fast-training of sparse gnn on dense hardware

Fast Training of Sparse Graph Neural Networks on Dense Hardware

paper: [https://arxiv.org/pdf/1906.11786.pdf]

code: []

Important Concepts:

- TPU: Tensor Processing Units
- DSA: Domain Specific Architectures
- primitive of virtual machine: appropriately-sized matrix multiplication
- B (bandwidth): the bandwidth of a square matrix $A = \{a_{ij}\}_{N \times N}$ is the smallest $B \in \mathbb{N}_0$ such that $a_{ij} = 0$ when ever $|i - j| > B$
- GGNN: gated GNN $E^{(t+1)} = GRU(\sum_{p=1}^{P} A_p E^t W_p, E^{(t)})$
    - edges are of P discrete types
    - W_p: weights that map from embeddings to message of each edge type
- sparse batching & supergraph: multiple training graphs can be packed into a single supergraph of a fixed maximum size, until no more graphs fit
- graph layout problem: Finding an ordering of graph nodes to optimize an objective
- einsum: [https://docs.scipy.org/doc/numpy/reference/generated/numpy.einsum.html] and [https://stackoverflow.com/questions/26089893/understanding-numpys-einsum]
    - The great thing about `einsum` however, is that is does not build a temporary array of products first; it just sums the products as it goes. This can lead to big savings in memory use.
    - Imagine that we have two multi-dimensional arrays, `A` and `B`. Now let's suppose we want to...
        - *multiply* `A` with `B` in a particular way to create new array of products; and then maybe
        - *sum* this new array along particular axes; and then maybe
        - *transpose* the axes of the new array in a particular order
    - There's a good chance that `einsum` will help us do this faster and more memory-efficiently that combinations of the NumPy functions like `multiply`, `sum` and `transpose` will allow.
- BMM (batch matrix multiplication): Given two tensors of compatible shapes $[d_1, \ldots, d_{n-2}, m, k]$ and $[d_1, \ldots, d_{n-2}, k, n]$, it performs the $d_1 \cdots d_{n-2}$ matrix multiplications in the last two dimensions, yielding a tensor of shape $[d_1, \ldots, d_{n-2}, m, n]$

- RCMK algorithm (Reverse Cuthill Mckee Algorithm): [https://www.geeksforgeeks.org/reverse-cuthill-mckee-algorithm/]

Key Idea:

- low bandwidth structure, express GNN propagation in terms of application of dense matrix multiply primitive

- permute nodes to expose low bandwidth structure

- training pipeline:

    1. compilation (RCMK algorithm): finding a permutation of nodes in each training graph such that the resulting adjacency matrix has low bandwidth

    2. input pipeline: efficient reading of compiled training data from disk, assembly of supergraph via sparse batching

    3. model: perform message passing logic for low-bandwidth graphs by batch matrix multiplication

        1. message passing by batch matrix multiplications
        2. efficient implementation with a memory layout to avoid tensor transpositions

```pseudocode
/** ------------------------ **/
/** Algorithm1: Low-bandwidth **/
/** ------------------------ **/
input: tensors of the following shapes
    * Cks [K,     S*P, S]  /** adjacents **/
    * Uks [K - 1, S*P, S]  /** adjacents **/
    * Lks [K - 1, S*P, S]  /** adjacents **/
    * Eks [K, S, H]        /** embeddings **/
    * Wps [P*H, H]         /** edge weights **/
output: new_node_embeddings [K, S, H]
1: pre_messages        = einsum('kzs,ksh->kzh', Cks, Eks)
2: pre_messages[0:k-1]+= einsum('kzs,ksh->kzh', Uks, Eks[1:K])
3: pre_messages[1:K]  += einsum('kzs,ksh->kzh', Lks, Eks[0:K - 1])
4: pre_messages        = reshape(pre_messages, [K, S, P*H])
5: incoming_messages   = einsum('ksy,yh->ksh', pre_messages, Wps)
6: new_node_embeddings = GRU_cell(incoming_messages, Eks)
/** ------------------------ **/
```

Problem definition:

1. Scaling up sparse graph neural networks using a platform targeted at dense computation on fixed-size data
2. sparse GNN models <-- suitability --> hardware for speeding up
3. GNN propagation on sparse graph -- compiling (only use primitives) --> dense hardware

Precessed Work:

https://arxiv.org/pdf/1711.00740.pdf

Metrix gain:

datasets: code graphhttps://arxiv.org/pdf/1711.00740.pdf

- GNN propagation: O(N^2H) --> O(NBH)
- program graphs often exhibit low-bandwidth structure
- fast processing of training data
- same validation accuracy

Challenge:

- how about other sparse graphs
- add more runtime primitives
- design graph structures that encode relevant domain knowledge while respecting constraints that could lead to even faster computation