Sparse Heterogenous Graph Transformers

Accelerating Segmented Matrix Multiplication

Kanv Khare
Computer Science
North Carolina State University
Raleigh, North Carolina, USA
kkhare@ncsu.edu

Kevin Schwarz
Computer Science
North Carolina State University
Raleigh, North Carolina, USA
kmschwa3@ncsu.edu

ABSTRACT

A graph neural network (GNN) is a type of neural network wherein graphs are used to represent data. Heterogenous GNNs are a special category of GNNs where each node and edge can have its own data type. To work with heterogenous GNNs in a scalable and efficient manner, sparse heterogenous graph transformers are used.

A heterogenous graph transformer allows for the model to work with the different types of data in the graph and understand complex relationships which might be otherwise challenging to work with. Using a heterogenous graph transformer, heterogenous GNNs can perform a variety of tasks including edge prediction, node prediction, node clustering, classifying nodes and edges, classifying entire graphs, or even generating similar graphs.

To perform all of these operations quickly, heterogenous graph transformers rely on certain other operations being implemented efficiently. One of these operations is segment matrix multiplication or segment_matmul. In this paper, we implement an efficient version of segment_matmul using Triton. Triton is an open-source python-like GPU focused programming language designed specifically for neural networks that was developed by OpenAI.

CCS CONCEPTS

• Parallel architectures • Performance • Neural networks

KEYWORDS

Graph Neural Networks, Parallel architecture, Artificial Intelligence, Optimization

ACM Reference format:

Kanv Khare, Kevin Schwarz. 2023. Sparse Heterogenous Graph Transformers: Accelerating Segmented Matrix Multiplication. Parallel Systems *(CSC 548 Spring 2023). North Carolina State University, Raleigh, NC, USA, 5 pages.*

1 Graph Neutral Network

A Graph Neural Network (GNN) is a type of neural network that uses graphs to represent data. Since nodes and edges can easily be added to graphs, GNNs are flexible in their structure and allow for more data to be added easily without having to worry about constraints that exist in many other representations. A traditional homogenous GNN uses two data representations – one for the nodes, and one for the edges. GNNs can be used to perform many tasks including edge prediction, node prediction, node clustering, edge classification, node classification, graph classification, and creating similar graphs.

GNNs work by nodes passing messages to their neighbors. Nodes then update their features based on the features of their neighbors. In this manner, changes propagate across the entire graph. [1]

Heterogenous GNNs are a type of GNN that allows for each node and edge to have its own representation. Figure 1 shows an example of a heterogenous GNN. In this example, the Institution, Author, Paper, and Field of Study nodes all have different representations. The has topic, writes, cites, and affiliated with edges all also have different representations. An author has a writes relationship with a paper but an affiliated with relationship with an Institution. As can be seen, this lets real world systems like social networks, computer networks, or drug reactions to be represented a lot more accurately than a traditional homogenous GNN.

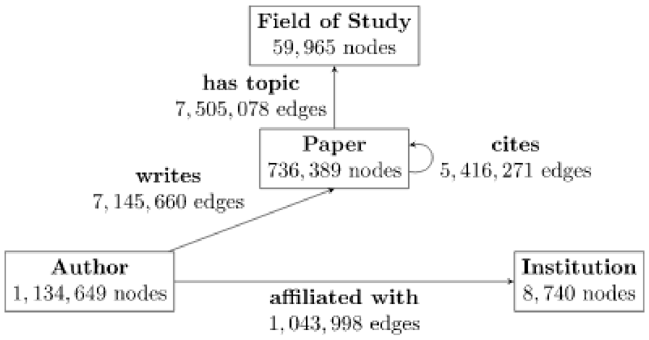

Figure 1: An example heterogenous graph neural network

In order for GNNs to work with heterogenous graphs, heterogenous graph transformers (HGT) are used. HGTs allow for heterogenous GNNs to be scalable and efficient. HGTs rely on several operations to function and require efficient implementations of these operations to themselves be efficient.

2 Segment Matrix Multiplication

One operation that is widely used with heterogenous GNNs is segment matrix multiplication or segment_matmul. This operation is a variation of matrix matrix multiplication that takes three parameters – input, ptr, and other. These three parameters are a 2-D matrix, a list, and a 3-D matrix respectively. It returns a 3-D matrix.

