GRIP: A Graph Neural Network Accelerator Architecture

Kevin Kiningham, Philip Levis, *Member, IEEE*, and Christopher Ré

Abstract—We present GRIP, a graph neural network accelerator architecture designed for low-latency inference. Accelerating GNNs is challenging because they combine two distinct types of computation: arithmetic-intensive *vertex-centric* operations and memory-intensive *edge-centric* operations. GRIP splits GNN inference into a three edge- and vertex-centric execution phases that can be implemented in hardware. GRIP specializes each unit for the unique computational structure found in each phase. For vertex-centric phases, GRIP uses a high performance matrix multiply engine coupled with a dedicated memory subsystem for weights to improve reuse. For edge-centric phases, GRIP use multiple parallel prefetch and reduction engines to alleviate the irregularity in memory accesses. Finally, GRIP supports several GNN optimizations, including an optimization called vertex-tiling that increases the reuse of weight data. We evaluate GRIP by performing synthesis and place and route for a $28\,\mathrm{nm}$ implementation capable of executing inference for several widely-used GNN models (GCN, GraphSAGE, G-GCN, and GIN). Across several benchmark graphs, it reduces 99th percentile latency by a geometric mean of $17\times$ and $23\times$ compared to a CPU and GPU baseline, respectively, while drawing only $5\,\mathrm{W}$.

Index Terms—Accelerator architectures, neural networks, hardware, system-on-chip, graph neural networks

1 Introduction

Traditional deep neural networks (DNNs) rely on regularly structured inputs, such as vectors, images, or sequences. This makes them difficult to use on irregular graphs, such as user connections on social media. Graph neural networks (GNNs) tackle these domains by extending DNNs to allow arbitrarily structured graph-valued inputs, where feature vectors are associated with the edges and vertices of a graph.[1] GNNs have found significant success in a range of practical tasks, from recommendations on social media [2] to improving circuit testability [3].

GNNs have two types of operations [4], [5]: *vertex-centric* and *edge-centric*. Vertex-centric operations are similar to traditional feed-forward neural networks operating on features associated with each vertex. This results in regular computational and memory access patterns with significant opportunities for data reuse. Edge-centric operations are similar to those found in graph analytics (e.g., neighborhood reduction [6]). Their computational structure depends on the sparse and irregular input graph structure, resulting in random memory accesses and limited data reuse.

While traditional architectures optimize for each type of computation individually, they do not optimize for both types combined. For example, most DNN accelerators (e.g., the TPU [7]) are optimized for dense, regular computation, which is inefficient for edge operations [8]. Graph analytics accelerators (e.g., Graphicionado [9]) are designed for workloads that require little computation per-vertex and have difficulty exploiting data reuse in vertex-centric operations. Likewise, CPUs and GPUs are not effectively optimized for GNNs, and prior work has demonstrated inference is limited by cache and memory bandwidth bottlenecks [10].

The demand for improved performance has lead to the development several accelerators designed for GNNs [11]. These designs largely optimize for high throughput, e.g., computing embeddings for all vertices within a large graph [2], [12]. However, they give much smaller speedups on latency. This poses a challenge as GNNs are increasingly deployed in applications that benefit from lower latency inference. For example, online deployments where inference is performed on a small number of vertices, such as recomputing recommendations as a social media user interacts with new content. Today, because of limitations on latency, embeddings for these applications are typically precomputed and cached, resulting in stale recommendations. By reducing GNN inference latency to less than a few milliseconds, the recommendations can instead be updated in near real-time, improving the results and the user experience.

This paper proposes GRIP (GRaph Inference Processor), an accelerator architecture designed for low-latency GNN inference. The two principal challenge in reducing GNN inference latency are 1) the amount of time spent on overhead tasks like sampling edges and 2) the high latency penalty of random access for graph information. GRIP addresses this by operating on a preprocessed representation of the the graph called a

1. Following the convention in prior work [1], for clarity we call a GNN's input a *graph* and the GNN itself a *network*.

- *Kevin Kiningham is with the Department of Electrical Engineering, Stanford University, Stanford, CA 94305 USA. E-mail: kkiningh@stanford.edu.*
- *Philip Levis is with Computer Systems Lab, Stanford University, Stanford, CA 94305 USA. E-mail: pal@cs.stanford.edu.*
- *Christopher Ré is with the Department of Computer Science, Stanford University, Stanford, CA 94305 USA. E-mail: chrismre@cs.stanford.edu.*

Manuscript received 24 May 2021; revised 11 February 2022; accepted 10 July 2022. Date of publication 17 August 2022; date of current version 13 March 2023.
This work was supported in part by the National Science Foundation under Grant 1505728, and in part by the Stanford Secure Internet of Things Project and the Stanford System X Alliance.
(Corresponding author: Kevin Kiningham.)
Recommended for acceptance by A. Karanth.
Digital Object Identifier no. 10.1109/TC.2022.3197083

nodeflow [13], allowing for operations like sampling to be precomputed as the graph is constructed and can be easily updated as edges are added or removed. In addition, using the nodeflow instead of the original graph allows all memory accesses to be determined statically, eliminating the need for complex dynamic caching mechanisms, such as used in state-of-the-art throughput optimized designs like HyGCN [10]. It also increases locality by locating edge information required for inference on a particular vertex in a single linear block of memory.

GRIP also uses a programming model that decomposes GNN inference into three phases following GReTA [14]: *aggregate*, *combine*, and *update*. GRIP implements each phase with specialized on-chip memory and execution units. For example, GRIP alleviates irregularity in the edge-centric *aggregate* phase by using multiple parallel prefetch engines to load data. This increases flexibility compared to prior approaches that use two stages (e.g., E/MVM [10]) by allowing computation to happen per-edge.

Finally, we also introduce and implement several optimizations that can be applied to general GRIP programs, such as caching partitions of feature data, pipelining computation between phases, preloading weights, and a optimization called vertex-tiling that substantially improves latency by increasing the reuse of weight values during inference.

1.1 Contributions

GRIP achieves low-latency inference on a wide range of GNNs by using three novel features in its design:

1) it proposes preprocessing a GNN into a nodeflow data structure, which allows it to access graph data with high locality;
2) it divides computation into three stages (aggregate, combine, and update) with specialized memory controllers for each stage, allowing it to execute a wider set of GNNs with higher performance than prior approaches;
3) it shows the high memory locality afforded by a nodeflow representation allows GRIP to be cacheless and have a small (128 KiB) feature memory,

Evaluated across several benchmark graphs, GRIP reduces 99th percentile latency by a geometric mean of 17× compared to an Intel Xeon CPU and 23× compared an Nvidia P100 GPU. It reduces latency by 4.5× compared to HyGCN [10], a GCN accelerator, and 8.1× compared to Graphicionado [9], a graph analytics accelerator.

2 Background and Motivation

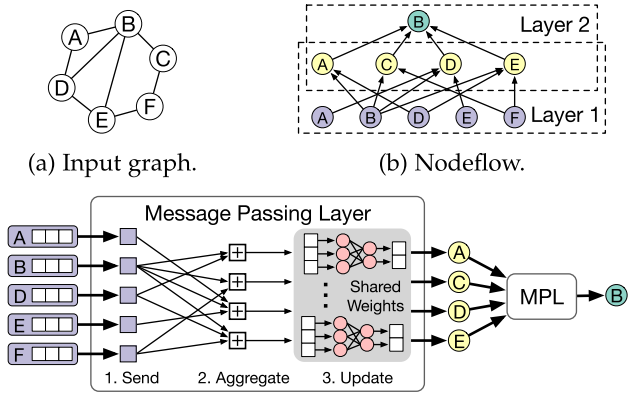
2.1 Graph Neural Networks

GNNs [15] are a class of DNN that operate on graph-valued data. GNNs differ from traditional DNNs by directly taking advantage of graph structure during learning and inference. For example, consider the task of classifying web-pages by topic. A pure content approach (e.g., a classic recurrent neural network) considers only features derived from a page's content. However, a significant amount of information is stored in the structure of links *between* pages. By modeling these links as a graph, a GNN can natively leverage both page content and link structure.

