batch size led to accuracy tradeoffs.

Global Neighbor Sampling (GNS) is another technique used to reduce sampling and offload time [13]. The GNS algorithm samples a set of vertices and stores them in the GPU memory. To select the next mini-batch, the algorithm prioritizes vertices with embedding vectors already located on GPU. The intent is to reduce data movement between CPU and GPU and speed up sampling time.

2 Characterization Methodology

GCN was profiled using nine benchmarks from the OGB datasets [15], listed in Table 1. The baseline system used in this performance characterization was a dual-socket Intel(R) Xeon(R) Platinum 8380 CPU with 40 cores per socket and 512 GB of main memory using DDR4-3200. This system is one of the largest non-distributed CPU systems available. To profile GPU, an NVIDIA A100 GPU was used with 40 GB memory capacity [16] with 32GB/sec **PCIe4.0** network fabric between host and GPU. The CPU host was a dual socket Intel(R) Xeon(R) Platinum 8380 CPU with 40 cores per socket and 512 GB of main memory using DDR4-3200.

Table 1: OGB Dataset Descriptions

Name	Num Vertices	Num Edges	Density
ddi	4,267	1,334,889	$7.33 * 10^{-2}$
proteins	132,534	39,561,252	$2.25 * 10^{-3}$
arxiv	169,343	1,166,243	$4.06 * 10^{-5}$
collab	235,868	1,285,465	$2.31 * 10^{-5}$
ppa	576,289	30,326,273	$9.13 * 10^{-5}$
mag	1,939,743	21,111,007	$5.61 * 10^{-6}$
products	2,449,029	61,859,140	$1.03 * 10^{-5}$
citation2	2,927,963	30,561,187	$3.56 * 10^{-6}$
papers	111,059,956	1,615,685,872	$1.31 * 10^{-7}$

The methodology for profiling GCN for full-graph and full-neighborhood-sampling approaches is detailed in this section. The characterization used the Torch-Geometric and Torch-Sparse frameworks. An embedding dimension sweep study is conducted across the hidden layers, while the input and output embedding dimensions of the graph do not change.

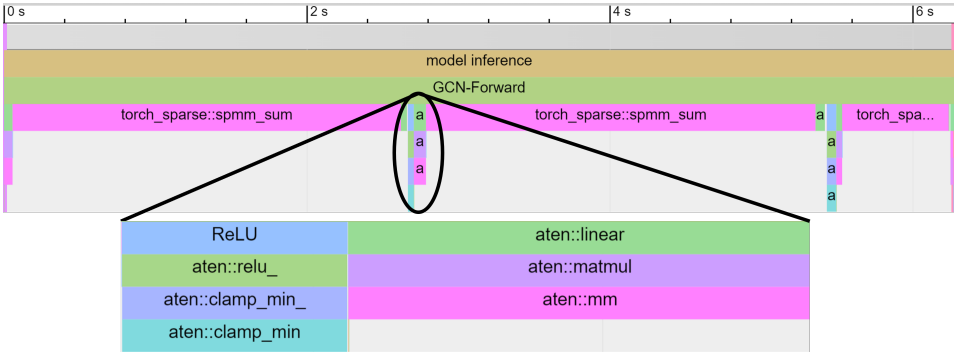

Figure 1: CPU execution timeline for *products* embedding dimension 256. This trace shows the large impact of the sparse kernel (torch_sparse:spmm_sum) on total execution time. The expanded section shows the activation function (ReLU) for the first GCN layer, and the dense linear matrix multiplication during the start of the second layer.

2.1 Full-Graph GCN Characterization

Each OGB graph was profiled on CPU and GPU for GCN using the following methodology.

1. Load the OGB graph in CSR format and feature matrix into CPU memory.
2. Construct GCN model and load weights for convolution kernels.
3. (GPU only) Offload GCN model weights to GPU. Do not include offload time of model in performance characterization because model parameters are assumed to be pre-loaded on GPU in production settings.
4. Begin profiling GCN using the PyTorch profiling tool and record execution time for key kernels.
5. (GPU only) Offload adjacency matrix and feature matrix from CPU to GPU.
6. Run forward inference of graph through GCN model. See Figure 1 for detailed execution time plot on CPU and Figure 2 for GPU.
7. (GPU only) Offload result of GCN from GPU to CPU.

