Har Wavelet Feature Compression for Quantized Graph Convolutional Networks

Moshe Eliasof, Benjamin J. Bodner, and Eran Treister

Abstract—Graph convolutional networks (GCNs) are widely used in a variety of applications and can be seen as an unstructured version of standard convolutional neural networks (CNNs). As in CNNs, the computational cost of GCNs for large input graphs (such as large point clouds or meshes) can be high and inhibit the use of these networks, especially in environments with low computational resources. To ease these costs, quantization can be applied to GCNs. However, aggressive quantization of the feature maps can lead to a significant degradation in performance. On a different note, the Haar wavelet transforms are known to be one of the most effective and efficient approaches to compress signals. Therefore, instead of applying aggressive quantization to feature maps, we propose to use Haar wavelet compression and light quantization to reduce the computations involved with the network. We demonstrate that this approach surpasses aggressive feature quantization by a significant margin, for a variety of problems ranging from node classification to point cloud classification and both part and semantic segmentation.

Index Terms—Graph convolutional networks (GCNs), graph wavelet transform, network compression, quantized neural networks.

I. INTRODUCTION

Graph convolutional networks (GCNs) have been shown to be highly successful when applied to a wide array of problems and domains, including social analysis [1], [2], recommendation systems [3], computational biology [4], and computer vision and graphics [5], [6], [7]. Conceptually, GCNs are an unstructured version of standard convolutional neural networks (CNNs) where instead of a 2-D or a 3-D grid, an unstructured graph or a mesh is given [7].

The computational cost of GCNs, as with structured CNNs, is directly affected by the size of their inputs and the intermediate feature maps throughout the network. In many real-world applications, such as LiDAR-based point cloud segmentation, large point clouds or graphs are required during training and deployment, leading to high computational and memory costs. These tremendous costs inhibit the use and deployment of these networks, especially in environments with low computational resources such as smartphones, autonomous vehicles, and specialized edge devices.

A variety of approaches have been proposed to reduce the computational costs of neural networks in recent years. Among the popular approaches are weight pruning or sparsification [8], [9]. Such methods reduce the number of nonzero entries in the weight tensors, intending to perform computations on the nonzero entries only. However, to reduce the computational load, the sparsification needs to be structured [10].

Another popular approach that we focus on in this work is quantization, where the numerical precision of both the weights and feature maps throughout the network is reduced [11]. This enables the operations to be carried out using low-cost fixed-point integer arithmetic and significantly reduces the memory overhead. Recent works have demonstrated that standard CNNs can be quantized to 4-bit precision, with minimal losses in performance [12], [13]. In addition, quantization methods were also applied to GCNs [14], [15], [16], showing similar gains in computational efficiency with minimal loss in performance.

However, in many scenarios, it is evident that aggressive quantization (less than 4-bit precision) is prone to accuracy degradation. This is particularly apparent for the feature maps’ bit-width in tasks such as semantic segmentation [17] where we wish to classify each element of the input, as opposed to classification [12], where a single label is globally assigned to the input. Evidently, the majority of quantization techniques are applied and tested on image or graph classification [18]. In the recent [17], which targets quantized U-Nets, the feature map bit rates are relatively high (8 bits), while those can be low for the learned weights (2 bits). In this work, we wish to improve feature maps’ compression in GCNs beyond the existing quantization methods, by introducing a novel graph Haar wavelet compression mechanism. Our compression method further eases the computations of GCNs, especially for semantic segmentation on dense point clouds.

To this end, we harness wavelet transformations [19] which are well-known for their ability to compress images through sparse representation in the wavelet domain, as in the popular JPEG2000 format [20]. Following their success in images, wavelet transformations have been adapted for graph signals [21] and were later used in various works such as [22], [23], [24] to define spectral wavelet-based graph convolutions. Similarly, a spatial graph Haar wavelet transformation was suggested in [25] based on a chain representation of the graph. Later, a similar principle was adapted and applied to GCN...
convolution and pooling [26], [27], [28]. The advantage of the wavelet transforms lies in the sparsity of the transformed signals, which can be exploited for denoising and compression. Furthermore, wavelet convolutions typically have a wider receptive field than standard spatial convolutions, yet with similar computational costs [22], making them favorable in terms of computational efficiency.

In this work, we use both the Haar wavelet compression and quantization methods to obtain highly compressed GCNs, with a minimal performance loss. The Haar compression is applied to the feature maps, and quantization is applied to both the weights and the Haar-transformed feature maps. This combination saves a significant amount of computations in the inference and training of the network. Our Haar transform is a parallelized version of [25], which is efficiently applied on GPUs, using graph clustering, and is similar to the transform in [26]. The transform is applied with linear complexity for each channel separately, using binary operations only, due to the simplicity of the Haar basis. The key component of our approach is to keep only the top k elements of the transformed feature maps—across all the channels, which we refer to as the wavelet shrinkage scheme, yielding a compressed tensor of the wavelet transform. The final component is the convolution that is applied to the compressed tensor. We show that the transform and shrinkage operations commute with the convolution step, which is at the core of CNNs and GCNs, can be performed on a significantly smaller input. This idea is executed along with quantization to further reduce the activation size and cost of convolution.

Using the approach described above, we compress the feature maps with minimal loss of information, which, as we demonstrate, provides better compression ratios and less performance degradation than having aggressive quantization on the feature maps. We propose two versions of our compression scheme: the first can be applied with minimal change to the network’s weights (i.e., it is possible to use pretrained weights in the spatial domain), and the other is a new neural architecture in the graph Haar wavelet domain, which is suitable for large feature maps and requires less wavelet transforms. To summarize, our contribution is as follows.

1) We are the first, to the best of our knowledge, to apply both the quantization-aware training and hierarchical graph Haar transforms in the context of compression of GCNs.

2) We show that the sparsity of the Haar wavelet transform can be used for the compression of the graph channels. By applying joint channel shrinkage, the combination of the transform and shrinkage operations commutes with a $1 \times 1$ convolution, obtaining computational savings using dense arithmetic only. We show that this approach surpasses aggressive quantization of the activation in both performance and efficiency.

3) Two wavelet convolution schemes are suggested: the first can be applied to pretrained weights of a network without the Haar transforms (i.e., as a scheme for compression only), and the other applies multiple convolutions in the wavelet domain, requiring less forward and inverse transforms, but needs to be trained with the Haar transform from scratch.

II. RELATED WORK

A. Graph Convolutional Networks (GCNs)

GCNs come in two main types—involving spatial and spectral convolutions [29]. The spectral convolution is built on the graph Fourier transform, using learned low-order Chebyshev polynomials over the graph Laplacian [30]. This formulation is also featured in the works [31], [32], using first-order polynomials. Such methods are mostly used for graph node classification problems. One of the most debatable issues in the literature with such methods is the over-smoothing problem that leads to degradation in performance as more layers are added. For example, the purpose of additions in [32] over [31] are to prevent over-smoothing.

Another family of GCNs targets more geometric tasks such as classification and semantic segmentation of point clouds and meshes where the input features include locations of the points in a 3-D space. See DGCNN [6], MPNN [33], DiffGCN [7], and references therein. Usually, given a point cloud, the underlying graph is produced by the k-nearest neighbor (kNN) algorithm, and the spatial convolution is defined by computing values in the graph edge space. For instance, in DGCNN neighboring features are subtracted, while in [7] and [34] graph differential operators are used, and graph pooling is applied by graph clustering, e.g., Graclus [35].

B. Wavelet Transforms, Compression, and Convolution

Wavelet transforms are widely used for images and can also be used to define convolutions and architectures in standard CNNs for a variety of imaging tasks—see [36], [37], [38]. Analogously, graph wavelet transforms [21] have been used to define convolutions in GCNs. Similar to the graph convolution, graph wavelet convolutions can be spectral [22], [23], [24], and like the graph Fourier convolution, they primarily rely on the eigenbasis of the graph Laplacian, along with a heat kernel applied on the spectrum. Graph wavelets lead to sparse feature maps for node classification networks [22]. Since the full eigen-decomposition of the graph Laplacian is too costly, other works [22], [39] propose to use a Chebyshev polynomial parameterization.