Message-Passing Layer. Modern GNNs are typically composed of multiple message-passing layers [4], shown in Algorithm 1. A layer takes as input a graph G consisting of a set of vertices V and edges E. Each vertex and edge is assigned a feature vector $\boldsymbol{h}_v$ and $\boldsymbol{h}_{(u,v)}$ respectively. Computation is split into three operations:

- *Send* computes a message vector $\boldsymbol{m}_{u,v}$ for each edge.
- *Aggregate* reduces incoming messages for each vertex to a vector $\boldsymbol{a}_v$. The neighborhood function $N(v)$ determines which messages are considered, typically using a fixed size random sample of vertices.
- *Update* combines each vertex's prior state with the result of aggregation to produce a new vector $\boldsymbol{z}_v$.

By iteratively applying K of these layers, the final state for each vertex captures information about the structure of its K-hop neighborhood.



(a) Input graph. (b) Nodeflow. (c) Inference dataflow.

Fig. 1. An example of inference on vertex B using a two-layer GCN. The nodeflow (b) describes the propagation of features (c) within a message-passing layer (MPL).

Algorithm 1. Message Passing Layer Forward Pass

Input: Graph $G = (V, E)$; Vertex and edge features $\boldsymbol{h}_v$, $\boldsymbol{h}_{(u,v)}$
Output: Updated vertex features $\boldsymbol{z}_v$
for (u, v) **in** E **do**
  $\boldsymbol{m}_{u,v} \leftarrow \mathrm{Send}(\boldsymbol{h}_v, \boldsymbol{h}_u, \boldsymbol{h}_{(u,v)})$
for v **in** V **do**
  $\boldsymbol{a}_v \leftarrow \mathrm{Aggregate}(\{\boldsymbol{m}_{u,v} \mid u \in N(v)\})$
  $\boldsymbol{z}_v \leftarrow \mathrm{Update}(\boldsymbol{h}_v, \boldsymbol{a}_v)$

Other Layers. In this paper, we treat other GNN layers as slightly modified versions of message passing. For example, *Pooling* and *Readout* layers are treated as a special case of message-passing where nodes are connected to dummy nodes representing a node cluster [16].

Nodeflow. A nodeflow [13] is a bipartite data structure that describes how features propagate during message-passing. It is usually generated in a preprocessing step before inference, but is also cheap to create on-demand (e.g., for dynamic graphs). The nodeflow speeds determining which edges and vertices are needed to update a specific vertex, making it particularly useful when performing inference on a subset of vertices in a graph. It also allows sampling operations to be computed before inference by simply sampling the nodeflow rather than the original graph. Fig. 1 shows an example of using the nodeflow to compute inference.

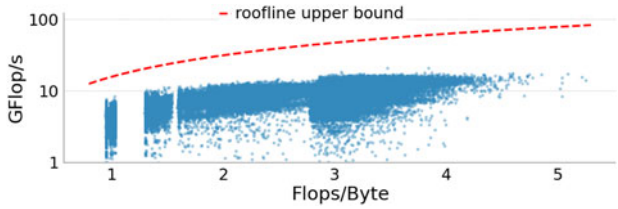


Fig. 2. CPU performance of GCN inference for vertices in the Pokec [20] dataset. Bottlenecks in cache bandwidth result in a significant gap between measured performance and the roofline upper bound.

GCN. We use the Graph Convolutional Network [17] (GCN) as a concrete example of a GNN. GCN uses multiple message-passing layers with the following send, aggregate, and update operations

$$\begin{aligned} \boldsymbol{m}_{u,v} &\leftarrow \boldsymbol{h}_u \\ \boldsymbol{a}_v &\leftarrow \text{mean}(\{\boldsymbol{m}_{u,v} \,|\, u \in N(v)\}) \\ \boldsymbol{z}_v &\leftarrow \text{ReLU}(W\boldsymbol{a}_v) \end{aligned}$$

where W is a trainable weight matrix. We can rewrite this to use sparse-dense matrix multiplication (SpMM)

$$Z \leftarrow \text{ReLU}(\hat{A}HW) \tag{1}$$

where $\hat{A}$ is a sparse matrix derived from the nodeflow and H and Z are dense matrices formed from the set of input and output features respectively. This allows GCN inference to be implemented using operations from highly optimized sparse matrix libraries, such as Intel MKL [18] or cuSPARSE [19].

2.2 Performance Challenges of GNNs

To demonstrate the performance challenges of GNNs in practice, we implement 2-layer GCN using the SpMM form in Eq. (1). Our implementation uses Tensorflow compiled with Intel MKL run on a single socket of an Intel Xeon E5-2690v4. Fig. 2 plots measured latency versus arithmetic intensity for each vertex in the Pokec dataset. Arithmetic intensity depends on the number of unique neighbors that must be read during inference, which is determined by the local graph structure of each vertex.

Inference latency in this dataset is theoretically bottlenecked by off-chip memory bandwidth for all vertices. However, there is a significant gap between the theoretical upper bound and the actual measured latency at higher levels of arithmetic intensity. The primary bottleneck is last level cache bandwidth, a result consistent with prior analysis of GPU performance [21]. Here, the highest arithmetic intensities occur when a vertex appears in multiple neighborhoods and its feature vector can be reused. However, this also results in higher utilization of cache bandwidth since multiple cores read and write the vertex in parallel.

Opportunities for Acceleration. GNN inference has high latency in existing architectures because a GNN exhibits a wide variety of memory access patterns and operations. We propose that by carefully decomposing GNNs into sub-operations, an accelerator can optimize each one separately and the overall architecture can stitch the units into a complete execution pipeline. To specialize for vertex-centric operations, we propose using a dedicated high performance matrix-multiplication unit. Weights are stored on-chip in dedicated memory with a level of caching to improve reuse. For edge-centric operations, we propose prefetching data for multiple edges in parallel and specialize the on-chip feature memory to enable fast gather and reduction operations. Finally, since the nodeflow is known statically, we propose improving off-chip access efficiency by scheduling bulk transfers of feature data rather than loading on demand during execution.

3 Related Work

DNN Accelerators. A significant number of custom neural network accelerators have been developed, mostly focused on dense operations [7], [22], [23]. However, edge-centric operations are difficult to implement efficiently on these architectures [8], which are much more computationally irregular than traditional DNNs. GRIP natively supports edge-centric operations by using a graph-processing based programming model (Section 4) and by a combination of specialized memory for edge accesses and software techniques. In Section 8.6 we estimate GRIP to be $2.4\times$ faster than a comparable TPU-like accelerator modified to improve GNN inference latency.

GCN Accelerators. A number of accelerators have been recently been developed for GNNs [11]. For example, HyGCN [10] and GraphACT [24] are two accelerators designed for graph convolutional networks, a subclass of GNNs. Like GRIP, these accelerators use separate edge- and vertex-centric units for GNN computation. GRIP builds on these designs by handling a much more general set of GNNs that includes models that use computation associated with edges. This is important for many emerging state-of-the-art GNNs, such as Graph Attention Networks [25]. Additionally, GRIP's support for vertex-tiling reduces the amount of on-chip bandwidth required for loading weights during vertex-oriented operations. In Section 8.6 we estimate this improves latency by $4.5\times$ compared to HyGCN.

Graph Analytics Accelerators. Specialized accelerators have also been proposed for graph analytics [26], [27], [28]. However, these workloads require relatively little computation per-vertex and typically use scalars rather than large feature vectors. Thus, the computation and memory access patterns are very different. In Section 8.6, we estimate GRIP to be $8.1\times$ faster than the approach of Graphicionado [9].

4 GRIP's Programming Model: GReTA

GRIP's programming model is based on GReTA [14], a graph-processing abstraction specialized for implementing GNNs. GReTA decomposes computation within a GNN layer into four stateless user-defined functions (UDFs): `gather`, `reduce`, `transform`, and `activate`. GRIP invokes each UDF in a series of three execution phases: *aggregate*, *combine*, and *update*. GRIP programs can also be composed by using the result of one program as the features or accumulator in another. This flexibility allows implementing a wide range of GNN models. Algorithm 2 shows the semantics of a GRIP program, with the phases, UDFs, and access patterns summarized in Table 1.

