Learning to Reorder Operations in Sparse Neural Networks

CS 3543 Final Project

Alok Kamatar

February 2023

1 Introduction

State of the art neural network models now consist of billions of parameters and require trillions of operations to perform inference, putting increased strain on the hardware performing inference and training. As these networks have grown, sparsification of DNNs is an activate area of research to reduce parameter count and accelerate execution. However, these sparse networks present new computational challenges in terms of lower computational intensity and irregular memory accesses [12]. Specifically, irregular memory accesses produce cache misses, triggering costly data movement. This data movement can cost hundreds of times the number of cycles as a floating point operation, and ends up being the most costly part of many machine learning applications.

Recently, [5] showed that a layer by layer inference algorithm introduced unnecessary IO-cost. They presented the edge ordering problem for sparse neural network inference, where the edge order determines the order of computation. To solve this problem, they gave a simulated annealing algorithm to create a more efficient inference algorithm. However, this algorithm discards the graph structure when reordering the edges.

In this work we tackle the problem of ordering the edges in the graph of a sparse neural network using the graph structure. First, we draw a connection from the ordering of the edges in a DNN to the problem of reordering vertices for data locality in graph processing algorithms. This inspires the design of the EGO algorithm. EGO uses the structure of the graph to prioritize the edges to the next edge to compute. However, this algorithm uses a greedy heuristic which may be sub-optimal. To overcome the limitations of a greedy algorithm, we also propose a learning based approach to the same problem. We formulate the edge ordering problem as a MDP and experiment with a deep reinforcement learning approach to explore the ordering space. Experiments are shown on randomly sparse neural networks as well as networks from the Amazon/MIT Graph Challenge [10]. We find that the EGO algorithm can significantly improve inference time and reduce cache misses in sparse networks. However, we also find scalability issues in both algorithms.

The rest of this paper is laid out as follows. Section 2 gives the context of this project by discussing the landscape of related work. Section 3 presents the two proposed methods as well as the intuition behind their design. Section 4 discusses the evaluation methodology and results. Section 5 discusses the limitations of this project as well as presents interesting directions for future work. Finally, Section 6 concludes the paper.

2 Related Work

2.1 Machine Learning Compilers

The rise of deep learning has necessitated the development of necessitated domain specific compilers that take high level model definitions and generate efficient machine code [13]. Existing deep learning compilers leverage DAG based intermediate representations for efficient code generation. For example, in TVM, GLOW, Intel nGraph, and Google XLA, nodes represent atomic deep learning operations such as matrix multiplication convolution, and edges represent tensors. After constructing the computation graph, DL compilers use graph transformations to perform optimization. A variety of transformations happen at the node-level, neighborhood (block) level and global level. Node-level optimizations include eliminating no-op nodes (i.e. summations over dimension 1) and node replacement (replacing a high-cost node with a lower cost alternative. Block level optimizations are of more interest. Such optimizations consider a sequence of nodes, then use commutativity and associativity of deep learning operations to reorder, fuse, and eliminate nodes. Among the problems addressed at this stage is the optimization of computation order. Traditionally, this is implemented by in finding algebraic patterns in primitive operations (for example transforming $A^T B^T$ to $(AB)^T$). Graph level optimizations include static memory planning and layout transformations. Static memory planning performs optimizations to maximize the reuse of buffers. [1] performs offline memory scheduling of irregular neural networks using dynamic programming to minimize the total memory footprint. Similarly, layout transformations finds the best data layout for a sequence of tensors to maximize efficiency.

The combination and application of these transformations creates many correlated optimization problems that the ML compiler needs to solve. Traditionally compilers rely on heuristics to solve these problems, leading to sub-optimal solutions. [17] uses deep reinforcement learning to direct a genetic algorithm that searches over compiler optimizations. Similarly, [23] uses a graph attention and demonstrates the ability to generalize to a wider range of computational graphs. [11] goes a step further to show the potential of GNNs to optimize scheduling of generic computation graphs.

While optimizing the computation graph is a common pattern across machine learning compilers those works do not overlap with this. The ML compilers surveyed here operate on a graph of coarse grained operations, such as matrix transposes or multiplies. However, this assumes that efficient implementations

of those operations exist. For sparse matrices this is often not the case. Furthermore, by only considering those operations, they miss the potential for fine grained optimizations.

2.2 Sparse Neural Networks

As deep neural networks grow larger in size and put increased strain on the hardware needed to implement them, much research over the past decade has focused on sparsification [10]. Sparsity has proven an effective approach to save parameters and resources [4]. However, these sparse networks are in general harder to optimize due to lower computational intensity and irregular memory accesses [12]. To encourage the community to pursue research into this area, the MIT/Amazon HPEC Graph Challenge introduced the Sparse Deep Neural Network Challenge [10]. The challenge provides the weights and inputs for sparse neural networks that range from 1024 to 65536 neurons wide and 120 to 1920 neurons deep. The 2020 winner of that challenge decomposes sparse inference into a task graph of highly optimized kernels and then efficiently partitions those tasks among GPUs to maximize data and model parallelism [14]. In this work we leverage the data and the networks in that challenge, however, we focus only on inference on CPUs.