Of particular interest in this work is the graph Haar transform [25] which has been designed to generate highly sparse image representations based on a 1-D chain representation of the graph. Likewise, a parallel version of the Haar transform is used to define graph convolution and pooling [26], [27], [28], where parallel clustering methods are used to generate a tree-based hierarchy. Other Haar-based GNNs [22], [39] use a Chebyshev polynomial parameterization to construct low- and high-pass filter banks. In our work, we build on the idea of approximating the solution of the perfect graph-matching problem, to obtain an orthogonal basis of the graph, paving the road to a wavelet transform, as described in Section IV. The advantage of the Haar transform lies in its simplicity and computational efficiency, as it does not require multiplications and can be carried out using efficient binary operations.
We exploit the sparsity of the transformed features to save computations by compressing them, and while here we use the Haar transform, the standard wavelet transform [22] can also be used in our framework.

C. Quantized Neural Networks

Research on quantization methods has been active in recent years, mostly in the context of CNNs. A plethora of quantization schemes tailored for different scenarios have been proposed, including post-training and quantization-aware training schemes, uniform and nonuniform schemes, etc.; see [12], [13], [18] and references therein. In the context GCNs, per-node bit allocation was suggested in [15] and [14], while in [16] the bit allocation is performed for each layer, using neural architecture search methods. In this work, we focus on the contribution of the wavelet shrinkage scheme. With that in mind, and to ensure hardware compatibility, we harness uniform quantization-aware training, with per-layer quantization parameters.

III. BACKGROUND

A. Neural Network Quantization and Training

The uniform quantization schemes we consider can typically be split into three steps: clip, scale, and round. In the first step, the original values of the weights or feature maps are clipped to be within a range of values: i.e., \([-\alpha, \alpha \cdot r_b] = \alpha\cdot \lfloor 2^b \cdot x \rfloor/2^b\) for signed quantization (typically, for the weights) and \([0, \alpha \cdot r_b] = \alpha\cdot \lfloor 2^b \cdot x \rfloor\) for unsigned quantization (typically, for the feature maps)\(^1\). Here, \(\alpha\) is referred to as the clipping parameter and \(r_b = (2^{b-1})/2^b\), where \(b\) is the number of bits. Next, the dynamic range is scaled to the target integer range of \([-2^{b-1}, 2^{b-1} - 1]\) for signed quantization and \([0, 2^b - 1]\) for unsigned quantization. Finally, the values are rounded to the nearest integer.

Each layer in the neural network typically contains two clipping parameters, one for the weights and one for the feature maps. These parameters are optimized across all the layers during the training procedure such that they yield the lowest possible loss. Several approaches have been proposed in recent years to optimize the clipping parameters, such as those based on quantization error minimization [40], backpropagation [13], and more [18]. Here, we base our uniform quantization-aware training on [12] and [13], which enables the quantized network to be trained in an end-to-end manner.

To formally introduce the pointwise quantization operations, we first define the quantization operator

\[
Q_b(x) = \text{round}(2^b \cdot x) / 2^b \quad (1)
\]

where \(x\) is a real-valued tensor in \([-1, 1]\) or \([0, 1]\) for signed or unsigned quantization, respectively.

Given this operator, we use the reparameterized clipping function [13] to define the quantized weights and feature maps

\[
W_b = \alpha_W Q_{b-1} \left( \text{clip} \left( W / \alpha_W, -1, r_{b-1} \right) \right) \quad (2)
\]

\[
X_b = \alpha_X Q_{b} \left( \text{clip} \left( X / \alpha_X, 0, r_b \right) \right) \quad (3)
\]

Here, \(W\) and \(W_b\) are the original and quantized weight tensors, \(X\) and \(X_b\) are their associated clipping parameters, respectively. The function clip\((x, a, b)\) denotes the clipping of the input \(x\) onto the section \([a, b]\). Note that (1)–(3) are used for training only. During inference, all the weights and feature maps are quantized, and all the operations are performed using fixed-point arithmetic.

Here, we also follow the common practice in the quantization-aware training schemes, where feature maps and weights are quantized during the forward pass, and in the backward pass, the straight through estimator (STE) [41] is used to optimize the weights. That is, we use \(W_b\) in the forward pass, but ignore \(Q_b\) in the backward pass, and estimate the gradients using the floating point values of \(W\) in the optimization process. Furthermore, given (2) and (3), the STE can also be used to calculate the gradients with respect to, for example, \(\alpha_W\)

\[
\frac{\partial W_b}{\partial \alpha_W} = \begin{cases} -1, & \text{if } W \leq -\alpha_W \\ r_{b-1}, & \text{if } W \geq \alpha_W \cdot r_{b-1} \\ \frac{W_b}{\alpha_W} - \frac{W}{\alpha_W}, & \text{if } -\alpha_W < W < \alpha_W \cdot r_{b-1}. \end{cases} \quad (4)
\]

The contributions are then summed over all the elements of \(W_b\), to calculate the gradient values used to update \(\alpha_W\). \(\alpha_X\) is updated similarly, only with respect to the activation maps. For more details, see [12], [13]. To help the optimization, we also use weight normalization before the quantization [13]: \(\hat{W} = ((W - \mu) / (\sigma + \epsilon))\). Here, \(\mu\) and \(\sigma\) are the mean and standard deviation of the weight tensor, respectively, and \(\epsilon = 10^{-6}\).

B. Graph Operations and Node Pairing

Assume we are given an undirected graph \(G = (\mathcal{V}, \mathcal{E})\) where \(\mathcal{V}\) is a set of \(n\) nodes and \(\mathcal{E}\) is a set of \(m\) edges. Let us denote by \(\mathbf{f}_i \in \mathbb{R}^c\) the feature vector that resides at the \(i\)th node of \(G\) with \(c\) being the number of channels. Also, we denote the degree of the \(i\)th node by \(d(i)\). We define \(\mathbf{D}\), the gradient operator that acts on the graph node features \(\mathbf{f}\) as follows:

\[
(\mathbf{Df})_{(i,j) \in \mathcal{E}} = \frac{1}{\sqrt{2}}(\mathbf{f}_i - \mathbf{f}_j) \quad (5)
\]

where nodes \(i\) and \(j\) are connected via the \((i, j)\)th edge. This operator is a mapping from the node space to the edge space and can be thought of as a weighted directional derivative in the direction defined by the \((i, j)\)th edge. Similar to (5), we define the node-averaging operator

\[
(\mathbf{Af})_{(i,j) \in \mathcal{E}} = \frac{1}{\sqrt{2}}(\mathbf{f}_i + \mathbf{f}_j) \quad (6)
\]

which also maps from the node space to the edge space. The transpose of the nodal average, i.e., \(\mathbf{A}^\top\), is an averaging operator for the edge features.
Fig. 1. Example of a perfect matching. Circles and lines represent nodes and edges, respectively. The node features are denoted by \( \mathbf{f} \). The perfect matching edges \( E_p \) from (8) are labeled in red.

1) Pair-Graph: Given a graph \( \mathcal{G} = (\mathcal{V}, \mathcal{E}) \), we define its undirected pair-graph

\[
\mathcal{G}_p = (\mathcal{V}, \mathcal{E}_p)
\]

where \( \mathcal{E}_p \subset \mathcal{E} \) is a subset of the edges, which forms a perfect matching of the nodes. That is, we construct a graph where each node is paired to exactly one (different) node, and the degree of each node is exactly 1. See the example in Fig. 1. Formally, we wish to find a subset of the edges such that

\[
\mathcal{E}_p = \{ (i, j) \in \mathcal{E} \mid d_p(i) = d_p(j) = 1 \}
\]

where \( d_p(i) \) is the degree of the \( i \)th node of the graph \( \mathcal{G}_p \). Note that this subset is not unique and can be found under different criteria. In this work, we focus on the graph Haar wavelet transform and feature maps’ compression. Therefore, we construct \( \mathcal{E}_p \) such that the similarity between connected nodes is maximal, subject to the constraints in (8), to promote the sparsity of the Haar wavelet transform and to enable better compression of the transformed features maps. This can be approximately achieved by choosing nodes that are closest in their feature norm (\( \ell_2 \) or \( \ell_1 \)). Specifically, we use the Gracus [35] algorithm with edge weights that are computed as \( w_{ij} = \| \mathbf{f}_i - \mathbf{f}_j \|_2 \) to efficiently obtain the pair-graph. While not optimal, this clustering algorithm is fast and parallel (in many cases, it is also used in GCNs for pooling). Following this process, if a few nodes are left with no match, we match those nodes randomly. Typically, only a small fraction of the nodes do not belong to a pair (in our experiments, under 2%). Hence, this has an insignificant impact on the performance.

