

---

# ADAPTIVE FILTERS AND AGGREGATOR FUSION FOR EFFICIENT GRAPH CONVOLUTIONS

---

Shyam A. Tailor<sup>1</sup> Felix L. Opolka<sup>1</sup> Pietro Liò<sup>1</sup> Nicholas D. Lane<sup>1,2</sup>

## 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 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). Common approaches include techniques such as quantization (Jacob et al., 2018) and pruning (Blalock et al., 2020). Another 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 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 ranging from computer vision (Sarlin et al., 2020; Shi & Rajkumar, 2020) to code analysis (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. We use a novel approach, *adaptive filtering*, to obtain our results.
- 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

---

<sup>1</sup>Department of Computer Science and Technology, University of Cambridge, UK <sup>2</sup>Samsung AI Center, Cambridge, UK. Correspondence to: Shyam A. Tailor <sat62@cam.ac.uk>.

across 5 large graph datasets covering both transductive and inductive use-cases. 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 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  $\phi$  that calculates messages from node  $u$  to node  $v$ , a differentiable and permutation-invariant aggregator  $\oplus$ , and an update function  $\gamma$  to calculate representations at layer  $l + 1$ :  $\mathbf{h}_{l+1}^{(i)} = \gamma(\mathbf{h}_l^{(i)}, \oplus_{j \in \mathcal{N}(i)} [\phi(\mathbf{h}_l^{(i)}, \mathbf{h}_l^{(j)}, \mathbf{e}_{ij})])$ , where  $\mathcal{N}(i)$  denotes the in-neighbours of  $i$ .

In this work we evaluate against popular and recent high-performance graph architectures: GCN (Kipf & Welling, 2017), GAT (Veličković et al., 2018), GIN (Xu et al., 2019), MPNN (Gilmer et al., 2017) and PNA (Corso et al., 2020). We draw the reader’s attention to the fact that GCN and GIN can be implemented using sparse matrix multiplication (SpMM) rather than using the node-wise formulation that relies on scattering, which has favorable hardware characteristics that we expand on later.

**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).

For many applications, *deploying* our models and handling unseen graphs is the challenge—not scaling 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. 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).

## 3 OUR ARCHITECTURE: EFFICIENT GRAPH CONVOLUTION (EGC)

In this section we describe our architecture, *EGC*. We present two versions: *EGC-S*, which uses a single aggregator, and *EGC-M* which generalizes our approach by incorporating multiple aggregation functions. We also present *aggregator fusion* to enable us to implement *EGC-M* with minimal latency overheads.

### 3.1 Architecture Description

For a layer with in-dimension of  $F$  and out-dimension of  $F'$  we use  $B$  basis weights  $\Theta_i \in \mathbb{R}^{F' \times F}$ . 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  $\Theta_i$ . We calculate  $\mathbf{w}^{(i)} = \Phi \mathbf{x}^{(i)} + \mathbf{b}$ , where  $\Phi \in \mathbb{R}^{B \times F}$  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:

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

where  $\alpha(i, j)$  is some function of nodes  $i$  and  $j$ . A popular method pioneered by GAT to boost representational power is to represent  $\alpha$  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  $\alpha$  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 and consistent results across a wide variety of tasks.

**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. 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  $\parallel$  as the concatenation operator, and making the use of symmetric normalization explicit, we obtain the **EGC-S** layer:

$$\mathbf{y}^{(i)} = \parallel_{h=1}^H \sum_{b=1}^B w_{h,b}^{(i)} \sum_{j \in \mathcal{N}^{(i)} \cup \{i\}} \frac{1}{\sqrt{\deg(i)\deg(j)}} \Theta_b \mathbf{x}^{(j)} \quad (2)$$

### 3.2 Boosting Representational Capacity

Recent work by Corso et al. (2020) has shown theoretically and empirically that using only a single aggregator is sub-optimal. 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 representations calculated by  $\Theta_b \mathbf{x}^{(j)}$ . 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:

$$\mathbf{y}^{(i)} = \parallel_{h=1}^H \sum_{\oplus \in \mathcal{A}} \sum_{b=1}^B w_{h,\oplus,b}^{(i)} \bigoplus_{j \in \mathcal{N}^{(i)} \cup \{i\}} \Theta_b \mathbf{x}^{(j)} \quad (3)$$

where  $\oplus$  is an aggregator. 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. 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.