In another line of work investigates generating sparse structures that are easier to optimize [15]. Fine-grained sparsity is the removal of individual weights, and has proved successful in a variety of contexts and architectures [6]. This type of sparsity can be easily generated through techniques such as magnitude pruning, but is also the most challenging to accelerate because of the irregularity of memory accesses. Vector level and kernel level sparsity, which reduce the number of neurons or the number of layers respectively, better preserve the regularity of memory accesses, but typically come at the expense of lower accuracy [15]. In this work we focus only on fine grained sparsity, the most challenging and most general case.

The work in [5] most directly inspires our work here. That work introduces the edge ordering problem for sparse neural networks. They come up with theoretical bounds for IO-complexity of sparse neural networks, and use a genetic algorithm to optimize the order of computation to be more IO-efficient. Their reordering algorithm does not use any structural information from the graph. By leveraging the structural information, we believe that our approach can use common patterns across a network to find a better ordering. In addition, they do not provide the code to reproduce the algorithm.

2.3 Vertex Reordering

Optimizing the computation graph of a sparse neural network is a specific instance of the broader class of work on graph ordering. Graph ordering attempts to find a "good" ordering of the vertices in memory, or one that enhances the performance of traditional graph processing algorithms [21]. For instance, [2] computes a compression friendly ordering of the vertices π by minimizing the

distance in the permutation between neighbors. Similarly, [3] computes an ordering of vertices that improves the performance of nearest neighbor search. [21] proves that the graph ordering problem is NP-hard in general and gives a bounded approximation algorithm. The algorithm iteratively selects the next node in the permutation by maximizing the number of common neighbors between the current node and a sliding window of nodes previously added to the permutation. As we adapt this algorithm for our application, we present more details in Section 3. [22] developed the Deep Order Network (DON) which uses a deep reinforcement learning approach that outperforms previous approximation and heuristic methods. DON learns a function $Q(S, v)$, a scoring function from a partial ordering of the vertices S and a new vertex v to replace the common-neighbor heuristic in [21]. These graph reordering approaches for graph processing are driven by cache and memory concerns. Nodes that are processed or accessed close together should live on the same page and/or cache line in memory. This is similar to the concern of data-locality in sparse neural networks, where the goal is to find the next computation that minimizes the need to transfer data. However, the vertex ordering problem does not have to respect the constraints imposed by the topological ordering of the DAG.

3 Methods

3.1 Problem Formulation

We begin with a traditional feed forward neural network (FFNN). Each layer can be described as $\sigma(x_i^T W^{(i)})$ where x_i are the feature inputs, $W^{(i)}$ are the learnt weights, and σ is some non-linear activation function (i.e $ReLU$, tanh, etc.). A sparse neural network is a network with a low proportion of connectivity compared to all possible connections - that is a high number of zero entries in each matrix $W^{(i)}$. In this project, we only focus on performing inference after the sparsity has been fixed.

A sparse network admits a representation as a directed acyclic graph. Each input element and intermediate neuron translate to vertices, and each non-zero element in a matrix $W^{(i)}$ translates to an edge in the graph as shown in figure 1 [19]. A compute efficient inference algorithm traverses this graph in topological order and performs a multiply and accumulate on each edge. For this work we adopt the general inference algorithm given in [5] which is shown in Algorithm 1. This inference algorithm is competitive with a more traditional algorithm based on layer-wise sparse matrix multiplication, which we will illustrate in section 4. The algorithm operates on a topological ordering of the edges $e_1, e_2 ... e_M$. This order directly impacts the memory performance and therefore the speed of inference. When the edge order displays temporal and spatial locality, values can be kept in the cache to minimize data movement. Transformations on dense matrix-matrix multiplication like loop reordering and tiling are designed to maximize locality to take advantage of this fact. However, traditional layer by layer inference in sparse networks does not exhibit the same pattern [5] The goal

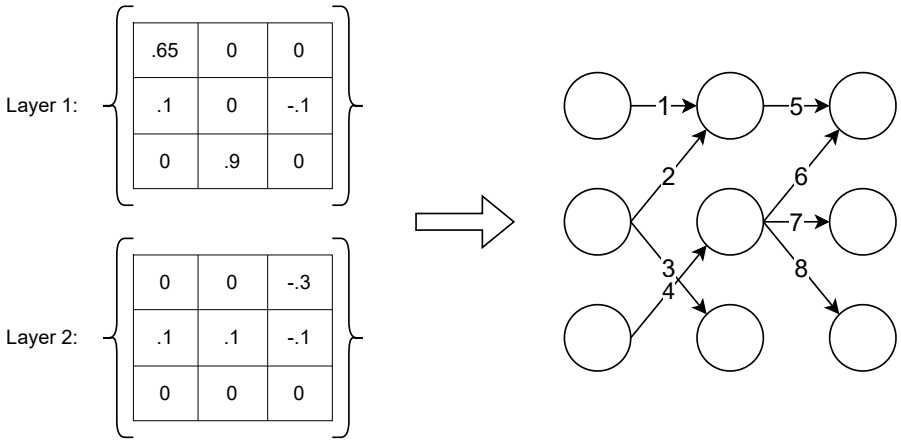


Figure 1: Example of the graph representation of a sparse matrix. Non-zero entries from the matrix become edges in the graph. The edges in the graph are then ordered in a way that is consistent with the topological order of the nodes. The ordering in the figure corresponds to the default layer-wise ordering.

of this work is then to find a permutation ϕ of the edges that exhibits a higher degree of locality, which we expect to be correlated with better performance.