Segment_matmul works by splitting up input into n – 1 matrices, where n is the number of elements in ptr. The matrix is split up based on row indices with ptr[0] and ptr[1] representing the first split up matrix, ptr[1] and ptr[2] representing the second split up matrix, and so on. The split is done such that the left index is inclusive and the right one is not. The split up matrices are then multiplied with other, which can be thought of as a list of 2-D matrices. The first split matrix is multiplied with the first matrix from other and so on. Figure 2 shows an example of how segmented matrix multiplication works. [3]

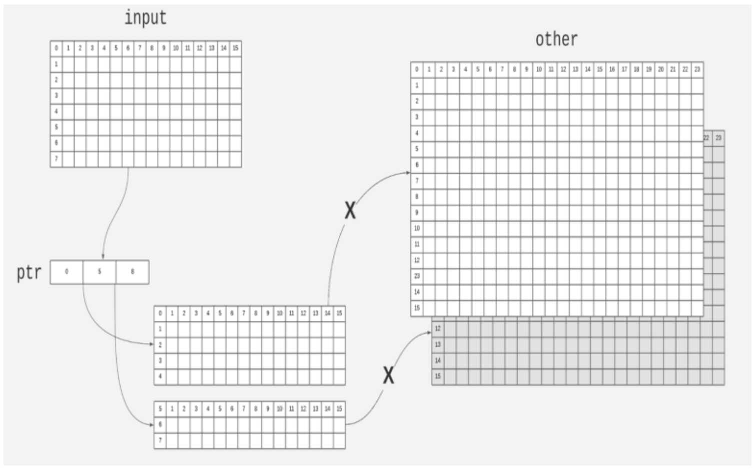

Figure 2: Segment Matrix Multiplication

2.1 Grouped Matrix Multiplication

Segment_matmul is an extension of matrix multiplication or matmul. It can be thought of as a series of matmul operations. These can be trivially parallelized since each matmul is its own operation. However, we can go even further by optimizing matmul. We achieved this by using grouped matmul instead of traditional matmul.

With traditional matmul, row-major order is generally used (with column-major ordering sometimes being used, but generally avoided because it has poorer spatial locality). As can be seen in Figure 3, when multiplying two 9x9 matrices, both row-major and column-major ordering load 9 elements from one of the matrices and 81 elements from the other matrix for a total of 90 elements to write 9 elements to the output matrix.

When using segment_matmul, grouped ordering is used instead of row-major or column-major ordering. As Figure 3 shows, when multiplying two 9x9 matrices, 3 rows are loaded from one matrix and 3 columns from the other for a total of 54 elements to write 9 elements to the output matrix. Segment_matmul loads fewer elements to write the same number of elements to the output and encourages data reuse with increased temporal locality. The L2 cache hit rate is also improved when data is loaded into SRAM. [4]