Figures 1 and 2 show the execution timeline for full-graph GCN on CPU and GPU respectively. Both plots use the *products* graph. CPU executes *products* GCN in ∼6 seconds while GPU executes *products* GCN ∼800 milliseconds. Notice that the offload features and offload adjacency matrix must occur before any GCN kernels can be started. This offload time is unavoidable and is determined by the interconnect bandwidth between the CPU and GPU. This bandwidth is typically in the 10x GBs range whereas GPU memory to GPU die is typically in the 100xGBs-TBs range. Systems with different CPU-GPU bandwidth will see this offload time scale proportionally. A performance characterization not included in this study is the impact of data movement on total energy. Data movement at the system level is expensive; future studies should reveal the impact on energy efficiency of using a GPU versus a CPU.

All the OGB graphs were profiled on CPU for an embedding dimension sweep from 8 to 256. All graphs which fit on the GPU (all except *papers*) where characterized for the same set of embedding dimensions.

2.2 Full-Neighborhood-Sampling GCN Characterization

To address the challenge of graphs which do not fit in GPU memory, neighborhood sampling is required. Two methods for full-neighborhood sampling are investigated; batch-wise and layer-wise sampling.

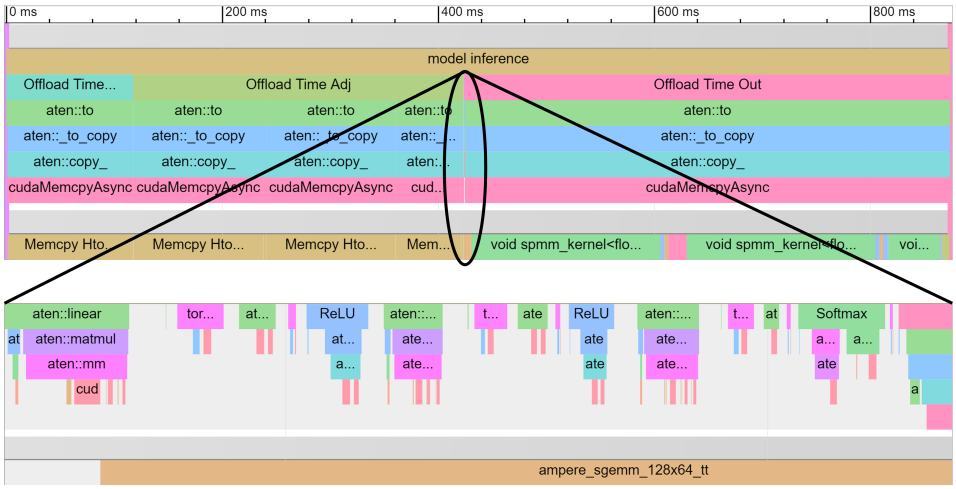

Figure 2: GPU execution timeline for *products* using embedding dimension 256. This trace shows the performance of the host CPU and GPU hardware. In the top figure, the first group of traces are for the CPU while the bottom trace is the GPU. The impact of offload time is almost 50% for this graph when running on GPU. The sparse kernel dominates most of the remaining execution time. The expanded section shows the host offloading the GCN kernels to the GPU. Notice that after the CPU initializes a cuda kernel within the *aten:linear*, the GPU *ampere_sgemm_128x64_tt* kernel starts (bottom row).