Algorithm 1 Inference Algorithm

```
1: procedure INFER
2:     Let N be the array of neurons
3:     for each e_i in [e_1...e_m] do
4:         n_i, n_o, w ← e_i
5:         Read a = N[n_i]
6:         Read b = N[n_o]
7:         Update N[n_o] = a * w + b
8:         if e_i is the last connecting with output n_o then
9:             Update N[n_o] = σ(N[n_o])
10:        end if
11:    end for
12: end procedure
```

3.2 Reordering Algorithms

3.2.1 EGO: Edge GO

The first approach that we experiment with is a modification of the GO algorithm from [21]. That work attempts to find an ordering of nodes in memory to speed up traditional graph processing algorithms. They base the node ordering

off the observation that the dominant computation of most graph processing algorithms involves iterating through the outgoing edges of each node. This leads them to define the optimization problem:

$$\max_{\phi} \sum_{0 \leq \phi(v) - \phi(u) \leq w} S(u, v)$$

$S(u, v)$ is a locality (similarity) score defined by

$$S(u, v) = S_n(u, v) + S_s(u, v)$$

where $S_n(u, v)$ is weather there is an edge between u and v, and $S_s(u, v)$ is the number of times u and v occur in sibling relationships. W is a parameter defining the window size to consider. This parameter is set to approximate the behavior of fast memory. They present a priority queue implementation of the algorithm which is reproduced in Algorithm 2.

Unlike in [21] we attempt to find an ordering on the edges of the graph rather than the vertices. Also, the edges must be ordered in a topologically consistent manner. Naively, we could transform G into the line-graph of G, the graph H where each edge in G becomes a node in H, and there exists an edge between two nodes in H if and only if the two corresponding edges in G intersect [8]. A topological ordering of the nodes in this graph would correspond to an ordering of the edges in G. However H explodes in size, which becomes a problem for runtime and memory. To address these challenges, we modify Algorithm 2 to produce EGO, shown in Algorithm 3. First, the algorithm is initialized with the edges in the priority queue rather than the nodes. We consider sibling edges those that share the same source or destination. We also consider a successor to edge (u, v) an edge (v, w), or a pair of edges whose source and destination intersect. We also modify the priority of each edge to have two components. The second component is just the priority from the original algorithm. The first component keeps track of the number of dependencies that this edge has. It is initialized to be the (negative) in degree of the source vertex. This priority is then incremented every time a successor edge is added to the ordering. Since the first component is more significant than the second when considering the priority, this ensures that the edges are popped off the priority queue in a topologically consistent order.

3.2.2 EON: Edge Ordering Network

There is some intuition for why learning the function $\phi : E(G) \rightarrow \{1...m\}$ could be a more efficient approach. First, similar problems for optimizing computation graphs have been effectively addressed by learning approaches [23]. In addition, [22] showed that a reinforcement learning approach could out perform heuristic approaches for vertex ordering. Second, there are structural patterns in the graph of sparse networks that admit could similar ordering patterns. For example, the optimal ordering of a "path" of singly connected neurons would be similar across the network. However, there are numerous challenges to learning

Algorithm 2 Node Vertex Ordering Algorithm [21]

```
1: procedure Reorder Nodes
2:     Priority Queue Q
3:     for each node v ∈ V(G) do
4:         Insert v into Q such that k(v) ← 0
5:     end for
6:     i ← 1
7:     while i ≤ n do
8:         Pop v_e from Q, P[i] ← v_e
9:         for each node u ∈ N_O(v_e) do
10:            if u ∈ Q then k(u) ← k(u) + 1
11:            end if
12:        end for
13:        for each node u ∈ N_I(v_e) do
14:            if u ∈ Q then k(u) ← k(u) + 1
15:            end if
16:            for each node v ∈ N_O(u) do
17:                if v ∈ Q then k(v) ← k(v) + 1
18:                end if
19:            end for
20:        end for
21:        if i > w then
22:            for each node u ∈ N_O(v_e) do
23:                if u ∈ Q then k(u) ← k(u) − 1
24:                end if
25:            end for
26:            for each node u ∈ N_I(v_e) do
27:                if u ∈ Q then k(u) ← k(u) − 1
28:                end if
29:                for each node v ∈ N_O(u) do
30:                    if v ∈ Q then k(v) ← k(v) − 1
31:                    end if
32:                end for
33:            end for
34:        end if
35:    end while
36: end procedure
```