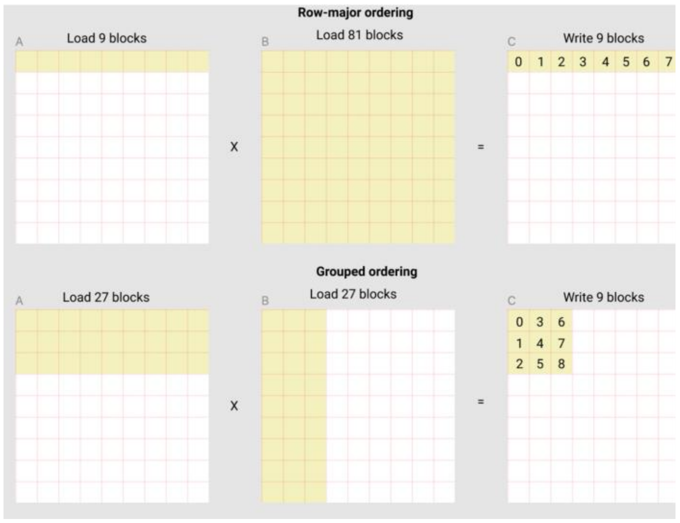

Figure 3: Grouped Matrix Multiplication

3 Implementation

For our implementation of segment_matmul, the program is split into multiple threads that each calculate a [BLOCK_SIZE_M, BLOCK_SIZE_N] grid of values in the output matrix. To do this, each thread will load a [BLOCK_SIZE_M, BLOCK_SIZE_K] grid from the first ('input') matrix, and a [BLOCK_SIZE_K, BLOCK_SIZE_N] grid from the second ('other') matrix. Since we want to cover the entire rows (BLOCK_SIZE_M) and columns (BLOCK_SIZE_N), we continue iterating through the rows and columns in BLOCK_SIZE_K intervals, multiplying the grids/matrices and accumulating them. Once fully accumulated, the resulting grid is stored into the assigned [BLOCK_SIZE_M, BLOCK_SIZE_N] grid in the output matrix. However, the problem here is that there are multiple 'other' matrices that we need to multiply, depending on which grid in the output that we need to calculate. To handle this, every program checks what row their block begins in. If it is before the split point denoted by the ptr array, then it loads from the first matrix in the 'other' tensor. If it is past it, then it instead loads the second matrix in the 'other' tensor. It then multiplies it by the first ('input') array.[2]

4 Performance

We compared our implementation of segment_matmul against PyG's (Pytorch Geometric) implementation. For our benchmarks, we used an NVIDIA Tesla T4 GPU. We used randomly generated square matrices ranging from 256 to 4992 in dimension, going up in increments of 128. We did 500 iterations and ensured that for a given iteration, both implementations were given the same matrix.

Figure 4 shows a performance comparison for segment_matmul between our implementation (Triton) and PyG's implementation. The x-axis represents the size of the matrix, and the y-axis represents the number of floating point operations per second (TFLOPS). The dotted line represents PyG's performance while the solid line represents our implementation using Triton. The gradient shows the minimum and maximum times across executions. As can be seen by the graph, the Triton implementation performs better than the PyG implementation in most cases, with the performance difference increasing as the input size increases.

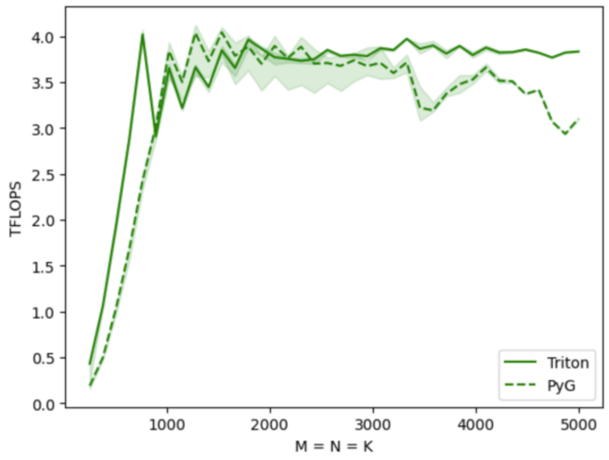

Figure 4: Triton vs PyG Segment Matrix Multiplication Benchmark

5 Limitations and Future Research

Our implementation of segment_matmul currently has some limitations. These are not assumptions we make but rather edge cases that we can incorporate in future research. For our current implementation, the size of ptr must be 3 with the second element in ptr being a multiple of BLOCK_SIZE_M. Additionally, K (y-dimension) must be a multiple of BLOCK_SIZE_K.

ACKNOWLEDGMENTS

We would like to thank our CSC 548 professor, Dr. Li (North Carolina State University), for introducing us to this project and looking over our project proposal. We would also like to thank Dr. Zhou (OpenAI) for guiding us through this project and answering all our questions.

REFERENCES

[1] Benjamin Sanchez-Lengeling, Emily Reif, Adam Pearce, and Alex Wiltschko. 2021. A gentle introduction to graph neural networks. Distill 6, 8 (2021). DOI:http://dx.doi.org/10.23915/distill.00033

[2] Philippe Tillet, H.T.Kung ,and David Cox. 2019. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL '19), June 22, 2019, Phoenix,AZ, USA. ACM, New York, NY, USA, 10 pages.https://doi.org/10.1145/3315508.3329973

[3] PyG Developers. Pyg Documentation. Retrieved April 19, 2023 from https://pytorchgeometric.readthedocs.io/en/latest/

[4] Triton Developers. Welcome to Triton's documentation! Retrieved April 19, 2023 from https://tritonlang.org/main/index.html