Accelerating Graph Convolutions

A Simple Python Runtime System Built for CS263

Yuke Wang and **Sirius Zhang**

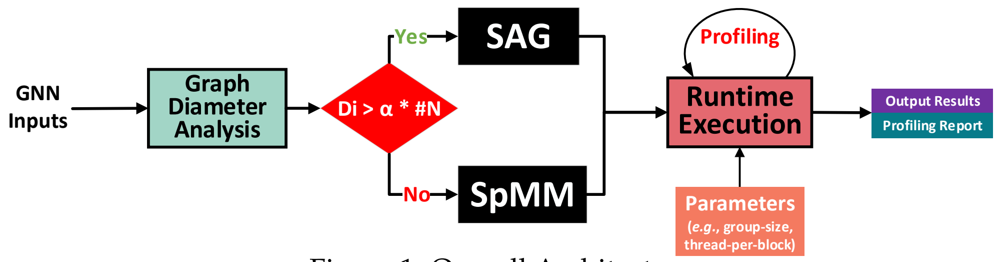


Figure 1: Overall Architecture.

1 Introduction

Inspired by the recent success of 2D convolutional neural network (CNN) [1] on euclidean data (such as image pixels defined on a 2D uniform grid), recently the deep learning research community generates huge interest in a new form of neural networks that can process graph data (such as social networks, graphs that describe protein to protein interaction, etc) directly, called Graph Convolutional Networks (GCN).

In this project, we surveyed how different Python runtime systems/libraries accelerate the research and deployment of ideas coming out from this new research field. In addition, we implemented a simple Python runtime system that can switch between two GPU kernels of our own, specifically, Scatter and Gather (SAG) and Sparse Matrix Multiplication (SpMM) kernels. The kernel switching call is made at runtime based on graph's characteristic. Finally, we profiled the performance of our runtime system on different benchmarks and reported our findings. The overall architecture is illustrated in Figure 1. Implementation details will be discussed in Section 4.

2 Related Work & Survey

From our survey, we noticed that the majority of the runtime systems and machine learning libraries used by researchers and developers in this field use high-level programming language such as Python to boost productivity. A lot of the algorithms from this field are math-heavy. Data are usually stored as Numpy objects. All the math operations and extra features are built around Numpy objects. These systems and libraries use CUDA (when NVIDIA GPU is available) and C++ under the hood for performance critical code. We picked a few representatives behind different approaches and presented them in following subsections.

2.1 Numba

Numba [2] is a just in time (JIT) compiler that works best on python code that uses NumPy arrays, functions, and loops. Numba is not a full-fledged Python runtime compiler. It serves as an add-on runtime compiler for a complete runtime compiler such as CPython. Numba specific python decorators can be used to provide hints to Numba on how to optimize their code at runtime.

2.2 PyCuda

PyCuda [3] uses Python to wrap the complete NVIDIA CUDA API set. It works with Numpy. In addition, it has garbage collector support. Object cleanup is tied to object's lifetime. Both the CPU and GPU memory allocated for a object will be automatically freed when the object is out of scope. PyCuda also wraps every CUDA error using Python exceptions to output more traceable errors. Scikit-Cuda [4] is similar to PyCuda's effort. But it's less developed compared to PyCuda.

2.3 PyTorch

PyTorch [5] is a popular machine learning library based on Torch developed by Facebook's AI Research lab. It supports automatic differentiation for all operations on Tensors. It allows an easy integration of mixed CUDA/C++/Python code. TensorFlow is Google's similar effort to PyTorch.

3 Graph Convolution & Graph Neural Network

Graph convolution can be seen as a relaxation of the standard 2D convolution seen in work such as [1]. Graph convolution aims to do the same thing as the standard 2D convolution but with only one difference: instead of operating on data defined on uniform grid, Graph convolution operates on data defined on arbitrary graph structures.

Graph Neural Networks (GNN) are now a major way for machine learning on graph structures. It generally consists of several graph convolutional layers, each of which consists of a neighbor aggregation and a node update step. It computes the embedding for node v at layer $l+1$ based on its embedding at layer l, where $l \geq 0$.

Under GNN's formulation, each data point serves as a graph node v, and a graph edge defines the relationship between any two connected nodes. One specific example is to defined the edge as the L2 euclidean distance between any two nodes. When all the edges have the same distance and are perpendicular to each other, we have our original 2D convolution defined on uniform grid.

We present a formal definition of GNN following this work from Thomas Kipf and Max Welling [6]. Every neural network layer can be written as a non-linear function

$$H^{l+1} = f(H^{l}, A) \tag{1}$$

with $H^0 = X$ (X for initial graph embedding input), $H^L = Z$ (or z for graph-level outpus), L being the number of layers, and A being the adjacency matrix encodes the graph connectivity. The specific models then differ only in how f is chosen and parameterized.