Algorithm 3 EGO: Edge Graph Ordering

procedure REORDER EDGES
  Priority Queue Q
  for each edge $(u, v) \in V(G)$ **do**
    Insert (u, v) into Q such that $k((u, v)) \leftarrow (-d_I(u), 0)$
  end for
  $i \leftarrow 1$
  while $i \leq m$ **do**
    Pop (u_e, v_e) from Q, $P[i] \leftarrow (u_e, v_e)$
    for each node v, s.t. $(u_e, v) \in E$ **do**
      if $(u_e, v) \in Q$ **then** $k_2((u_e, v)) \leftarrow k_2((u_e, v)) + 1$
      end if
    end for
    for each node u, s.t. $(u, v_e) \in E$ **do**
      if $(u, v_e) \in Q$ **then** $k_2((u, v_e)) \leftarrow k_2((u, v_e)) + 1$
      end if
    end for
    for each node u, s.t. $(v_e, u) \in E$ **do**
      if $(v_e, u) \in Q$ **then** $k_2((v_e, u)) \leftarrow k_2((v_e, u)) + 1$
      end if
    end for
    ...
  end while
end procedure

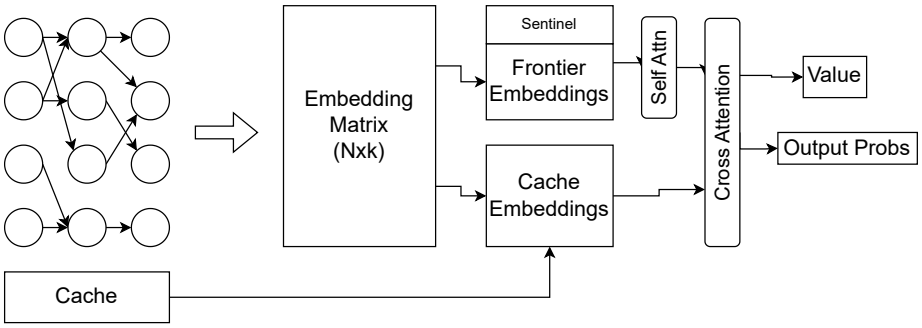


Figure 2: The architecture of EON: the edge ordering network.

an edge ordering of the graph. First, the space of training samples is exponential in the number of edges. Second, it is unclear how to enforce the constraints of a permutation as well as the topological order of the edges. The rest of this section presents the design of EON, edge ordering net, and explains how we attempt to solve theses challenges.

Following previous work, we first phrase the problem as a Markov Decision Process. A state is the current ordering of a subset of the edges as well as the state of the cache. We simplify the model of the cache as a fixed size set of vertices. We define an action as adding an edge to the partial ordering. On this action, the source and destination vertex of the edge are pushed onto the cache. We model cache eviction using a least recently used policy. Since we are interested in creating a policy that maximizes cache performance, we also use the number of cache misses for an edge as a penalty (negative reward) for any action. By considering edge reordering as a MDP, the learning problem becomes more clear. We need to design a policy $\pi : S \to E(G)$ that takes the state of the system (the current order as well as the cache) and outputs the next edge to compute. This formulation lends itself to applying reinforcement learning. Rather than trying to construct explicitly training pairs, reinforcement learning jointly builds a policy function to choose the best action, and a value function to evaluate the state of the system by simulating actions on the environment and collection observation and reward information.

Since the size of the state space is exponential in the length of the ordering and the cache, and the size of the action space is proportional to $|E(G)|$, it is infeasible to build a tabular representation of the policy and value functions as is done in traditional Policy/Value iteration or Q-learning. Thus, we adopt deep reinforcement learning and design a parametric network that is trained to estimate the policy and values, called an actor-critic network. The design of the network is illustrated in figure 2.

The first step in the network is creating a representation of the nodes and the edges in the network. If we examine the objective function of the GO algorithm from above, we see that the critical information for calculating the locality $S(u, v)$ is the neighborhood of any node. When looking at edge ordering,

the neighborhood of an edge is created by the neighborhood of its source and destination nodes. We leverage the un-directed graph Laplacian matrix to create a vertex representation that captures the neighborhood information. Then, we represent an edge as the concatenation the representation of two nodes. This representation can reproduce the same information as the EGO algorithm. If two edges appear as siblings, then the destination or source nodes match, so they will be similar in the embedding space. For dependent edges, we only consider the intersecting source and destination nodes which will share the same embedding. This also captures higher degree information. Even if two edges do not share the same source or destination, if their endpoints share similar neighbors, the embedding of those nodes will be similar. Additionally, using the Laplacian makes the network invariant to a reordering of the nodes as desired for any graph learning algorithm. While the Laplacian has desirable properties, using it directly is impractical because of its length and sparsity. Therefore we use a low rank decomposition of the Laplacian based off the k largest eigenvalues. The largest eigenvectors correspond to the high frequency features of the graph and are therefore appropriate for capturing the local structure of the graph around a node. The final representation is given by:

$$L = D - A = Q\Lambda Q^{-1}$$
$$v_i \rightarrow Q[i, : k]$$

Next we discuss the construction of the policy and value heads of the network. First, before the network is called, the current ordering of the edges is used to build the edge frontier - the edges that have all dependencies filled. The network only sees this frontier, as well as the nodes that are in the cache. This simplifies the design and training of the network since the network can pick an action from this frontier, eliminating the possibility of choosing an invalid action. Once the frontier is constructed, the edges in the frontier as well as the nodes in the cache are embedded as discussed above. After the embedding, the frontier is past through a self attention layer. This is to satisfy the intuition that the best choice of the next edge to calculate might depend on the other edges that need to be calculated. The policy head is then calculated through a cross attention layer between the nodes in the cache and the edges in the frontier. The output corresponding to any action in the frontier is the maximum across all values in the cache. This is transformed into a probability using a softmax activation. Finally the value head is calculated by letting every value of the frontier and the cache attend to a sentinel embedding, similar to the [SEP] token used in NLP. Since attention only depends on the values of the keys and the queries, it is permutation equivariant. In addition, the max function is invariant to the order of the inputs. Thus, this design is invariant against a reordering of the cache and equivariant against a reordering of the frontier.

