ADAPTIVE FILTERS FOR LOW-LATENCY AND MEMORY-EFFICIENT GRAPH NEURAL NETWORKS

Anonymous authors
Paper under double-blind review

ABSTRACT

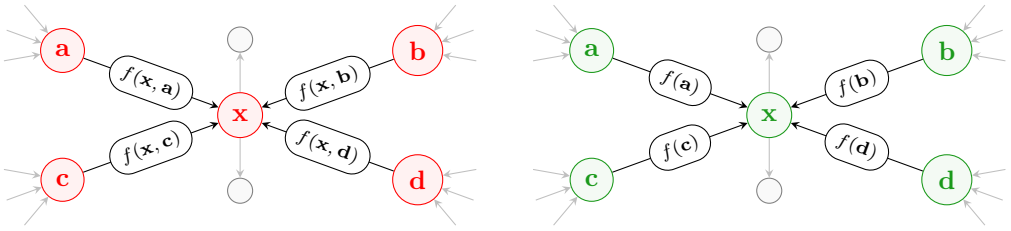
Scaling and deploying graph neural networks (GNNs) remains difficult due to their high memory consumption and inference latency. In this work we present a new type of GNN architecture that achieves state-of-the-art performance with lower memory consumption and latency, along with characteristics suited to accelerator implementation. Our proposal uses memory proportional to the number of vertices in the graph ($\mathcal{O}(V)$), in contrast to competing methods which require memory proportional to the number of edges ($\mathcal{O}(E)$); surprisingly, we find our efficient approach actually achieves higher accuracy than competing approaches across 6 large and varied datasets against strong baselines. We achieve our results by using a novel *adaptive filtering* approach, which can be interpreted as enabling each vertex to have its own weight matrix, and is not directly related to attention. Following our focus on efficient hardware usage, we demonstrate that our method achieves lower latency and memory consumption for the same accuracy when compared to competing approaches.

1 INTRODUCTION

Graph Neural Networks (GNNs) have emerged as an effective way to build models over arbitrarily structured data. For example, they have successfully been applied to computer vision tasks: GNNs can deliver high performance on point cloud data (Qi et al., 2017) and for feature matching across images (Sarlin et al., 2020). Recent work has also shown that they can be applied to physical simulations (Pfaff et al., 2020; Sanchez-Gonzalez 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 make a step towards enabling these applications, and many more, by investigating methods to cheaply boost GNN representational power. Succinctly, we investigate whether it is possible to design GNN architectures that achieve high performance without compromising on efficiency. Importantly, our method delivers benefits at both training and inference time, and can be successfully applied to tasks requiring generalization to unseen graphs.

The development of hardware-efficient techniques is key to the scaling and deployment of deep learning. We have seen the deployment of convolutional neural networks (CNNs) to enable novel applications at the edge, due to innovation at the hardware and algorithmic level. Investigating methods to improve the efficiency of GNNs will enable us to train larger models, on larger datasets, along with scaling down our models so that we can deploy them for applications at the edge.

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), with common examples including quantization and pruning. Another key area is *designing neural network architectures with efficiency as a design goal*. To date, research on GNN architectures has typically focused on boosting accuracy, with efficiency being a secondary consideration; many works focusing on efficiency tend not to address the underlying architecture design, and instead consider orthogonal approaches, such as quantization or graph sampling. We have seen memory consumption and latency for state-of-the-art GNN architectures increase in recent years, predominantly due to models incorporating *anisotropy*—treating neighbors differently using learnt functions—to boost accuracy. Surprisingly, we find that we can achieve state-of-the-art performance without explicitly incorporating anisotropy into our architecture, challenging conventional wisdom in the community.



Messages are functions of *source and target* nodes and hence must be materialized

Messages are functions of *source only*; propagation can be implemented using matrix multiplication-style approaches

Figure 1: Many GNN architectures (e.g. GAT (Veličković et al., 2018), PNA (Corso et al., 2020)) incorporate sophisticated message functions to improve accuracy (left). This is problematic as we must materialize messages, leading to $\mathcal{O}(E)$ memory consumption and OPs to calculate messages; these dataflow patterns are also difficult to optimize for at the hardware level. **This work demonstrates that we can use simple message functions, requiring only $\mathcal{O}(V)$ memory consumption** (right) **and improve performance over state-of-the-art.**

Contributions. We propose a new GNN architecture, Efficient Graph Convolution (*EGC*). Our proposal's memory consumption is *linear in the number of vertices in the graph* ($\mathcal{O}(V)$)—not the number of edges ($\mathcal{O}(E)$). Our architecture's latency is also lower than competing approaches, and is a *drop-in replacement* on a variety of tasks. We provide a rigorous evaluation of our architecture across 6 large graph datasets covering both transductive and inductive use-cases. Application domains ranging from citation graphs to code analysis and molecular property prediction are covered, and demonstrate that our approach consistently achieves better results than strong baselines, and that our approach can handle heterogeneous graphs. We also demonstrate that our method can achieve lower memory consumption and latency for a given accuracy.

2 Background

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 (Vaswani et al., 2017). 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 baseline architecture are provided in Table 1, with further details supplied in Table 11 in the Appendix.

This work builds upon the concept of using basis matrices. The idea of combining basis matrices was proposed in R-GCN (Schlichtkrull et al., 2018) to handle multiple edge types; Xu et al. (2021) demonstrated that this concept can also be applied to point cloud analysis. In this work we are solv-

ing a fundamentally different problem: we are interested in *designing efficient GNN architectures*, not merely boosting expressivity.

Scaling and Deploying GNNs While GNNs have seen success across a range of domains, there remain challenges associated with scaling and deploying them. Graph sampling is one approach to scaling training for large graphs or models 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). Alternatively, systems for distributed GNN training have been proposed (Jia et al., 2020) to scale training beyond the limits of a single accelerator. Some works have proposed architectures that are designed to accommodate scaling: graph-augmented MLPs, such as SIGN (Rossi et al., 2020), are explicitly designed as a shallow architecture, as all the graph operations are done as a pre-processing step. Other work includes applying neural architecture search (NAS) to arrange existing GNN layers (Zhao et al., 2020), or building quantization techniques for GNNs (Tailor et al., 2021). Finally, a recent work has shown that using memory-efficient reversible residuals (Gomez et al., 2017) for GNNs (Li et al., 2021) enables us to train far deeper and larger GNN models than before, thereby progressing the state-of-the-art accuracy.