For example, our SpMM matrix kernal based GCN can be formulated as following

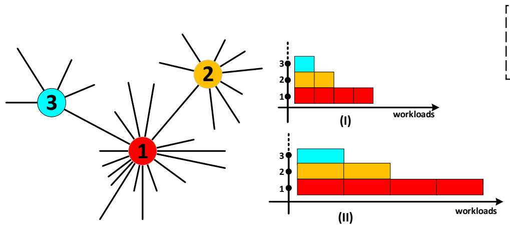
$$f(H^{l}, A) = \sigma(AH^{l}W^{l}) \tag{2}$$

where W^l is a weight matrix for the l-th neural network layer and σ is a non-linear activation function like the ReLU. In the actual implementation, we implement the matrix product AH^lW^l as sparse matrix multiplication instead of the standard dense matrix multiplication.

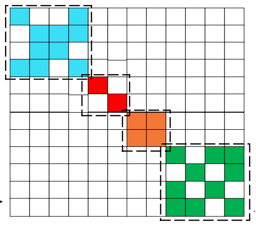
For our Scatter and Gather (SAG) kernel, we have the following discrete formulation of the general Equation 1:

$$\begin{aligned} a_v^{l+1} &= Aggregate^{l+1}(h_u^{l+1} | u \in Neighbor(v)) \\ h_v^{l+1} &= Update^{l+1}(a_v^{l+1}, h_v^{l}) \end{aligned} \tag{3}$$

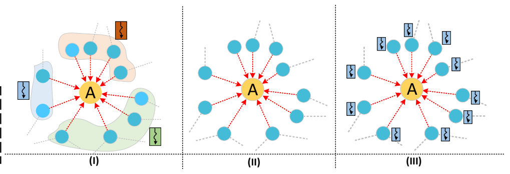
As shown in Equation 3, h_v^l is the embedding vector for node v at layer l. a_v^{l+1} is the aggregation results through collecting neighbors' information (*e.g.*, node embeddings). The aggregation method could vary across different GNNs. Some methods just purely depend on the properties of neighbors while others also leverage the edge properties, such as weights. The update function is generally composed of a single fully connected layer or multi-layer perceptron (MLP) for NN-operations on both aggregated neighbor results and its current embedding at layer l.



(a)



(b)



(c)

Figure 2: (a) Workload Imbalance; (b) Block-Diagonal Graph; (c) Group-based Workload.

4 Implementation Details & Contributions

4.1 Overall Architecture & Runtime Kernel Switching

As shown in Figure 1, our GNN runtime library takes the input of the graph, and then apply graph diameter analysis on the graph structure. Note that such analysis is lightweight and can be completed while loading the graph as input. After that, we make a decision based on the diameter value of the graph. If its value is large than a certain threshold then we can take the Scatter-And-Gather (SAG) kernel, otherwise, we take the SpMM kernel.

What also worth mentioning is that α is a hyperparameter for setting threshold (in general we set $\alpha = 0.1$ that can handle diverse graphs) and $\#N$ is the number of nodes of a input graph. We also justify this statement via the experiments results in Section 5. After then, we put the selected kernel on GPU for computation with the default or manually-defined parameters (*e.g.*, group-size). Meanwhile, our profiling tools will collect runtime profiling results. Finally, we generate the output running results and profiling report in human-readable .csv format.

4.2 Scatter and Gather (SAG)

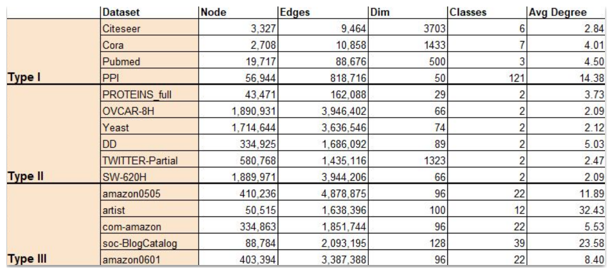
Scatter and Gather (SAG) kernel is originated from graph processing, such as, PageRank, BFS. Its major work is to propagate the information among nodes. Specifically, there are two types of SAG operations depending on their direction, *Push* and *Pull*. **Push-based SAG** updates the neighbor nodes based on the center node. For Push-based SAG, we would consider data write conflicts under the parallel setting since each thread works on a center node. While the **Pull-based SAG**, updates the center nodes by ”drag” the information from its neighbors. For Pull-based SAG, we would concern about the data irregular read access since neighbors are stored in non-continuous memory space in general. In our implementation, based on our initial study, applying the Pull operation during the GCN computation can effectively reduce the atomic operations that are major performance bottleneck to improve the overall performance. We incorporate SAG in our design to demonstrate its benefits on some input graphs with high irregularity in their edge connections.