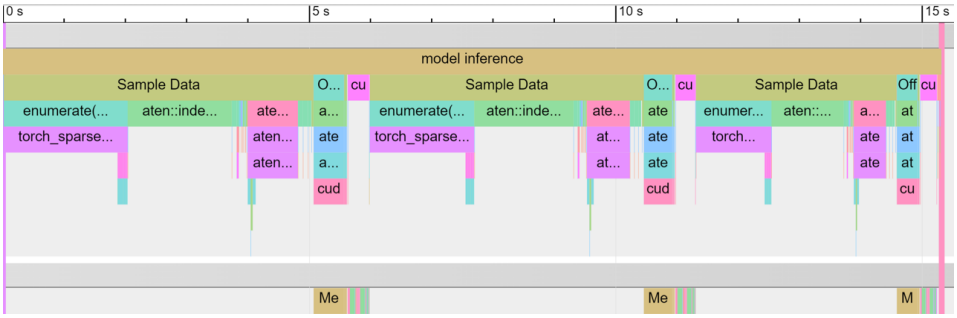

Figure 3: GPU execution timeline for *papers* using batch-wise neighborhood sampling for embedding dimension 256. The first group of traces are for the CPU while the bottom trace is the GPU. In this trace, three mini-batches are shown out of ∼1.7 million. The impact of neighborhood sampling dominates the total execution time (shown as *Sample Data* in the CPU timeline). Offload time is the next highest impact shown in teal immediately following *Sample Data*.

2.2.1 Batch-Wise Neighborhood-Sampling

Figure 3 shows an execution timeline for three batches of *papers* using batch-wise neighborhood sampling. Notice that the GPU has low utilization during each batch. This is because the CPU must first collect the data which feeds into the GPU. The GPU completes its tasks well before the CPU can prepare the next set of data, leading to severe GPU under utilization.

Additionally, to minimize latency, the largest mini-batch size which fit on GPU was required. Experimentally, the largest batch-size per embedding dimension were discovered. Further research can optimize this batch size as the results of this work led to conservative batch sizes.

Finding the largest batch size for a graph is important because it minimizes the data-movement between CPU and GPU. Every batch moves feature vectors from the CPU to the GPU. Only a small percent of those feature vectors are actually updated during each batch, thus the data-movement cost during sampled inference on GPU is incredibly high. The base memory footprint of *papers* with embedding dimension 256 is ∼100GB. The additional data-movement for *papers* is on the order of 100s / 1000s TBs of additional data movement!

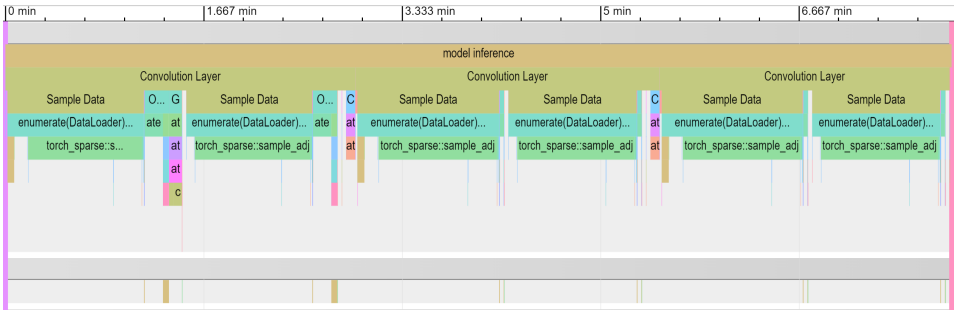

Figure 4: GPU execution timeline for *papers* using layer-wise neighborhood sampling for embedding dimension 256. The first group of traces are for the CPU while the bottom trace is the GPU. In this trace, two mini-batches are shown for each GCN layer, totalling six sampling occurrences. Only two mini-batches out of ∼100 per layer are shown to highlight the separation of each layers computation.

The largest batch size which could run on GPU for *papers* with embedding dimension 256 was a batch size of 64. Using neighborhood sampling, the number of vertices for a three layer GCN expands to 1-4M vertices; meaning 99.99% of embedding vectors were auxiliary. This led to a data-movement cost of ∼4GB per batch and a memory footprint of 30GB - almost reaching full memory-capacity on the A100-40GB GPU when including the model weights and other data structures.

Furthermore, 1.7 million mini-batches (111M total vertices / 64 vertices per mini-batch) were required for *papers* with embedding dimension 256, leading to an additional data-movement cost of ∼**2-6 Petabytes** when using full-neighborhoods.

The data movement requirement can be approximately computed by taking the batch size and multiplying it by the average edges per vertex raised to the number of layers. In the case of *papers* with embedding dimension 256, the number of embedding vectors per mini-batch equals $64 * 30^3$.

Clearly, offloading large graphs to GPU using batch-wise full-neighborhood sampling is intractable. Fixed-neighborhood batch-wise sampling still faces the exploding neighborhood problem, which will likely cause a similar data-movement issue.