Why Are Existing Approaches Not Sufficient? It is worth noting that many of these approaches have significant limitations that we hope to address with our work. Sampling methods are often ineffective when applied to many problems which involve model generalization to unseen graphs—a common use-case for GNNs. We evaluated a variety of sampling approaches and observed that even modest sampling levels, which provide little benefit to memory or latency, cause model performance to decline noticeably. In addition, these methods do not accelerate the underlying GNN, hence they may not provide any overall benefit to inference latency. There is also no evidence that we are aware of that graph-augmented MLPs perform adequately when generalizing to unseen graphs; indeed, they are known to be theoretically less expressive than standard GNNs (Chen et al., 2021). We also investigated this setup, and found that these approaches do not offer competitive accuracy with state-of-the-art approaches. Experiment details and results, along with further discussion of the limitations of existing work, is provided in Appendix B.

In summary, efficient GNN architecture design is of interest to the community as it enables us to scale our models further, potentially yielding improvements in accuracy. In addition, for tasks where we need to generalize to unseen graphs, such as code analysis or point cloud processing, we reduce memory consumption and latency, thereby enabling us to deploy our models to more resource-constrained devices than before. 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 is more parameter efficient. In contrast to competing architectures, which require $\mathcal{O}(E)$ memory, our proposal requires only $\mathcal{O}(V)$ memory.

3 Our Architecture: Efficient Graph Convolution (EGC)

In this section we describe our approach, and delay theoretical analysis to the next section. We present two versions: *EGC-S*(ingle), using a single aggregator, and *EGC-M*(ulti) which generalizes our approach by incorporating multiple aggregators. Our approach is visualized in Figure 2.

3.1 Architecture Description

For a layer with in-dimension of F and out-dimension of F' we use B basis weights $\mathbf{\Theta}_b \in \mathbb{R}^{F' \times F}$. 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 $\mathbf{\Theta}_b$. 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. The output for node i is computed in three steps. First, we perform the aggregation with each set of basis weights $\mathbf{\Theta}_b$. Second, we compute the weighting coefficients $\mathbf{w}^{(i)} = \mathbf{\Phi}\mathbf{x}^{(i)} + \mathbf{b} \in \mathbb{R}^B$ for each node i, where $\mathbf{\Phi} \in \mathbb{R}^{B \times F}$ and $\mathbf{b} \in \mathbb{R}^B$ are weight and bias parameters for calculating the combination weighting coefficients. Third, the layer

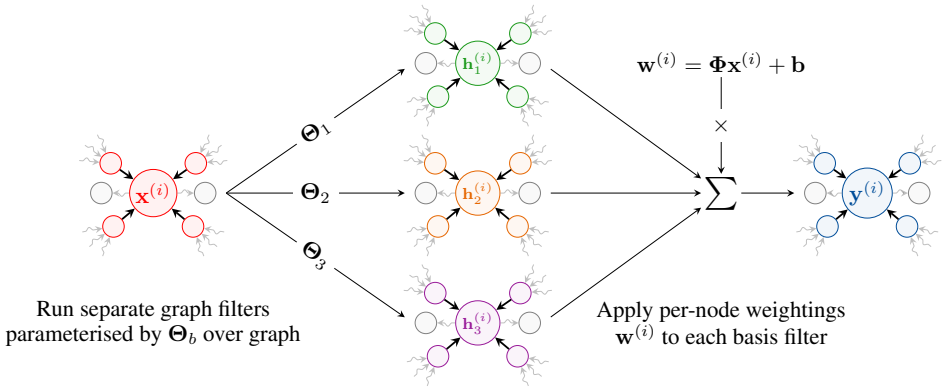


Figure 2: Visual representation of our EGC-S layer. In this visualization we have 3 basis filters (i.e. $B = 3$), which are combined using per-node weightings $\mathbf{w}$. This simplified figure does not show the usage of heads, or multiple aggregators, as used by EGC-M.

output for node i is the weighted combination of aggregation outputs:

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

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 (Veličković et al., 2018) 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 $\alpha(i,j) = 1$ to recover the add aggregator used by GIN (Xu et al., 2019), or $\alpha(i,j) = 1/\sqrt{\text{deg}(i)\text{deg}(j)}$ to recover symmetric normalization used by GCN (Kipf & Welling, 2017)—then we can implement our message propagation phase using sparse matrix multiplication (SpMM), and avoid explicitly materializing each message, even for the backwards pass. 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; more formal justification is provided in section 4.2.

Adding Heads as a Regularizer We can extend our layer through the addition of 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 that adding this degree of freedom aids regularization when the number of heads (H) is larger than B, as bases are discouraged from specializing (see section 5.3), without requiring the integration of additional loss terms into the optimization—hence requiring no changes to code for downstream users. To normalize the output dimension, we change the basis weight matrices dimensions to $\frac{F'}{H} \times F$. Using $\|$ as the concatenation operator, and making the use of symmetric normalization explicit, we obtain the **EGC-S** layer:

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

3.2 Boosting Representational Capacity

Recent work by Corso et al. (2020) has shown that using only a single aggregator is sub-optimal: instead, it is better to combine several different aggregators. In Equation (2) we defined our layer to use only symmetric normalization. To improve performance, we propose applying different aggregators to the representations 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

stddev, min or max which are not based on summation. It is also possible to use directional aggregators proposed by Beaini et al. (2021), however this enhancement is orthogonal to this work. If we have a set of aggregators $\mathcal{A}$, we can extend Equation (2) to obtain our **EGC-M** layer:

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

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 It would appear that adding more aggregators would cause latency and memory consumption to grow linearly. However, this is not true in practice. Firstly, since sparse operations are typically memory bound in practice, we can apply extra aggregators to data that has already arrived from memory with little latency penalty. EGC can also efficiently inline the node-wise weighting operation at inference time, thereby resulting in relatively little memory consumption overhead. The equivalent optimization is more difficult to apply successfully to PNA due to the larger number of operations per-node that must be performed during aggregation, caused by scaling functions being applied to each aggregation, before all results are concatenated and a transformation applied. More details, including profiling and latency measurements, can be found in Appendix C.

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* directly correspond to attention.

