Adaptive Filters and Aggregator Fusion for Efficient Graph Convolutions

Shyam A. Tailor [1] **Felix L. Opolka** [1] **Pietro Liò** [1] **Nicholas D. Lane** [1 2]

Abstract

Training and deploying graph neural networks (GNNs) remains difficult due to their high memory consumption and inference latency. In this work we present a new type of GNN architecture that achieves state-of-the-art performance with lower memory consumption and latency, along with characteristics suited to accelerator implementation. Our proposal uses memory proportional to the number of vertices in the graph, in contrast to competing methods which require memory proportional to the number of edges; we find our efficient approach actually achieves higher accuracy than competing approaches across 5 large and varied datasets against strong baselines. We achieve our results by using a novel *adaptive filtering* approach inspired by signal processing; it can be interpreted as enabling each vertex to have its own weight matrix, and is not related to attention. Following our focus on efficient hardware usage, we propose *aggregator fusion*, a technique to enable GNNs to significantly boost their representational power, with only a small increase in latency of 19% over standard sparse matrix multiplication. Code and pretrained models can be found at this URL: `https://github.com/shyam196/egc`.

1. Introduction

The development of hardware-efficient techniques is key to the deployment of deep learning. We have seen the deployment of convolutional neural networks (CNNs) to enable previously unthinkable applications at the edge, due to innovation at the hardware and algorithmic level. Recently, we have seen research efforts aimed at tackling the deployment challenges associated with language models in both the data center and at the edge (Tay et al., 2020; Iandola et al., 2020). Research into efficiency often focuses on hardware-software co-design: the development of techniques to enable the software to take better advantage of the hardware, and vice-versa. There are several facets to this field (Sze et al., 2020). Usage of low precision arithmetic is one example: representing values with lower bit-widths enables us to compress models substantially, and decrease training and inference latency when we are memory-bandwidth limited. Another common approach is pruning, where we remove weights from the model; this will reduce model sizes, and may enable inference acceleration. A key area, however, is *designing neural network architectures with efficiency as a design goal*. This requires domain-specific knowledge, and approaches may not be directly transferable from one domain to another. A dominant approach for CNNs is to use separable convolutions, which can provide significant latency reductions in exchange for a small loss in accuracy (Iandola et al., 2016; Howard et al., 2017). For Transformers (Vaswani et al., 2017), there have been many proposals to reduce the memory consumption required for self-attention from $\mathcal{O}(n^2)$, where n is the number of tokens in the sequence (Tay et al., 2020).

Graph Neural Networks (GNNs) have emerged as an effective way to build models over arbitrarily structured data, with successes in many different application domains. For example, recent work has shown that they can be applied to physical simulations (Pfaff et al., 2020; Sanchez-Gonzalez et al., 2020). There has also been success on computer vision tasks: GNNs can deliver high performance on point cloud data (Shi & Rajkumar, 2020) and for feature matching across images (Sarlin et al., 2020). Code analysis is another application domain where GNNs have found success (Guo et al., 2020; Allamanis et al., 2017). Our work aims to enable these applications, and many more, by investigating approaches to design GNN architectures that enable us to obtain high accuracy without requiring large increases in memory consumption or latency.

Our work makes the following contributions:

- We propose a new GNN architecture, Efficient Graph Convolution (EGC), which does not require trading accuracy for runtime memory or latency reductions. Our proposal's memory consumption is linear in the number of vertices in the graph—not the number of

[1]Department of Computer Science & Technology, University of Cambridge, United Kingdom [2]Samsung AI Center, Cambridge, United Kingdom. Correspondence to: Shyam A. Tailor <sat62@cam.ac.uk>.

Preprint, Copyright 2021 by the author(s).

edges. We achieve our results through a novel adaptive filtering approach, which has no correspondence to attention. Our architecture is a *drop-in replacement* on a wide variety of tasks.

- We make hardware considerations a core aspect of our architecture design. Our architecture is well suited to existing accelerator designs, while offering substantially better accuracy than existing approaches. Further to this, we propose a novel technique, *aggregator fusion*, to further accelerate our architecture at training and inference time.
- We provide a rigorous evaluation of our architecture across 5 large graph datasets covering both transductive and inductive use-cases. We cover application domains ranging from citation graphs through to code analysis and molecular property prediction, and demonstrate that our approach consistently achieves better results than strong baselines.

2. Background

In this section we will discuss hardware-software co-design techniques that are commonly used for neural networks. We will then discuss GNNs, and existing attempts to improve their efficiency and scalability.

2.1. Hardware-Software Co-Design for Deep Learning

Several of the popular approaches for co-design have already been described in the introduction: quantization, pruning, and careful architecture design are all common for CNNs and Transformers. In addition to enabling better performance to be obtained from general purpose processors such as CPUs and GPUs, these techniques are also essential for maximizing the return from specialized accelerators; while it may be possible to improve performance over time due to improvements in CMOS technology, further improvements plateau without innovation at the algorithmic level (Fuchs & Wentzlaff, 2019). As neural network architecture designers, we cannot simply rely on improvements in hardware to make our proposals viable for real-world deployment.

2.2. Graph Neural Networks

Many GNN architectures can be viewed as a generalization of CNN architectures to the irregular domain: as in CNNs, representations at each node are built based on the local neighborhood using parameters that are shared across the graph. GNNs differ as we cannot make assumptions about the the size of the neighborhood, or the ordering. One common framework used to define GNNs is the message passing neural network (MPNN) paradigm (Gilmer et al., 2017). A graph $\mathcal{G} = (V, E)$ has node features $\mathbf{X} \in \mathbb{R}^{N \times F}$, adjacency matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ and optionally D-dimensional edge features $\mathbf{E} \in \mathbb{R}^{E \times D}$. We define a function ϕ that calculates messages from node u to node v, a differentiable and permutation-invariant aggregator $\oplus$, and an update function γ to calculate representations at layer $l+1$: $\mathbf{h}_{l+1}^{(i)} = \gamma(\mathbf{h}_l^{(i)}, \oplus_{j \in \mathcal{N}(i)}[\phi(\mathbf{h}_l^{(i)}, \mathbf{h}_l^{(j)}, \mathbf{e}_{ij})])$. Propagation rules for architectures we evaluate against are presented in Table 1. The reader should note that we can also implement GCN and GIN using sparse matrix multiplication (SpMM) rather than using the node-wise formulation.