TABLE 1
Summary of GReTA's Phases, UDFs, and Access Patterns

Phase	UDFs	Data In	Data Out	Access Pattern Acc.	Feat.	Wgts.
Aggregate	`gather`, `reduce`	$\boldsymbol{h}_u$, $\boldsymbol{h}_v$, $\boldsymbol{h}_{(u,v)}$	$\boldsymbol{e}_v$	Seq.	Rnd.	–
Combine	`transform`	$\boldsymbol{e}_v$, W	$\boldsymbol{a}_v$	Seq.	–	Seq.
Update	`activate`	$\boldsymbol{a}_v$	$\boldsymbol{z}_v$	Seq.	–	–

Data Model. GRIP programs operate on four types of data: (1) A nodeflow $NF = (U, V, E)$ defines how input vertices U and output vertices V are related using the set of directed edges E. (2) Feature vectors $\boldsymbol{h}_u$, $\boldsymbol{h}_v$, and $\boldsymbol{h}_{(u,v)}$ associated with nodeflow input vertices, output vertices, and edges respectively. (3) A set of constant layer weights W. (4) Edge- and vertex-accumulators ($\boldsymbol{e}_v$ and $\boldsymbol{a}_v$) associated with each output vertex.

Algorithm 2. GRIP Program Execution Semantics

Input: Layer nodeflow (U, V, E); Vertex data $\boldsymbol{h}_u$ and $\boldsymbol{h}_v$; Edge data $\boldsymbol{h}_{(u,v)}$; Accumulators $\boldsymbol{e}_v$ and $\boldsymbol{a}_v$; Weights and biases W
Output: Updated vertex data z_v

```
/* Aggregate Phase */
for (u, v) in E do
    e_v = reduce(e_v, gather(h_u, h_v, h_(u,v)))
/* Combine Phase */
for v in V do
    a_v = transform(a_v, e_v, W)
/* Update Phase */
for v in V do
    z_v = activate(a_v)
```

Execution. A GRIP program executes in three phases. First, the *Aggregate* phase iterates over the nodeflow edges and invokes `gather` and `reduce`. `Gather` reads features associated with the current edge and adjacent vertices and produces a message value. `Reduce` accumulates message values associated with the same output vertex into $\boldsymbol{e}_v$, resulting in a single accumulated value per output vertex.

Second, the *Combine* phase iterates over each output vertex and combines $\boldsymbol{e}_v$ with the previous accumulator state $\boldsymbol{a}_v$ using `transform`. `Transform` is restricted to being an affine function (e.g., matrix multiplication) and is the only UDF with access to layer weights.

Finally, the *Update* phase again iterates over each output vertex and applies `activate` to produce $\boldsymbol{a}_v$. The main difference between the *Update* and *Combine* phases is that `activate` is not restricted to being affine and cannot access weights. It is typically used to implement the non-linear operations required in a layer (e.g., an activation function). This produces a final accumulated value for each vertex z'_v.

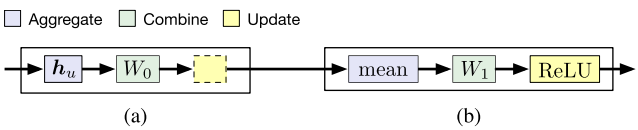


Fig. 3. Modifying the GCN *Send* operation Eq. (2) requires splitting the layer into two programs (a) and (b). The dashed box in (a) indicates a phase with no computation.

4.1 Edge Computation

GReTA is expressive enough to implement a wide variety of GNNs [14]. However, to keep vertex units and weight memories simple and fast (Section 5), only *Combine* can access layer weights. Some models rely on computation associated with edges; these models must be decomposed into multiple GReTA programs. For example, consider a modified version of the GCN *Send* operation

$$\boldsymbol{m}_{u,v} \leftarrow W_0 \boldsymbol{h}_u. \tag{2}$$

Since *aggregate* cannot read W_0, this layer must be implemented by splitting it into two programs, shown in Fig. 3. Note that splitting the layer results in each program iterating over a different nodeflow. In this case, the program in Fig. 3a iterates over an identity nodeflow where all vertices are self-connected. Fig. 4 demonstrates the flexibility of this approach by showing the implementation of a variety of different GNN models.

5 The GRIP Architecture

GRIP is an accelerator architecture for low-latency GNN inference. In this section, we describe an overview of GRIP and the design of the three core execution units: 1) the edge unit, which operates on nodeflow edges 2) the vertex unit, which operates on nodeflow vertices, and 3) the update unit, which performs non-linear operations such as an activation function. A high level overview of GRIP is shown in Fig. 5.

5.1 Overview

Control. A host system controls GRIP by sending commands to execute different operations or transfer data. The control

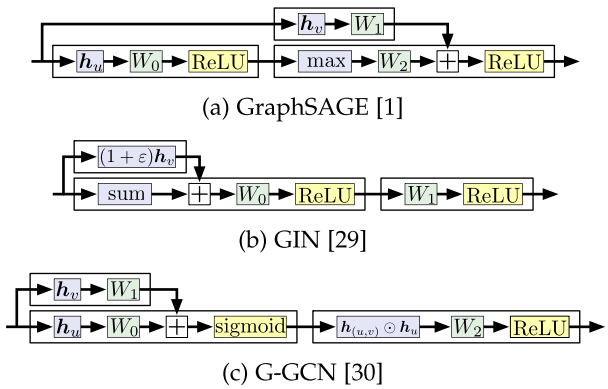


Fig. 4. GRIP implementation of several GNN models. Plus-boxes indicate the output of one program is used as the edge or vertex-accumulator of another. Phases with no associated computation are omitted.

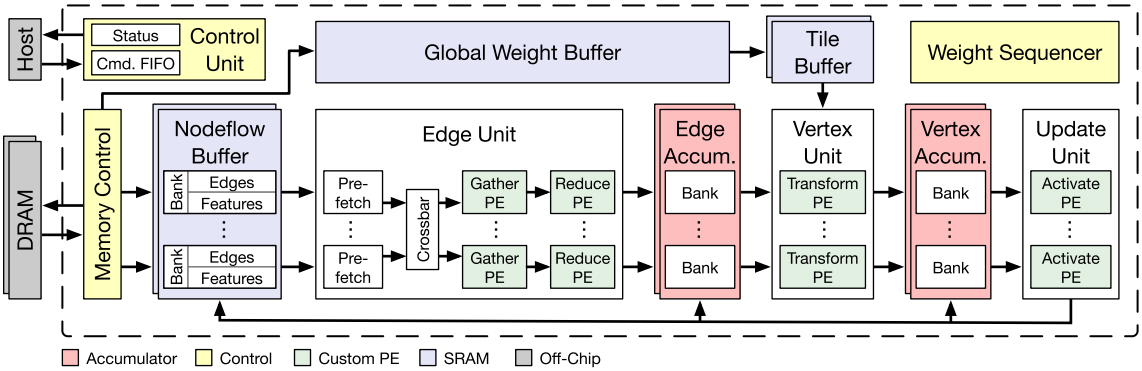


Fig. 5. High-level overview of GRIP.

unit dequeues each command in-order and issues them asynchronously to the GReTA execution pipeline or the data-supply unit. A barrier command enforces dependencies by preventing new commands from being issued until all previous commands have completed. Each command updates a global status register on completion, which can be queried by the host to monitor execution.

GReTA Execution Pipeline. The core compute elements in GRIP are contained within the GReTA execution pipeline. The pipeline is composed of three units: the edge unit, the vertex unit, and the update unit. The edge unit performs *aggregate* by iterating over the edges of the nodeflow, which is stored in the nodeflow buffer. For each edge, it reads the associated features, executes `gather`, and accumulates the result into the edge accumulator using `reduce`.

The vertex unit performs the *combine* phase. It iterates over the output vertices corresponding to the accumulated edge values, executes `transform`, and then accumulating the result into the vertex accumulator. The vertex unit also reads weight values from the tile buffer, which caches tiles of weight values from the global weight buffer. The weight sequencer synchronizes the tile buffer and the vertex unit to iterate over the tiles as described in Section 6.2.

Finally, the update unit performs the *update* phase by reading the accumulated values for each vertex and passing the values to the activate PE. The result is written to the nodeflow buffer as an updated feature, or to the edge or vertex accumulator. This allows efficiently passing values between different GRIP programs when they are executed in sequence.

PE Implementation. The GreTA execution pipeline has four processing elements (PEs) that correspond to the UDFs introduced in Section 4. Each PE can use a customized hardware design to accommodate different implementation requirements. For example, each PE could be implemented using a reconfigurable fabric (e.g., an FPGA) for maximum flexibility. Alternatively, they could be implemented using a model specific circuit to optimize for area or performance.