To train the actor-critic network, we use proximal policy optimization [18].

4 Evaluation

4.1 Implementation Details

The code for conducting the experiments was written in Python and run using CPython 3.10.9. The code for inference algorithm was written in C++ compiled using clang version 10.0. The C++ code exposed a python interface for the function using PyBind11 [9]. The overhead of this wrapper is negligible, and is included in the results of all timings. The EON network was written using PyTorch . All experiments were conducted on a desktop with an Intel Core i7-10700 CPU with 16 cores at a clock speed of 2.9 GHz. The L1 cache size was 256 KiB, the L2 was 2 MiB, the L3 was 16 MiB, and the RAM memory was 16 GiB. All cache performance data was collected with perf, a Linux kernel-based subsystem for performance analysis. All of the results presented are the average of 10 runs.

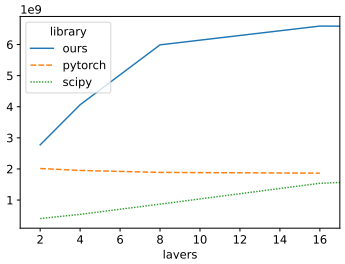
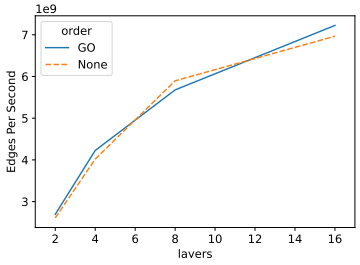


Figure 3: Performance of the inference algorithm on the Radix networks from the Graph Challenge. y-axis is the number of edges calculated per second (higher is better).

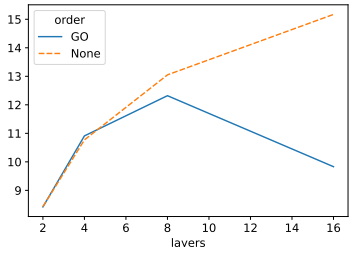
Two categories of networks are evaluated in this work. First, the we take the small network from the Amazon/MIT Graph Challenge [10]. This network is 1024 neurons wide and has up to 1920 layers, although we only evaluate up to 16 layers. The network is constructed using Radix-Net, randomly permuting the layers and repeating the construction till the network reaches the desired depth. As such, each neuron has exactly 32 inputs and outputs. Secondly, we create random sparse networks by choosing non-zero edges uniformly at random. We generate random networks at various levels of sparsity with a width of 1024, and a width of 16. To ingest the networks, each layer is written in COO format to a file, which is read by the inference or reordering algorithm.

4.2 Experiments

To begin, we evaluate the inference algorithm against layer-wise sparse matrix multiplication implemented in Scipy and and pyTorch. We evaluate the inference speed on the Radix-Net graphs from the Graph Challenge. The results are presented across a varying number of layers in figure 3. We can see that the inference algorithm that we use is competitive with the two comparison libraries. The algorithm is up to 4x faster than Scipy and up to 3x faster than pyTorch. Although not shown graphically, the speed up compared to Scipy is persistent at 120 layers (the first official size in the graph challenge). At 120 layers, pyTorch runs out of memory for an unknown reason. These results are



(a) Edges traversed per second as number of layers scale (higher is better).



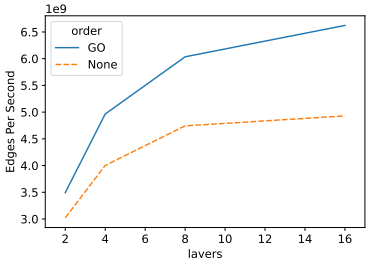
(b) Percentage of cache misses as number of layers scale (lower is better).

Figure 4: Performance of the EGO algorithm compared to the default layer-wise ordering on sparse radix graphs.

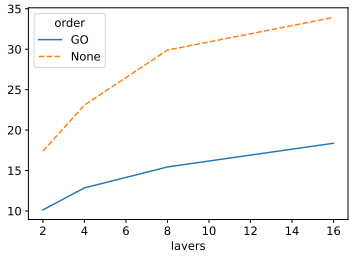
consistent with the results shown in [5] and validates that working with this algorithm is promising research direction.