Scaling and Deploying GNNs. While GNNs have seen success across a wide range of domains, there remain challenges associated with scaling and deploying them. Graph sampling is one approach to scaling training for extremely large graphs which will not fit in memory. Rather than training over the full graph, each iteration is run over a sampled sub-graph; approaches vary in whether they sample node-wise (Hamilton et al., 2017), layer-wise (Chen et al., 2018a; Huang et al., 2018), or sub-graphs (Zeng et al., 2019; Chiang et al., 2019). Other works have investigated distributed GNN training (Jia et al., 2020). Some works have proposed architectures that are designed for these large graphs: SIGN (Rossi et al., 2020) is explicitly designed as a shallow architecture, as all the graph operations are done as a pre-processing step.

For many applications, *deploying* our models is the challenge—not scaling them at training time. Although semi-supervised learning on large graphs is a popular task in the literature, it represents only a small slice of real world applications, as described in the introduction. The techniques for scaling training are not generally applicable to tasks where we need to generalize to unseen graphs at test time, as in these tasks the graphs tend to be orders of magnitude smaller. One approach to reduce latency and memory consumption is to learn a shallow GNN (Yan et al., 2020a); however, this proposal only applies to the case where we are adding new nodes to a previously seen graph, and not to cases where we need to generalize to unseen graphs. Other work includes applying neural architecture search to arrange existing GNN layers (Zhao et al., 2020), or building quantization techniques for GNNs (Tailor et al., 2021).

To date, the graph community has focused on designing architectures that prioritize accuracy as the primary metric, but as the computer vision and NLP communities have shown, it is possible to make a small trade in accuracy for a large boost in efficiency. In this work, we propose an architecture that is not only far superior to competing architectures in terms of memory consumption and inference latency, but which also achieves better accuracy on a wide variety of tasks. In contrast to competing architectures, which require $\mathcal{O}(|E|)$ memory, our proposal requires only $\mathcal{O}(|V|)$ memory. Our approach is *designed for the hardware*: it is suited to existing accelerators, and due to our proposed aggregator

Table 1. Propagation rules for GNN architectures we compare against in this work; rules are provided using node-wise formulations. We evaluate against popular architectures, and a recent proposal that has achieved state-of-the-art performance, PNA. We also provide asymptotic memory consumption: in general, architectures that require explicit materialization of messages require $\mathcal{O}(|E|)$ memory.

Method	Propagation Rule	Memory	Notes
GCN (Kipf & Welling, 2017)	$\mathbf{y}^{(i)} = \mathbf{\Theta} \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{1}{\sqrt{\deg(i)\deg(j)}} \mathbf{x}^{(j)}$	$\mathcal{O}(\|V\|)$	Formally defined for undirected graphs with self-loops; motivated by graph signal processing.
GAT (Veličković et al., 2018)	$\mathbf{y}^{(i)} = \alpha_{i,i} \mathbf{\Theta} \mathbf{x}^{(i)} + \sum_{j \in \mathcal{N}(i)} \alpha_{i,j} \mathbf{\Theta} \mathbf{x}^{(j)}$	$\mathcal{O}(\|E\|)$	Attention coefficients calculated using dot-product attention: $\alpha_{i,j} = \frac{\exp\left(\mathrm{LeakyReLU}\left(\mathbf{a}^\top [\mathbf{\Theta}\mathbf{x}^{(i)} \parallel \mathbf{\Theta}\mathbf{x}^{(j)}]\right)\right)}{\sum_{k \in \mathcal{N}(i) \cup \{i\}} \exp\left(\mathrm{LeakyReLU}\left(\mathbf{a}^\top [\mathbf{\Theta}\mathbf{x}^{(i)} \parallel \mathbf{\Theta}\mathbf{x}^{(k)}]\right)\right)}$. Common to define multiple attention heads and concatenate.
GIN (Xu et al., 2019)	$\mathbf{y}^{(i)} = f_{\mathbf{\Theta}}[(1+\epsilon)\mathbf{x}^{(i)} + \sum_{j \in \mathcal{N}(i)} \mathbf{x}^{(j)}]$	$\mathcal{O}(\|V\|)$	f is a learnable function, typically parameterized as an MLP or linear layer; ϵ may be fixed or learned.
MPNN (Gilmer et al., 2017)	$\mathbf{y}^{(i)} = U(\mathbf{x}^{(i)}, \bigoplus_{j \in \mathcal{N}(i)} M(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}, \mathbf{e}_{ij}))$	$\mathcal{O}(\|E\|)$	U, M typically defined as linear layers acting on concatenated features; $\bigoplus$ may be any valid aggregator, typically sum or max.
PNA (Corso et al., 2020)	$\mathbf{y}^{(i)} = U(\mathbf{x}^{(i)}, \bigoplus_{j \in \mathcal{N}(i)} M(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}, \mathbf{e}_{ij}))$	$\mathcal{O}(\|E\|)$	Similar to MPNN, but with $\bigoplus$ defined to use 4 aggregators (mean, standard deviation, max, and min) scaled by 3 different functions of node degree, resulting in 12 different aggregations.

fusion technique, offers a large jump in representational power with only a minor increase in latency.

3. Our Architecture: Efficient Graph Convolution (EGC)

In this section we provide a description of our approach; we delay interpretation and discussion until the next section. We present two versions: *EGC-S*, which uses a single aggregator, and *EGC-M* which generalizes our approach by incorporating multiple aggregation functions, and which can be accelerated by our *aggregator fusion* approach. Our overall approach is visualized in Figure 1.

3.1. Architecture Description

For a layer with in-dimension of F and out-dimension of F' we use B basis weights $\mathbf{\Theta}_i \in \mathbb{R}^{F' \times F}$. We compute the output for node i by calculating combination weighting coefficients $\mathbf{w}^{(i)} \in \mathbb{R}^B$ *per node*, and weighting the results of each aggregation using the different basis weights $\mathbf{\Theta}_i$. We calculate $\mathbf{w}^{(i)} = \mathbf{\Phi}\mathbf{x}^{(i)} + \mathbf{b}$, where $\mathbf{\Phi} \in \mathbb{R}^{B \times F}$ and $\mathbf{b} \in \mathbb{R}^B$ are weight and bias parameters associated with calculating the combination weighting coefficients. Then our layer output for node i is:

$$\mathbf{y}^{(i)} = \sum_{b=1}^{B} w_b^{(i)} \sum_{j \in \mathcal{N}(i)} \alpha(i,j) \mathbf{\Theta}_b \mathbf{x}^{(j)} \tag{1}$$

Where $\alpha(i,j)$ is some function of nodes i and j, and $\mathcal{N}(i)$ denotes the in-neighbours of i. A popular method pioneered by GAT to boost representational power is to represent α using a learned function of the two nodes' representations. While this enables anisotropic treatment of neighbors, and can boost performance, it necessarily results in memory consumption of $\mathcal{O}(|E|)$ due to messages needing to be explicitly materialized, and complicates hardware implementation for accelerators. If we choose a representation for α that is not a function of the node representations—such as 1 to recover the add aggregator used by GIN, or $1/\sqrt{\deg(i)\deg(j)}$ to recover symmetric normalization used by GCN—then we can implement our message propagation phase using SpMM, and avoid explicitly materializing each message. In this work, we assume $\alpha(i,j)$ to be symmetric normalization as used by GCN unless otherwise stated; we use this normalization as it is known to offer strong results across a variety of tasks; formal justification is provided in section 4.2.