4.1 Spatial Interpretation: Node-Wise Weight Matrices

In our approach, each node effectively has its own weight matrix. We can derive this by re-arranging eq. (2) by factorizing the $\mathbf{\Theta}_b$ terms out of inner sum:

$$\mathbf{y}^{(i)} = \mathop{\Big\|}_{h=1}^{H} \underbrace{\mathbf{\Theta}_h^{(i)}}_{\text{Varying per Node}} \underbrace{\left(\sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{1}{\sqrt{\deg(i)\deg(j)}} \mathbf{x}^{(j)} \right)}_{\text{Computable via SpMM}} \tag{4}$$

In contrast, GAT shares weights, and pushes complexity into the message calculation phase by calculating per-message weightings. MPNN (Gilmer et al., 2017) and PNA (Corso et al., 2020) further increase complexity by explicitly calculating each message—resulting in substantial latency overhead due to the number of dense operations increasing by roughly $\frac{|E|}{|V|}$. Specifically, we have:

$$\mathbf{y}_{\text{GAT}}^{(i)} = \mathop{\Big\|}_{h=1}^{H} \underbrace{\mathbf{\Theta}}_{\text{Shared Weights}} \underbrace{\left(\sum_{j \in \mathcal{N}(i) \cup \{i\}} \alpha_{h,i,j} \mathbf{x}^{(j)} \right)}_{\text{Calculated Message Weighting}} \qquad \mathbf{y}_{\text{PNA}}^{(i)} = U(\mathbf{x}^{(i)}, \bigoplus_{j \in \mathcal{N}(i)} \underbrace{M(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})}_{\text{Explicit Message Calculation}})$$

From an efficiency perspective we observe that our approach of using SpMM has better characteristics due to it requiring only $\mathcal{O}(V)$ memory consumption—no messages must be explicitly materialized to use SpMM. We note that although it is possible to propagate messages for GAT with SpMM, there is no way to avoid storing the weightings during training as they are needed for back-propagation, resulting in $\mathcal{O}(E)$ memory consumption. We also note that fusing the message and aggregation steps for certain architectures may be possible at inference time, but this is a difficult pattern for hardware accelerators to optimize for.

Relation To Attention Our method is *not directly related to attention*, which relies upon pairwise similarity mechanisms, and hence results in a $\mathcal{O}(E)$ cost when using the common formulations. Alternatives to attention-based Transformers proposed by Wu et al. (2019a) are a closer analogue to our technique, but rely upon explicit prediction of the per-timestep weight matrix. This approach is not viable for graphs, as the neighborhood size is not constant.

Architecture	Propagation Rule	Memory	ZINC (MAE ↓) Unseen Graph Regression	CIFAR (Acc. ↑) Unseen Graph Classification	MolHIV (ROC-AUC ↑) Unseen Graph Classification	Code-V2 (F1 ↑) Unseen Graph Classification	Arxiv (Acc. ↑) Transductive Node Classification
GCN	$\mathbf{y}^{(i)} = \mathbf{\Theta} \sum_j \frac{1}{\sqrt{\deg(i)\deg(j)}} \mathbf{x}^{(j)}$	$\mathcal{O}(V)$	0.459 ± 0.006	55.71 ± 0.38	76.14 ± 1.29	0.1480 ± 0.0018	71.92 ± 0.21
GIN	$\mathbf{y}^{(i)} = f_{\mathbf{\Theta}}[(1+\epsilon)\mathbf{x}^{(i)} + \sum_j \mathbf{x}^{(j)}]$	$\mathcal{O}(V)$	0.387 ± 0.015	55.26 ± 1.53	76.02 ± 1.35	0.1481 ± 0.0027	67.33 ± 1.47
GraphSAGE	$\mathbf{y}^{(i)} = \mathbf{\Theta}_1 \mathbf{x}^{(i)} + \bigoplus_j \mathbf{\Theta}_2 \mathbf{x}^{(j)}$	$\mathcal{O}(V)$	0.468 ± 0.003	65.77 ± 0.31	75.97 ± 1.69	0.1453 ± 0.0028	71.73 ± 0.26
GAT	$\mathbf{y}^{(i)} = \alpha_{i,i}\mathbf{\Theta}\mathbf{x}^{(i)} + \sum_j \alpha_{i,j}\mathbf{\Theta}\mathbf{x}^{(j)}$	$\mathcal{O}(E)$	0.475 ± 0.007	64.22 ± 0.46	77.17 ± 1.37	0.1513 ± 0.0011	$*\ 71.81 \pm 0.23$
MPNN-Sum	$\mathbf{y}^{(i)} = U(\mathbf{x}^{(i)}, \sum_j M(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}))$	$\mathcal{O}(E)$	0.381 ± 0.005	65.39 ± 0.47	75.19 ± 3.57	0.1470 ± 0.0017	$*\ 66.11 \pm 0.56$
MPNN-Max	$\mathbf{y}^{(i)} = U(\mathbf{x}^{(i)}, \max_j M(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}))$	$\mathcal{O}(E)$	0.468 ± 0.002	69.70 ± 0.55	77.07 ± 1.37	0.1552 ± 0.0022	$*\ 71.02 \pm 0.21$
PNA	$\mathbf{y}^{(i)} = U(\mathbf{x}^{(i)}, \bigoplus_j M(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}))$	$\mathcal{O}(E)$	0.320 ± 0.032	70.21 ± 0.15	$\mathbf{79.05 \pm 1.32}$	$*\ 0.1570 \pm 0.0032$	$*\ 71.21 \pm 0.30$
EGC-S *(Ours)*	Equation 2	$\mathcal{O}(V)$	0.364 ± 0.020	66.92 ± 0.37	77.44 ± 1.08	0.1528 ± 0.0025	$\mathbf{72.21 \pm 0.17}$
EGC-M *(Ours)*	Equation 3	$\mathcal{O}(V)$	$\mathbf{0.281 \pm 0.007}$	$\mathbf{71.03 \pm 0.42}$	78.18 ± 1.53	$\mathbf{0.1595 \pm 0.0019}$	71.96 ± 0.23

Table 1: Results (mean ± standard deviation) for *parameter-normalized* models run on 5 datasets. Details of the specific aggregators chosen per dataset and further experimental details can be found in the supplementary material. Results marked with $*$ ran out of memory on 11GB 1080Ti and 2080Ti GPUs. EGC obtains best performance on 4 of the tasks, with consistently wide margins.