Our implementation uses a programmable ALU based approach. Since most common GNNs only require a small number of operations in practice, this allows us to support a range of models on the same hardware while remaining reasonably efficient in practice. Specifically, we allow `gather` to be identity (e.g., $\boldsymbol{h}_u$ or $\boldsymbol{h}_v$), element-wise sum, product, or scale by constant; `reduce` to be element-wise sum, max, or mean; `transform` to be matrix multiplication followed by element-wise sum; and `activate` to be either ReLU or a LUT operation which we describe in Section 5.4. While these cover most GNN models we investigated, expanding the set of supported operations may be required for other GNNs. We leave exploring other possible implementations for future work.

Data Supply Unit. The data supply unit moves data associated with nodeflow partitions on- and off-chip. Since all off-chip memory accesses are simple to determine statically using the nodeflow, data-transfers are scheduled by the host before inference begins, rather than being requested on-demand by the execution units. This approach eliminates the need for a complex caching system, and prevents units from stalling on external memory access since their required data is guaranteed to be on-chip. It also hides memory access latency by allowing loading data for partitions access to overlap with execution of the different GReTA phases (Section 6.1).

5.2 Edge Unit

The edge unit pipeline is split into two distinct halves (Fig. 6). Stages P0-P2 implement *prefetch*, which iterates over the edges of the nodeflow and reads the features corresponding to the source vertex. The result is passed to *reduce* (stages R0-R4), which reads the corresponding destination feature and then applies `gather` and `reduce`, accumulating the result into the edge accumulator. GRIP allows optionally disabling stage R0 since most models do not require reading source features.

Parallelization. The performance of the edge unit can be improved by parallelizing computation associated with each edge. A simple method to parallelize execution is to duplicate the elements of the edge unit into N identical copies. Each copy can then be assigned a subset of output vertices to process in parallel (e.g., by a random hash of the vertex ID). However, since the nodeflow buffer is read every

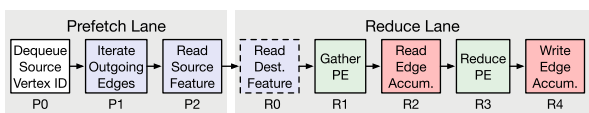


Fig. 6. The edge unit pipeline is split into source-oriented (P0-P2) and destination-oriented (R0-R4) sections called lanes that can be independently replicated. Stage R0 is only used for models that require reading source features.

cycle by each lane, this approach requires adding $2N$ ports to the nodeflow buffer.

Instead, GRIP duplicates prefetch and reduce into N and M copies called lanes. Each lane is statically assigned a partition of input vertices (for prefetch) or output vertices (for reduce). Similarly, edges are assigned to a prefetch lane based on the edge's source vertex. During execution, each prefetch lane iterates over its assigned edges and reads each edge's corresponding feature. It then sends the feature data through an $N \times M$ crossbar to the reduce lane assigned to the destination vertex. This design restricts each lane to accessing only its assigned subset of features and edges, allowing GRIP to partition the nodeflow buffer into $N + M$ separate SRAMs. As a result, GRIP scales to a much larger number of lanes than the simpler design. Additionally, our implementation of GRIP extends this scheme to include off-chip memory by storing feature data pre-partitioned and setting the number of prefetch lanes equal to the number of DRAM channels.

5.3 Vertex Unit

The vertex unit implements the *combine* phase by iterating over the output vertices and applying `transform`. Our implementation restricts `transform` to a matrix multiplication, which we implement using a 16×32 weight stationary PE array [23]. Each PE contains a 16-bit multiplier, as well as a local double buffered weight register. The PE array is broken into two 16×16 blocks. Blocks can be configured to use one of two modes: cooperative, where both blocks operate on the same vertex, or parallel, where blocks operate on different vertices in parallel. Parallel mode broadcasts weight values to both blocks, allowing for slightly lower energy consumption at the expense of higher latency when there is only a single output vertex.

To implement matrix multiplication, GRIP broadcasts inputs across rows and accumulates results down columns using a reduction tree. The entire operation is pipelined to allow multiple matrix operations to occur without stalling, even as weights are transferred in and out of the array.

5.4 Update Unit

The update unit applies `activate` to each vector in the vertex accumulator. Our implemenation allows two possible operations: element-wise ReLU and a two-level, linearly interpolated lookup-table (LUT) similar to the implementation in NVDLA [29]. The LUT supports approximating most common activation functions, including sigmoid, which is required for models such as G-GCN.

6 Optimizations

GRIP implements two major GNN optimizations: execution partitioning and vertex-tiling. Execution partitioning splits a GRIP program to operate on partitions of a nodeflow, reducing the needed on-chip memory. GRIP supports pipelining operations on different partitions, improving performance. Vertex-tiling improves the locality of weights in the *combine* phase, reducing the memory bandwidth needed by the vertex unit. These optimizations reduce inference latency for GRIP by $2.5\times$ and $8.0\times$ respectively (Section 8.5).

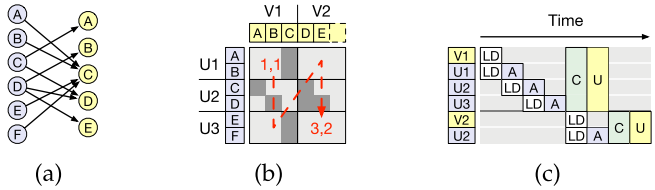


Fig. 7. An example of a nodeflow (a) and corresponding partitions (b) which are processed column-wise. GRIP also pipelines loading data (LD) with Aggregate (A), Combine (C), and Update (U) (c).

6.1 Execution Partitioning

A common GNN optimization is to split the graph into partitions that can be computed on separately [5], [10], [14]. This reduces the peak amount of on-chip memory required to compute inference since only a portion of the graph must be loaded at once. However, performing inference on a single vertex within a partitioned graph can be slow since in the worst case it requires loading a new partition for each neighboring vertex. GRIP instead uses a similar optimization performed on the nodeflow, which we refer to as execution partitioning (Fig. 7). First, the nodeflow is partitioned offline by splitting the input and output vertices into fixed chunks of size N and M. Likewise, the edges are partitioned into blocks of size $N \times M$, where block $NF_{i,j}$ stores the edges connecting input vertices in chunk U_i to output vertices in chunk V_j. During inference, GRIP executes *aggregate* for each partition in a column, skipping blocks that are empty. Then, GRIP executes the *combine* and *update* phases once, updating values in the corresponding partition of output vertices. This ensures every incoming edge for each output vertex is processed before *combine*.

GRIP also implements two kinds of pipelining related to partitioning. First, GRIP pipelines loading data from off-chip with the *aggregate* phase. This allows overlapping execution with bulk loading feature data for an entire partition. If enough space is available in the nodeflow buffer, GRIP also optionally caches partition feature data loaded during the processing of the first column to avoid reloading data while processing later columns. Second, transferring weights from the global buffer can be pipelined with processing an entire column. GRIP performs inter-layer pipelining by loading the weights of the next layer while processing the last column, and preloads the tile buffer before processing the first column.

6.2 Vertex-Tiling

The bandwidth required to load layer weights can be a significant bottleneck. For example, consider a GCN layer with a feature size of 256. Since our implementation of `transform` cannot hold the entire $1\,\mathrm{MB}$ weight matrix locally, new weight values must be loaded every cycle. At an operating frequency of $1\,\mathrm{GHz}$, this requires a maximum of $2\,\mathrm{TB/s}$ of tile buffer bandwidth, which we found difficult to implement physically. While this could be resolved by increasing the number of weights stored within the multiplier array, this increases energy usage and lacks flexibility since a model with a larger feature size would still run into the same limitation.

GRIP addresses this bottleneck by reordering the loop over the feature vectors. We call this optimization "vertex-

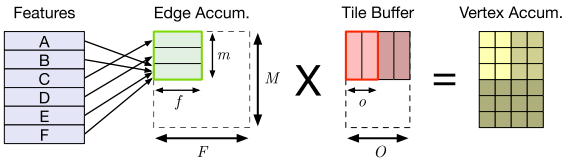


Fig. 8. Vertex-tiling allows materializing a small tile of edge accumulator values ($m \times f$) instead of the full $M \times F$ matrix. This reduces the memory bandwidth required since a tile of weight values can be reused across m vertices.

tiling" to emphasize that portions of vertex feature vectors are accumulated and transformed together instead of full feature vectors as in HyGCN [10]. This is similar to the reordering optimizations of GCNAX [30], but applied to GReTA programs with an affine `transform` UDF instead of specifically to GCNs.