Adding Heads. We can further extend our layer through the addition of multiple heads, as used in architectures such as GAT or Transformers (Vaswani et al., 2017). These heads share the basis weights, but apply different weighting coefficients per head; we find in practice adding this extra degree of freedom aids regularization when an appropriate number of bases are chosen ($B \leq H$, section 5.3). To normalize the output dimension, we change the basis weight matrices to have dimensions $\frac{F'}{H} \times F$, where H is the number of heads. Using $\|$ as the concatenation operator, and making the use of symmetric normalization explicit, we obtain the **EGC-S** layer:

$$\mathbf{y}^{(i)} = \Big\Vert_{h=1}^{H} \sum_{b=1}^{B} w_{h,b}^{(i)} \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{1}{\sqrt{\deg(i)\deg(j)}} \mathbf{\Theta}_b \mathbf{x}^{(j)} \tag{2}$$

3.2. Boosting Representational Capacity

Recent work by Corso et al. (2020) has shown theoretically and empirically that using only a single aggregator is suboptimal. Instead, it is better to combine several different aggregators. In Equation (2) we defined our layer to use only a summation-derived aggregator. To improve performance, we propose applying different aggregators to the represen-

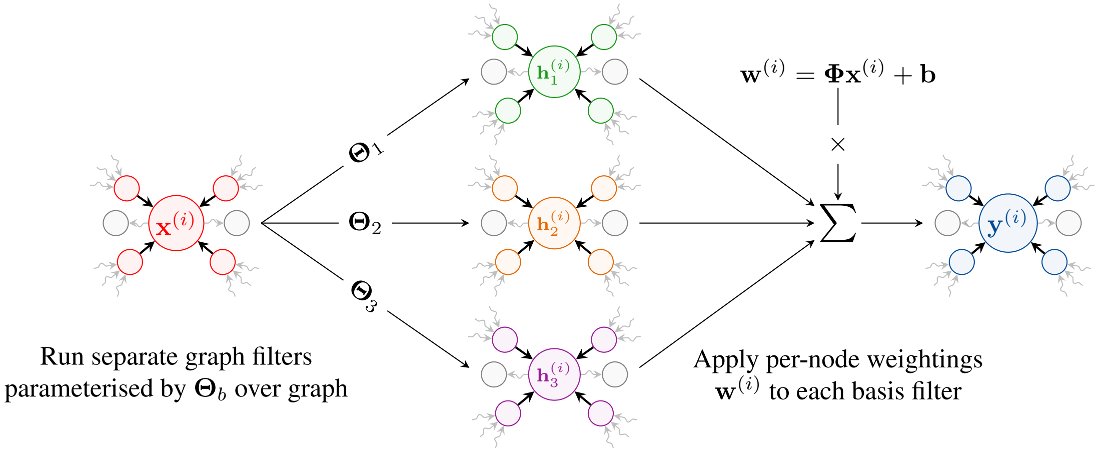

$\mathbf{x}^{(i)}$
$\mathbf{\Theta}_1$
$\mathbf{\Theta}_2$
$\mathbf{\Theta}_3$
$\mathbf{h}_1^{(i)}$
$\mathbf{h}_2^{(i)}$
$\mathbf{h}_3^{(i)}$
$\mathbf{w}^{(i)} = \mathbf{\Phi}\mathbf{x}^{(i)} + \mathbf{b}$
×
$\sum$
$\mathbf{y}^{(i)}$
Run separate graph filters parameterised by $\mathbf{\Theta}_b$ over graph
Apply per-node weightings $\mathbf{w}^{(i)}$ to each basis filter

Figure 1. Visual representation of our EGC-S layer. In this visualization we have 3 basis filters (i.e. $B = 3$), which are combined using per-node weightings $\mathbf{w}$. This simplified figure does not show the usage of heads, or multiple aggregators, as used by EGC-M.

tations calculated by $\mathbf{\Theta}_b\mathbf{x}^{(j)}$. The choice of aggregators could include different variants of summation aggregators e.g. mean or unweighted addition as opposed to symmetric normalization that was proposed in the previous section. Alternatively, we can use aggregators such as standard deviation, min or max which are not based on summation. If we have a set of aggregators $\mathcal{A}$, we can extend Equation (2) to obtain our **EGC-M** layer:

$$\mathbf{y}^{(i)} = \bigg\Vert_{h=1}^{H} \sum_{\oplus\in\mathcal{A}} \sum_{b=1}^{B} w_{h,\oplus,b}^{(i)} \bigoplus_{j\in\mathcal{N}(i)\cup\{i\}} \mathbf{\Theta}_b \mathbf{x}^{(j)} \tag{3}$$

where $\oplus$ is an aggregator. With this formulation, we are reusing the same messages we have calculated as before—but we are applying several aggregation functions to them at the same time.

Aggregator Fusion. At first glance it would appear that having $|\mathcal{A}|$ aggregators would cause inference latency to increase by approximately $|\mathcal{A}|\times$. However, this is not the case if we carefully consider the ordering in which we perform the aggregations. The naive approach of performing each aggregation sequentially would cause this linear increase—but there is a better way to order our computation. The key observation to note is that we are *memory-bound*, and not compute-bound: the bottleneck with sparse operations is waiting for the data to arrive from memory. This observation applies to both GPUs and CPUs, and justified through profiling. Using a profiler on a GTX 1080Ti we observed that SpMM using the Reddit graph (Hamilton et al., 2017) with feature sizes of 256 achieved just 1.2% of the GPU's peak FLOPS, with 88.5% of stalls being caused by unmet memory dependencies. The fastest processing order, which we refer to as *aggregator fusion*, performs as much work as possible with data that has already been fetched from memory, rather than fetching it multiple times.