In other words, the loop over our aggregators should be the inner-most loop; performing the aggregations sequentially would correspond to having this loop over aggregators at the outer-most level. 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. We note that it may be faster to interleave these row-processing calls to utilize the processor’s functional units in parallel; this trade-off will be architecture specific.

## 4 INTERPRETATION AND BENEFITS

This section will provide explanations for our design choices. We will also explain why our proposal is better suited to hardware implementation than competing approaches.

### 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 factorizing the  $\Theta_b$  terms out of inner sum. Building upon Equation (2) we obtain:

$$\mathbf{y}^{(i)} = \parallel_{h=1}^H \underbrace{\Theta_h^{(i)}}_{\text{Varying per Node}} \underbrace{\left( \sum_{j \in \mathcal{N}^{(i)} \cup \{i\}} \frac{\mathbf{x}^{(j)}}{\sqrt{\deg(i)\deg(j)}} \right)}_{\text{Computable via SpMM}}$$

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

$$\mathbf{y}^{(i)} = \underbrace{\Theta}_{\text{Shared Weights}} \underbrace{\left( \sum_{j \in \mathcal{N}^{(i)} \cup \{i\}} \alpha_{i,j} \mathbf{x}^{(j)} \right)}_{\text{Explicit Message Materialization}}$$

where  $\alpha_{i,j}$  represent attention coefficients that are calculated from  $\mathbf{x}^{(i)}$  and  $\mathbf{x}^{(j)}$  as defined by Veličković et al. (2018). From an efficiency perspective, we can observe that our approach has significantly better characteristics, and, as we will demonstrate in the evaluation, our method is still sufficiently expressive to obtain state-of-the-art results.

**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 importance for filtering digital signals and images. Many GNNs 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. 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 an application of the Adaptive Linear Combiner method (Widrow & Winter, 1988), which to the best of our knowledge has not been applied in modern deep learning literature.

## 4.2 Interaction with Hardware

Our architecture is substantially more hardware friendly than existing state-of-the-art graph architectures. As mentioned earlier, by choosing a fixed representation for  $\alpha(i, j)$ , we can implement the message propagation step using SpMM alone i.e.  $\mathbf{Y}_b = \mathbf{S}\mathbf{X}\Theta_b$ . We do not need to explicitly materialize the messages between nodes, cutting our memory usage from  $\mathcal{O}(|E|)$  to  $\mathcal{O}(|V|)$ . 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 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 is challenging 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). Adding support for flexible architectures to an accelerator inherently requires trading design complexity & area, peak performance, and efficiency. We also note that SpMM acceleration is likely to become common due to the popularity of pruning.

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 memory accesses. Aggregator fusion also benefits energy consumption since it reduces data move-

ment. Reducing data movement is a critical for achieving low energy consumption: a 32-bit floating-point add costs 0.9pJ, but a 32-bit DRAM read costs 640pJ (Horowitz, 2014)—3 orders of magnitude higher.

## 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). 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 used in Dwivedi et al. (2020) and Corso et al. (2020); this corresponds to 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. All experiments were run 10 times. We do not use sampling, which is not applicable to 4 of the datasets. Full details, including choices of aggregators for EGC-M, and pre-trained models can be found in the released code.

### 5.2 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 architectures.** GAT and MPNN are comparable architectures using one aggregator; we see across all tasks that we obtain similar, or better, performance. The only exception is MPNN-Max on CIFAR-10 and Code, where the choice of max aggregator provides a better inductive

Table 1. 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 ↓)	CIFAR (Acc. ↑)	MolHIV (ROC-AUC ↑)	Arxiv (Acc. ↑)	Code-V2 (F1 ↑)
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 ± 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 ± 0.56	0.1470 ± 0.0017
MPNN-Max	0.468 ± 0.002	69.70 ± 0.55	77.07 ± 1.37	* 71.02 ± 0.21	0.1552 ± 0.0022
PNA	0.320 ± 0.032	70.21 ± 0.15	<b>79.05 ± 1.32</b>	* 71.21 ± 0.30	* 0.1570 ± 0.0032
EGC-S	0.364 ± 0.020	66.63 ± 0.26	77.21 ± 1.10	<b>72.19 ± 0.16</b>	0.1528 ± 0.0025
EGC-M	<b>0.281 ± 0.008</b>	<b>71.04 ± 0.45</b>	78.18 ± 1.53	71.96 ± 0.23	<b>0.1595 ± 0.0019</b>

bias than the sum-based aggregator used by EGC-S.