Fig. 8 shows a graphical representation of this strategy. Here, *aggregate* produces f elements for m output vertices. This requires storing $f \times m$ elements in the edge accumulator instead of the full $F \times M$ matrix. Then, we run *combine* (in this case matrix multiplication), which loads each $f \times o$ tile from the tile buffer. We then repeat this process, first for all vertex tiles and then for all weight slices, maximizing the locality of the weights. This reduces the bandwidth between the tile buffer and the matrix unit by a factor of $1/m$. Tuning f and m can trade-off the required bandwidth with the amount of storage required for tiles and *aggregate* values.

7 Experimental Methodology

Datasets. Table 2 shows the datasets chosen for evaluation, selected from previous evaluations of GNNs [1], the SNAP project [20], and the UF sparse matrix collection [31]. Included datasets were designed to be similar to the workloads used by GNNs and have a range of connectivity. We prepossessed each dataset using the same procedure outlined by the authors of GraphSAGE [1]. The column "2-Hop" denotes the median number of unique vertices within the 2-hop neighborhood of a vertex selected uniformly at random, taking into account the sampling procedure.

Models. We implemented four GNNs which cover a broad range of model types: GCN [17], the max variant of GraphSage [1], GIN [32], and G-GCN [33]. For our neighborhood function, we use the same sampling proceedure as described by the authors of GraphSage. Specifically, we deterministically map a given vertex to a fixed-sized, uniform sample of its neighbors. For all models, we use two layers with sample sizes 25 and 10 for the first and second layer, respectively. Samples between layers are independent. Additionally, we use a feature size of 602 (the feature size of the Reddit dataset), a hidden dimension of 512, and an output dimension of 256 for all layers.

TABLE 2
Datasets Used for Evaluation

Dataset	Nodes	Edges	2-Hop
Youtube (YT)	1,134,890	2,987,624	25
Livejournal (LJ)	3,997,962	34,681,189	65
Pokec (PO)	1,632,803	30,622,564	167
Reddit (RD)	232,383	47,396,905	239

TABLE 3
Architectural Characteristics of Baseline and GRIP

	CPU	GRIP
Compute	1.164 TOP/s @ 2.6 GHz	1.088 TOP/s @ 1.0 GHz
On-chip memory	L1D: 14 × 32 KiB L2: 14 × 256 KiB LLC: 35 MiB	Nodeflow: 4 × 20 KiB Tile: 2 × 64 KiB Weight: 2 MiB
Off-chip memory	4× DDR4-2400 76.8 GiB/s	4× DDR4-2400 76.8 GiB/s
Total Area	306.18 mm^2	11.27 mm^2
Power	135 W	4.9 W

Baseline. Our CPU baseline was a dual socket server containing two, 14-core 2.60 GHz Intel Xeon E5-2690 v4 CPUs, each with four channels of DDR4-2400 memory. We restricted our experiments to a single socket to adhere to Tensorflow performance guidelines [34] and to avoid NUMA latency variation. In this configuration, we measured a sustained 1.084 TFlop/s in a matrix multiply benchmark (93% of 1.164 TFlop/s theoretical peak) and 64.5 GiB/s of off-chip memory bandwidth (84% of 76.8 GiB/s theoretical peak).

We implemented both the baseline and our optimized inference algorithm in Tensorflow v2.0 [35] with eager mode disabled and compiled with the Intel Math Kernel Library [18]. To discount the overhead of the Tensorflow library for each model, we measured the time to evaluate an equivalent model with all tensor dimensions set to zero and subtract it from the latency measurement. We also perform a warm-up inference before all measurements to allow Tensorflow to compile and optimize the network.

ASIC Synthesis: We implemented GRIP in SystemVerilog, choosing the architectural parameters to have similar compute and memory bandwidth as our CPU baseline (Table 3). The implementation uses 16-bit fixed point, which maintains suitable inference accuracy in the models we evaluate. We then performed synthesis and place and route in a 28 nm CMOS process, targeting a 1 GHz operating frequency and worst case PVT corner. The critical path of GRIP was determined to be 0.93 ns, inside the weight SRAMs. Power estimates of each unit was performed by generating activity factors from a cycle accurate simulation of our implementation and applying them to our synthesized design. We used Cacti v6.5 [36] to estimate the area and power of the SRAM memories. We also integrated Ramulator [37] into our simulator to estimate DRAM timings and produce a command trace. These traces were fed to DRAMPower [38] to estimate DRAM power. Note that area and power estimates of the DRAM PHY and IOs are not included in the totals presented in table 3 for consistency with prior work [9], [10].

Latency Metric. In most experiments where we measure latency as the end-to-end execution time required to perform inference for a single vertex with the maximum neighborhood size within each dataset after sampling. There are two exceptions. In Section 8.2 we report 99th-percentile

TABLE 4
99%-Ile Inference Latency for GRIP, CPU, and GPU

Model	Dataset	GRIP	CPU		GPU	
			s	×	s	×
GCN	youtube	15.4	309.2	(20.1)	1082.4	(70.5)
	livejournal	15.8	466.8	(29.5)	1313.6	(83.1)
	pokec	16.0	477.1	(29.8)	1085.6	(67.7)
	reddit	16.3	407.1	(25.0)	813.2	(50.0)
G-GCN	youtube	134.1	2315.9	(17.3)	1332.5	(9.9)
	livejournal	146.3	2493.2	(17.0)	1837.6	(12.6)
	pokec	146.7	2637.9	(18.0)	1409.2	(9.6)
	reddit	147.0	2864.2	(19.5)	1133.9	(7.7)
GS	youtube	113.7	1545.1	(13.6)	1309.0	(11.5)
	livejournal	124.4	1947.4	(15.7)	2193.8	(17.6)
	pokec	124.9	2075.7	(16.6)	1759.1	(14.1)
	reddit	125.3	2099.0	(16.8)	1252.8	(10.0)
GIN	youtube	30.5	344.7	(11.3)	1387.6	(45.5)
	livejournal	30.9	416.1	(13.5)	1221.5	(39.5)
	pokec	31.1	340.7	(10.9)	855.5	(27.5)
	reddit	31.4	354.8	(11.3)	1009.4	(32.2)

latency measured across all vertices in each dataset. In Section 8.4 we report latency as the neighborhood size changes.

8 Evaluation

We evaluate GRIP's ability to perform low latency GNN inference on a wide range of models by measuring overall inference latency for four different models and compare to a CPU and GPU baseline (Section 8.1). To better understand GRIP's performance, we then breakdown the contribution of each architectural feature (Section 8.2) and how the overall speedup changes as we modify both architectural (Section 8.3) and model parameters (Section 8.4). We also measure the impact of each GNN optimization we implemented (Section 8.5). Finally, we compare GRIP to alternative approaches (Section 8.6).

8.1 Overall Performance

To evaluate GRIP's overall performance, we measured the total end-to-end execution time (latency) for inference using a variety of models and datasets. Table 4 shows GRIP's inference latency and speedup versus our CPU and GPU implementation. We use 99th percentile latency for consistency with prior evaluations of inference latency [39].

Latency versus CPU. Compared to our CPU implementation, GRIP achieves a latency improvement of between $29.8\times$ (GCN, Pokec) and $10.9\times$ (GIN, Pokec) with a geometric mean of $17.0\times$ across all datasets and models. GRIP tends to give a smaller speedup on models that perform a larger portion of their computation during the *Update* step of the message-passing layer. For example, GIN's *Update* uses a two-layer MLP that requires roughly double the computation of GCN's single matrix multiplication. However, the additional computation results in similar overall CPU inference latency since our implementation is largely bottlenecked by non-computational factors (Section 2.2). This results in GRIP achieving a smaller performance improvement of 10.9-13.5× compared to an improvement of more than $13.6\times$ for all other models.