Algorithm 1 Aggregator Fusion with aggregators $\mathcal{A}$. This method is a modification of the Compressed Sparse Row (CSR) SpMM algorithm, where we maximize re-use of matrix $\mathbf{B}$. Maximizing re-use enables us to obtain significantly better accuracy with minimal impact on memory and latency. For simplicity, pseudocode assumes $H = B = 1$.

```
Input: CSR $\mathbf{A} \in \mathbb{R}^{N\times N}$, Dense $\mathbf{B} \in \mathbb{R}^{N\times F}$, Combination weightings $\mathbf{w} \in \mathbb{R}^{N\times|\mathcal{A}|}$
Output: Dense $\mathbf{C} \in \mathbb{R}^{N\times F}$
for $i = 0$ to $\mathbf{A}$.rows $- 1$ do
  for $jj = \mathbf{A}$.row_pointer$[i]$ to $\mathbf{A}$.row_pointer$[i+1]$ do
    $j = \mathbf{A}$.column_index$[jj]$
    Init temp arrays of length $F$ per aggregator
    $a_{ij} = \mathbf{A}$.values$[jj]$
    // May be faster to interleave these calls:
    for $\oplus \in \mathcal{A}$ do
      process_row$_\oplus(a_{ij}, \mathbf{B}[i,:], \text{temp}_\oplus)$
    end for
  end for
  // Can be generalized to $H, B > 1$:
  $\mathbf{C}[i,:] = \sum_{\oplus\in\mathcal{A}} \mathbf{w}[i,\oplus] \cdot \text{temp}_\oplus[:]$
end for
```

In other words, the loop over our aggregation functions should be the inner-most loop; performing the aggregations sequentially would correspond to having this loop over aggregators at the outer-most level. This concept is illustrated in Algorithm 1. We can perform all aggregations as a lightweight modification to the standard compressed sparse row (CSR) SpMM algorithm: each aggregator defines a function for how to process a row in the dense matrix. The implementation as presented is suitable for inference time; while this technique can be applied on the forward pass during training, it may be necessary to retain extra information for the backwards pass (e.g. for the max aggregator), and it is not possible in general to fuse the backwards pass.

4. Interpretation and Benefits

This section will explain our design choices, and why they are better suited to the hardware. We emphasize that our approach does *not* correspond to attention.

4.1. Spatial Interpretation

The idea of combining basis matrices has been proposed in Schlichtkrull et al. (2018) to handle multiple edge types. The core technique involved learning a weight matrix per edge type, however this can lead to overfitting; instead, learning each edge weight matrix as a combination of shared basis matrices was found to be an effective regularizer. While our approach is related as we also utilize basis matrices, we are solving a fundamentally different problem: we are investigating how to boost representational power without incurring high computational overheads.

In our approach, each node effectively has its own weight matrix. We can derive this by re-arranging our equation for EGC-S by factorizing the $\mathbf{\Theta}_b$ terms out of inner sum. Building upon Equation (2) we obtain:

$$\begin{aligned}\mathbf{y}^{(i)} &= \Big\Vert_{h=1}^{H} \left(\sum_{b=1}^{B} w_{h,b}^{(i)} \mathbf{\Theta}_b\right) \left(\sum_{j\in\mathcal{N}(i)\cup\{i\}} \frac{\mathbf{x}^{(j)}}{\sqrt{\deg(i)\deg(j)}}\right) \\ &= \Big\Vert_{h=1}^{H} \underbrace{\quad \mathbf{\Theta}_h^{(i)} \quad}_{\text{Varying per Node}} \underbrace{\left(\sum_{j\in\mathcal{N}(i)\cup\{i\}} \frac{\mathbf{x}^{(j)}}{\sqrt{\deg(i)\deg(j)}}\right)}_{\text{Computable via SpMM}}\end{aligned}$$

In contrast, GAT shares weights, and pushes the complexity into the message calculation phase. Specifically, we have:

$$\begin{aligned}\mathbf{y}^{(i)} &= \Big\Vert_{h=1}^{H} \alpha_{h,i,i}\mathbf{\Theta}\mathbf{x}^{(i)} + \sum_{j\in\mathcal{N}(i)} \alpha_{h,i,j}\mathbf{\Theta}\mathbf{x}^{(j)} \\ &= \Big\Vert_{h=1}^{H} \underbrace{\quad \mathbf{\Theta} \quad}_{\text{Shared Weights}} \underbrace{\left(\sum_{j\in\mathcal{N}(i)\cup\{i\}} \alpha_{h,i,j}\mathbf{x}^{(j)}\right)}_{\text{Message Materialization}}\end{aligned}$$

where $\alpha_{h,i,j}$ is defined in Table 1. From an efficiency perspective, we can observe that our approach of using SpMM has significantly better characteristics: while it may still be possible to implement some architectures by fusing the message and aggregation steps to reduce overheads from materialization, this is a more difficult pattern for accelerators to optimize for.

4.2. Localised Spectral Filtering with Multiple Kernels

We can alternatively interpret our EGC-S layer through the lens of graph signal processing (Sandryhaila & Moura, 2013). The convolution operation for the Euclidean domain is of paramount important for filtering digital signals and images. Many modern graph neural networks build on the observation that an analogous operation defined for the graph domain has similarly strong inductive biases: it respects the structure of the domain and preserves the locality of features by being an operation localised in space. Our method can be viewed as a method of building adaptive filters in the graph domain. Adaptive filters are a common approach when signal or noise characteristics vary with time or space; for example, they are commonly applied in adaptive noise cancellation. Our approach can be viewed as constructing adaptive filters by linearly combining learnable filter banks with spatially varying coefficients; to the best of our knowledge this type of approach has not been used in the modern machine learning literature.

The graph convolution operation is typically defined on the spectral domain with the convolution theorem $\mathcal{F}(x * f) = \mathcal{F}(x) \cdot \mathcal{F}(f)$, where $\mathcal{F}(x)$ denotes the Fourier transform of signal $\mathbf{x} \in \mathbb{R}^N$ on a graph with N nodes. As on the Euclidean domain, the Fourier transform on the graph-domain is defined as the basis decomposition with the orthogonal eigenbasis of the Laplace operator, which for a graph with adjacency matrix $\mathbf{A} \in \mathbb{R}^{N\times N}$ is defined as $\mathbf{L} = \mathbf{D} - \mathbf{A}$, where $\mathbf{D}$ is the diagonal degree matrix with $D_{ii} = \sum_{j=1}^{N} A_{ij}$. The Fourier transform of a signal $\mathbf{x} \in \mathbb{R}^N$ is $\mathcal{F}(\mathbf{x}) = \mathbf{U}^\top\mathbf{x}$, where $\mathbf{L} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^\top$, with orthogonal eigenvector matrix $\mathbf{U} \in \mathbb{R}^{N\times N}$ and diagonal eigenvalue-matrix $\mathbf{\Lambda} \in \mathbb{R}^{N\times N}$. The result of a signal $\mathbf{x}$ filtered by g_θ is $\mathbf{y} = g_\theta(\mathbf{L})\mathbf{x} = \mathbf{U}g_\theta(\mathbf{\Lambda})\mathbf{U}^\top\mathbf{x}$ where the second equality holds if the Taylor expansion of g_θ exists.