4.3 Sparse Matrix-Matrix Multiplication (SpMM)

We implemented the Equation 2 based on SpMM and demonstrate its performance benefits on the input graphs with block-diagonal property (Figure 2b) in their adjacent matrix representations. We also figured out **WHEN** should we use them in place of the SAG kernel for seeking better performance by a simple yet low-cost graph diameter analysis.

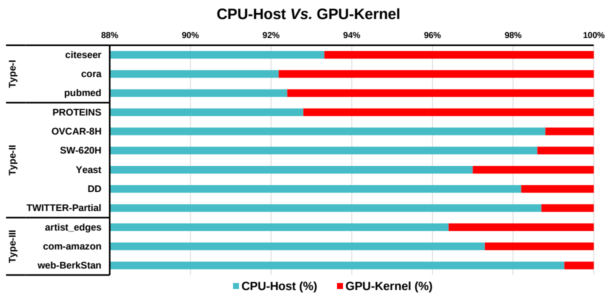
4.4 Group-based Workload

Directly applying SAG operations during the GCN aggregation step would lead to sub-optimal performance due to the excessive amount of inter-thread synchronization overhead and workload imbalance resulted from various node degrees. Note that such a workload imbalance would even be exacerbated in multi-layered GCN (Figure 2a-II) because of the high node embedding dimension compared with the setting in the traditional graph processing (Figure 2a-I).



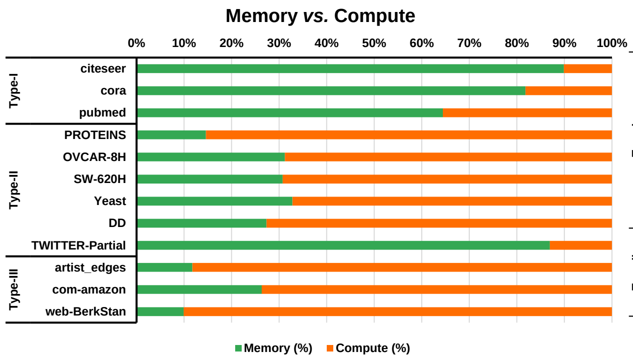
	Dataset	Node	Edges	Dim	Classes	Avg Degree
	Citeseer	3,327	9,464	3703	6	2.84
	Cora	2,708	10,858	1433	7	4.01
	Pubmed	19,717	88,676	500	3	4.50
Type I	PPI	56,944	818,716	50	121	14.38
	PROTEINS_full	43,471	162,088	29	2	3.73
	OVCAR-8H	1,890,931	3,946,402	66	2	2.09
	Yeast	1,714,644	3,636,546	74	2	2.12
	DD	334,925	1,686,092	89	2	5.03
	TWITTER-Partial	580,768	1,435,116	1323	2	2.47
Type II	SW-620H	1,889,971	3,944,206	66	2	2.09
	amazon0505	410,236	4,878,875	96	22	11.89
	artist	50,515	1,638,396	100	12	32.43
	com-amazon	334,863	1,851,744	96	22	5.53
	soc-BlogCatalog	88,784	2,093,195	128	39	23.58
Type III	amazon0601	403,394	3,387,388	96	22	8.40

(a)

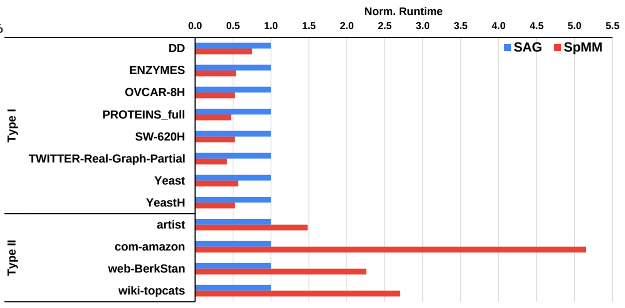


(b)

Figure 3: (a) Datasets, and (b) CPU Host Vs. GPU SAG Kernel.



(a) SAG (Memory Vs. Compute).



(b) Kernel Performance.

Figure 4: Runtime Breakdown and SAG Kernel Performance.

To improve SAG's performance, we introduced Group-based workload partitioning as a load balancing technique. It breaks down the neighbors of a node into different groups and assigns the group-based neighbor aggregation workload (first step in Equation 3) to each thread. Specifically, it relies on the input-level information – node degree and node embedding – to determine the group-based aggregation workload assigned to each thread. As exemplified in Figure 2c-I, the neighbors of a node are divided into 3 groups with a pre-determined group size of 4.