2.2.2 Layer-Wise Neighborhood-Sampling

Layer-wise neighborhood sampling is an alternative sampling algorithm to batch-wise sampling which trades off larger memory footprint with faster execution time. Figure 4 shows an execution timeline for layer-wise sampling. Each GCN layer is broken into mini-batches, and each mini-batch requires a sampling step. In this case, there are three GCN layers where only two mini-batches per layer are shown.

The larger memory footprint is because for each layer, the intermediate feature vectors are saved in CPU memory. The change in memory footprint depends on the input-output channel sizes for the GCN.

Layer-wise sampling may have a larger memory footprint than batch-wise sampling because all the intermediate features must be saved on CPU before computing the next layer. For example, if a graph has an input dimension size of 128 and a hidden dimension size of 256, then the CPU memory footprint for the first GCN layer using layer-wise sampling is #-vertices * 128 + #-vertices * 256 whereas with batch-wise sampling, the CPU memory footprint for the first GCN layer was only #-vertices * 128.

The case of *papers* with embedding dimension 256 was examined. The largest minibatch-size for this embedding dimension was 1M vertices, leading to a maximum GPU memory footprint of ∼30 GB and ∼100 total batches. This minibatch-size was much higher than batch-wise sampling because layer-wise only expands the neighborhood of a mini-batch to 1-hop neighborhoods; reducing the impact of the exploding neighborhood problem with batch-wise sampling.

The GPU memory requirement can be derived by taking the GCN layer with the largest convolution dimensions. The largest GCN layer in this example was the hidden layer with input dimension

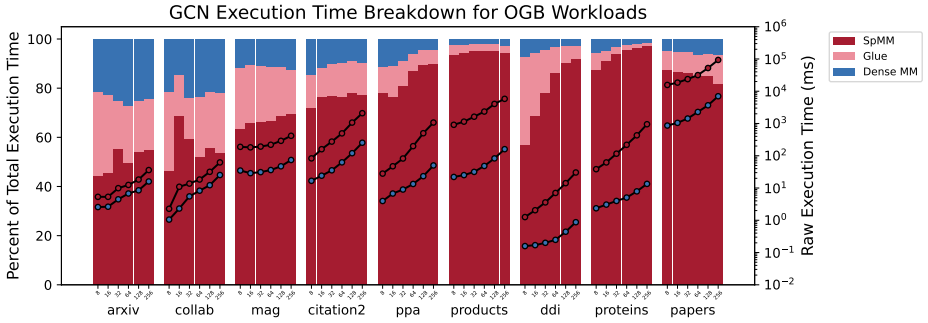

Figure 5: Execution time breakdown for Xeon CPU.

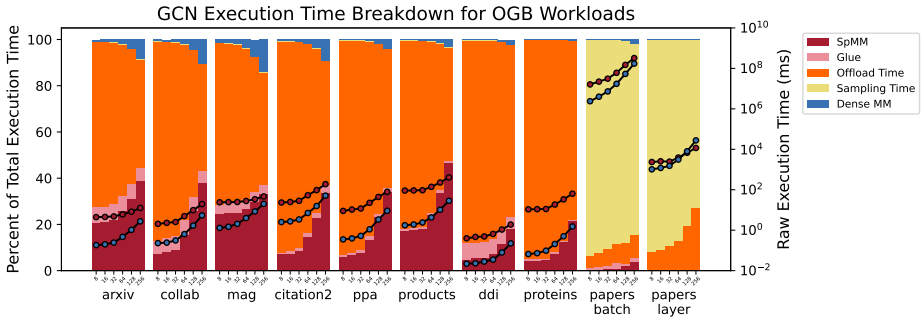

Figure 6: Execution time breakdown for A100 GPU.

256 and output dimension 256. After expanding the 1-hop neighbors for the 1M vertices in a mini-batch, the total number of vertices used in the GCN layer was $\sim$ 15M. Thus, the total memory capacity requirement on GPU was the number of vertices times the (input channel + output channel) times the data size $= 15M * (256 + 256) * 4 =\sim$ 30GB.

The total data-movement can be calculated by multiplying the data-movement on and off the GPU per mini-batch by the number of mini-batches per layer by the number of layers. In the case of *papers* with GCN embedding size 256, the total data movement is $\sim$**4 Terabytes**. This is three orders of magnitude lower than batch-wise neighborhood sampling, but still orders of magnitude higher than desired data movement between CPU and GPU.