Our approach corresponds to learning multiple filters and computing a linear combination of the resulting filters with weights depending on the attributes of each node locally. The model therefore allows applying multiple filters for each node, enabling to obtain a spatially-varying frequency response, while staying far below $\mathcal{O}(|E|)$ in computational complexity. Using a linear combination of filters, the filtered signal becomes $\mathbf{y} = \sum_{b=1}^{B} \mathbf{w}_b \odot g_{\theta_b}(\mathbf{L})\mathbf{x}$, where $\mathbf{w}_b \in \mathbb{R}^N$ are the weights of filter b for each of the N nodes in the graph. If we parameterize our filter using first-order Chebyshev polynomials as used by Kipf & Welling (2017) our final expression for the filtered signal becomes:

$$\mathbf{Y} = \sum_{b=1}^{B} \mathbf{w}_b \odot (\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}})\mathbf{X}\mathbf{\Theta}_b, \tag{4}$$

where $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}_N$ is the adjacency matrix with added self-loops and $\tilde{\mathbf{D}}$ is the diagonal degree matrix of $\tilde{\mathbf{A}}$ as defined earlier. This justifies the symmetric normalization aggregator we chose in Equation (2).

An approach for localized filtering was proposed by Cheng et al. (2021). However, their approach does not generalize to graphs with unseen topologies or scale to large graphs as it requires learning the coefficients of several sparse filter matrices $\mathbf{S}_k$ of size $N \times N$. Our approach does not suffer from these constraints.

4.3. Interaction with Hardware

As explained earlier, we do not need to explicitly materialize the messages between nodes, cutting our memory usage from $\mathcal{O}(|E|)$ to $\mathcal{O}(|V|)$. In the limiting case, we have $\mathcal{O}(|E|) = \mathcal{O}(|V|^2)$, meaning that our architecture asymptotically reduces the memory cost to a square root of the previous cost; we note, however, that the precise benefit is topology-specific. Another key benefit is that SpMM is an algorithm that has been studied for decades (Eisenstat et al., 1977): we can build upon this work. It is worth noting that we can optimize our implementation by concatenating our basis matrices into a single matrix: $\mathbf{\Theta} = (\mathbf{\Theta}_1, \ldots, \mathbf{\Theta}_B)$ and performing one SpMM (i.e. $\mathbf{SX\Theta}$). It is also worth noting that for small or dense graphs, it may be faster to implement message propagation using dense matrix multiplication; this is not possible for architectures relying on materialization.

Supporting arbitrary GNN architectures is difficult for hardware designers, and will be more difficult to accelerate. This is already demonstrated by the work in the GNN accelerator literature: the majority of works claiming to have built a GNN accelerator only support SpMM (Geng et al., 2020; Chen et al., 2020; Yan et al., 2020b)—they do not support more recent architectures. It is worth invoking Amdahl's Law (Amdahl, 1967): as a hardware designer, we can obtain the best performance by *optimizing for the common case*. Adding support for flexible architectures to an accelerator inherently requires trading design complexity & area, peak performance, and efficiency, and there is no guarantee these accelerator designs will be deployed in the real world. We believe that SpMM accelerators are a realistic target as they can also be used to accelerate pruned models.

In addition to concerns about accelerators, our approach is also beneficial for data center and mobile workloads. One aspect is data movement: since there is no need to materialize edges, we can reduce the number of memory accesses. Reducing data movement is a key contributor to achieving low energy consumption: a single 32-bit floating-point add costs 0.9pJ, but a 32-bit DRAM read costs 640pJ (Horowitz, 2014)—3 orders of magnitude higher. Aggregator fusion also benefits energy consumption since it reduces data movement. Another aspect is cache performance: on CPUs—which remain common for data center inference (Hazelwood et al., 2018)—caching affects inference latency significantly for sparse workloads (Tailor et al., 2021). By avoiding materialization we have smaller activations to fit into cache, and we can take advantage of cache-blocking approaches to SpMM to boost performance (Zhang et al., 2017).

5. Evaluation

In this section we demonstrate that our proposal outperforms competing approaches, and provide studies investigating how to choose the hyperparameters of our model. We also show that aggregator fusion enables our architecture to be implemented with little overhead.

5.1. Protocol

We evaluate our approach on 5 datasets taken from recent works on GNN benchmarking. We use ZINC and CIFAR-10 Superpixels from Dwivedi et al. (2020) and Arxiv, MolHIV and Code from Open Graph Benchmark (Hu et al., 2020). These datasets cover a wide range of domains, cover both transductive and inductive tasks, and are larger than datasets which are typically used in GNN works. We use evaluation metrics specified by these papers.

In order to provide a fair comparison we standardize all parameter counts, architectures and optimizers in our experiments. For ZINC, CIFAR-10 and Arxiv we use models with 100k parameters; for MolHIV we use 300k, and for Code we use 11M—most of which are associated with the fully-connected layers required to predict tokens. For benchmarks from Dwivedi et al. (2020) we use 100k as this is the count they normalize all architectures to; for the OGB datasets, no normalized benchmarks exist, therefore we chose parameter counts which were representative of models that have already been submitted to their leaderboards. All experiments were run using Adam (Kingma & Ba, 2014).

We normalize the architectures against those in Dwivedi et al. (2020) and Corso et al. (2020); this corresponds to stacking 4 layers with residual connections. We apply the same architecture to Arxiv and Code, where there are no existing normalized baselines; the only change we make is to use 3 layers for Arxiv. We do not use edge features in our experiments. We do not use sampling, which is not applicable to 4 datasets; for the remaining dataset, Arxiv, we believe it is not in the interests of making results comparable by introducing an additional variable. All experiments were run 10 times. Further details, including aggregator choices for EGC-M, can be found in the supplementary material.

5.2. Results

Our results across the 5 tasks are shown in Table 2. We draw attention to the following observations:

- **EGC-S is competitive with anisotropic architectures**. GAT and MPNN are architectures using one aggregator; we see across all tasks that we obtain similar, or better, performance. The exception is MPNN-Max on CIFAR-10 & Code, where the max aggregator provides a better inductive bias.