- 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 obtains good performance 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, 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  $\mathbf{a}_{\text{source}}^{\top} \Theta \mathbf{x}^{(i)}$  and  $\mathbf{a}_{\text{target}}^{\top} \Theta \mathbf{x}^{(j)}$  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 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 (specifically EGC-S) retains many of the desirable properties of spectral models, and although we do not assess it in this work, we note that it may be

possible to apply our approach to higher order (spectral) convolutions which include 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.

### 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 is shown in Figure 1. 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 1(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 1(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  $H = 4$ .

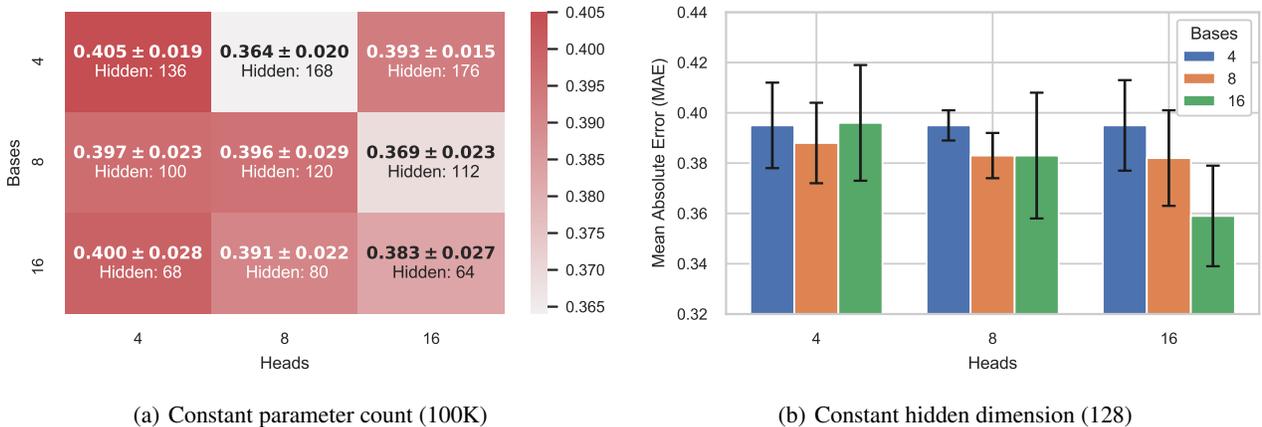


Figure 1. Ablation study over the number of heads ( $H$ ) and bases ( $B$ ). Study run on ZINC dataset with EGC-S. Metric is MAE (mean and standard deviation); lower is better. We study two regimes: keeping the total parameter count constant, and fixing the hidden dimension while varying  $H$  and  $B$ . 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 2. Inference latency (mean and standard deviation) for CSR SpMM and aggregator fusion. Assuming a feature dimension of 256 and  $H = B = 1$ . 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, justifying the value of aggregator fusion for end-to-end inference latency.

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

## 5.4 Latency Benchmarks

We evaluated aggregator fusion across four different topologies, on both CPU and GPU; our results can be found in Table 2. 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), one of the largest graph datasets evaluated on in the GNN literature.

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 sparse matrix multiplication. 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: the CSR SpMM operation is **7.9** $\times$  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. Our approach can be interpreted as enabling each node to have its own weight matrix. Despite our improved results, our proposal is more efficient than existing approaches: we require only  $\mathcal{O}(|V|)$  memory, which dramatically reduces data movement, improving energy-efficiency, and which enables us to train on larger graphs. Additionally, our approach is directly suited to the hardware: we can re-use existing accelerator designs, and incur only a minor latency increase due to our aggregator fusion approach.

## 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.
- Blalock, D., Ortiz, J. J. G., Frankle, J., and Gutttag, J. What is the state of neural network pruning? *arXiv preprint arXiv:2003.03033*, 2020.
- 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.
- 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., Haggi, 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.
- 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 outperforms 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.
- Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., and Kalenichenko, D. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2704–2713, 2018.
- 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.
- 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.
- Taylor, 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.
- Widrow, B. and Winter, R. Neural nets for adaptive filtering and adaptive pattern recognition. *Computer*, 21(3):25–39, 1988.
- 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.
- Zeng, H., Zhou, H., Srivastava, A., Kannan, R., and Prasanna, V. Graphsaint: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931*, 2019.
- Zhao, Y., Wang, D., Gao, X., Mullins, R., Lio, P., and Jammik, M. Probabilistic dual network architecture search on graphs. *arXiv preprint arXiv:2003.09676*, 2020.