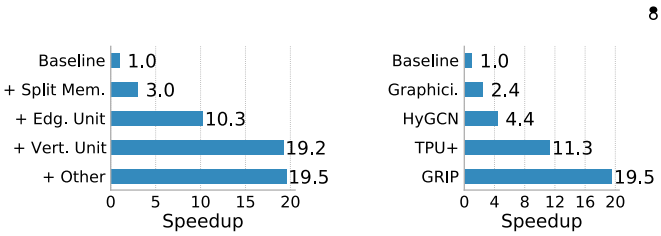
Latency versus GPU. Practical deployments of online GNN inference most often use CPUs due to the large memory requirements for graph features and low utilization at small batch sizes. However, for completeness we also benchmark GRIP against an Nvidia P100 GPU implementation for each model. GRIP's speedup on GPU ranges from $83.1\times$ (Livejournal, GCN) to $7.7\times$ (Reddit, G-GCN) with a geometric mean of $23.4\times$. Models with relatively low overall latency (GCN, GIN) see a large speedup, ranging from 27.5-83.1×. This is largely due to the overhead of transferring embeddings from host to GPU memory (~200-500 s, depending on neighborhood size). GRIP does not incur this penalty since features and weights are already stored in device DRAM and do not have to be transferred from the host. Note that this implies GRIP's speedup could be reduced by ~2-4× compared to a platform that does not separate host and GPU memory (e.g., some SoC based GPUs). On models with higher total execution time (e.g., G-GCN), GRIP still achieves a significant speedup due to low GPU utilization. With a batch size of 1, there is not sufficient computation during each layer to fully utilize the computational resources of the GPU and overhead of launching each kernel tends to dominate.

8.2 Breakdown of Latency

This subsection breaks down the latency impact of each architectural feature of GRIP. Specifically, we modify our cycle-accurate simulator to match the bottlenecks exhibited by our CPU implementation and then progressively remove each modification to measure the impact of different units. We use the geometric mean speedup of GCN for the largest neighborhood in each dataset as a performance benchmark.

Baseline Configuration. Our baseline configuration emulates each core being assigned independent vertices and performing all GReTA phases, with weights and partition data being first loaded into L3 cache and intermediate values accumulated directly in L2. This results in the following simulator modifications. First, we modify our vertex unit to use 14, 8×2 matrix multiply units, with each unit assigned independent vertices within a partition. This emulates the effect of each CPU core using two 8-element SIMD units. Second, we increase the number of fetch and gather units to 14 and the crossbar width to 32 bytes, matching the number of cores and L2 cache bandwidth, respectively. We also disable pipelining between the edge and vertex units to emulate a single core performing both functions. Third, we merge the weight and nodeflow buffers into a single SRAM and limit the maximum read bandwidth to 16 bytes per cycle per fetch unit, matching the bandwidth of the L3 cache. Finally, we disable pipelining between the vertex and update unit to model both operations being performed by the same core. This configuration overestimates the performance of the CPU since it models ideal performance and no additional computation required for auxiliary operations, such as indexing calculations. In particular, with a $2.6\,\text{GHz}$ clock and an element width of 4-bytes, our model is $2.07\times$ faster than the measured CPU latency.

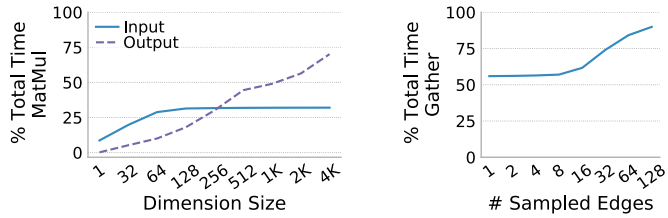
Breakdown. Fig. 9a shows the impact of different units in GRIP by progressively removing each modification from our baseline in reverse order. First, we split the weight and



(a) Speedup breakdown for each component of GRIP versus baseline.

(b) Estimated speedups of prior work versus baseline and GRIP.

Fig. 9. Breakdown of performance improvements.



(a) Impact of feature dimensions.

(b) Impact of sampling.

Fig. 11. The impact of scaling different GCN parameters on the balance of time spent in each operation. Scaling the output feature size increases the amount of time spent performing matrix multiplication, while increasing the number of edges decreases it.

nodeflow memories into separate SRAMs. This results in a 2.8× speedup due to removing contention between fetching features and weights from the same SRAM (2.0×), as well doubling the bandwidth available to load weight values into the vertex unit (1.4×). Second, we add the edge unit, resulting in an additional improvement of $3.4\times$. While this is partially due to increased crossbar bandwidth after adjusting the number of fetch and gather units (1.14×), the majority of the speedup is due to allowing loading data, *aggregate*, and *combine* phases to overlap by using a dedicated unit for each phase (2.97×). Third, we enable the vertex unit and revert to using a single 16×32 matrix multiply unit, resulting in an additional 1.87× speedup. This is due to increased overall TOP/s (1.63×) and using a single unit rather than multiple units, which allows units to not be wasted when the overall number of output vertices is small (1.15×). Finally, separating and pipelining the update unit produces a small speedup of $1.02\times$.

8.3 Architectural Parameters

Here, we discuss the impact of several high level architectural parameters on inference latency.

Number of DRAM Channels. The number of DRAM channels determines the overall memory bandwidth available to transfer data on- and off-chip. In Fig. 10a, we observe that GRIP's performance is strongly related to the number of channels until around 8 channels (∼150 GiB/s). This indicates that GRIP's performance is primarily limited by off-chip memory bandwidth.

Weight Bandwidth. The weight bandwidth determines how many values can be read from the global weight buffer each cycle. If this is set too low, loading weight values can become a bottleneck during *combine*. We observe this effect in Fig. 10b below 128 GiB/s, which corresponds to loading 64 weight values each cycle.

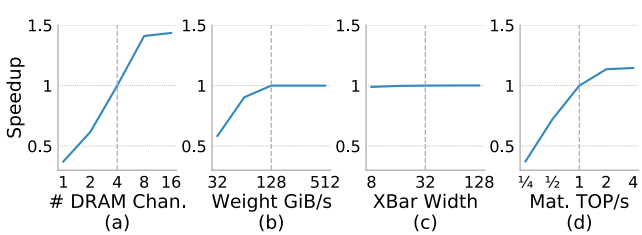


Fig. 10. Impact of scaling architectural parameters. Dashed vertical line indicates our implementation's parameters. In Fig. 10a the number of edge unit lanes is kept equal to the number of channels.

Crossbar Port Width. The crossbar port width determines the number of elements accumulated by each gather unit in a single cycle. In our experiments, the average number of edges per vertex is fairly small (sampled to be less than 25). Since *aggregate* typically takes much less time than *combine* or loading data from DRAM, increasing the width has a limited impact on performance (Fig. 10c).

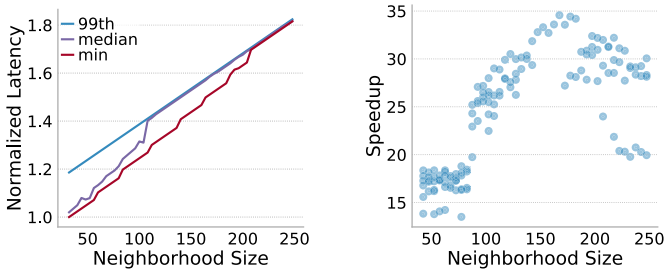
Matrix Multiply TOP/s. The total number of TOP/s GRIP can achieve is determined primarily by the size of the matrix multiply unit. In Fig. 10d we see that performance is strongly related to the size of this unit, until reaching around 2 TOP/s at which point GRIP is limited by memory bandwidth. Thus, our implementation of GRIP would see a relatively small benefit from a substantially larger matrix unit (1.14× for a 4× larger unit).

8.4 Model Parameters

A key aspect GRIP's design is balancing the performance between GReTA's edge and vertex-centric phases. Here, we evaluate how this balance changes as the parameters of the GNN model change.

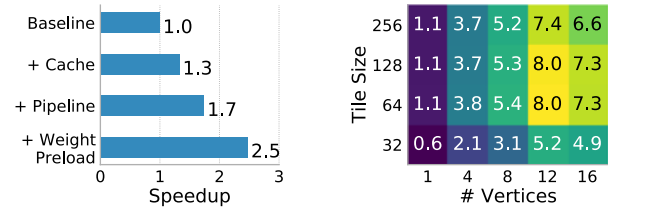
Feature Dimensions. In Fig. 11a, we evaluate how varying the number of the input and output features impacts the percent of time spent in matrix multiplication. The proportion is initially low (∼8%) for small features and increases linearly until 32 features. This is due to the fact that when the feature size is smaller than the native width of the DRAM interface, DRAM bandwidth is poorly utilized due to many random accesses. In our implementation, we use two dual-channel DDR4 controllers, which each have an interface of 64 2-byte elements. Above this point, the proportion of time spent performing vertex-accumulation stays flat, reflecting the fact that each additional feature results in a constant amount of additional computation during inference. However, this analysis does not hold for the output features, which can be increased without needing to increase the number of values loaded from DRAM. We see that increasing the output feature size always increases the percent of time performing *combine*. Thus, models with large output feature sizes are likely to be limited by compute rather than memory.