Table 2. Results (mean and standard deviation) for EGC run on 5 datasets against normalized baselines. Details of the specific aggregators chosen per dataset and further experimental details can be found in the supplementary material. Any results marked with $*$ ran out of memory on the popular Nvidia 1080Ti or 2080Ti GPUs. EGC obtains best performance on 4 of the tasks, with consistently wide margins.

Architecture	**ZINC (MAE $\downarrow$)**	**CIFAR (Acc. $\uparrow$)**	**MolHIV (ROC-AUC $\uparrow$)**	**Arxiv (Acc. $\uparrow$)**	**Code-V2 (F1 $\uparrow$)**
GCN	0.459 ± 0.006	55.71 ± 0.38	76.14 ± 1.29	71.92 ± 0.21	0.1480 ± 0.0018
GAT	0.475 ± 0.007	64.22 ± 0.46	77.17 ± 1.37	$* \, 71.81 \pm 0.23$	0.1513 ± 0.0011
GIN	0.387 ± 0.015	55.26 ± 1.53	76.02 ± 1.35	67.33 ± 1.47	0.1481 ± 0.0027
MPNN-Sum	0.381 ± 0.005	65.39 ± 0.47	75.19 ± 3.57	$* \, 66.11 \pm 0.56$	0.1470 ± 0.0017
MPNN-Max	0.468 ± 0.002	69.70 ± 0.55	77.07 ± 1.37	$* \, 71.02 \pm 0.21$	0.1552 ± 0.0022
PNA	0.320 ± 0.032	70.21 ± 0.15	$\mathbf{79.05 \pm 1.32}$	$* \, 71.21 \pm 0.30$	$* \, 0.1570 \pm 0.0032$
EGC-S	0.364 ± 0.020	66.63 ± 0.26	77.21 ± 1.10	$\mathbf{72.19 \pm 0.16}$	0.1528 ± 0.0025
EGC-M	$\mathbf{0.281 \pm 0.008}$	$\mathbf{71.04 \pm 0.45}$	78.18 ± 1.53	71.96 ± 0.23	$\mathbf{0.1595 \pm 0.0019}$

- **EGC-M obtains state-of-the-art performance**. The addition of multiple aggregator functions improves performance of EGC to, or even beyond, that obtained by PNA. This is a significant achievement: our architecture performs excellently on a wide variety of tasks, but with lower resource requirements. We hypothesize that our improved performance over PNA is related to PNA's reliance on multiple degree-scaling transforms. While this approach can boost the representational power of the architecture, we hypothesize it can result in a tendency to overfit to the training set.

- **EGC performs strongly without running out of memory.** We observe that EGC is one of only three architectures that did not exhaust the VRAM of the popular Nvidia 1080/2080Ti GPUs, with 11GB VRAM, when applied to Arxiv: we had to use an RTX 8000 GPU to run these experiments. PNA, our closest competing technique accuracy-wise, exhausted memory on the Code benchmark as well. We note that optimizations can be made to GAT to reduce memory footprint required at training time by storing only derived values $\mathbf{a}_{\text{source}}^{\top}\boldsymbol{\Theta}\mathbf{x}^{(i)}$ and $\mathbf{a}_{\text{target}}^{\top}\boldsymbol{\Theta}\mathbf{x}^{(j)}$ per-edge for backpropagation; however, this still corresponds to an asymptotic cost of $\mathcal{O}(|E|)$. Additionally, this approach is specific to GAT, and cannot be applied to MPNN or PNA.

- **Our approach performs well on transductive tasks**. Many transductive tasks are homophilous i.e. the closer two nodes, the more similar the graph signal—hence why spectral techniques tend to perform well, as they are tend to smooth graph signals. We note that the current state-of-the-art results on Arxiv can be achieved with label propagation (Huang et al., 2020) due to this homophilous property, and that our architecture can be combined with this approach. We hypothesize that our architecture retains many of the desirable properties of spectral models, and although we do not assess it in this work, we note that it is possible to apply our approach to higher order (spectral) convolutions including nodes from more than 1 edge away (Defferrard et al., 2016).

Overall, EGC obtains the best performance on 4 out of the 5 datasets; on the remaining dataset (MolHIV), EGC is the second best architecture. This represents a significant achievement: our architecture demonstrates that we do not need to choose between efficiency and accuracy.

5.3. Ablation Study: Varying Heads and Bases

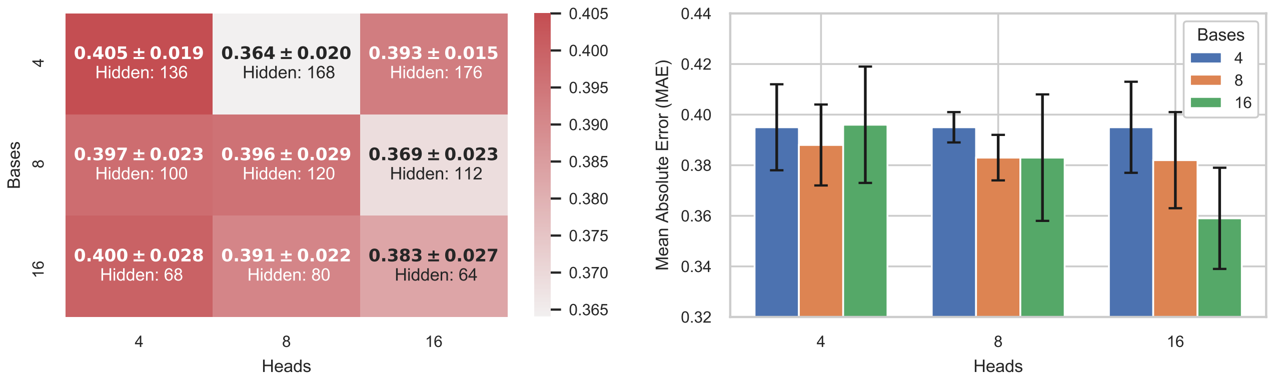
In order to understand the trade-off between the number of heads (H) and bases (B), we ran an ablation study on ZINC using EGC-S; this in shown in Figure 2. We run experiments controlling for parameter count, and study varying H and B with a constant hidden dimension.