4.2 Spectral Interpretation: Localised Spectral Filtering

We can also interpret our EGC-S layer through the lens of graph signal processing (Sandryhaila & Moura, 2013). Many modern graph neural networks build on the observation that the convolution operation for the Euclidean domain when generalised to the graph domain has 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 for 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.

The graph convolution operation is typically defined on the spectral domain as filtering the input signal $\mathbf{x} \in \mathbb{R}^N$ on a graph with N nodes with a filter g_θ parameterized by θ. This requires translating between the spectral and spatial domain using the Fourier transform. 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 $\mathbf{x} \in \mathbb{R}^N$ then 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 us 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 $\mathbf{Y} = \sum_{b=1}^{B} \mathbf{w}_b \odot (\tilde{\mathbf{D}}^{-\frac{1}{2}} \tilde{\mathbf{A}} \tilde{\mathbf{D}}^{-\frac{1}{2}}) \mathbf{X} \mathbf{\Theta}_b$, 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).

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

5 Evaluation

5.1 Protocol

We primarily 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

Model	Peak Training Memory (MB)	GPU Training Latency (ms)	GPU Inference Latency (ms)
GCN	1905	159.8 ± 4.6	35.2 ± 0.1
GIN	1756	155.2 ± 3.9	35.2 ± 0.1
GraphSAGE	1352	113.9 ± 5.4	25.1 ± 0.4
GAT	10841	324.3 ± 1.2	84.7 ± 0.3
MPNN-Sum	14323	768.2 ± 0.8	230.7 ± 0.3
MPNN-Max	14623	797.8 ± 0.9	258.2 ± 0.4
PNA	14533	892.7 ± 1.1	305.7 ± 0.5
EGC-S	2430	177.7 ± 2.2	37.3 ± 0.1
EGC-M	4068	220.9 ± 0.6	42.2 ± 0.3

Table 2: Memory and latency statistics for ***parameter-normalized*** models (used in table 1) on Arxiv. Note that EGC-M memory consumption and latency can be reduced with aggregator fusion at inference time.

Method	Test Accuracy % ↑
MLP	26.92 ± 0.26
GCN	30.43 ± 0.25
GraphSAGE-Mean	31.53 ± 0.15
EGC-S	32.13 ± 0.73
EGC ($\oplus$ = Mean)	33.22 ± 0.50
R-GCN (Full Batch)	46.29 ± 0.45
R-EGC (Full Batch)	47.21 ± 0.40

Table 3: Results on OGB-MAG. EGC performs strongly, and can be extended to heterogeneous graphs, obtaining a 0.9% improvement.

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 and splits specified by these papers. Baseline architectures chosen reflect popular general-purpose choices (Kipf & Welling, 2017; Xu et al., 2019; Hamilton et al., 2017; Veličković et al., 2018; Gilmer et al., 2017), along with the recent state-of-the-art PNA (Corso et al., 2020). Code and pre-trained models are supplied in the supplementary material.

In order to provide a fair comparison we standardize all parameter counts, architectures and optimizers in our experiments. All experiments were run using Adam (Kingma & Ba, 2014). Further details on how we ensured a fair evaluation can be found in the appendix.

We do not use edge features in our experiments as for most baseline architectures there exist no standard method to incorporate them. We do not use sampling, which, as explained in Section 2.2, is ineffective for 4 datasets; for the remaining dataset, Arxiv, we believe it is not in the scientific interest to introduce an additional variable. All experiments were run 10 times.

5.2 Main Results

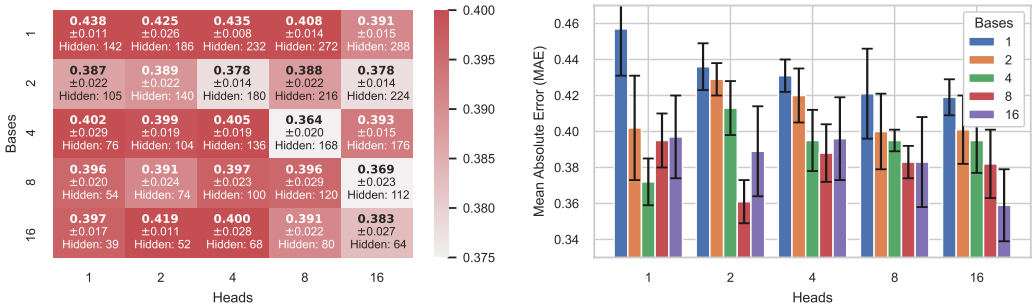
Our results across the 5 tasks are shown in Table 1. We draw attention to the following observations:

- **EGC-S is competitive with anisotropic approaches**. We outperform GAT and MPNN-Sum on all benchmarks, despite our resource efficiency. The exception is MPNN-Max on CIFAR & Code, where the max aggregator provides a stronger inductive bias.
- **EGC-M obtains state-of-the-art performance**. The addition of multiple aggregator functions improves performance of EGC to beyond that obtained by PNA. 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 believe that 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 with 48GB VRAM to run these experiments. PNA, our closest competing technique accuracy-wise, exhausted memory on the Code benchmark as well. Detailed memory consumption figures are provided in Table 2.

Overall, EGC obtains the best performance on 4 out of the 5 main 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 Additional Studies

How Should Heads and Bases be Chosen? 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 3.



(a) Constant parameter count (100k)

(b) Constant hidden dimension (128)

Figure 3: Study over the number of heads (H) and bases (B). Study run on ZINC dataset with EGC-S. Metric is MAE (mean $\pm$ standard deviation): lower is better. We study keeping the total parameter count constant, and fixing the hidden dimension. Each experiment was tuned individually. Setting $B > H$ does not necessarily improve performance due to the risk of overfitting, and forces the usage of a smaller hidden dimension to retain a constant parameter count.

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 3. 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 3 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 $B = H$ or $B = \frac{H}{2}$. We find $H = 8$ to be effective with EGC-S; for EGC-M, where more parameters are spent on combination weights, we advise setting $H = 4$. This convention is applied consistently for Table 1; full details are supplied in the appendix and code.

Should The Combination Weightings (w) Be Activated? Any activation function will shrink the space the per-node weights $\mathbf{\Theta}_h^{(i)}$ can lie in, hence we would expect it to harm performance; this is verified in Table 4. Activating w may improve training stability, but we did not observe to be an issue in our experiments. Another issue is that different aggregators result in outputs with different means and variances (Tailor et al., 2021), hence they need to be scaled by different factors to be combined.