IV. Method

In this section, we describe our Haar wavelet convolution and feature compression method in detail. We aim to reduce the computational cost associated with convolutions performed on intermediate feature maps. Here, we use the Haar transform for its computational efficiency and simplicity. When the spatial operations and fully coupled operations like \( 1 \times 1 \) convolutions are separable, we perform the latter in the wavelet domain on a fraction of the original input size. This property is desired, as \( 1 \times 1 \) convolutions are typically the most expensive component of neural networks.

Prior to the \( 1 \times 1 \) convolution step, a spatial graph Haar transform is applied, which is separable between the channels—i.e., there is no coupling between the different feature maps. We then get sparse feature maps and perform a joint shrinkage operation, dropping the entries with the smallest feature norms, yielding a compressed representation of the input features. The locations of those nonzeros in the original graph are kept in a single index list (alternatively, a bit-map), as they are necessary for the inverse transform from the wavelet to the spatial domain. To further improve the compression rates, we also apply light quantization (8 bits) to the transformed signals and quantize the convolution weights, as described in Section III-A.

We consider two versions of compression. The first (\( V_1 \)) is where our transform is used for compression only, and the network itself (its weights) can, in principle, remain the same as without the compression, as we show in Section IV-C. This version is important, for example, for scenarios where we want to compress networks without retraining (or without access to the data to train on) using pretrained weights. In this case, in each convolution, we apply the forward and inverse Haar transforms in every convolution. In our second version (\( V_2 \)), we apply the nonlinear feature maps on the compressed signals in the wavelet domain, which involves fewer Haar transforms throughout the network.

A. Graph Haar Wavelet Transform

1) Single-Level Graph Haar Transform: Given a pair-graph \( \mathcal{G}_p \), we define a single level graph Haar operator as the combination of the pair-graph gradient matrix \( D \) and average operator \( A \)

\[
W = \begin{bmatrix} D \mid A \end{bmatrix} \in \mathbb{R}^{n \times n}.
\]

Given node feature maps \( \mathbf{f} \in \mathbb{R}^{n \times c} \) defined over the nodes of the graph, we compute the (single level) Haar wavelet transform of the features as follows:

\[
\mathbf{p} = W \mathbf{f}.
\]

Note that by the construction of the pair-graph, the two operators satisfy \( A^T D = 0 \), since they correspond to the orthogonal basis vectors \( (1/\sqrt{2})[1,1] \) and \( (1/\sqrt{2})[1,-1] \). Also, both \( DD^T \) and \( AA^T \) are identity matrices. This means that the operator in (9) is orthogonal as well, i.e., it holds that \( WW^T = I \). Thus, it can represent the feature maps \( \mathbf{f} \) without loss of information. Also, since we have that the degree of each node is 1 in the pair-graph, it follows that the number of edges is exactly half the number of the nodes, i.e., \( |\mathcal{E}_p| = (|\mathcal{V}|)/2 \). Therefore, the gradient and averaging operators coarsen (downsample) the number of graph nodes \( \mathbf{f} \) by a factor of 2.

2) Multilevel Graph Haar Transform: The description above concludes the first Haar level, and to achieve satisfactory compression, we repeat the construction of the wavelet operator (9), each time based on the graph averaged features \( A \mathbf{f} \) to obtain a coarser representation of the original feature maps \( \mathbf{f} \). More precisely, denote by \( \mathbf{p}^{(l)} \) the transformed features at the \( l \)th graph resolution level, and by \( \mathbf{f}^{(l)} \) the averaged signal at the
Fig. 2. Example of the tree-like hierarchical coarsening of the graph in Fig. 1 during the Haar transform in (11).

A hierarchy of graphs is shown in Fig. 2. The transformed signal is the concatenation of all the signals, including the final average

$$p = [p^{(1)}, p^{(2)}, \ldots, p^{(L)}, f^{(L)}] = Wf. \quad (12)$$

Note that the total size of $p$ is $n^2$, as $f$, and since $W$ is orthogonal, we have that $\|p\|_2^2 = \|f\|_2^2$, and no information is lost in the process. The main feature of the Haar transform is that if the signal $f$ is piecewise constant or at least smooth, then $p$ will be sparse, on which we elaborate Section IV-B.

Fig. 3. Flow of the Haar wavelet compression. (a) Input features—16 channels (columns) and 32 points (rows) from the second layer of DGCNN for the ModelNet40 dataset. (b) Haar-transformed graph signal. (c) $\times 4$ compressed signal and its bitmap using joint channel sparsity.

B. Wavelet Signal Compression

The wavelet compression methods involve the lossless wavelet transformation described above, followed by a lossy shrinkage operation, which reduces the small-magnitude nonzero entries of the transformed input signal $p$. Thus, we have a sparse representation of the signal in the wavelet domain, which can be transformed back to the original spatial domain via the corresponding inverse wavelet transform.

However, naively storing a multichannel signal as a sparse matrix imposes the overhead of maintaining nonzero entries indices per channel, leading to high memory costs and inefficient memory access. Furthermore, multiplying general sparse matrices has overhead and is costly to perform in hardware. Hence, we apply the shrinkage selection jointly across all the channels, such that all the channels share the same nonzero location list. Then, we gather the nonzeros of all the channels to a dense 2-D tensor, which is fed into a standard $1 \times 1$ convolution operator that operates efficiently in dense arithmetic on a significantly smaller input signal. Using the nonzero list, we apply the inverse transform after zero-filling the channels (also called “scattering”).

For example, consider the feature map in Fig. 3(a). Its wavelet-transformed tensor $p \in \mathbb{R}^{n \times n}$ is presented in Fig. 3(b) and is obtained by the process described in Fig. 2, and (11). Typically, for a piecewise smooth feature map $f$, the tensor $p$ is effectively sparse, as depicted in Fig. 3(b). We compress $p$ by choosing a subset of the $n$ entries according to the feature norm of each entry to minimize the information loss. To this end, we choose $[\alpha n]$ entries of $p$, where $0 < \alpha < 1$, with the largest $\ell_2$ norm, and denote the ordered indices of the chosen nodes by $C \in \mathbb{R}^{[\alpha n]}$. More formally, we construct the shrinkage operator $T \in \mathbb{R}^{[\alpha n] \times n}$ as follows:

$$T_{ij} = \begin{cases} 1, & j = C_i \\ 0, & \text{Otherwise.} \end{cases} \quad (13)$$

Namely, $T$ is an operator that projects $p$ onto the entries corresponding to indices in $C$. The compressed wavelet transform and a bit-map whose nonzeros indicate $C$ of the prior example is given in Fig. 3(c). For $C$, we choose the node indices with the largest feature $\ell_2$ norm so that the squared error norm, $\|Tp - p\|_2^2$, is minimized. Analogously, we define the opposite operator of $T$, which interpolates (zero-fills) from $[\alpha n]$ to $n$ entries as its transpose $T^\top$. For a signal $f$ to be efficiently compressed by a wavelet transform, the transformed signal $Wf$ needs to be sparse, and then

$$Wf \approx T^\top Wf$$

since $T^\top T$ replaces the smallest entries in $Wf$ with zeros, leading to a minimal information loss with respect to the $\ell_2$ norm of $Wf$. From the orthogonality of $W$, it follows that:

$$f \approx W^\top T^\top Wf. \quad (14)$$

We elaborate on this claim experimentally in Section V-A.

