Hardware Acceleration of Graph Neural Networks

Adam Auten
Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
auten2@illinois.edu

Matthew Tomei
Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
tomei2@illinois.edu

Rakesh Kumar
Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
rakeshk@illinois.edu

***Abstract*—Graph neural networks (GNNs) have been shown to extend the power of machine learning to problems with graph-structured inputs. Recent research has shown that these algorithms can exceed state-of-the-art performance on applications ranging from molecular inference to community detection. We observe that existing execution platforms (including existing machine learning accelerators) are a poor fit for GNNs due to their unique memory access and data movement requirements. We propose, to the best of our knowledge, the first accelerator architecture targeting GNNs. The architecture includes dedicated hardware units to efficiently execute the irregular data movement required for graph computation in GNNs, while also providing high compute throughput required by GNN models. We show that our architecture outperforms existing execution platforms in terms of inference latency on several key GNN benchmarks (e.g., 7.5x higher performance than GPUs and 18x higher performance than CPUs at iso-bandwidth).**

I. Introduction

There has been considerable recent interest in machine learning algorithms that operate on graph-structured data. One such class of algorithms, Graph Neural Networks (GNNs) [13], has generated particular interest due to their ability to outperform existing techniques in many applications [18], [16]. GNNs can be described as an extension of the popular Deep Neural Network (DNN) [5] that allows the networks to natively support graph-structured inputs, outputs, and/or internal state. While this extension improves inference performance metrics, it also introduces computations that are not a good fit for existing machine learning accelerators (Section II).

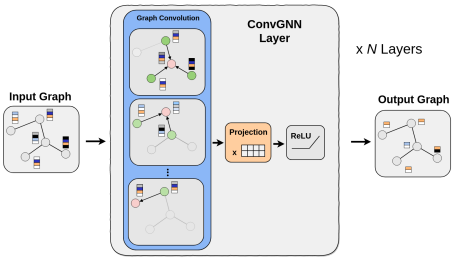

Fig. 1: General structure of a convolutional GNN network.

An almost entirely dominant sub-class of GNNs focuses on graph convolutions [11], [15], [7], [3]. A generic diagram of the structure of a convolutional GNN (ConvGNN) is shown in Figure 1. A graph with features is presented as input to a sequence of ConvGNN layers. Each layer propagates and aggregates features according to the underlying graph structure, and subsequently projects this feature onto a new subspace via a learned projection matrix. Finally, a nonlinear activation is optionally used on the resulting node features. The projection and nonlinear step can be seen as a traditional batched fully-connected layer or convolutional layer, depending on whether a single or the whole set of node states are considered as input. The final output layer produces a graph with transformed node features.

In this paper, we analyze ConvGNN model implementations to develop an understanding of the hardware requirements for efficient execution. We then leverage our understanding of GNN models and their implementations to develop an architecture for efficient acceleration of GNN-based inference. To the best of our knowledge, this is the first architecture for accelerating GNNs. Finally, we compare the performance of our inference accelerator against existing platforms and report significant performance benefits (7.5x higher performance than GPUs and 18x higher performance than CPUs at iso-bandwidth).

II. Do GNNs Need A New Accelerator?

GNN models can be thought of a composition of traditional graph and DNN algorithms. For example, projections of vertex features onto a subspace through a learned projection matrix are equivalent to a batched fully-connected layer in a neural network, while message passing or graph convolution steps are common operations in graph algorithms. While hardware accelerators for both classes of applications have been developed, the resulting architectures have important differences. Graph accelerators are typically MIMD architectures to extract irregular parallelism and typically provide high random-access memory bandwidth with relatively small access sizes. DNN accelerators typically opt for a SIMD architecture to exploit regularity in the application, and include caches or scratchpads to exploit data locality. An accelerator architecture for GNN inference will need elements of each class of accelerators. However, the exact nature of composition is non-obvious.

In order to determine the degree to which the graph components of a GNN algorithm introduce inefficiencies when run on a DNN accelerator, we modeled the execution of a Graph Convolutional Network (GCN- a popular GNN) on a Eyeriss-like [2] spatial architecture (Table I) with 182 processing elements (PEs). We describe the GCN algorithm as a series of convolutional and fully connected layers for three input graphs: Cora, Citeseer, and Pubmed, whose sizes can be found in Table V. The graph convolution step is modeled as a matrix multiplication with the adjacency matrix, and implemented as a convolution with the adjacency matrix as the weights. NN-Dataflow [6] is used for dataflow scheduling and analysis of inference latency, required off-chip bandwidth, and PE utilization. To measure how much computation and memory bandwidth is wasted due to computation on zero entries in the adjacency matrix, we separately measure the compute operations and memory accesses due to non-zero entries in the matrix for layers that operate on the adjacency matrix.