Activation	EGC-S	EGC-M
Identity	0.364 ± 0.020	0.281 ± 0.007
Hardtanh	0.435 ± 0.010	0.293 ± 0.013
Sigmoid	0.366 ± 0.008	0.303 ± 0.016
Softmax	0.404 ± 0.010	0.307 ± 0.013

Table 4: Activating the combination weightings w harms performance. Run on ZINC; lower is better.

Applying EGC to Large-Scale Heterogeneous Graphs We evaluated EGC on the heterogeneous OGB-MAG dataset, containing 2M nodes and 21M edges. On a simplified, homogeneous, version of the graph, we exceed the baselines' performance by 1.5-2%; we experiment with both symmetric normalization (EGC-S) and the mean aggregators to demonstrate that the mechanism utilized by EGC is effective, regardless of which aggregator provides the stronger inductive bias for the dataset. Our architecture can be expanded to handle different edge types, yielding the R-EGC architecture which improves performance over R-GCN by 0.9%. We expect that accuracy can be further improved by using sampling techniques to regularize the optimization, or using pretrained embed-

Accuracy-Normalized **Model**	**Parameters**	**GPU Training Latency (ms)**	**GPU Inference Latency (ms)**	**Peak Memory (MB)**
GCN	184k	208.6 ± 2.2	44.9 ± 0.4	2549
GIN	*FAIL*	*FAIL*	*FAIL*	*FAIL*
GraphSAGE	593k	335.8 ± 2.4	60.2 ± 0.2	3208
EGC-S	100k	177.7 ± 2.2	37.3 ± 0.1	2430

Table 5: Latency and memory statistics for ***accuracy-normalized*** models on Arxiv. To achieve the same accuracy as EGC-S, we must boost the size of the baseline models. We observe that EGC-S offers noticeable reductions to both memory consumption and latency for a given accuracy level.

dings; however, adding these techniques makes comparing the results more difficult as it is known that sampling techniques can affect each architecture's performance differently (Liu et al., 2020).

5.4 Memory and Latency Benchmarks

We now assess our model's resource efficiency. For CPU measurements we used an Intel Xeon Gold 5218 and for GPU we used an Nvidia RTX 8000.

Aggregator Fusion We evaluated aggregator fusion across several topologies on both CPU and GPU. For space reasons, we leave full details to Appendix C. In summary, we observe that using 3 aggregators over standard SpMM incurs an additional overhead of only **14%**, enabling us to improve model performance without excessive computational overheads at inference time.

End-to-End Latency We provide latency and memory statistics for the *parameter-normalized* models in Table 2. We draw attention to how slow and memory-intensive the $\mathcal{O}(E)$ models are for both training and inference. The reader should note that the inference latency and memory consumption for MPNN and PNA rises by approximately 6-7$\times$ relative to EGC, corresponding to the large $\frac{|E|}{|V|}$ ratio for Arxiv. EGC-M offers substantially lower latency and memory consumption than its nearest competitor, PNA, as $|E|$ is often $\geq 2|V|$ due to the graphs being directed. We report results on other datasets in Appendix D.

So far we have not demonstrated that our approach is more efficient than the baselines on Arxiv, which is the only dataset where EGC-M and PNA are not the best performing. To demonstrate that we are more efficient we must show that we achieve lower memory consumption and latency for a given accuracy-level. We evaluate increasing the parameter count for baseline models until they achieve the same accuracy as EGC-S; we gave these models an extra advantage over our method by increasing the hyperparameter search budget. The results are shown in Table 5, where we observe that EGC-S is more efficient once we are comparing models achieving the same accuracy.

6 Discussion and Conclusion

How Surprising Are Our Results? Our results contradict common wisdom in the GNN community: they directly challenge the observations regarding anisotropy improving performance described in the widely cited benchmarking work by Dwivedi et al. (2020). Our observations raise important questions regarding our understanding of GNN optimization: we believe that although certain baselines are more expressive, they are more difficult to train. Related observations to ours have been made in the NLP community: it has been demonstrated that convolutional-style networks can be competitive with Transformers in many cases (Tay et al., 2021).

This work has made a 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 6 different benchmark datasets compared to strong baselines. We have demonstrated that our approach can be used to replace anisotropic mechanisms currently pervasive in GNN architecture design to improve parameter efficiency, memory consumption and latency. In summary, our work raises important questions for the research community, while offering significant practical benefits with regard to resource consumption.

ETHICS STATEMENT

The method described in this paper is generic enough that it can be applied to any problem GNNs are applied to. The ethical concerns associated with this work are related to enabling more efficient training and deployment of GNNs. This may be positive (drug discovery) or negative (surveillance), but these concerns are inherent to any work investigating efficiency for machine learning systems.

REPRODUCIBILITY STATEMENT

We have supplied the code required to regenerate our results, along with the hyperparameters required. In addition, we supply pre-trained models. Please refer to the supplementary material for more details.

REFERENCES

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

Dominique Beaini, Saro Passaro, Vincent Létourneau, William L. Hamilton, Gabriele Corso, and Pietro Liò. Directional graph networks, 2021.

Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247*, 2018a.

Lei Chen, Zhengdao Chen, and Joan Bruna. On graph neural networks versus graph-augmented {mlp}s. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=tiqI7w64JG2.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 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.

Xiuyuan Cheng, Zichen Miao, and Qiang Qiu. Graph convolution with low-rank learnable local filters. In *International Conference on Learning Representations*, 2021.

Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 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.

Intel Corporation. Intel® 64 and ia-32 architectures optimization reference manual.

Gabriele Corso, Luca Cavalleri, Dominique Beaini, Pietro Liò, and Petar Veličković. Principal Neighbourhood Aggregation for Graph Nets. *arXiv:2004.05718 [cs, stat]*, June 2020. URL http://arxiv.org/abs/2004.05718. arXiv: 2004.05718.

Vijay Prakash Dwivedi, Chaitanya K. Joshi, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Benchmarking Graph Neural Networks. *arXiv:2003.00982 [cs, stat]*, July 2020. URL http://arxiv.org/abs/2003.00982. arXiv: 2003.00982.

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

Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International Conference on Machine Learning*, pp. 1263–1272. PMLR, 2017.