Sampled Edges. Another important model parameter is the number of sampled edges per output vertex. In Fig. 11b, we evaluate how the number of edges impacts the percent of total time spent performing *aggregate*. For less than 8 edges per vertex, GRIP's latency is mostly limited by computation and overhead related to accessing data from DRAM. Above



(a) (b)

Fig. 12. Impact of different neighborhood sizes on latency for the GCN model. GRIP's latency linearly increases with neighborhood size due to more computation being required for inference. The speedup is roughly constant until a neighborhood size of about 95, at which point intermediate values no longer fit into the cache of a single CPU core.



(a) Impact of partitioning and pipelining (b) Speedup of vertex-tiling

Fig. 13. Impact of partitioning and tiling optimizations.

this threshold, the memory and crossbar bandwidth becomes a bottleneck, and GRIP spends an increasing portion of execution loading data.

Neighborhood Size. The neighborhood size of a vertex heavily impacts GRIP's inference latency and is influenced by local graph structure. To demonstrate, we plot GRIP's minimum, median, and 99th percentile inference latency for GCN across different neighborhoods of the LiveJournal dataset in Fig. 12a. The result is a strong linear relationship between the neighborhood size and latency across the entire distribution. Each vertex added to the neighborhood results in a roughly constant increase in the amount of work during inference. Additionally, we observe that as the neighborhood size increases, the median latency moves closer to the 99th percentile. This is the result of larger neighborhoods being more likely to be densely connected, leading to a larger number of reductions that must be computed.

In Fig. 12, we evaluate the latency speedup compared to the CPU baseline across different neighborhood sizes. Below a neighborhood size of 95, we see a roughly constant speedup of between 12× and 18×. For these neighborhood sizes, all intermediate values fit into the L1 and L2 cache of a single CPU core. After this point, some feature values must be stored in the L3 cache and inference becomes limited by the cache bandwidth (Section 2.2).

8.5 Optimizations

In this subsection, we evaluate the impact of each optimization used by GRIP.

Partitioning and Pipelining. In Fig. 13a we show the cumulative speedups of each optimization enabled by partitioning. GRIP uses a nodeflow partition size of 12 by 4 nodes. We compare to an unoptimized baseline, where feature values are loaded from off-chip on demand and no pipelining exists between stages. First, by caching feature data on-chip, GRIP achieves a $1.3\times$ speedup. This is due to the decreased memory traffic required to reload data between partition columns and by the improved throughput from bulk loading data for an entire partition. Second, pipelining operations between different partitions gives an additional $1.3\times$ speedup from overlapping execution with memory transfers. Finally, we can also pipeline the transfer of weights from the global weight buffer into the update unit. This increases the overall speedup to a total of $2.5\times$.

Vertex-Tiling. In Fig. 13b, we show the speedup compared to no tiling as we alter the two tiling parameters M (the number of vertices in a tile) and F (the number of elements per vertex). We see that the speedup generally reaches a maximum around $F = 64$ elements. Above $F = 64$, increasing F causes the vertex unit to stall more often while waiting for a tile to be produced by the edge unit. The latency degradation is not linear because the time taken to accumulate a tile depends on the connectedness of the nodeflow. We also see degraded latency below $F = 64$. This is because F features are loaded from memory for each vertex. As F decreases, more random DRAM accesses are required to load features, degrading DRAM throughput. Increasing M also increases the speedup until around 12 vertices. The maximum number of output vertices in our model is 11. Increasing M beyond this only adds additional latency associated with processing empty dummy vertices.

8.6 Comparisons to Prior Work

Several other approaches have been proposed to accelerate neural networks and graph algorithms. Here, we analyze the bottlenecks present in each approach and compare latency with GRIP.

HyGCN. HyGCN [10] is a GCN accelerator optimized for high throughput inference (e.g., performing inference on all vertices in a large graph). The HyGCN authors propose a decomposition of GCN inference referred to as the edge and matrix-vector multiplication programming model (E/MVM). E/MVM forms the basis of HyGCN, with the architecture split into two units optimized for each phase. The combination phase is accelerated using a conventional systolic array, similar to the one found in the TPU [7]. The aggregation phase is accelerated using a more sophisticated unit (referred to as the *Aggregation Engine*) which features an edge sampler, sparsity eliminator, and scheduler that feeds features to multiple SIMD cores.

GRIP improves on HyGCN in three ways. First, GRIP can easily and efficiently handle per-edge computations that require access to layer weights (e.g., Eq. (2)). In the E/MVM model, all edge processing happens in the Aggregation engine, which does not have access to the weight buffer. GRIP achieves this with program splitting (Section 4.1).

Second, GRIP reduces latency by operating on nodeflow partitions, which directly encodes sampling, sparsity elimination, and scheduling as the result of a prepossessing step. HyGCN performs these operations dynamically on the entire graph in hardware during inference. The overhead of doing this dynamically is acceptable in HyGCN because it is designed for high-throughput operation across an entire

graph, but introduces an unacceptable latency for inference on a small number of vertices.

Third, vertex-tiling allows GRIP to materialize a small, fixed sized tile when it processes a vertex. This significantly reduces the amount of memory required for the edge-accumulator. HyGCN, in contrast, requires computing and storing full feature vectors before performing vertex-oriented operations, which use a $16\,\mathrm{MB}$ buffer. The HyGCN authors report that shrinking this buffer harms performance (1.3 – $4\times$ for a $16\times$ smaller buffer.) GRIP, in contrast, uses a $1.5\,\mathrm{KiB}$ edge-accumulator, four orders of magnitude smaller than the equivalent buffer in HyGCN.

We measure the latency impact of these differences by modifying our simulator to emulate the HyGCN approach. Specifically, we set the number of gather and fetch units to 1 and the crossbar width to 256 to match the number of SIMD lanes. We disable all tiling and force feature vectors to be fully accumulated before *combine*. We then set all other parameters to be the same as GRIP, including the same partitioning used in our evaluation of GRIP. This configuration results in a speedup of $4.4\times$ the baseline, shown in Fig. 9b[2]. However, it performs $4.5\times$ slower than GRIP due to limits in the available on-chip memory bandwidth for weights. Incorporating vertex tiling would allow for a much smaller edge accumulate buffer and reduce the required bandwidth by increasing the reuse of the weights.

TPU-GNN. The TPU [7] is a DNN accelerator designed around a large 2-D systolic array. GNNs are difficult to implement efficiently for the TPU due to a lack of support for edge-oriented operations [8]. Instead, we compare GRIP to an extension of the TPU architecture that incorporates features from GRIP to address these limitations. We refer to this modified design as the TPU-GNN.

TPU-GNN has an additional unit similar to GRIP's edge-unit between the TPU's unified buffer and the systolic data setup. This allows the TPU-GNN to natively support the GReTA programming model by mapping *aggregate* onto the new edge-unit, *combine* onto TPU's systolic array, and *update* onto the activation pipeline. This design also supports both the execution partitioning and vertex-tiling optimizations described in Section 6.

We estimate the latency of the TPU-GNN by modifying our cycle-accurate model to use a single fetch and gather unit. We also replace the vertex-unit with an identically sized 16×32 systolic array. As in the original TPU design, weights are stored off-chip and the dedicated weight bandwidth is limited to $30\,\mathrm{GiB/s}$. All other parameters remain unchanged compared to our evaluation of GRIP, including the use of $4\times$ DDR4-2400 for off-chip memory and the same partitioning and vertex-tiling optimizations.

This configuration achieves a $11.3\times$ speedup (Fig. 9b) compared to our baseline in Section 8.2. The main bottleneck in this approach is the limited bandwidth dedicated to weights. Moving weights on-chip as in GRIP results in a $1.72\times$ speedup. Higher performance memory for weights (e.g., HBM as used by later versions of the TPU) could also address this bottleneck. We leave a fuller exploration of these differences for future work.

Graphicionado. Graphicionado [9] is an accelerator architecture designed for graph analytics and can be used for GNN inference. However, it is designed for algorithms that use a small amount of state per-vertex. As a result, it suffers from two bottlenecks. First, like HyGCN, it cannot perform vertex-tiling since it requires full feature vectors to be accumulated. This results in a bottleneck similar to HyGCN since weight data cannot be easily reused between different vertices. Second, each lane has independent vertex units instead of using a single shared unit, increasing the required weight bandwidth by an amount proportional to the number of lanes.