The relationship between these parameters is non-trivial. There are several aspects to consider: (1) increasing H and B means that we spend more of our parameter budget to create the combinations, which reduces hidden dimension—as shown in Figure 2(a). This is exacerbated if we use multiple aggregators: our combination dimension must be $HB|\mathcal{A}|$. (2) Increasing B means we must reduce the hidden size substantially, since it corresponds to adding more weights of size $\frac{F'}{H} \times F$. (3) Increasing H allows us to increase hidden size, since each basis weight becomes smaller. We see in Figure 2(b) that increasing B beyond H does not yield significant performance improvements: we conjecture that bases begin specializing for individual heads; by sharing, there is a regularizing effect, like observed in Schlichtkrull et al. (2018). This regularization stabilizes the optimization and we observe lower trial variance for smaller B. We also evaluated whether applying orthogonality constraints to the bases improved performance, but observed no benefit.

We advise initially setting $B = H$ or $B = \frac{H}{2}$ for a given parameter count. In general, we find $H = 8$ to be effective with EGC-S. For EGC-M, where more parameters must be spent on the combination weights, we advise using $H = 4$.

5.4. Latency Benchmarks

We evaluated aggregator fusion across four different topologies, on both CPU and GPU; our results can be found in



(a) Constant parameter count (100K)

(b) Constant hidden dimension (128)

Figure 2. Ablation study over the number of heads (H) and bases (B). Study run on ZINC dataset with EGC-S. Metric is MAE (mean and standard deviation): lower is better. We study two regimes: keeping the total parameter count constant, and fixing the hidden dimension while varying H and B. Each experiment was tuned individually and evaluated across 10 seeds. Setting $B > H$ does not improve performance, forces the usage of a smaller hidden dimension to retain a constant parameter count, and may induce overfitting.

Table 3. Inference latency (mean and standard deviation) for CSR SpMM, used by GCN/GIN, and aggregator fusion. Assuming a feature dimension of 256 and $H = B = 1$ per Algorithm 1. We observe that aggregator fusion results in an increase of 40% in the worse case; in contrast, the naive implementation has a worst case increase of 460%. Also included are timings for dense multiplication with a square weight matrix; we observe that sparse operations dominate latency measurements.

	CPU (Xeon Gold 5218)				**GPU (RTX 8000)**			
Method	**Reddit / s**	**Code / s**	**Arxiv / s**	**ZINC / s**	**Reddit / ms**	**Code / ms**	**Arxiv / ms**	**ZINC / ms**
Weight Matmul	0.07 ± 0.01	0.391 ± 0.006	0.055 ± 0.005	0.068 ± 0.001	2.12 ± 0.00	13.90 ± 0.00	1.88 ± 0.00	2.56 ± 0.02
CSR SpMM	25.24 ± 0.18	1.888 ± 0.013	0.642 ± 0.006	0.307 ± 0.001	186.50 ± 0.03	20.00 ± 0.01	6.46 ± 0.01	4.43 ± 0.04
Naive Fusion	74.15 ± 0.89	8.253 ± 0.131	2.307 ± 0.012	1.369 ± 0.005	596.14 ± 0.16	111.90 ± 0.30	23.73 ± 0.05	19.09 ± 0.17
Our Fusion	34.92 ± 0.26	1.709 ± 0.012	0.821 ± 0.012	0.274 ± 0.002	208.36 ± 0.03	26.66 ± 0.08	8.03 ± 0.01	5.62 ± 0.01

Table 3. We assumed all operations are 32-bit floating point, and that we were using three aggregators: summation-based, max, and min; these aggregators match those used for EGC-M Code. Our benchmarks were conducted on a batch of 10k graphs from the ZINC and Code datasets, Arxiv, and the popular Reddit dataset (Hamilton et al., 2017), which is one of the largest graph datasets commonly evaluated on in the GNN literature. Our SpMM implementation on GPU is based on Yang et al. (2018). Code for the kernels are provided in our repo.

As expected, our technique optimizing for input re-use achieves significantly lower inference latency than the naive approach to applying multiple aggregators. While the naive approach results in a mean increase in latency of 305%, our approach incurs a mean increase of **only 19%** relative to ordinary SpMM, used by GCN and GIN. The increase is topology dependent, with larger increases in latency being observed for topologies which are less memory-bound. We also provide timings for dense matrix multiplication (i.e. $\mathbf{X}\mathbf{\Theta}$) to justify our focus on optimizing sparse operations in this work: the CSR SpMM operation is **7.9**× slower (geomean) than the corresponding weight multiplication. We believe further optimizations of the sparse and dense operations used by architecture are achievable through the use of auto-tuning frameworks e.g. TVM (Chen et al., 2018b), but this lies beyond the scope of this work.

6. Conclusion

This work has made an important step towards improving the runtime efficiency of GNNs. Our proposed layer can be used as a *drop-in replacement* for existing GNN layers, and achieves better results across 5 different benchmark datasets compared to strong baselines, while also being more efficient memory and latency-wise. Our approach requires memory proportional to the number of vertices, in contrast to approaches with competitive accuracy which require memory proportional to the number of edges. Additionally, we propose a useful technique for reducing latency, aggregator fusion, that can be applied outside of this work. Throughout this work we have carefully considered the interaction between our proposal and the underlying hardware: we believe that our approach can be accelerated by realistic upcoming accelerator designs that incorporate support for sparse matrices. We believe the next step for our work is efficient incorporation of edge features.

Acknowledgements

This work was supported by Samsung AI and by the UK's Engineering and Physical Sciences Research Council (EPSRC) with grants EP/M50659X/1 and EP/S001530/1 (the MOA project) and the European Research Council via the REDIAL project (Grant Agreement ID: 805194). FLO acknowledges funding from the Huawei Studentship at the Department of Computer Science and Technology of the University of Cambridge.

References

Allamanis, M., Brockschmidt, M., and Khademi, M. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

Amdahl, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pp. 483–485, 1967.

Chen, J., Ma, T., and Xiao, C. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247*, 2018a.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. Tvm: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pp. 579–594, USA, 2018b. USENIX Association. ISBN 9781931971478.

Chen, X., Wang, Y., Xie, X., Hu, X., Basak, A., Liang, L., Yan, M., Deng, L., Ding, Y., Du, Z., Chen, Y., and Xie, Y. Rubik: A Hierarchical Architecture for Efficient Graph Learning. *arXiv:2009.12495 [cs]*, September 2020. URL http://arxiv.org/abs/2009.12495. arXiv: 2009.12495.

Cheng, X., Miao, Z., and Qiu, Q. Graph convolution with low-rank learnable local filters. In *International Conference on Learning Representations*, 2021.

Chiang, W.-L., Liu, X., Si, S., Li, Y., Bengio, S., and Hsieh, C.-J. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 257–266, 2019.

Corso, G., Cavalleri, L., Beaini, D., Liò, P., and Veličković, P. Principal Neighbourhood Aggregation for Graph Nets. *arXiv:2004.05718 [cs, stat]*, June 2020. URL http://arxiv.org/abs/2004.05718. arXiv: 2004.05718.