Aidan N. Gomez, Mengye Ren, Raquel Urtasun, and Roger B. Grosse. The reversible residual network: Backpropagation without storing activations, 2017.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Jian Yin, Daxin Jiang, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. *arXiv preprint arXiv:1706.02216*, 2017.

Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687*, 2020.

Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. Adaptive sampling towards fast graph representation learning. *arXiv preprint arXiv:1809.05343*, 2018.

Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems*, 2:187–198, 2020.

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

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

Guohao Li, Matthias Müller, Bernard Ghanem, and Vladlen Koltun. Training graph neural networks with 1000 layers, 2021.

Ziqi Liu, Zhengwei Wu, Zhiqiang Zhang, Jun Zhou, Shuang Yang, Le Song, and Yuan Qi. Bandit samplers for training graph neural networks, 2020.

Tobias Pfaff, Meire Fortunato, Alvaro Sanchez-Gonzalez, and Peter W Battaglia. Learning mesh-based simulation with graph networks. *arXiv preprint arXiv:2010.03409*, 2020.

Charles R Qi, Li Yi, Hao Su, and Leonidas J Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. *arXiv preprint arXiv:1706.02413*, 2017.

Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. Dropedge: Towards deep graph convolutional networks on node classification, 2020.

Emanuele Rossi, Fabrizio Frasca, Ben Chamberlain, Davide Eynard, Michael Bronstein, and Federico Monti. Sign: Scalable inception graph neural networks. *arXiv preprint arXiv:2004.11198*, 2020.

Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter W Battaglia. Learning to simulate complex physics with graph networks. *arXiv preprint arXiv:2002.09405*, 2020.

Aliaksei Sandryhaila and José MF Moura. Discrete signal processing on graphs. *IEEE transactions on signal processing*, 61(7):1644–1656, 2013.

Paul-Edouard Sarlin, Daniel DeTone, Tomasz Malisiewicz, and Andrew Rabinovich. 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.

Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*, pp. 593–607. Springer, 2018.

Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks. *Synthesis Lectures on Computer Architecture*, 15(2):1–341, 2020.

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

Yi Tay, Mostafa Dehghani, Jai Gupta, Dara Bahri, Vamsi Aribandi, Zhen Qin, and Donald Metzler. Are pre-trained convolutions better than pre-trained transformers?, 2021.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (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.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018.

Felix Wu, Angela Fan, Alexei Baevski, Yann N. Dauphin, and Michael Auli. Pay less attention with lightweight and dynamic convolutions, 2019a.

Felix Wu, Tianyi Zhang, Amauri Holanda de Souza Jr. au2, Christopher Fifty, Tao Yu, and Kilian Q. Weinberger. Simplifying graph convolutional networks, 2019b.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How Powerful are Graph Neural Networks? *arXiv:1810.00826 [cs, stat]*, February 2019. URL http://arxiv.org/abs/1810.00826. arXiv: 1810.00826.

Mutian Xu, Runyu Ding, Hengshuang Zhao, and Xiaojuan Qi. Paconv: Position adaptive convolution with dynamic kernel assembling on point clouds, 2021.

Carl Yang, Aydin Buluc, and John D. Owens. Design principles for sparse matrix multiplication on the gpu, 2018.

Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. Graph-saint: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931*, 2019.

Yiren Zhao, Duo Wang, Xitong Gao, Robert Mullins, Pietro Lio, and Mateja Jamnik. Probabilistic dual network architecture search on graphs. *arXiv preprint arXiv:2003.09676*, 2020.

Dataset	Add	Mean	Symmetric Normalization	Max	Min	Std	Var	Result
Zinc	✓			✓		✓		0.281 ± 0.007
Zinc			✓	✓	✓			0.284 ± 0.045
CIFAR	✓			✓		✓		71.03 ± 0.42
CIFAR			✓	✓	✓			70.05 ± 1.14
MolHIV	✓	✓		✓				78.19 ± 1.54
MolHIV	✓			✓			✓	77.40 ± 1.02
MolHIV	✓			✓		✓		77.98 ± 1.65
Arxiv		✓	✓	✓				71.96 ± 0.23
Arxiv	✓	✓	✓					70.59 ± 0.67
Arxiv	✓	✓		✓				70.38 ± 0.76
Code-V2			✓	✓	✓			0.1595 ± 0.0019
Code-V2				✓	✓	✓		0.1572 ± 0.0029
Code-V2	✓			✓		✓		0.1578 ± 0.0021

Table 6: Possible aggregators tried for EGC-M. Up to 3 combinations (from a possible 35) were tried, as we limited ourselves to always using 3 aggregators. Our conclusions do not change, and it is likely that better results can be found with more optimal aggregator choices.

A Further Experiment Details

A.1 Ensuring Fairness

We expand on our experimental protocol, with a particular focus on describing measures that we took to ensure that the results we report are not unfairly biased towards EGC.

For EGC-S, we use $H = 8$ and $B = 4$, as implied by our ablation for all experiments, with the single exception of OGB-Code, where we use $H = B = 8$. The benefit of using a smaller set of bases is that we can increase the hidden dimension, but this is not viable in this case since most of the 11M parameters in the model correspond to the token read-out layers, which quickly increases as the model hidden dimension grows. As shown by section 5.3, if we cannot increase the hidden dimension, it is better to increase the bases. This is the only exception we make.

For EGC-M, we use $H = B = 4$ for all experiments. The main challenge is aggregator selection, and this remains a major challenge for our work. We were unable to find a satisfactory technique for automated discovery of aggregator choices, hence we rely on heuristics to find them. We restrict ourselves to use 3 aggregators for each model (yielding 35 possible choices). In order to determine the aggregators, we use two heuristics: (1) aggregators should be "diverse" and (2) aggregators should be chosen based on inductive bias for task. Using these two rules, we try up to 3 possible choices of aggregators; all choices considered are shown in Table 6. We note that while some choices do improve performance, our conclusions are not invalidated; it is also worth noting that it is likely that better aggregator choices can be found.

A.2 Cluster Details

Most of our experiments were run on several machines in our SLURM cluster using Intel CPUs and NVIDIA GPUs. Each machine was running Ubuntu 18.04. The GPU models in our cluster were RTX 2080Ti and GTX 1080Ti. High-memory experiments were run on V100s in our cluster and an RTX 8000 virtual machine we had access to.