Next we compare the effect of the EGO algorithm. Figure 4a shows the performance of the algorithm on the Radix networks. There does not seem to be an appreciable difference between the edge reordered inference performance. However, when looking at the cache misses in figure 4b we see that the reordered edges produce significantly fewer cache misses, especially at 16 layers. We can compare this to the analogous graph in figure 5 which shows the inference and cache performance for a randomly sparse neural network. Unlike the Radix network, the edge reordering of the random network produces a significant speed up. This is also reflected in the cache performance. The edge reordered random network experiences nearly half as many cache misses as the original ordering. And the original ordering of the random network experiences twice as a many cache misses as the radix network. The discrepancy between the randomly sparse network and the Radix network can be explained by regularity of the sparse network. The initial ordering of the edges operates in a layer-wise fashion, computing each output neuron in sequential order. Since there are 32 connections to every neuron, and considering a very simplified model of the cache, this ordering would only experience a miss 1/32nd of the time. Compared to random connections, this is is a relatively good cache miss rate. The true cache miss rate is much higher than those projections, but it is clear that the regularity in the connections allows the radix network to perform much better with the original ordering.

To further investigate the performance of the EGO algorithm we compare the performance across a range of sparsity conditions. The results are show in figure 6a. We observe that as the density of the network increases, the EGO algorithm sustains a persistent performance advantage over the baseline ordering. Again, the observed speedup in the rate of edge processing is matched by a corresponding reduction in cache misses. Unfortunately, the EGO algorithm does not prove scalable to higher densities. At a sparsity of 0.5, the EGO algorithm

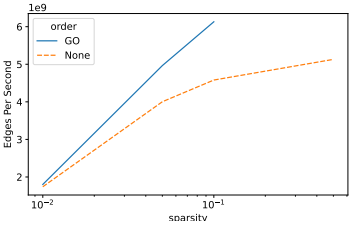


(a) Edges traversed per second as number of layers scale.

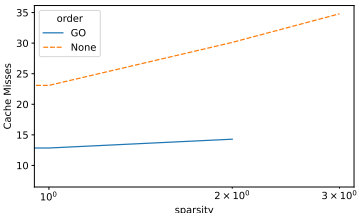


(b) Percentage of cache misses as number of layers scale.

Figure 5: Performance of the EGO algorithm compared to the default layer-wise ordering on randomly sparse graphs 1024 neurons wide with connection sparsity 0.05.



(a) Edges traversed per second versus network sparsity on a semi-log scale

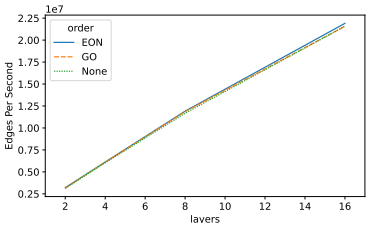


(b) Percentage of cache misses versus network sparsity on a semi-log scale

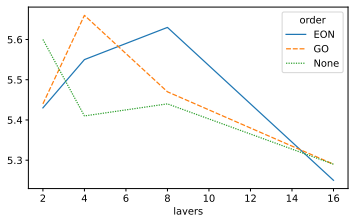
Figure 6: Performance of the EGO algorithm compared to the default layer-wise ordering on randomly sparse graphs 1024 neurons wide and 4 layers deep.

terminates with an out-of memory exception for a network of depth 4. This is a significant limitation to the usefulness of the algorithm as we cannot scale to networks with many connections. At first analysis it is unclear where the increased memory pressure comes from. The priority queue scales linearly with respect to the number of edges, so the algorithm should only take a constant multiple of the size of the graph to run.

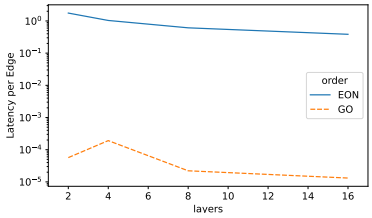
Finally we show the results of EON on a toy network with a width of 16 and a sparsity of 0.05. Figure 7 shows the results compared to EGO and the baseline ordering. Unsurprisingly on a network of this size, there is no significant performance difference between any of the three orderings. We also observe that percentage of cache misses is nearly indistinguishable for the three approaches. While this is very unconvincing, these graphs show a working example of the EON algorithm. Two things prevented the evaluation of EON on a wider range of networks. The first is simply the pragmatic concern of this project. We did not have the necessary time or resources to scale the training and tune the



(a) Edges traversed per second as the number of layers scales



(b) Percentage of cache misses as the number of layers scales



(c) Latency per edge of reordering algorithms, on a semi-log y scale (lower is better).

Figure 7: Performance of the EON algorithm compared to EGO and the default layer-wise ordering on a randomly sparse graph of 16 neurons with a sparsity of 0.05.

parameters to get EON to work on a larger network. The second is a design limitation of EON. Figure 7c shows the time cost per edge of reordering an edge EON compared to EGO. We can see that since EON requires the training of the actor-critic network for new network seen, it requires orders of magnitude more time than the EGO algorithm. In addition to just the overhead of training, the design of the network isn't scalable. In the proposed design, each element of the cache attends to every element of the frontier. For the toy network in the experiment, this frontier can only be as large as 256 edges wide, and because the network is sparse, the length is never longer than 92 edges during the training process. However, for a network that is 1024 neurons wide, the frontier could be as long as 1e6 elements, and will practically be 10,000 edges long. The complexity of self attention scales with the square of the sequence length and the complexity of cross attention scales linearly with sequence length [20]. As such, when scaling EON to operate on larger networks, we quickly exhaust the memory on the machine.