Figure 2 shows the bandwidth and PE utilization for GCN executing on the reference DNN spatial architecture accelerator. Table II lists the measured inference latency for each input graph assuming an aggressive clock frequency of 2.4 GHz. For all GCN input graphs that were considered, most of the compute and memory bandwidth is wasted due to the sparsity of the input graph. The degree of wasted compute and memory bandwidth depends largely on the sparsity of the input graph. For the sparsest input (Pubmed, at 99.989% sparse), only 1% of the memory requests and 2% of the compute are useful in performing the final inference. In general, this results in a significant amount of energy being wasted on unnecessary memory accesses. It also increases inference latency on systems where off-

978-1-5386-5541-2/18/$31.00

Number of PEs	182
PE configuration	13 x 14
Register File Size	512B
Global Buffer Size	108kB
Precision	32-bit fixed point

TABLE I: Configuration of the spatial architecture DNN accelerator, modeled after the silicon-proven Eyeriss DNN accelerator.

Input Graph	Inference Latency (ms)	
	Unlimited BW	68GBps BW
Cora	0.791	1.597
Citeseer	1.434	2.661
Pubmed	22.129	64.636

TABLE II: Inference Latencies of GCN on DNN spatial architecture accelerator, assuming a 2.4 GHz clock.

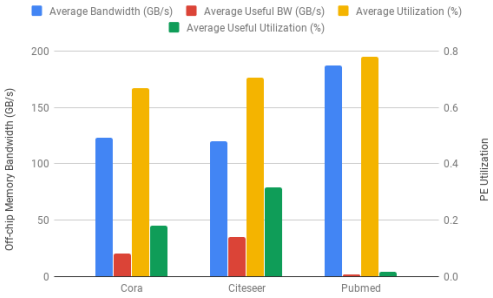

Fig. 2: Measured off-chip bandwidth and PE utilization of GCN model running on a DNN spatial architecture accelerator. Useful bandwidth and utilization counts only non-zero entries in operations on the adjacency matrix

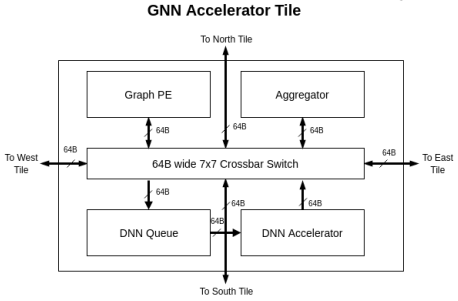

Fig. 3: Block diagram of a tile in the GNN accelerator showing hardware modules, NoC, and connections to external tiles

chip bandwidth is constrained (Section VI). This motivates the need to design accelerators specifically targeting GNNs.

Several recent works optimize DNN accelerators to exploit sparsity in matrices for traditional DNN workloads (e.g., [9]) and demonstrate significant benefits. However, this sparsity is at least two orders of magnitude lower than the sparsity in a sparse graph typically inputted to a GNN. For example, in the work by Han et al[9], 88.9-92.3% of the elements can be removed without impacting accuracy of learning. For our graph inputs, on the other hand, 99.8% values in the dense vertex adjacency matrices are 0. I.e., useful elements occur at a rate $\sim 100\times$ lower than in our graph inputs. Due to this multiple order of magnitude difference in useful elements, previous DNN accelerator-based approaches are inadequate for GNNs. For example, even though the input and output to their compute logic is sparse, they work with dense representations of the inputs when scheduling the useful operations to be performed. For very sparse matrices, that scheduler can lead to low PE utilization since so little of the work considered is useful.

One approach to efficiently deal with massive sparsity is to generate a sparse representation of the graph and dynamically generate a work list. This is the approach used by graph accelerators(e.g., [17]). Unfortunately, while graph accelerators are a good fit the inter-vertex communication components of GNNs, they are a poor fit for the per-vertex components. Unlike DNN accelerators, graph accelerators do not exploit the memory locality and regular parallelism of the per-vertex computations in GNNs. They do not have the large arrays of compute units required to efficiently perform the statically predictable portion of the computation. They are also built to favor the small memory accesses typical to graph applications[8] due to their small per-vertex state. GNNs, on the other hand, often have large per-vertex state and could, therefore, benefit from wide accesses.