3 Results

This section explains the performance results of GCN characterized on CPU and GPU. One focus of this characterization was to breakdown the execution time of GCN into its primary components on each hardware platform; sparse matrix multiplication (SpMM), dense matrix multiplication (Dense MM), Glue Code (activation functions and kernel initialization), offload time, and sampling time.

The results shown in Figure 5 reveal several trends in CPU performance. First, SpMM is the dominant kernel in GCN, which is well understood in literature. A trend was identified that as the embedding dimension increased, the impact of the Glue Code decreased leading to higher dominance of SpMM. For example, in *mag*, as the embedding dimension increased, the impact of SpMM increased while Dense MM stayed almost constant. Graphs *ppa*, *ddi*, and *proteins* have the opposite effect; as the embedding dimension increases, the impact of the dense kernel decreases significantly. Finally, in *papers*, as the embedding dimension increases, the fraction of execution time spent in SpMM decreases.

Next, the execution time breakdown for GCN on GPU is shown in Figure 6. All the OGB graphs except *papers* fit into GPU memory. Therefore, only *papers* required a sampling approach to run using GPU. The first observation was that offload time had a large impact on performance. This is

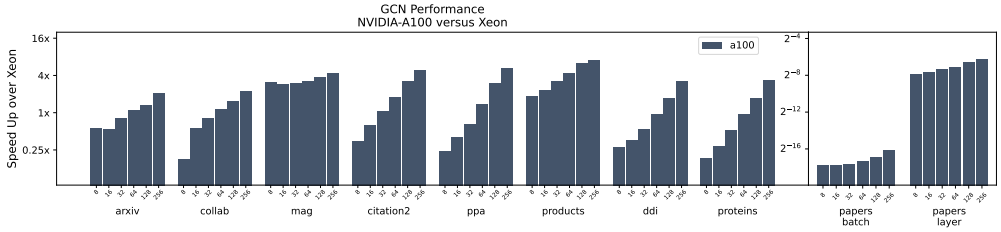

Figure 7: Speedup of A100 GPU versus Xeon for OGB workloads. The first eight graphs fit entirely in GPU memory, while *papers* required sampling, leading to much lower performance than CPU.

well understood in literature and prior works have attempted to reduce long offload times through embedding clustering.

Another observation was that as the embedding dimension increased, the impact of offloading to GPU decreased. Although data-transfer generally increased linearly with embedding dimension, sparse/dense multiplication increased exponentially with embedding dimension. This explains why dense and sparse had higher impact on execution time at higher embedding dimensions compared to offload time.

Another interesting observation was that the workloads which were dominated by SpMM on CPU were not necessarily the most SpMM dominant graphs on GPU. For example, *proteins* was the most SpMM dominant workload on CPU; however, *products* was the most SpMM dominant on GPU.

Finally, the *papers* result shown in Figure 6 revealed the execution time breakdown for sampled GCN on GPU. From this result it is clear that sampling had the largest effect on performance. In fact, sampling and offload time constituted more than 95% of execution time using batch-wise sampling and more than 99% of the execution time using layer-wise sampling.

The execution times for the CPU and GPU experiments are compared in Figure 7. At higher embedding dimensions, full-graph on GPU clearly outperformed CPU; however, at low embedding dimensions, CPU had higher performance. This result is largely explained by the long offload times for GPU. As the amount of compute increased with embedding dimension, the impact of offload time decreased, as shown in Figure 6.

The sampling solution for *papers* performs very poorly in both batch and layer sampling scenarios when compared to the CPU implementation. This was do to the performance bottleneck of CPU based sampling and data movement to GPU.