4.5 Python Wrapper

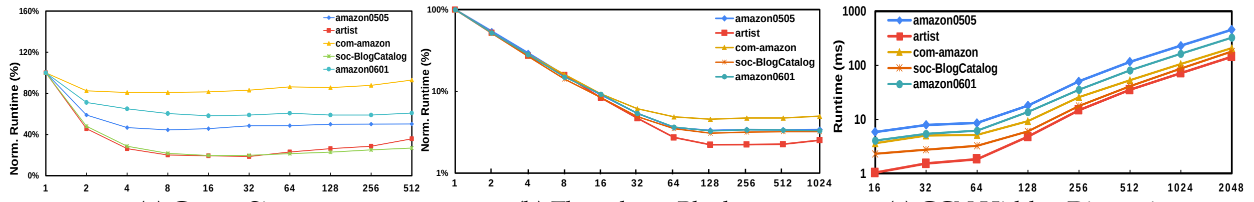
For the actual implementation, we followed PyTorch's design for an easy integration of mixed CUDA, C++, and Python code. **In the Python layer**, we define the interface to our kernel and configure the path to statically compiled source code. **In the C++ layer**, the C++ code mirrors the interface from the python layer, and articulates what each python variable's corresponding c++ type is. The C++ layer is also responsible for launching the CUDA kernel dynamically at runtime. The major lifting code is written **in CUDA**, taking advantage of parallel computing hardware for best performance. This design allows us to implement light weight graph diameter based kernel switching optimization in the Python layer while loading the input graph without sacrificing too much performance. Please refer to our source code [1] for more details.

5 Evaluation

5.1 Runtime Breakdown

As shown in Figure 3b, we analyze the cost of our runtime system at both the CPU host and the GPU sides. We further decompose the GPU kernels' running time cost into its memory manage-

[1] https://github.com/YukeWang96/CS263-project.git



(a) Group-Size (b) Thread-per-Block (c) GCN Hidden Dimension.

Figure 5: Case Studies.

ment time cost and its computation time cost, both measured in unit time.

We can see that the CPU host takes the majority of running time overall, and the main reason behind such a high CPU overhead is that our python front-end on the CPU side is not optimized with multi-thread parallelization, thus, suffering from inferior performance. And such CPU-side performance bottleneck will be exacerbated when the input graphs are at scale. In contrast, our GPU kernel is highly optimized with our SAG kernel with group-based optimization for balancing single-thread efficiency and multi-thread parallelism. Therefore, it can maximize the performance gain through efficient GPU execution.

In the GPU kernel running time cost decomposition, as shown in Figure 4a, the running time of memory operation will dominate the overall kernel runtime cost as the node embedding dimension increases. Because, in this case, more memory access and data movements are required to handle high dimensional node embeddings.

As shown in Figure 4b, we can justify our early statement in Section 4. For Type II graphs, which have much better regularity in their edge connections, SpMM kernel performed better than SAG kernel due to the good data spatial and temporal locality. For Type III graphs, however, the benefits of SpMM disappeared due to the fact that edge connections of graphs are highly irregular without any block-diagonal pattern. Thus, in this case, the SAG kernel demonstrated its strengths in handling such irregular cases. These results also showed the importance of applying appropriate kernels for seeking better performance for GCN on GPUs.

5.2 Case Studies for SAG Kernel

Group-size: As shown in Figure 5a, we can see that with the increase of the group size, the running time will decrease before the performance benefit saturates. Since such an increase will fully exploit the compute capability of each thread, and meanwhile enjoy the benefit of data locality and atomic operation reduction (*i.e.*, inter-thread synchronization overhead).

Thread-per-block: As shown in Figure 5b, the performance impact of thread-per-block parameter follows a similar pattern of the group size effect described above. Increasing the thread-per-block would first improve the overall performance, and then compromise the performance when it crosses over a certain threshold that is conditioned on different input-level properties (*e.g.*, node degrees, and node embedding size).

Hidden Dimensions: As shown in Figure 5c, we observe that with the increase of hidden dimensionality of GCN, the running time is also increased due to more computation cost (*e.g.*, additions), more memory operations (*e.g.*, data movements) involved during the aggregation phase, and a lot more node embedding updates during the update phase for a larger node embedding matrix.

6 Conclusion

We surveyed different runtime systems, machine learning libraries, and python wrappers options that are popular among data scientists and machine learning researchers. We also built our own GNN runtime library with Scatter and Gather (SAG) and Sparse matrix multiplication (SpMM) kernels. We implemented our system follows PyTorch's design of mixed CUDA/C++/Python code. The entire project along with its profiling code can be found on our github repo.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. Red Hook, NY, USA: Curran Associates Inc., 2012.

[2] “Numba.” [Online]. Available: numba.pydata.org/numba-doc/latest/user/5minguide.html

[3] “Pycuda.” [Online]. Available: documen.tician.de/pycuda/

[4] “scikit-cuda.” [Online]. Available: scikit-cuda.readthedocs.io/en/latest/

[5] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems (NIPS) 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d Alch’e-Buc, E. Fox, and R. Garnett, Eds., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[6] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” *arXiv preprint arXiv:1609.02907*, 2016.