III. An Accelerator Architecture for GNNs

To design a GNN accelerator, we analyzed several GNN benchmarks (Section V) and observed that the operations required for the execution of vertex-programmed GNNs fall into three distinct categories with unique computational characteristics: graph traversal, DNN computation, and aggregation. Graph traversal includes operations required for navigating the underlying graph structure, and typically requires sequences of two or more dependent memory accesses. DNN computation includes vertex-local operations that operate on dense data representations, such as the projection of a vertex feature onto an intermediate space through a learned matrix (i.e. a fully-connected neural network layer). Aggregation includes reduction operations with input and output sizes that depend on the graph input to the GNN.

Figure 3 shows the key hardware blocks and datapath within our accelerator tile that supports those three categories. The Graph Processing Element (GPE) is responsible for graph traversal and sequencing computation steps which are dependent on the underlying graph structure (e.g. aggregation). The DNN Accelerator (DNA) is responsible for executing the DNN computation within the GNN model. The AGG (aggregator) is responsible for performing aggregation of the features, and is coordinated by the GPE according to the graph traversal. Finally, the DNN queue (DNQ) is responsible for buffering memory requests and intermediate results as they are passed to the DNA. Each block is connected to a configurable bus which allows various types of GNN dataflows to be supported.

Graph PE: At a high level, the GPE (Figure 4) functions as a control core, coordinating other elements on the system. The GPE consists of a general purpose CPU which executes a lightweight runtime. The runtime manages a pool of software threads and schedules them according to system load. It also performs global synchronization, which is required between layers to ensure the executing vertex program reaches a consistent state across all accelerator modules.

The GPE also has a scratchpad memory to hold the application state and binary. The interface to main memory is specialized to allow the GPE to issue indirect asynchronous memory requests. These requests are supported by a dedicated flit buffer from which data is written into the scratchpad as data is received from the NoC without requiring core intervention. Finally, an allocation bus connects the GPE to the DNN Queue (DNQ) and the Aggregator (AGG) modules, allowing the GPE to request scratchpad space on each module for dataflow mapping.

DNN Queue: The DNN Queue (DNQ) (Figure 6) is responsible for staging inputs to the spatial architecture accelerator and providing support for multiple simultaneous DNN models. The queue supports delayed enqueues, which allow queue space to be allocated before

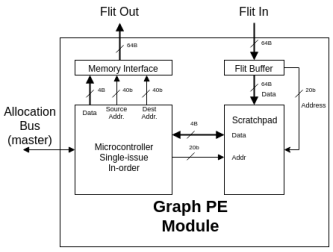

Fig. 4: Block diagram of the GraphPE module

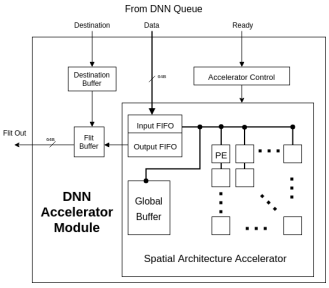

Fig. 5: Block diagram of the DNN Accelerator module

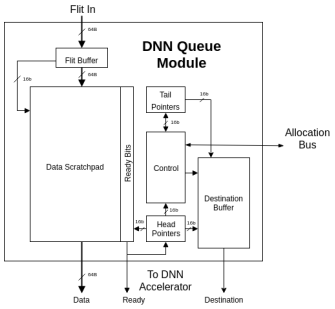

Fig. 6: Block diagram of the DNN Queue module

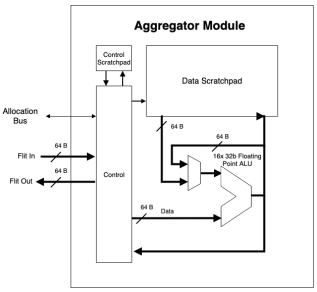

Fig. 7: Block diagram of the Aggregator module

data is written. A large scratchpad (62kB) provides storage for the queue and ready bits, which are required due to the delayed writes. A second scratchpad (2kB) stores destination information for routing responses. A control logic manages queue allocation, while the destination buffer stores allocation-related information.