B Limitations of Existing Approaches

As explained in the related work, existing approaches to improving GNN efficiency have severe limitations. In this section we elaborate upon them, and provide experimental evidence where necessary.

Model	ZINC (MAE ↓)	CIFAR (Acc. ↑)	MolHIV (ROC-AUC ↑)	Code-V2 (F1 ↑)
GA-MLP	0.510 ± 0.037	58.13 ± 0.65	75.50 ± 1.32	0.1485 ± 0.0027
GCN	0.459 ± 0.006	55.71 ± 0.38	76.14 ± 1.29	0.1480 ± 0.0018
EGC-S	0.364 ± 0.020	66.92 ± 0.37	77.44 ± 1.08	0.1528 ± 0.0025

Table 7: Results of applying graph-augmented MLPs (GA-MLPs) to tasks requiring generalization to unseen graphs. We see that the performance is broadly similar to the corresponding GCN model—and far weaker than EGC-S, which uses the same aggregator. We also considered using the add aggregation on ZINC, and achieved 0.444 ± 0.019. By comparison, the equivalent GIN model (which uses the add aggregation) achieved 0.387 ± 0.015.

Experiment	ZINC (MAE ↓)	CIFAR (Acc. ↑)	MolHIV (ROC-AUC ↑)	Code-V2 (F1 ↑)
EGC-S	0.364 ± 0.020	66.92 ± 0.37	77.44 ± 1.08	0.1528 ± 0.0025
EGC-S + DropEdge ($p = 0.1$)	0.468 ± 0.007	66.37 ± 0.28	77.41 ± 1.32	0.1553 ± 0.0021
EGC-S + DropEdge ($p = 0.5$)	0.629 ± 0.023	64.60 ± 0.58	75.33 ± 0.82	0.1527 ± 0.0019
EGC-S + GraphSAINT Node Sampler ($p = 0.1$)	0.631 ± 0.012	61.37 ± 0.75	73.65 ± 1.41	0.1461 ± 0.0027

Table 8: Results of applying sampling approaches to EGC-S. We do not enable sampling at test time—hence these approaches do not offer any test time reductions to computation. Sampling, in principle, is applicable to any underlying GNN: our conclusions will transfer to other underlying GNNs. We see that DropEdge (Rong et al., 2020) with a low drop probability ($p = 0.1$) can aid model performance; however, setting p this low does not significantly reduce memory consumption or computation. Increasing p to 0.5 does reduce resource consumption noticeably, but results in noticeable degradation to model performance.

We first examine graph-augmented MLPs (GA-MLPs); to our knowledge there are few experimental results assessing the performance of these models when they are applied to problems requiring generalization to unseen graphs. In the literature they are generally applied to large scale node classification benchmarks, such as those found in the OGB benchmark suite (Wu et al., 2019b; Rossi et al., 2020). It is known that these models are theoretically less expressive than standard GNNs, however their performance when applied to node classification datasets has been acceptable.

We consider models of the form:

$$\mathbf{Y} = \text{Readout}(\text{MLP}(\mathop{\Big\|}_{k=0}^{4} \mathbf{S}^k \mathbf{X} \mathbf{W})) \tag{5}$$

We use up to the 4th power of the diffusion operator $\mathbf{S}$ to emulate the depth of the corresponding GNNs, which all use 4 layers. We set $\mathbf{S}$ to use symmetric normalization, as used by GCN and EGC-S; we also consider setting $\mathbf{S}$ to the adjacency matrix on ZINC, to emulate the operations used by GIN. The results are provided in Table 7. We see that the GA-MLP models offer similar (but often worse) performance than the corresponding GNN baselines. The models are not competitive with approaches such as GAT or MPNN, and is outperformed by a wide margin by EGC-S. Achieving state-of-the-art performance with GA-MLPs does not appear to be possible, at least with our current understanding of these models.

We now proceed to investigate sampling, in which each training step does not run over the entire graph, but over some sampled subgraph. These methods have seen great popularity when applied to tasks such as node classification, and they are able to deliver sampling ratios in excess of $20\times$, hence yielding noticeable improvements to memory consumption (the primary limitation with large scale training). However, we note that the benefit of these methods has not been examined carefully for many graph topologies; while they have been shown to be effective for many "small-world" graph topologies—which arise in many graphs—it is not the case that all graphs fall into this category. For example, molecule graphs would not fit this category.

We first assess the sampling strategy from GraphSAINT (Zeng et al., 2019) by applying it to EGC-S models. For our experiments, we use the node-centric sampler. The results are presented in Table 8; we disable sampling at test time. Even with a relatively low dropping probability of 10% (i.e. 90% of nodes are retained), the model performance degradation is severe. We note that the computational savings achieved are modest when using this drop probablility. We also observed similar results when using the edge-centric sampler from GraphSAINT.

We also attempted a different approach for sampling proposed by DropEdge (Rong et al., 2020); as before, we apply it to EGC-S models. In this simple scheme, elements of the adjacency matrix are dropped; this scheme was shown to be effective for training deep graph networks, but it is also useful for reducing the computational footprint of models, since it effectively reduces the number of messages that have to be computed. We also provide results in Table 8. The results are significantly better than we observe when using GraphSAINT's sampling strategies: at 10% drop probabilities we even see an improvement in some cases, due to the regularization effect. However, once we increase the drop probability to levels where we would observe a noticeable reduction in computational demand, we observe that model performance declines. In summary, while DropEdge is a more effective strategy for sampling on many inductive tasks, it is not beneficial as a method to reduce the computational burden.

Finally, we discuss the limitations of other approaches proposed in the literature. Quantization is an approach that is typically applied at inference time; mixed-precision approaches can be applied at training time, however care must be taken for GNNs to avoid biasing the gradients (Tailor et al., 2021). Additionally, while neural architecture search (NAS) may be useful to find memory-efficient models if the search objective is set appropriately, they suffer from some limitations—primarily search time and memory consumption. Finally, approaches such as reversible residuals are useful to architecture design, they do not tackle issues such as high peak memory usage induced by the message passing step.

C AGGREGATOR FUSION

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 $H = B = 1$. This version demonstrates how we can remove memory overheads at inference time.