We estimate the impact of these bottlenecks by modifying our simulator by disabling tiling and splitting the vertex unit into two units lanes that share a single tile buffer port. We also use the same partitioning scheme used for GRIP. This configuration results in a small speedup of $2.4\times$ over the baseline, shown in Fig. 9b. However, this is $8.1\times$ slower than GRIP due to bottlenecks in weight bandwidth.

9 Conclusion

This paper presents GRIP, an accelerator architecture designed for low latency GNN inference. GRIP splits GNN operations into a series of edge- and vertex-centric phases. Each phase is implemented independently in hardware, allowing for specialization of both the memory subsystem and execution units to improve latency. Additionally, GRIP has hardware support for several optimizations that further reduce latency, including pipelining operations between nodeflow partitions and vertex-tiling. We then implement GRIP as $28\,\mathrm{nm}$ ASIC capable of executing a range of different GNNs. On a variety of real graphs, our implementation improves 99th percentile latency by a geometric mean of $17\times$ and $23\times$ compared to a CPU and GPU baseline, respectively, while drawing only $5\,\mathrm{W}$.

References

[1] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 1024–1034.

[2] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2018, pp. 974–983.

[3] Y. Ma et al., "High performance graph convolutional networks with applications in testability analysis," in *Proc. 56th Annu. Des. Autom. Conf.*, 2019, pp. 1–6.

[4] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *Proc. 34th Int. Conf. Mach. Learn.*, 2017, pp. 1263–1272.

[5] L. Ma et al., "NeuGraph: Parallel deep neural network computation on large graphs," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 443–458.

[6] Y. Wang et al., "Gunrock: GPU graph analytics," *ACM Trans. Parallel Comput.*, vol. 4, no. 1, pp. 1–49, 2017.

[7] N. P. Jouppi, C. Young, N. Patil, and D. Patterson, "A domain-specific architecture for deep neural networks," *Commun. ACM*, vol. 61, no. 9, pp. 50–59, Aug. 2018.

[8] M. Balog, B. van Merriënboer, S. Moitra, Y. Li, and D. Tarlow, "Fast training of sparse graph neural networks on dense hardware," 2019, *arXiv:1906.11786*.

2. This is different from the CPU speedup reported by the HyGCN authors since we report latency speedup rather than throughput. In addition, we were able to replicate the HyGCN throughput speedup when our baseline was run on multiple sockets. Our methodology limits CPU inference to a single socket as described in Section 7.

[9] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2016, pp. 1–13.
[10] M. Yan et al., "HyGCN: A GCN accelerator with hybrid architecture," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2020, pp. 15–29.
[11] S. Abadal, A. Jain, R. Guirado, J. López-Alonso, and E. Alarcón, "Computing graph neural networks: A survey from algorithms to accelerators," *ACM Comput. Surv.*, vol. 54, no. 9, Oct. 2021, Art. no. 191.
[12] A. Jain, I. Liu, A. Sarda, and P. Molino, "Food discovery with Uber Eats: Using graph learning to power recommendations," 2019. Accessed: Feb. 20, 2020. [Online]. Available: https://eng.uber.com/uber-eats-graph-learning/
[13] Z. Huang, D. Zheng, Q. Gan, J. Zhou, and Z. Zhang, "Nodeflow and sampling," 2019. [Online]. Available: https://doc.dgl.ai/tutorials/models/5_giant_graph/1_sampling_mx.html#nodeflow
[14] K. Kiningham, P. Levis, and C. Re, "GReTA: Hardware optimized graph processing for GNNs," in *Proc. Workshop Resour.-Constrained Mach. Learn.*, 2020.
[15] P. W. Battaglia et al., "Relational inductive biases, deep learning, and graph networks," 2018, *arXiv:1806.01261*.
[16] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Trans. Neural Netw.*, vol. 20, no. 1, pp. 61–80, Jan. 2009.
[17] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proc. Int. Conf. Learn. Representations*, 2017.
[18] Intel Corporation, *Intel Math Kernel Library. Reference Manual*. Santa Clara, CA, USA: Intel Corporation, 2019.
[19] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, "cuSPARSE library," in *Proc. GPU Technol. Conf.*, 2010.
[20] J. Leskovec and A. Krevl, "SNAP datasets: Stanford large network dataset collection," Jun. 2014. [Online]. Available: http://snap.stanford.edu/data
[21] Gunrock developers, "Hive workflow report for GraphSage GPU implementation," 2019. Accessed: Feb. 20, 2020. [Online]. Available: https://gunrock.github.io/docs/hive/hive_graphSage.html
[22] Y. Chen et al., "DaDianNao: A machine-learning supercomputer," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2014, pp. 609–622.
[23] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.
[24] H. Zeng and V. Prasanna, "GraphACT: Accelerating GCN training on CPU-FPGA heterogeneous platforms," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2020, pp. 255–265.
[25] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," in *Proc. Int. Conf. Learn. Representations*, 2018.
[26] M. M. Ozdal et al., "Graph analytics accelerators for cognitive systems," *IEEE Micro*, vol. 37, no. 1, pp. 42–51, Jan./Feb. 2017.
[27] E. Nurvitadhi et al., "GraphGen: An FPGA framework for vertex-centric graph computation," in *Proc. IEEE 22nd Annu. Int. Symp. Field-Programmable Custom Comput. Mach.*, 2014, pp. 25–28.
[28] T. Oguntebi and K. Olukotun, "GraphOps: A dataflow library for graph analytics acceleration," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2016, pp. 111–117.
[29] NVDLA Developers, "NVDLA documentation - LUT programming," 2020. [Online]. Available: http://nvdla.org/hw/v1/ias/lut-programming.html
[30] J. Li, A. Louri, A. Karanth, and R. Bunescu, "GCNAX: A flexible and energy-efficient accelerator for graph convolutional neural networks," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit.*, 2021, pp. 775–788.
[31] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, 2011, Art. no. 1.
[32] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?," in *Proc. Int. Conf. Learn. Representations*, 2019.
[33] X. Bresson and T. Laurent, "Residual gated graph ConvNets," 2017, *arXiv:1711.07553*.
[34] N. Greeneltch and J. Xu, "Maximize TensorFlow performance on CPU: Considerations and recommendations for inference workloads," 2019. [Online]. Available: https://software.intel.com/en-us/articles/maximize-tensorflow-performance-on-cpu-considerations-and-recommendations-for-inference
[35] T. Developers, "Tensorflow," 2020. Accessed: Oct. 15, 2020. [Online]. Available: https://github.com/tensorflow/tensorflow
[36] P. Shivakumar and N. P. Jouppi, "CACTI 3.0: An integrated cache timing, power, and area model," Compaq Comput. Corporation, Palo Alto, CA, USA, Tech. Rep., 2001.
[37] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible DRAM simulator," *IEEE Comput. Archit. Lett.*, vol. 15, no. 1, pp. 45–49, Jan.–Jun. 2016.
[38] K. Chandrasekar et al., "DRAMPower: Open-source DRAM power & energy estimation tool," 2012. [Online]. Available: http://www.drampower.info
[39] M. Developers, "MLPerf inference rules," 2021. Accessed: Oct. 15, 2021. [Online]. Available: https://github.com/mlcommons/inference_policies/blob/master/inference_rules.adoc

Kevin Kiningham received the BS degree from the University of Michigan, in 2014. He is currently working toward the PhD degree in electrical engineering with Stanford University. His current research interests include hardware acceleration for machine learning, graph neural networks, and computer architecture.

Philip Levis (Member, IEEE) is an associate professor of computer science and electrical engineering. He heads the Stanford Information Networking Group (SING) and co-directs Lab64, the EE maker space at Stanford. He researches operating systems, networks, software design, and the hardware/software interface. He has received an NSF CAREER Award, a Microsoft New Faculty Fellowship, an Okawa Research Grant, numerous best paper awards and three test of time awards.

Christopher Ré is an associate professor with the Department of Computer Science, Stanford University. He is in the Stanford AI Lab and is affiliated with the Statistical Machine Learning Group. His recent work is to understand how software and hardware systems will change as a result of machine learning along with a continuing, petulant drive to work on math problems.