REFERENCES

[1] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering," Feb. 2017, number: arXiv:1606.09375 arXiv:1606.09375 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1606.09375

[2] Z. Gong, H. Ji, Y. Yao, C. W. Fletcher, C. J. Hughes, and J. Torrellas, "Graphite: optimizing graph neural networks on CPUs through cooperative software-hardware techniques," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*. New York New York: ACM, Jun. 2022, pp. 916–931. [Online]. Available: https://dl.acm.org/doi/10.1145/3470496.3527403

[3] V. Md, S. Misra, G. Ma, R. Mohanty, E. Georganas, A. Heinecke, D. Kalamkar, N. K. Ahmed, and S. Avancha, "DistGNN: scalable distributed training for large-scale graph neural networks," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. St. Louis Missouri: ACM, Nov. 2021, pp. 1–14. [Online]. Available: https://dl.acm.org/doi/10.1145/3458817.3480856

[4] M. K. Rahman, M. H. Sujon, and A. Azad, "FusedMM: A Unified SDDMM-SpMM Kernel for Graph Embedding and Graph Neural Networks," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Portland, OR, USA: IEEE, May 2021, pp. 256–266. [Online]. Available: https://ieeexplore.ieee.org/document/9460486/

[5] G. Huang, G. Dai, Y. Wang, and H. Yang, "GE-SpMM: General-Purpose Sparse Matrix-Matrix Multiplication on GPUs for Graph Neural Networks," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. Atlanta, GA, USA: IEEE, Nov. 2020, pp. 1–12. [Online]. Available: https://ieeexplore.ieee.org/document/9355302/

[6] O. Selvitopi, B. Brock, I. Nisa, A. Tripathy, K. Yelick, and A. Buluç, "Distributed-memory parallel algorithms for sparse times tall-skinny-dense matrix multiplication," in *Proceedings of the ACM International Conference on Supercomputing*. Virtual Event USA: ACM, Jun. 2021, pp. 431–442. [Online]. Available: https://dl.acm.org/doi/10.1145/3447818.3461472

[7] Z. Zhang, J. Leng, L. Ma, Y. Miao, C. Li, and M. Guo, "Architectural Implication of Graph Neural Networks," *IEEE Computer Architecture Letters*, pp. 1–1, 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9075391/

[8] M. Yan, Z. Chen, L. Deng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Characterizing and Understanding GCNs on GPU," *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 22–25, Jan. 2020. [Online]. Available: https://ieeexplore.ieee.org/document/8976117/

[9] T. Baruah, K. Shivdikar, S. Dong, Y. Sun, S. A. Mojumder, K. Jung, J. L. Abellan, Y. Ukidave, A. Joshi, J. Kim, and D. Kaeli, "GNNMark: A Benchmark Suite to Characterize Graph Neural Network Training on GPUs," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Stony Brook, NY, USA: IEEE, Mar. 2021, pp. 13–23. [Online]. Available: https://ieeexplore.ieee.org/document/9408205/

[10] L. Zhang, Z. Lai, Y. Tang, D. Li, F. Liu, and X. Luo, "PCGraph: Accelerating GNN Inference on Large Graphs via Partition Caching," in *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. New York City, NY, USA: IEEE, Sep. 2021, pp. 279–287. [Online]. Available: https://ieeexplore.ieee.org/document/9644829/

[11] Z. Wang, Y. Wang, C. Yuan, R. Gu, and Y. Huang, "Empirical analysis of performance bottlenecks in graph neural network training and inference with GPUs," *Neurocomputing*, vol. 446, pp. 165–191, Jul. 2021. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0925231221003659

[12] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive Representation Learning on Large Graphs," Sep. 2018, arXiv:1706.02216 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1706.02216

[13] J. Dong, D. Zheng, L. F. Yang, and G. Karypis, "Global Neighbor Sampling for Mixed CPU-GPU Training on Giant Graphs," Jun. 2021, arXiv:2106.06150 [cs]. [Online]. Available: http://arxiv.org/abs/2106.06150

[14] M. Ramezani, W. Cong, M. Mahdavi, A. Sivasubramaniam, and M. Kandemir, "Gcn meets gpu: Decoupling "when to sample" from "how to sample"," *Advances in Neural Information Processing Systems*, vol. 33, pp. 18 482–18 492, 2020.

[15] 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," Feb. 2021, number: arXiv:2005.00687 arXiv:2005.00687 [cs, stat]. [Online]. Available: http://arxiv.org/abs/2005.00687

[16] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky, "NVIDIA A100 Tensor Core GPU: Performance and Innovation," *IEEE Micro*, vol. 41, no. 2, pp. 29–35, Mar. 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9361255/