5 Discussion

5.1 Limitations

As mentioned several times through out this work, there are several limitations with the methods that are suggested here. In this section we analyze some of those limitations and probe possible solutions.

We begin with the design of the node embeddings. For this work the graph Laplacian had several nice properties, and was very convenient to use. However, there are some immediate drawbacks. First, by using the graph Laplacian, we fix the embeddings of each node, meaning the information from the graph that the network can use is limited by the initial embeddings. Second, by using only the features of the graph Laplacian that correspond to the largest eigenvalues, we only give the network local information. While it is reasonable to expect this to be the most important information, an optimal ordering may depend on global structure as well. A reasonable alternative to using the graph Laplacian is to learn a representation using a message passing GNN such as graph SAGE [7]. Graph SAGE learns a custom embedding network and aggregation network to build a representation of each node or edge in the graph as a function of its neighborhoods. Deeper layers of graph SAGE integrate information from a wider neighborhood. Thus, for this work, neighborhoods with similar structure in different parts of the graph would produce the same embeddings, resulting in similar orderings. Furthermore, [23] showed that the aggregation and embedding functions learned by graph SAGE could transfer to unseen computation graphs. Although graph SAGE looks like an ideal alternative to the embeddings used, time and resources prevented us from trying this in an ablation study.

Moving on, we further analyze the design of EON. Specifically, the use of the frontier as a input to the network became a concern. The proximal issue is the memory and time that are required for the two attention layers in the network. However, attention still has the properties of equivariance/invariance that were considered when designing the network. Thus rather than removing the attention heads it may be possible to substitute a more efficient form of attention. Several alternatives are discussed in [20] and would be interesting to try. A larger concern is weather the frontier is a useful feature for the network or not. As mentioned in Section 3, the motivation behind this design was to restrict the action space of the network to valid actions rather than relying on the learning process to pickup on that fact. However, this design also could be limiting the behavior learned by the network. By presenting the information in this way, any information about edges not on the frontier (i.e. those that are not ready to be executed) must be inferred from the embedding of nodes on the frontier. Thus, we are biasing the network to act greedily because that is all the information that is available to it. Again this is an issue that could be potentially resolved by using a message passing GNN. If we encoded the cache as labels on the nodes in the graph, and edges in the ordering as labels on the edges in the graph, then the representation the GNN learns could contain the same information as if we past the frontier and cache explicitly.

5.2 Future Work

In addition to the limitations of this work, which can be seen as amending this method for better results, there are numerous directions for future research that these results suggest.

One of the primary directions is expanding the algorithm to different layer types. Specifically, there is a direct extension of the EGO algorithm to attention layers and convolution layers, which are two of the most commonly used layers[1]. First, convolution can be thought of as a feed forward layer with a very structured kind of sparsity. However, what allows dense convolution to run quickly on current hardware is that a small set of weights are shared across all the neurons in the layer. To capture this weight sharing the graph representation in figure 1 could be modified into a hyper-graph, where each edge joins a source, destination, and weight neuron. Then, the predecessors and successors in the EGO algorithm would be able to take into account weight sharing intelligently. Attention would require similar "hyper-edges". To calculate the attention weight, there must be a hyper-edge that joins two source neurons to one destination. Then to calculate output, we could use a second hyper-edge joining the attention weight with the value neuron.

A second possible extension of this work is to incorporate parallelism into the schedule of the edges. For sparse matrix multiplication, vendors provide specialized linear algebra libraries that can run on multiple cores to speed up calculation. Scipy can be linked with these libraries to provide higher performance. Furthermore, pyTorch by default runs on multiple cores (although it did not provide significant speed up in this case). For truly high performance sparse neural network inference the algorithm must be extended to run on multiple cores. One possible direction toward this goal would be adding an additional objective into EON to edges (tasks) into multiple orders while minimizing dependencies between orders. Another approach could be to integrate the static schedule provided by the edge ordering into a dynamic runtime that provides fine-grained parallelism. [16] introduces a low-overhead tasking runtime into OpenMP that allows very fine-grained tasks which may be an effective technique to limit the overhead of parallelism when tasks are the size of edges.

6 Conclusion

Spare neural networks pose an important and challenging problem for machine learning systems. In this work we tackle the problem of ordering the computation graph of sparse neural networks. First, we draw the connection between the computation of a sparse neural network and reordering the memory layout of nodes for traditional graph algorithms. This inspires the design of the EGO algorithm which uses common neighbor structure from the graph to produce an ordering of the edges. Next, we formulate the edge ordering problem as a MDP. This motivates the design of EON a actor-critic network trained by

[1] While I built these two formulations for this project, I ran out of time to implement them.

reinforcement learning to solve the MDP. Finally we evaluate the algorithms on randomly generated sparse networks as well as radix networks. We show that although the EGO algorithm can significantly reduce the number of cache misses and speed up the inference of sparse neural networks.

References

[1] Byung Hoon Ahn et al. "Ordering Chaos: Memory-Aware Scheduling of Irregularly Wired Neural Networks for Edge Devices". In: *Proceedings of Machine Learning and Systems*. Ed. by I. Dhillon, D. Papailiopoulos, and V. Sze. Vol. 2. 2020, pp. 44–57. URL: https://proceedings.mlsys.org/paper/2020/file/9bf31c7ff062936a96d3c8bd1f8f2ff3-Paper.pdf.