Defferrard, M., Bresson, X., and Vandergheynst, P. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems 29*, 2016.

Dwivedi, V. P., Joshi, C. K., Laurent, T., Bengio, Y., and Bresson, X. Benchmarking Graph Neural Networks. *arXiv:2003.00982 [cs, stat]*, July 2020. URL http://arxiv.org/abs/2003.00982. arXiv: 2003.00982.

Eisenstat, S., Gursky, M., Schultz, M., and Sherman, A. Yale sparse matrix package. ii. the nonsymmetric codes. Technical report, Department of Computer Science, Yale University, 1977.

Fuchs, A. and Wentzlaff, D. The accelerator wall: Limits of chip specialization. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 1–14. IEEE, 2019.

Geng, T., Li, A., Shi, R., Wu, C., Wang, T., Li, Y., Haghi, P., Tumeo, A., Che, S., Reinhardt, S., and Herbordt, M. AWB-GCN: A Graph Convolutional Network Accelerator with Runtime Workload Rebalancing. *arXiv:1908.10834 [cs]*, September 2020. URL http://arxiv.org/abs/1908.10834. arXiv: 1908.10834.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. In *International Conference on Machine Learning*, pp. 1263–1272. PMLR, 2017.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Yin, J., Jiang, D., et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

Hamilton, W. L., Ying, R., and Leskovec, J. Inductive representation learning on large graphs. *arXiv preprint arXiv:1706.02216*, 2017.

Hazelwood, K., Bird, S., Brooks, D., Chintala, S., Diril, U., Dzhulgakov, D., Fawzy, M., Jia, B., Jia, Y., Kalro, A., et al. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 620–629. IEEE, 2018.

Horowitz, M. 1.1 Computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 10–14, February 2014. doi: 10.1109/ISSCC.2014.6757323. ISSN: 2376-8606.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

Hu, W., Fey, M., Zitnik, M., Dong, Y., Ren, H., Liu, B., Catasta, M., and Leskovec, J. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687*, 2020.

Huang, Q., He, H., Singh, A., Lim, S.-N., and Benson, A. R. Combining label propagation and simple models out-performs graph neural networks. *arXiv preprint arXiv:2010.13993*, 2020.

Huang, W., Zhang, T., Rong, Y., and Huang, J. Adaptive sampling towards fast graph representation learning. *arXiv preprint arXiv:1809.05343*, 2018.

Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., and Keutzer, K. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

Iandola, F. N., Shaw, A. E., Krishna, R., and Keutzer, K. W. Squeezebert: What can computer vision teach nlp about efficient neural networks? *arXiv preprint arXiv:2006.11316*, 2020.

Jia, Z., Lin, S., Gao, M., Zaharia, M., and Aiken, A. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems*, 2:187–198, 2020.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Kipf, T. N. and Welling, M. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations*, 2017.

Pfaff, T., Fortunato, M., Sanchez-Gonzalez, A., and Battaglia, P. W. Learning mesh-based simulation with graph networks. *arXiv preprint arXiv:2010.03409*, 2020.

Rossi, E., Frasca, F., Chamberlain, B., Eynard, D., Bronstein, M., and Monti, F. Sign: Scalable inception graph neural networks. *arXiv preprint arXiv:2004.11198*, 2020.

Sanchez-Gonzalez, A., Godwin, J., Pfaff, T., Ying, R., Leskovec, J., and Battaglia, P. W. Learning to simulate complex physics with graph networks. *arXiv preprint arXiv:2002.09405*, 2020.

Sandryhaila, A. and Moura, J. M. Discrete signal processing on graphs. *IEEE transactions on signal processing*, 61 (7):1644–1656, 2013.

Sarlin, P.-E., DeTone, D., Malisiewicz, T., and Rabinovich, A. Superglue: Learning feature matching with graph neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4938–4947, 2020.

Schlichtkrull, M., Kipf, T. N., Bloem, P., Van Den Berg, R., Titov, I., and Welling, M. Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*, pp. 593–607. Springer, 2018.

Shi, W. and Rajkumar, R. Point-gnn: Graph neural network for 3d object detection in a point cloud. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1711–1719, 2020.

Sze, V., Chen, Y.-H., Yang, T.-J., and Emer, J. S. Efficient processing of deep neural networks. *Synthesis Lectures on Computer Architecture*, 15(2):1–341, 2020.

Tailor, S. A., Fernandez-Marques, J., and Lane, N. D. Degree-Quant: Quantization-Aware Training for Graph Neural Networks. In *International Conference on Learning Representations*, 2021.

Tay, Y., Dehghani, M., Bahri, D., and Metzler, D. Efficient transformers: A survey. *arXiv preprint arXiv:2009.06732*, 2020.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is All you Need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 30*, pp. 5998–6008. Curran Associates, Inc., 2017. URL http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. Graph attention networks. In *International Conference on Learning Representations*, 2018.

Xu, K., Hu, W., Leskovec, J., and Jegelka, S. How Powerful are Graph Neural Networks? *arXiv:1810.00826 [cs, stat]*, February 2019. URL http://arxiv.org/abs/1810.00826. arXiv: 1810.00826.

Yan, B., Wang, C., Guo, G., and Lou, Y. Tinygnn: Learning efficient graph neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, pp. 1848–1856, New York, NY, USA, 2020a. Association for Computing Machinery. ISBN 9781450379984. doi: 10.1145/3394486.3403236. URL https://doi.org/10.1145/3394486.3403236.

Yan, M., Deng, L., Hu, X., Liang, L., Feng, Y., Ye, X., Zhang, Z., Fan, D., and Xie, Y. HyGCN: A GCN Accelerator with Hybrid Architecture. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 15–29, February 2020b. doi: 10.1109/HPCA47549.2020.00012. ISSN: 2378-203X.

Yang, C., Buluc, A., and Owens, J. D. Design principles for sparse matrix multiplication on the gpu, 2018.

Zeng, H., Zhou, H., Srivastava, A., Kannan, R., and Prasanna, V. Graphsaint: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931*, 2019.

Zhang, Y., Kiriansky, V., Mendis, C., Amarasinghe, S., and Zaharia, M. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*, pp. 293–302, 2017. doi: 10.1109/BigData.2017.8257937.

Zhao, Y., Wang, D., Gao, X., Mullins, R., Lio, P., and Jamnik, M. Probabilistic dual network architecture search on graphs. *arXiv preprint arXiv:2003.09676*, 2020.