C. Compressed Wavelet Transform Convolution

We focus on $1 \times 1$ convolutions as these are fundamental in neural networks, particularly in GCNs, where the spatial
operations are done separately. The combination of the wavelet transform \( W \) with the compression matrix \( T \) in (14) together with a \( 1 \times 1 \) convolution matrix \( K_{1 \times 1} \) yields the compressed wavelet transform convolution

\[
\mathbf{f}^{\text{out}} = W^T T^T K_{1 \times 1} W \mathbf{f}^{\text{in}}
\]

(15)

where \( K_{1 \times 1} \) is a trainable convolution matrix. The application of the \( 1 \times 1 \) convolution \( K_{1 \times 1} \) to the compressed wavelet transform \( W \mathbf{f}^{\text{in}} \) is the main idea of our approach for obtaining an efficient and compact convolution. Since \( T \) and \( W \) jointly operate on all the channels, \( K_{1 \times 1} \) does not necessarily need to be squared (i.e., input channels need not be equal to the output channels).

1) Equivalence of the Convolution Order: A \( 1 \times 1 \) convolution operates on any channelwise input as follows:

\[
y^{\prime} = (K_{1 \times 1} f)^{\prime} = \sum_j k_{i,j} f^{\prime}
\]

(16)

where \( f^{\prime} \in \mathbb{R}^n \) is the \( j \)th input channel and \( k_{i,j} \in \mathbb{R} \) are the weights that comprise the matrix \( K_{1 \times 1} \in \mathbb{R}^{c_{\text{in}} \times c_{\text{out}}} \) where \( c_{\text{in}} \) and \( c_{\text{out}} \) are the input and output numbers of channels, respectively. Note that the operation in (16) is a simple matrix multiplication in the channel space. Assume that we are applying the spatial wavelet shrinkage operator \( TW \) that applies the same transformation and shrinkage on all the channels

\[
y^{\prime} = (TW K_{1 \times 1} f)^{\prime} = TW \sum_j k_{i,j} f^{\prime} = \sum_j k_{i,j} TW f^{\prime}
\]

(17)

It follows that the two operators commute

\[
TW K_{1 \times 1} = K_{1 \times 1} TW.
\]

(18)

In fact, the result above holds for any arbitrary spatial operation, such as node gradient and Laplacian, and also our inverse wavelet shrinkage transform \( W^T T^T \). Thus, the operation in (15) is equivalent to a wavelet compression (14) after a \( 1 \times 1 \) convolution

\[
\mathbf{f}^{\text{out}} = W^T T^T W K_{1 \times 1} \mathbf{f}^{\text{in}} = K_{1 \times 1} W^T T^T W \mathbf{f}^{\text{in}}
\]

(19)

The main difference between (15) and (19) is the computational cost. (19) applies the \( 1 \times 1 \) convolution \( K_{1 \times 1} \) to the original input shape with \( n \times c \) entries, while in (15) the \( K_{1 \times 1} \) convolution operates on the compressed wavelet transform that holds \( \lceil \alpha n \rceil \times c \) entries, where \( 0 < \alpha < 1 \). With our operation in (15), we present the following lemma.

**Lemma 1:** Given node features \( f \), it holds that a \( 1 \times 1 \) convolution layer \( K_{1 \times 1} \) in an activation function \( \sigma \) satisfies

\[
\sigma(K_{1 \times 1} W^T T^T W f) = \sigma(W^T T^T K_{1 \times 1} W f)
\]

(20)

and if no compression is used (\( T = I \)), then

\[
\sigma(K_{1 \times 1} f) = \sigma(W^T K_{1 \times 1} W f).
\]

(21)

The proof follows immediately from (15) and (19), and the fact that \( W \) is orthogonal. Thus, we can perform the convolution in the wavelet domain with dense \( \lceil \alpha n \rceil \) elements. As long as the wavelet compression is effective (i.e., \( Wf \) is sparse, thus allowing for more entries to be compressed), we obtain a similar result as applying the convolution on the original feature tensor \( f \) with \( n \) elements. The compression ratio \( \alpha \) is typically chosen to be \((1/4) \) or \((1/8)\). Note that in other works mentioned earlier, it is the spatial separable convolution that is applied in the wavelet domain, while the \( 1 \times 1 \) convolution is applied in the original domain. Also, note that the result above holds regardless of the entries in \( K_{1 \times 1} \). Hence, we can use our transform to compress the feature maps of a standard network with no wavelets, given pretrained weights. This scenario is especially important in cases where we have no access to the data and wish to compress the network without retraining (because of privacy issues, for example).

In addition, we propose an alternative scheme that includes fewer wavelet transformations by combining two subsequent convolution layers after applying one wavelet transform. That is, we stack multiple convolutions in the wavelet domain, instead of a single \( 1 \times 1 \) convolution. For instance, an application of two convolutions reads the following:

\[
\sigma(W^T T^T K_{1 \times 1}^1 \sigma(K_{1 \times 1}^2 W f))
\]

(22)

where \( K_{1 \times 1}^1 \) and \( K_{1 \times 1}^2 \) are different trainable weights. Note that this formulation does not satisfy lemma 1, and hence, it requires full training.

**D. Efficient Graph Wavelet Implementation**

In this section, we describe a fast and practical implementation of our method. Our goal is to avoid operations that require using the graph connectivity, due to the possibly arbitrary sparsity pattern in memory. To this end, we apply a preprocessing step where we rearrange the graph nodes into a 1-D sequence according to the computed node pairs, which are the edges of our pair-graph \( \mathcal{E}_p \) from (8). We keep the initial pairings for all the layers in the forward propagation. This construction allows to compute the forward and backward graph Haar wavelet transforms using standard 1-D Haar transforms built on efficient 1-D convolution operations. The result is exactly the same as the process described in Section IV-A.

Our custom GPU implementation applies the hierarchical graph Haar wavelet transform as an in-place operation such that each 1-D Haar transform costs about as a standard average pooling (i.e., \( \text{AvgPool1D} \) in the PyTorch [42]), which is a relatively cheap operation compared with a \( 1 \times 1 \) convolution. Our method is beneficial for devices where the multiplication-and-add operations are the main computational bottleneck. In such cases, for example, the average pooling is significantly cheaper than a \( 1 \times 1 \) convolution. We note that the complexity of the Haar transforms scales linearly in the number of channels, while the complexity of the \( 1 \times 1 \) convolution scales quadratically in the channels. Hence, our method is beneficial for wide architectures. Furthermore, being a pointwise operation, the Haar transforms can be fused together with other operations, such as a skip connection, activation, and also graph propagation. This saves memory readings and writings. Furthermore, we note that GPUs are different from low-resource edge devices in their design and performance.
characteristics. We demonstrate the run-times of our method in Section V-G.

E. Using Graph Wavelet Compression in GCNs

In this work, we focus on both the geometric and non-geometric tasks, which are typically treated with different networks. In particular, we focus on two popular networks: DGCNN [6] and GCNII [32], which we compress to various degrees to demonstrate the effectiveness of our approach.