[2] Flavio Chierichetti et al. "On Compressing Social Networks". In: *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '09. Paris, France: Association for Computing Machinery, 2009, pp. 219–228. ISBN: 9781605584959. DOI: 10.1145/1557019.1557049. URL: https://doi.org/10.1145/1557019.1557049.

[3] Benjamin Coleman et al. "Graph Reordering for Cache-Efficient Near Neighbor Search". In: *CoRR* abs/2104.03221 (2021). arXiv: 2104.03221. URL: https://arxiv.org/abs/2104.03221.

[4] Jonathan Frankle and Michael Carbin. "The Lottery Ticket Hypothesis: Training Pruned Neural Networks". In: *CoRR* abs/1803.03635 (2018). arXiv: 1803.03635. URL: http://arxiv.org/abs/1803.03635.

[5] Niels Gleinig, Tal Ben-Nun, and Torsten Hoefler. *A Theory of I/O-Efficient Sparse Neural Network Inference*. 2023. DOI: 10.48550/ARXIV.2301.01048. URL: https://arxiv.org/abs/2301.01048.

[6] Yiwen Guo, Anbang Yao, and Yurong Chen. "Dynamic Network Surgery for Efficient DNNs". In: *Advances in Neural Information Processing Systems*. Ed. by D. Lee et al. Vol. 29. Curran Associates, Inc., 2016. URL: https://proceedings.neurips.cc/paper/2016/file/2823f4797102ce1a1aec05359cc16dd9-Paper.pdf.

[7] Will Hamilton, Zhitao Ying, and Jure Leskovec. "Inductive Representation Learning on Large Graphs". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper/2017/file/5dd9db5e033da9c6fb5ba83c7a7ebea9-Paper.pdf.

[8] Frank Harary and Robert Z Norman. "Some properties of line digraphs". In: *Rendiconti del circolo matematico di palermo* 9 (1960), pp. 161–168.

[9] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. *pybind11 — Seamless operability between C++11 and Python*. https://github.com/pybind/pybind11. 2016.

[10] Jeremy Kepner et al. "Sparse Deep Neural Network Graph Challenge". In: *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. 2019, pp. 1–7. DOI: 10.1109/HPEC.2019.8916336.

[11] Mihael Kovac et al. "Towards intelligent compiler optimization". In: *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)*. 2022, pp. 948–953. DOI: 10.23919/MIPRO55190.2022.9803630.

[12] Hyungro Lee, Milan Jain, and Sayan Ghosh. "Sparse Deep Neural Network Inference Using Different Programming Models". In: *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. 2022, pp. 1–6. DOI: 10.1109/HPEC55821.2022.9926362.

[13] Mingzhen Li et al. "The Deep Learning Compiler: A Comprehensive Survey". In: *IEEE Transactions on Parallel and Distributed Systems* 32.3 (2021), pp. 708–727. DOI: 10.1109/TPDS.2020.3030548.

[14] Dian-Lun Lin and Tsung-Wei Huang. "A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism". In: *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. 2020, pp. 1–7. DOI: 10.1109/HPEC43674.2020.9286218.

[15] Huizi Mao et al. "Exploring the Regularity of Sparse Structure in Convolutional Neural Networks". In: *CoRR* abs/1705.08922 (2017). arXiv: 1705.08922. URL: http://arxiv.org/abs/1705.08922.

[16] Poornima Nookala et al. "Enabling extremely fine-grained parallelism via scalable concurrent queues on modern many-core architectures". In: *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE. 2021, pp. 1–8.

[17] Aditya Paliwal et al. "Reinforced Genetic Algorithm Learning for Optimizing Computation Graphs". In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=rkxDoJBYPB.

[18] John Schulman et al. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).

[19] Julian Stier, Harshil Darji, and Michael Granitzer. "Experiments on Properties of Hidden Structures of Sparse Neural Networks". In: *Machine Learning, Optimization, and Data Science*. Ed. by Giuseppe Nicosia et al. Cham: Springer International Publishing, 2022, pp. 380–394. ISBN: 978-3-030-95470-3.

[20] Yi Tay et al. "Efficient Transformers: A Survey". In: *ACM Comput. Surv.* 55.6 (Dec. 2022). ISSN: 0360-0300. DOI: 10.1145/3530811. URL: https://doi.org/10.1145/3530811.

[21] Hao Wei et al. "Speedup Graph Processing by Graph Ordering". In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 1813–1828. ISBN: 9781450335317. DOI: 10.1145/2882903.2915220. URL: https://doi.org/10.1145/2882903.2915220.

[22] Kangfei Zhao et al. "Graph Ordering: Towards the Optimal by Learning". In: *Web Information Systems Engineering – WISE 2021*. Ed. by Wenjie Zhang et al. Cham: Springer International Publishing, 2021, pp. 423–437. ISBN: 978-3-030-90888-1.

[23] Yanqi Zhou et al. “Transferable Graph Optimizers for ML Compilers”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 13844–13855. URL: https://proceedings.neurips.cc/paper/2020/file/9f29450d2eb58feb555078bdefe28aa5-Paper.pdf.