The control logic maintains two sets of head and tail pointers, allowing it to manage two virtual queues. The relative size of each queue is configurable at runtime via the allocation bus. Due to the single dequeue interface, only one queue may dequeue at a time. A lazy queue switching algorithm is used, whereby the queue eligible for dequeue is only switched when the DNA has been idle for 16 cycles. This acts to reduce the number of queue switches that need to occur.

The DNQ contains a single slave NoC port, which can accept flits from the network. Incoming flits are stored in the flit buffer, decoded, and subsequently written into the scratchpad at the appropriate queue location. Ready bits are maintained for every 4B word in the queue and set upon fill and unset upon dequeue. An allocation bus is connected to the GPE which allows it to request queue entries and fill in their associated destination entries. These destination entries represent the NoC addresses which will be used to route the final results once the queue entries have been processed by the DNA.

Once the entry at the head of the queue is marked ready, its entry is read out of the scratchpad and sent to the DNA along with its destination from the destination buffer.

DNN Accelerator: The DNN Accelerator (DNA) (Figure 5) is responsible for executing the DNN phases of the GNN algorithms. Because the DNN phases of GNNs are computationally similar to those supported by existing spatial architecture accelerators, we reuse such an architecture here.

The DNA has two interfaces to the rest of the GNN accelerator. A read port from the DNQ provides output routing information via the destination port, as well as input data to the DNN model. Outputs from the spatial architecture accelerator are sent to the flit buffer, where they are combined with their destination into a flit and sent to the NoC. The internal spatial architecture accelerator is sized according to Table I.

Aggregator: The Aggregator (AGG) (Figure 7) is responsible for performing the aggregation steps in a GNN model, and manages a pool of in-progress aggregations. The AGG only supports aggregation operations that are associative, which allows data to be aggregated in any order. It contains a pair of scratchpads for control (2kB) and data storage (62kB), a bank of 16 32-bit ALUs, control logic, and a bidirectional NoC port. Additionally, an allocation port is included, which allows the GPE to configure the AGG and allocate new aggregations.

As packets from the network arrive at the AGG, their address is used to determine which aggregation they belong to. A bank of ALUs is used to aggregate the incoming data with the existing aggregation read out of the data scratchpad. Additionally, a per-aggregation count is decremented to keep track of remaining elements in the aggregation. Once the count reaches 0, the aggregation is complete and the data are sent to the destination for that aggregation, which is configured at allocation time and stored in the control scratchpad.

A control logic manages allocation to the data scratchpad and sequences ALU operations in accordance with incoming NoC traffic. The scratchpad is managed by a control logic, which divides the scratchpad into a number of evenly-sized entries. The size of these entries are configurable at runtime. The control logic uses a 2kB control scratchpad which stores per-aggregation metadata, including the destination address used to send the result once the aggregation is complete.

Algorithm 1 Accelerator Runtime

```
1: global variables
2:     layers                          ▷ sequence of layers in the GNN model
3:     graph                           ▷ the input graph
4:     inQueue                         ▷ the input work queue
5:     outQueue                        ▷ the output work queue
6: end global variables
7: procedure INIT
8:     for n ∈ graph.nodes do
9:         inQueue.enqueue(n)
10:    end for
11: end procedure
12: procedure RUNTIME
13:    for layer ∈ layers do
14:        CONFIG(layer.config)        ▷ sets up DNA, DNQ and AGG
15:        SYNC                        ▷ ensure all modules are configured
16:        while inQueue.notEmpty() do
17:            n ← inQueue.dequeue()
18:            layer.runVertex(graph, n)
19:            outQueue.enqueue(n)
20:        end while
21:        swap(inQueue, outQueue)
22:        SYNC                        ▷ ensure all modules have completed work
23:    end for
24: end procedure
```

IV. RUNTIME

The GNN Accelerator program describes a GNN model as an ordered sequence of layers. Each layer takes as input a graph on which it performs a vertex program to produces an output graph. These layers are strung together in a sequence to implement a full GNN model. The first layer takes the model input as its input graph, while subsequent layers use the output of the preceding layer. The final layer produces the output graph.

The execution of these layers is managed by a software runtime, shown in Algorithm 1. In memory work queues, inQueue and outQueue, are used to track unprocessed vertices, and are shared across all GPEs. Global synchronization barriers between each layer

Parameter	Value
CPU	14-core Intel Xeon E5-2680v4 @ 2.4GHz
Memory	128GB of 4x DDR4-2133
GPU	NVIDIA Titan XP @ 1582 MHz
GPU Memory	12GB of GDDR5X @ 547.7 GB/s

TABLE III: Baseline system architecture

ensure all prior work has been completed and all hardware units are idle.