1) Wavelet-Compressed DGCNN: Given features tensor \( f \) defined on the nodes of the graph \( G = (V, E) \), the edge convolution [6] operation is given by the following equation:

\[
t_{l}^{(j+1)} = \square (K_{1 \times 1} [t_{l}^{(j)}, t_{l}^{(j)} - t_{j}^{(j)}])
\]

(23)

where \( K_{1 \times 1} \in \mathbb{R}^{c_{out} \times c_{in}} \) is the convolution matrix, and \( \square \) is a symmetric aggregation operator such as max or averaging. Alternatively, we could replace the order of the spatial and convolution operations as follows [43]:

\[
t_{l}^{(j+1)} = \square (K_{1 \times 1} t_{l}^{(j)} + t_{l}^{(j)} - t_{j}^{(j)})
\]

(24)

where \( y^{(j)} = K_{1 \times 1} f^{(j)} \) and \( t^{(j)} = K_{1 \times 1} t^{(j)} \), both of size \( c_{out} \times c_{in} \), and hence have the same number of parameters as in (23). Both (23) and (24) are equivalent, and the observation above significantly reduces the computational cost (a factor of \( 1/k \)) where \( k \) is the number of neighbors per node. Using Lemma 1, we simply compress the \( 1 \times 1 \) convolutions

\[
y^{(j)} = W^{T} T^{T} K_{1 \times 1}^{1} T W f^{(j)}, \quad t^{(j)} = W^{T} T^{T} K_{1 \times 1}^{2} T W f^{(j)}.
\]

2) Wavelet-Compressed GCNII: A GCNII layer is defined as follows [32]:

\[
f^{(j+1)} = \sigma (K_{gcnii}^{(j)} S^{(j)} (f^{(j)}))
\]

(25)

where \( S^{(j)} (\cdot) \) is the spatial operation

\[
S^{(j)} (f^{(j)}) = \left( (1 - \alpha_{j}) \tilde{P} f^{(j)} + \alpha_{j} f^{(0)} \right)
\]

(26)

where \( \tilde{P} = I - \tilde{L} \) and \( \tilde{L} \) is the normalized graph Laplacian for the graph \( G \) with one additional self-loop for each node. The addition of the initial feature helps prevent the over-smoothing phenomenon in GCNs. The channel mixture of GCNII, which is the learned part, is given by \( K_{gcnii}^{(j)} \)

\[
K_{gcnii}^{(j)} = \left( (1 - \beta_{j}) L + \beta_{j} K_{1 \times 1}^{(j)} \right)
\]

(27)

where \( K_{1 \times 1}^{(j)} \) is a trainable \( 1 \times 1 \) convolution matrix. The wavelet-compressed version of GCNII is written as follows:

\[
t^{(j+1)} = \sigma \left( W^{T} T^{T} K_{gcnii}^{(j)} T W S^{(j)} f^{(j)} \right)
\]

(28)

where \( \alpha_{j} \) and \( \beta_{j} \) are hyperparameters as in [32]. We note that a wavelet-compressed GCN [31] can also be obtained using (28) when \( \alpha_{j} = \beta_{j} = 1 \).

F. Quantization in GCNs

We apply quantization to several components of the GCNs discussed above. We quantize each network as is commonly done in CNNs and described in Section III-A. Specifically, we quantize the weights of all the layers in the GCN to a specified fixed precision (typically 1, 2, 4, or 8 bits). Following a common practice in quantized neural networks [12], [13], the first and last layers are given a high bit rate—32 bits. That is to prevent degradation of the input or final output, as quantizing these layers often leads to a large loss in performance, while providing minimal gains in terms of compression. The input feature maps of the spatial operations (wavelet, edge convolution, or the graph Laplacian) and MLPs are also quantized, as well as the input feature maps of the wavelet transformation. The channel mixing MLPs use quantized weights. Our quantization is uniform and uses the same bit rate for all the graph nodes for hardware efficiency. In addition, using more sophisticated and complex quantization [44] will improve accuracy, but they may be less efficient on hardware. This is a tradeoff that can be adapted given the situation.

V. Experiments

In this section, we demonstrate the efficiency of our wavelet approach for feature maps’ compression. We start with a basic experiment to quantify the performance of wavelets in feature maps’ compression. Following that, we carry various experiments both on the geometric and nongeometric datasets, for different tasks, such as node classification, point cloud classification, and point cloud segmentation. Our main goal is to demonstrate that the Haar wavelet compression is significantly better than aggressive feature quantization, and since the operations are carried using compact dense vectors, this is also efficient in hardware, i.e., we do not use sparse arithmetic in the convolutions.

The comparison is done using the primary performance metrics of each benchmark [e.g., mean square error (MSE), accuracy, mean intersection over union (mIoU)], as well as the feature maps compression ratios of the convolution operations, which is the most computation-intensive operation in GCNs. Unless stated otherwise, in all the experiments we apply 8-bit weight quantization to all the networks. Furthermore, we apply a wide array of feature maps quantization ratios, except the first and last \( 1 \times 1 \) convolutions, which are kept at the standard 32-bit precision. Our code is implemented using PyTorch and conducted on an Nvidia Titan RTX GPU.

A. Validation of Joint Sparsity

In our first experiment, we demonstrate the ability of the joint channel Haar wavelet transform to compress graph features that are taken from a network trained to classify the ModelNet-40 dataset [45]. That is, we extract learned feature maps \( f \) and measure the MSE between \( f \) and its compressed wavelet transform

\[
\frac{1}{n \cdot c_{in}} || f - W^{T} T^{T} W f ||_{2}^{2}
\]

(29)

for \( T \) that is applied with various compression percentages. Fig. 4 shows that our method is superior to standard (uniform)
quantization of feature maps and produces similar performance compared with an individual compression per channel, which, while accuracywise is preferred, has a tremendous cost. That is, such compression would require storing the indices of chosen elements per channel. For typical networks with dozens and hundreds of channels, this would defeat the purpose of the feature maps’ compression. More importantly, per-channel compression does not satisfy the property in Lemma 1, which we rely on for efficiency (compressed dense $1 \times 1$ convolutions). The approach of joint channel compression is suitable for both accuracy- and efficiencywise.

### B. Semi-Supervised Node Classification

In this section, we evaluate our method on three citation network datasets: Cora, Citeseer, and Pubmed [58]. The statistics of the datasets can be found in Table I. On each dataset, we use the standard training/validation/testing split as in [59], with 20 nodes per class for training, 500 validation nodes and 1000 testing nodes and follow the training scheme of [32], where we adopt the GCN [31] and GCNII [32] architectures and replace each convolution kernel with our analogous wavelet convolution. Those models are denoted as WGCN and WGCNII, respectively. We set the learning rate to 0.01, with a weight decay of $5 \cdot 10^{-4}$. The dropout is 0.60.7/0.5 on Cora/Citeseer/Pubmed, respectively. As in [32], we set the hyperparameters of WGCNII to a fixed $\alpha = 0.1$ for all the layers, and $\beta_l = \log((\lambda/l)+1)$ with $\lambda = 0.1$. To allow a fair comparison with various popular methods such as GCN, GCNII, and GAT, and other wavelet-based GNNs: HANet [26], DeepGWC [23], and M-GWNN [24], we use a two-layer network and do not perform quantization or compression. For JKNet [48] and Ince [47], we compare with the minimal number of layers that are reported. The results of our wavelet convolution can be seen in Table II, where we read a higher or same accuracy across all the datasets, compared with the considered methods.

Since the results reported in Tables II–IV are derived from networks with two layers, we also conduct a depth study, to ensure no over-smoothing occurs by the addition of our wavelet mechanism. To disentangle the influence of quantization or compression, and since the two latter are tangential to the influence of the addition of more layers, we compare the results with 32 bits (i.e., no feature maps quantization) and without wavelet compression. We present the results of our study in Fig. 5, where we see that our WGCNII does not over-smooth. In addition, we see that our results are superior or on par with those of GCNII.

Furthermore, we experiment with different compression ratios of the wavelet transform and feature map quantization.

---

**Table I**

| Benchmark | Classes | Label rate | Nodes | Edges | Features |
|-----------|---------|------------|-------|-------|----------|
| Cora      | 7       | 0.052      | 2708  | 5429  | 1433     |
| Citeseer  | 6       | 0.036      | 3327  | 4732  | 3703     |
| Pubmed    | 3       | 0.003      | 19717 | 44338 | 500      |

**Table II**

| Model       | Cora | Citeseer | Pubmed |
|-------------|------|----------|--------|
| DeepWalk    | 67.2 | 43.2     | 65.3   |
| Ince (4 layers) | 77.6 | 69.3     | 77.7   |
| ChebNet     | 81.2 | 69.8     | 74.4   |
| GCN         | 81.5 | 70.8     | 79.0   |
| JKNet       | 80.2 | 68.7     | 78.2   |
| GCNII       | 82.2 | 68.2     | 78.2   |
| HANet       | 81.9 | 70.1     | 79.3   |
| GWNII       | 82.8 | 71.7     | 79.1   |
| GAT         | 83.0 | 72.3     | 79.0   |
| NGCN        | 83.0 | 72.2     | 79.5   |
| DualGCN     | 83.5 | 72.4     | 79.3   |
| HGCN        | 84.2 | 72.9     | 79.1   |
| g-U-Nets    | 84.4 | 73.2     | 79.6   |
| DeepGWC     | 84.8 | 72.6     | 80.4   |
| M-GWNN      | 84.6 | 72.6     | 80.4   |
| UFGConv-R   | 83.6 | 71.0     | 79.4   |
| WGCN (ours) | 83.4 | 71.5     | 79.8   |
| WGCNII (ours) | 84.9 | 74.1     | 80.5   |