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

The naive approach of performing each aggregation sequentially would cause a linear increase in latency with respect to $|\mathcal{A}|$. However, a key observation to note is that we are *memory-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 performs as much work as possible with data that has

	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 ± 0.00	0.423 ± 0.023	0.055 ± 0.013	0.074 ± 0.010	2.36 ± 0.00	13.66 ± 0.12	1.74 ± 0.01	2.36 ± 0.02
CSR SpMM	29.08 ± 0.12	1.943 ± 0.040	0.760 ± 0.021	0.315 ± 0.006	186.44 ± 0.05	19.88 ± 0.10	5.56 ± 0.02	3.39 ± 0.01
Naive Fusion	88.25 ± 0.20	8.680 ± 0.094	2.631 ± 0.016	1.482 ± 0.010	595.91 ± 0.13	112.52 ± 0.22	23.81 ± 0.06	19.09 ± 0.05
+ **Faster Ordering**	40.05 ± 0.10	4.592 ± 0.054	1.303 ± 0.037	0.772 ± 0.008	214.60 ± 0.10	29.38 ± 0.12	7.63 ± 0.03	4.78 ± 0.02
+ **Store Weighted Result Only**	38.63 ± 0.22	1.752 ± 0.018	0.952 ± 0.026	0.278 ± 0.003	208.22 ± 0.13	26.64 ± 0.09	6.75 ± 0.03	4.38 ± 0.02

Table 9: 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 34% in the worse case; in contrast, the naive implementation has a worst case increase of 466%. Also included are timings for dense multiplication with a square weight matrix; we observe that sparse operations dominate latency measurements.

already been fetched from memory, rather than fetching it multiple times. This concept is illustrated in Algorithm 1 in the appendix. We can perform all aggregations as a lightweight modification to the standard compressed sparse row (CSR) SpMM algorithm.

The second observation we make is that storing the results of all aggregations is unnecessary at inference time. Note that the CSR SpMM algorithm processes each row in the output matrix sequentially: rather than storing the aggregations for every row, instead we should store only the weighted results. This approach not only reduces memory consumption, but also latency as we improve the effectiveness of our cache and reduce memory system contention. In practice, this optimization is especially important when performing inference on topologies which have more frequent periods where the processing has become compute-bound, since we reduce contention between load and store units Corporation. This is also demonstrated in Algorithm 1.

We evaluated aggregator fusion across four different topologies, on both CPU and GPU; our results can be found in Table 9. 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 331%, our approach incurs a mean increase of **only 14%** 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. $\mathbf{X\Theta}$) to justify our focus on optimizing sparse operations in this work: CSR SpMM operation is **4.7**$\times$ slower (geomean) than the corresponding weight multiplication. We believe further optimizations of the 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.

D Latency and Memory Consumption on Other Datasets

In the evaluation, we assessed the memory consumption and latency for the parameter-normalized models on Arxiv. In this section, we consider a similar exercise for OGB Code models. The results are provided in Table 10. Our conclusions remain broadly similar, with EGC-M offering clear improvements to memory consumption, latency, and parameter efficiency relative to PNA. EGC-S is superior to GAT, with similar inference latency, better model performance, and noticeably lower memory consumption.

We train models using batch size 128; the reader should note that the memory consumption figures can vary between runs (even for the same model), since graphs vary in the number of nodes and edges.

Model	Train Epoch Time (s)	Test Epoch Time (s)	Peak Train Memory (MB)
GCN	144.7 ± 0.5	6.3 ± 0.3	1337
GIN	134.5 ± 0.1	5.7 ± 0.3	1331
GraphSAGE	140.3 ± 0.3	5.6 ± 0.3	1226
GAT	176.0 ± 0.7	6.3 ± 0.3	2885
MPNN-Sum	305.3 ± 2.7	6.8 ± 0.3	3448
MPNN-Max	319.0 ± 1.3	7.6 ± 0.4	3901
PNA	575.4 ± 2.3	12.0 ± 0.2	9399
EGC-S	166.4 ± 2.7	6.2 ± 0.3	1842
EGC-M	225.7 ± 0.6	7.9 ± 0.3	3470

Table 10: Latency and memory results for *parameter-normalized* models on OGB Code-V2. Despite having a lower $\frac{|E|}{|V|}$ ratio of 2.75 relative to Arxiv (13.67), we see that the trends we observed in for Arxiv broadly remain. It is worth noting again that EGC-M is far more efficient both latency and memory-wise than PNA.

E GENERALIZING TO HETEROGENEOUS GRAPHS

Our R-EGC model is similar to the baseline R-GCN model included in the OGB repository. The OGB model deviates from the standard definition of R-GCN (Schlichtkrull et al., 2018) since it handles different *node* types, not just edge types. The baseline model has weights to generate messages for each relation-type, and a weight matrix to update each individual node type. This corresponds to:

$$\mathbf{y}^{(i)} = \mathbf{\Theta}_{\eta}\mathbf{x}^{(i)} + \sum_{r \in \mathcal{R}} \frac{1}{|\mathcal{N}_r(i)|} \sum_{j \in \mathcal{N}_r(i)} \mathbf{\Theta}_r \mathbf{x}^{(j)} \tag{6}$$

where η corresponds to the *node type* of node i, and $\mathcal{R}$ represents the set of relation types. Note that the mean aggregator is used.

We deviate from the baseline by using a single set of basis weights. Instead, we use a different weighting calculation layers ($\mathbf{w}^{(i)} = \mathbf{\Phi}\mathbf{x} + \mathbf{b}$) per node and relation type.

$$\mathbf{y}^{(i)} = \mathop{\Big\|}_{h=1}^{H} \sum_{b=1}^{B} w^{(i)}_{\eta,h,b} \mathbf{\Theta}_b \mathbf{x}^{(j)} + \sum_{r \in \mathcal{R}} \frac{1}{|\mathcal{N}_r(i)|} \mathop{\Big\|}_{h=1}^{H} \sum_{b=1}^{B} w^{(i)}_{r,h,b} \sum_{j \in \mathcal{N}(i)} \mathbf{\Theta}_b \mathbf{x}^{(j)} \tag{7}$$

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.
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.
GraphSAGE (Hamilton et al., 2017)	$\mathbf{y}^{(i)} = \mathbf{\Theta}_1 \mathbf{x}^{(i)} + \bigoplus_{j \in \mathcal{N}(i)} \mathbf{\Theta}_2 \mathbf{x}^{(j)}$	$\mathcal{O}(V)$	$\bigoplus$ typically parameterized as mean or max.
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.
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 by default.

Table 11: Propagation rules for general-purpose 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.