**Table III**

| Model       | Cornell | Texas | Wisconsin |
|-------------|---------|-------|-----------|
| GCN         | 52.7    | 52.2  | 45.9      |
| GAT         | 54.3    | 58.4  | 49.4      |
| APPNP       | 73.5    | 65.4  | 69.0      |
| JKNet (Drop) | 61.1   | 57.3  | 50.6      |
| Ince (Drop) | 61.6    | 57.8  | 74.1      |
| Geom-GCN    | 60.8    | 67.6  | 64.1      |
| GCNII       | 76.5    | 77.8  | 81.6      |
| H2GCN       | 82.7    | 84.9  | 87.6      |
| SD          | 86.5    | 85.9  | 89.4      |
| WGCNII (ours) | 88.9  | 85.2  | 89.3      |

---

Fig. 5. Depth study of WGCNII versus GCNII.
TABLE IV
ACCURACY (%) ON SEMI-SUPERVISED NODE CLASSIFICATION. W/F INDICATES THE WEIGHTS AND FEATURE MAPS, RESPECTIVELY.

| Model          | W/F | Comp. | Total-act. Comp. | Cora | Cite. | Pub. |
|---------------|-----|-------|-----------------|-----|-------|------|
| GCN [31]      | 8/8 | –     | × 4             | 82.5| 69.5  | 78.7 |
|               | 8/4 | –     | × 8             | 39.1| 26.1  | 40.7 |
|               | 8/2 | –     | × 16            | 20.7| 22.4  | 40.1 |
|               | 8/1 | –     | × 32            | 20.2| 20.3  | 33.0 |

| GCNII [32]    | 8/8 | –     | × 4             | 80.9| 69.8  | 80.0 |
|               | 8/4 | –     | × 8             | 31.9| 24.7  | 41.3 |
|               | 8/2 | –     | × 16            | 21.1| 18.3  | 40.7 |
|               | 8/1 | –     | × 32            | 20.0| 18.0  | 36.1 |

| QAT-GCN [15]  | 4/4 | –     | × 8             | 81.0| 71.3  | –    |
|               | 8/8 | –     | × 8             | 77.2| 64.1  | –    |

| nQAT-GCN [15] | 4/4 | –     | × 8             | 81.0| 70.7  | –    |

| DQ-GCN [15]   | 4/4 | –     | × 8             | 78.1| 65.8  | –    |

| SGQ-GAT (uniform) [14] | 32/4 | –     | × 8             | 81.1| –     | –    |

| SGQ-GCN (per-node) [14] | 32/1.2 | –     | × 26.6          | 81.7| 71.5  | 80.3 |

| WGCN (ours) [45] | 8/8 | × 1  | × 4             | 83.3| 71.1  | 80.0 |
|                 | 8/8 | × 2  | × 8             | 80.4| 70.3  | 80.1 |
|                 | 8/8 | × 4  | × 16            | 78.4| 69.6  | 79.3 |
|                 | 8/8 | × 8  | × 32            | 74.3| 62.9  | 77.0 |

| WGCNII (ours)   | 8/8 | × 1  | × 4             | 84.6| 74.0  | 80.5 |
|                 | 8/8 | × 2  | × 8             | 84.7| 73.7  | 80.6 |
|                 | 8/8 | × 4  | × 16            | 83.0| 73.1  | 79.1 |
|                 | 8/8 | × 8  | × 32            | 81.8| 71.8  | 77.0 |

We compare our method to several recent works such as QAT [15] and SGQuant [14]. We note that such methods apply component and layerwise as well as topology-aware quantization, which are considerably more sophisticated than our uniform (layer-, channel-, and nodewise) quantization. Such algorithmic upgrades can have significant costs on low computational resource edge devices. We apply our uniform quantization to the baseline methods of GCN and GCNII (followed by a rerun evaluation) and their counterparts WGCN and WGCNII. The results are reported in Table IV, where two contributions are portrayed. First, we see that compared with the GCN and GCNII, their wavelet variants WGCN and WGCNII (ours) yield significantly higher accuracy, even at larger compression rates. In addition, we see that compared with the quantization-designated methods, we obtain comparable or higher accuracy per compression ratio.

C. Fully Supervised Node Classification

We evaluate our method on fully supervised datasets Cora, Cornell, Texas, and Wisconsin [55], with the same training/validation/test splits of 48%, 32%, 20%, and 10% random splits from [55]. For each dataset, we use the hyperparameters reported in [32]. We first compare our nonquantized nor compressed WGCNII with recent methods such as GAT [49], APPNP [54], DropEdge [47], GCNII [32], H2GCN [56], and SD [57] in Table III. We find that our WGCNII performs better or on par with those methods. Then, we compare quantized and compressed, two-layer GCNII and WGCNII, where we see similar results when the feature maps’ compression is under ×16, and significantly better accuracy as the compression increases compared with more aggressive quantization—see Table V. In addition, to validate our method on a large graph dataset, we experiment with Ogbn-Arxiv [60] and compare our WGCNII with GCNII, using 16 layers.3 Also, we note that on heterophilic datasets like Cornell, we obtain better accuracy with ×8 rather than ×4 compression rate. The improved accuracy can be explained by that the compression of the wavelet transform also acts as an implicit denoiser by removing insignificant information as described in Section IV-C and [61].

D. Shape Classification

We demonstrate our method for the 3-D shape classification benchmark ModelNet40 [45]. The dataset consists of 12,311 CAD meshes, across 40 categories, with 9,843 and 2,468 samples in the train and test sets, respectively. We randomly sample 1024 points from each mesh and normalize the points to the unit cube. We follow [6] and construct a graph from each point cloud, using the kNN algorithm with k = 20.

Our network adopts the popular DGCNN architecture [6], in which each DGCCN block is replaced with the equivalent wavelet block from (24) and is denoted by WDGCNNV. Also, we examine the performance with the alternative convolution from (22), denoted by WDGCNNV. Our training scheme is as described in [32]. The author’s code and settings can be found at https://github.com/chennM/GCNII.
TABLE VII

| Model   | Feat. | Wav.  | Total-act. | Instance |
|---------|-------|-------|------------|----------|
|         | bits  | comp. | comp.      | mIoU     |
| DGCNN   | 8     | -     | \(\times 4\) | 84.1     |
|         | 4     | -     | \(\times 8\) | 80.7     |
|         | 2     | -     | \(\times 16\) | 75.6     |
|         | 1     | -     | \(\times 32\) | 38.1     |
| WDGCNNV1 | 8     | \(\times 1\) | \(\times 4\) | 84.5     |
|         | 8     | \(\times 2\) | \(\times 8\) | 83.6     |
|         | 8     | \(\times 4\) | \(\times 16\) | 82.0     |
|         | 8     | \(\times 8\) | \(\times 32\) | 81.7     |
|         | 8     | \(\times 16\) | \(\times 64\) | 77.8     |
| WDGCNNV2 | 8     | \(\times 1\) | \(\times 4\) | 84.2     |
|         | 8     | \(\times 2\) | \(\times 8\) | 83.6     |
|         | 8     | \(\times 4\) | \(\times 16\) | 83.0     |
|         | 8     | \(\times 8\) | \(\times 32\) | 80.5     |
|         | 8     | \(\times 16\) | \(\times 64\) | 78.1     |

TABLE VIII

| Model   | Feat. | Wav. comp. | Total-ct. comp. | mIoU   |
|---------|-------|------------|-----------------|--------|
| DGCNN   | 8     | \(\times 4\) | 56.5            |
|         | 4     | \(\times 8\) | 55.8            |
|         | 2     | \(\times 16\) | 53.1            |
|         | 1     | \(\times 32\) | 32.1            |
| WDGCNNV1 | 8     | \(\times 2\) | \(\times 8\)   | 56.8   |
|         | 8     | \(\times 4\) | \(\times 16\) | 56.0   |
|         | 8     | \(\times 8\) | \(\times 32\) | 54.3   |
|         | 8     | \(\times 16\) | \(\times 64\) | 51.0   |

TABLE IX

| Model   | Feat. | Wav. comp. | Wav. levels | Overall acc. % |
|---------|-------|------------|-------------|----------------|
| WDGCNNV1 | 8     | \(\times 8\) | 1            | 93.8           |
|         | 8     | \(\times 8\) | 3            | 93.4           |
|         | 8     | \(\times 8\) | 5            | 92.1           |

in [6], where we use the Adam [62] optimizer with a learning rate of 0.01 and a step scheduler with a decrease factor of 0.5 every 20 epochs, for 250 epochs. All our evaluations are reruns. The results are summarized in Table VI, where it is noted that our method outperforms feature maps’ quantizations, revealing a healthy margin when aggressive quantization (i.e., 2 bits or 1 bit) is applied.

E. Semantic and Part Segmentation

In this section, we evaluate our approach on two different segmentation datasets: the Shapenet part segmentation [63] and stanford large-scale 3D indoor spaces (S3DIS) [64].

The Shapenet part segmentation [63] dataset includes 16 881 3-D point clouds, across 16 shape categories, with a total of 50 part annotation classes (each shape contains two to six parts). We sample 2048 points from each shape, where the goal is to correctly classify these part annotations per point. The train, validation, and test sets are split according to [63].

The S3DIS [64], is another semantic segmentation benchmark. The dataset includes 3-D scans of 272 rooms from six different areas. Each point in these 3-D scans is annotated with one of the 13 semantic classes, and we wish to classify each point correctly. We adopt the preprocessing steps of splitting each room into \(1\times 1\) m blocks and sample 4096 points from the 3-D scan. Each point is represented by a 9-D vector (\(XYZ,\) RGB, normalized spatial coordinates). The train, validation, and test splits are the same as in [65] and follow the sixfold evaluation protocol [64].

For both the datasets, we use the same architecture, training, and testing schemes as in DGCNN, only replacing each edge-conv block [as in (23)] with ours from (24). For constructing the graph, we use \(k\)NN with \(k = 20, 40\) for Shapenet and S3DIS, respectively. Our results on Shapenet part segmentation are provided in Table VII, and for S3DIS in Table VIII, and for both the datasets, a rerun of DGCNN is performed. We note similar or slightly better performance when the compression ratio is under \(\times 8\). More significantly, we see that for extreme compression of over \(\times 16\), our WDGCNNV1 maintains high accuracy, compared with DGCNN where a major degradation of the accuracy takes place. Also, for ShapeNet, we report the accuracy of WDGCNNV1, where we see similar performance to WDGCNNV1, with less transformations.

F. Ablation Study

We delve on the influence of the number of wavelet levels on the accuracy for the ModelNet10 [45] dataset. All the configurations use 8-bit quantization and \(\times 8\) wavelet compression, and we compare one, three, and five levels in (11). Table IX summarizes the results and shows that three levels are the best option for this scenario.

G. Inference Times on a GPU Using a Custom Implementation

We start by analyzing the number of parameters and required floating-point operations (FLOPs) in our wavelet counterparts in Table X, where it is evident that our network requires significantly fewer FLOPs as the compression rate is increased. Note that the number of parameters is identical to the baseline model as our method focuses on the size of the feature maps rather than the number of parameters. To demonstrate the effectiveness of our implementation, we compare the inference time of a standard \(1\times 1\) convolution and our wavelet-compressed convolution in (15) in Fig. 6. Our time measurements in Fig. 6(a) show that using our wavelet compression mechanism with a baseline of 512 channels, a speedup of up to 51.3% is achieved. Also, Fig. 6(b) shows that as more channels are added, the gap between our wavelet-compressed convolution layer and a baseline \(1\times 1\) convolution further increases. We note that we accelerate the \(1\times 1\) convolutions only and not the graph propagation, as it is not the aim of our method. The computation graph node feature propagation is required both in our method and other GCNs and requires a similar time to a \(1\times 1\) convolution in our implementation. The timings were obtained using PyTorch and CUDA kernels using the ctypes package running on an NVIDIA RTX 2060Ti GPU and averaged over 400 consecutive runs.

VI. CONCLUSION

We propose an efficient compression method for GCNs, based on the combination of compressed Haar wavelet and
TABLE X
NUMBER OF PARAMETERS AND (FLOPs of OGRN-ARXIV USING GCNN AND OUR WGCNN WITH 16 LAYERS AND 128 CHANNELS

| Model     | Compression rate | Parameters [10^3] | FLOPs [10^9] |
|-----------|------------------|-------------------|--------------|
| GCNN      | ×2               | 283.6             | 47.19        |
| WGCNN     | ×4               | 283.6             | 13.91        |
|           | ×8               | 283.6             | 8.36         |
|           | ×16              | 283.6             | 5.58         |

Fig. 6. Inference times of our wavelet-compressed convolution compared with the baseline 1 × 1 convolution with 65 536 nodes. (a) Run-time versus compression ratio with 512 channels. (b) Run-time versus number of channels with ×4 compression rate.

quantization methods. We conduct an extensive set of experiments for both the geometrical and nongeometrical graph datasets, where it is demonstrated that our method is often associated with large performance gains, both in accuracy and computational efficiency terms. Furthermore, we demonstrate that 8-bit and 4-bit quantization can be applied to further reduce memory and computational costs, with minimal loss in performance, also while using baseline methods.

Our method can reduce inference times and computational costs related to deploying GCNs in real-world applications, such as LiDAR-based point cloud segmentation for autonomous vehicles. It is most beneficial for devices where the multiplication-and-add operations are the main computational bottleneck of the GCN application.

In our future work, we plan to see whether the low-precision performance can be improved using more advanced quantization schemes such as nonuniform and mixed-precision methods (which use different bit allocations for different layers), as well as topology-aware quantization. In addition, our method can be combined with various weight pruning methods for further savings, as our method is focused on feature map compression.

ACKNOWLEDGMENT

The authors would like to thank Tal Kopetz for valuable conversations on the efficiency of CNNs on dedicated edge hardware.

REFERENCES

[1] J. Qiu, J. Tang, H. Ma, Y. Dong, K. Wang, and J. Tang, “DeepInf: Social influence prediction with deep learning,” in Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining, 2018, pp. 2110–2119.
[2] C. Li and D. Goldwasser, “Encoding social information with graph convolutional networks for political perspective detection in news media,” in Proc. 57th Annu. Meeting Assoc. Comput. Linguistics, 2019, pp. 2594–2604.
[3] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, “Graph convolutional neural networks for web-scale recommender systems,” in Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining, Jul. 2018, pp. 974–983.
[4] M. Eliasof, T. Boesen, E. Haber, C. Keasar, and E. Treister, “Mimetic neural networks: A unified framework for protein design and folding,” Frontiers Bioinf., vol. 2, p. 39, May 2022.
[5] F. Monti, D. Boscaini, J. Masci, E. Rodola, J. Svoboda, and M. M. Bronstein, “Geometric deep learning on graphs and manifolds using mixture model CNNs,” in Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR), Jul. 2017, pp. 5425–5434.
[6] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon, “Dynamic graph CNN for learning on point clouds,” ACM Trans. Graph., vol. 38, no. 5, pp. 1–12, Oct. 2019.
[7] M. Eliasof and E. Treister, “DiffGCNN: Graph convolutional networks via differential operators and algebraic multigrid pooling,” in Proc. 34th Conf. Neural Inf. Process. Syst., 2020, pp. 1–12.
[8] J. Luo, J. Wu, and W. Lin, “ThiNet: A filter level pruning method for deep neural network compression,” in Proc. IEEE Int. Conf. Comput. Vis. (ICCV), Oct. 2017, pp. 5068–5076.
[9] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, “Learning structured sparsity in deep neural networks,” in Proc. Adv. Neural Inf. Process. Syst., 2016, pp. 2074–2082.
[10] A. Zhou et al., “Learning N:M fine-grained structured sparse neural networks from scratch,” in Proc. Int. Conf. Learn. Represent. OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=K9bw7vqg-ps
[11] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, “Quantized neural networks: Training neural networks with low precision weights and activations,” J. Mach. Learn. Res., vol. 18, pp. 187:1–187:30, 2017.
[12] S. K. Eisser, J. L. McKinstry, D. Bablani, R. Appuswamy, and D. S. Modha, “Learned step size quantization,” in Proc. Int. Conf. Learn. Represent. (ICLR). OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=kkgO6VXDS
[13] Y. Li, X. Dong, and W. Wang, “Additive powers-of-two quantization: An efficient non-uniform discretization for neural networks,” in Proc. Int. Conf. Learn. Represent. (ICLR). OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=BkgXT24tDS
[14] B. Feng, Y. Wang, X. Li, S. Yang, X. Peng, and Y. Ding, “SQQuant: Squeezing the last bit on graph neural networks with specialized quantization,” in Proc. IEEE 32nd Int. Conf. Tools Artif. Intell. (ICTAI), Nov. 2020, pp. 1044–1052.
[15] S. A. Tailor, J. Fernandez-Marques, and N. D. Lane, “Degree-quant: Quantization-aware training for graph neural networks,” in Proc. Int. Conf. Learn. Represent. (ICLR). OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=NSBrFgJAHg
[16] Y. Zhao, D. Wang, D. Bates, R. Mullins, M. Jannik, and P. Liu, “Learned low precision graph neural networks,” CoRR, vol. abs/2009.09232, 2020. [Online]. Available: https://arxiv.org/abs/2009.09232
[17] Z. Tang, P. Xi, K. Li, and D. N. Metaxas, “Towards efficient U-Nets: A coupled and quantized approach,” IEEE Trans. Pattern Anal. Mach. Intell., vol. 42, no. 8, pp. 2038–2050, Aug. 2020.
[18] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, “Model compression and acceleration for deep neural networks: The principles, progress, and challenges,” IEEE Signal Process. Mag., vol. 35, no. 1, pp. 126–136, Jan. 2018.
[19] I. Daubechies, Ten Lectures on Wavelets. Philadelphia, PA, USA: SIAM, 1992.
[20] M. Rabbani, “JPEG2000: Image compression fundamentals, standards and practice,” J. Electron. Imag., vol. 11, no. 2, p. 286, 2002.
[21] D. K. Hammond, P. Vandergheynst, and R. Gribonval, “Wavelets on graphs using eigenvectors and multiresolution,” Appl. Comput. Harmon. Anal., vol. 30, no. 2, pp. 129–150, Mar. 2011.
[22] B. Xu, H. Shen, Q. Cao, Y. Qiu, and X. Cheng, “Graph wavelet neural network,” in Proc. 7th Int. Conf. Learn. Represent. (ICLR). OpenReview.net, 2019. [Online]. Available: https://openreview.net/forum?id=H1ewdR5tQ
[23] J. Wang and Z. Deng, “A deep graph wavelet convolutional neural network for semi-supervised node classification,” 2021, arXiv:2102.09780.
[24] W. Zheng, F. Qian, S. Zhao, and Y. Zhang, “M-GWNN: Multi-granularity graph wavelet neural networks for semi-supervised node classification,” Neurocomputing, vol. 455, pp. 524–537, Sep. 2021.
[25] I. Ram, M. Elad, and I. Cohen, “Generalized tree-based wavelet transform,” IEEE Trans. Signal Process., vol. 59, no. 9, pp. 4199–4209, Sep. 2011.
[26] M. Li, Z. Ma, Y. G. Wang, and X. Zhuang, “Fast Haar transforms for graph neural networks,” Neural Netw., vol. 128, pp. 188–198, Aug. 2020.
[27] Y. G. Wang, M. Li, Z. Ma, G. Montufar, X. Zhuang, and Y. Fan, “Haar graph pooling,” in Proc. Int. Conf. Mach. Learn. (ICML), 2020, pp. 9952–9962.
Z. Wu et al., “3D ShapeNets: A deep representation for volumetric shapes,” in Proceedings of Machine Learning Research, vol. 115, 2019, pp. 841–851.

Y. Rong, W. Huang, T. Xu, and J. Huang, “DropEdge: Towards deep graph convolutional networks on node classification,” in Proc. Int. Conf. Learn. Represent. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=SJU4ayYgY

P. Li et al., “Efficient graph convolutional networks for point cloud handling,” in Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV), Oct. 2021, pp. 1608–1706.

Y. Li et al., “Towards efficient graph convolutional networks for point cloud handling,” in Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV), Oct. 2021, pp. 3732–3742.

L. Wang, X. Dong, Y. Wang, L. Liu, W. An, and Y. Guo, “Learnable look-up table for neural network quantization,” in Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR), Jun. 2022, pp. 12413–12423.

Z. Wu et al., “3D ShapeNets: A deep representation for volumetric shapes,” in Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR), Jun. 2015, pp. 1912–1920.

F. Perozzi, R. Al-Rfou, and S. Skiena, “DeepWalk: Online learning of social representations,” in Proc. 26th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining, Aug. 2014, pp. 701–710.

Y. Rong, W. Huang, T. Xu, and J. Huang, “DropEdge: Towards deep graph convolutional networks on node classification,” in Proc. Int. Conf. Learn. Represent. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=Hkts9FkP

K. Xu, C. Li, Y. Tian, T. Sonobe, K.-I. Kawarabayashi, and S. Jegelka, “Representation learning on graphs with jumping knowledge networks,” in Proc. Mach. Learn. Res., in Proceedings of Machine Learning Research, vol. 80, J. Dy and A. Krause, Eds. Stockholim, Sweden, Jul. 2018, pp. 5453–5462.

P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, “Graph attention networks,” in Proc. Int. Conf. Learn. Represent., 2018, pp. 1–12.

S. Abu-El-Haija, A. Kapoor, B. Perozzi, and J. Lee, “NGCN: Multi-scale graph convolution for semi-supervised node classification,” in Proc. 35th Conf. Uncertainty Artif. Intell., in Proceedings of Machine Learning Research, vol. 115, 2019, pp. 841–851.

C. Zhuang and Q. Ma, “Dual graph convolutional networks for graph-based semi-supervised classification,” in Proc. World Wide Web Conf., P. Chapan, F. Gandon, M. Almals, and P. G. Ieporiots, Eds., 2018, pp. 499–508.

F. Hu, Y. Zhu, S. Wu, L. Wang, and T. Tan, “Hierarchical graph convolutional networks for semi-supervised node classification,” in Proc. 28th Int. Joint Conf. Artif. Intell., J. Kraus, Ed., Aug. 2019, pp. 4532–4539.

Moshé Eliassof received the B.Sc. degree in computer science from the Technion—Israel Institute of Technology, Haifa, Israel, in 2017. He is currently pursuing the Ph.D. degree with the Department of Computer Science, Ben-Gurion University of the Negev (BGU), Be’er-Sheva, Israel.

His main research interests are graph neural networks, machine and deep learning, and inverse problems.

Benjamin J. Bodner was born in Zichron Yaakov, Israel, in 1991. He received the B.Sc. degree in physics from the Ben Gurion University of the Negev, Beer Sheva, Israel, in 2018, the M.Sc. degree in physics from Brown University, Providence, RI, USA, in 2022, and the M.Sc. degree in computer science from the Ben Gurion University of the Negev (BGU), in 2023. His current research interests include computer vision, machine learning, and optimization.

Eran Treister received the Ph.D. degree in computer science from the Technion—Israel Institute of Technology, Haifa, Israel, in 2014. He was a Post-Doctoral Researcher with The University of British Columbia, Vancouver, BC, Canada. He is currently an Associate Professor with the Department of Computer Science, Ben-Gurion University of the Negev, Be’er-Sheva, Israel. His current research interests include scientific computing, computational physics, optimization methods, and deep learning.

Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.