Each layer has two distinct components. The first is system configuration, shown on line 14, which describes the configuration of all hardware modules in the accelerator, i.e. the number of queues and size of each element in the `DNQ`, the number of elements and operation in each `AGG` aggregation, and the DNN layer dataflows for the `DNA`. Separated from the first component by a global barrier on line 15, the second component is a vertex program that describes the dataflow required to compute one output vertex in the layer. The loop starting on line 16 executes this vertex program for all vertices in the `inQueue`.

The runtime also manages a pool of software threads, allowing several vertices to be processed simultaneously. Whenever a memory load is requested, the system issues a non-blocking memory request, and stores a sentinel value in the scratchpad destination. The `GPE` then performs a software context switch to another thread. Since all program state is stored in the scratchpad, these context switches can be performed inexpensively. We estimate that with minimal hardware overhead, the latency of such a switch can be performed in a single cycle.

V. Methodology

GNN Benchmarks: We used the following four GNN models for our evaluations A) Graph Convolutional Networks (GCNs) [11]: The GCN is a spectral-based ConvGNN that reaches state-of-the-art accuracy on several text classification and knowledge graph datasets B) Graph Attention Networks (GATs) [15]: The GAT model augments the traditional graph convolution techniques used in ConvGNNs with a self-attention layer and drops the degree normalization term. It can be applied to directed graphs as well as to graphs unseen during training since it does not rely on the degree term. C) Message Passing Neural Networks (MPNNs) [7]: The MPNN is a generic class of spatial GNN that can be used to estimate real-valued functions over graph structured inputs. A message passing algorithm is iterated several times across nodes in the input graph, after which the global state is aggregated into an estimate. D). Power Graph Neural Networks (PGNN) [3]: The PGNN incorporates a multi-hop convolution and is often used as a component in a more complex Line Graph Neural Network model. Our selection of benchmarks provides adequate diversity across several dimensions in a GNN algorithm: spatial versus spectral convolution, different aggregation schemes, large vs small models, and different types of graph traversal.

GNN Accelerator Model: We developed a Booksim [12]-based custom simulation model for our accelerator. The simulator takes network topology and configuration as inputs (our network parameter values are in Table IV). During simulation, it connects the various accelerator tiles in our design and sequences the messages passed between tile modules. Booksim is a cycle accurate network simulator, so our simulation infrastructure can be thought of as a collection of packet generators connected to a network where the packet generators are models of the different components of the system.

For the memory controllers, we implement a simple bandwidth-latency model that enqueues up to 32 requests and services them in order according to the latency and bandwidth configuration. Each memory module is capable of servicing 68GBps of read/write traffic, which is roughly equivalent to 4 channels of DDR3-2400 memory. We assume a memory access granularity of 64B, and requests which are not integer multiples of 64B and properly aligned will result in wasted DRAM bandwidth but not wasted interconnect bandwidth.

For the GraphPE, we use an event-driven model where certain program steps require a certain latency. The program is broken up into steps such that any communication with other components in the system falls between steps and can have nondeterministic latency. The steps can reasonably be assumed to have deterministic latency due to the simplicity of the GraphPE cores. The sequence of program steps by the GraphPE is configurable at runtime to realize the execution of our different GNN benchmarks. Our GraphPE models a single threaded microprocessor with scratchpad for program and data. We assume that each ALU operation, memory access, or IO-operation executes in a single cycle.

The DNN Accelerator is modeled using a latency-throughput model similar to the memory controllers. NN-Dataflow[6] is used to map DNN models onto a Eyeriss-like single-tile spatial array accelerator with 182 PEs configured in a 13x14 array.

The Accumulator is modeled as a simple bank of 16 ALUs, a scratchpad, and a hardware FIFO to manage scratchpad allocation. We assume it takes one cycle to allocate an entry in the ALU scratchpad, and that the accumulator can issue one memory store per cycle, provided there is sufficient space in its 2kB flit buffer, which is drained one flit per cycle.

Input Datasets: Our choice of input graphs (Table V) for GCN and GAT come directly from their reference implementations: cora, pubmed, and citeseer for GCN, and cora for GAT. For MPNN, we choose the first 1000 graphs in the QM9 dataset (used in the reference implementation) to reduce simulation runtime. For PGNN, we use as the input a subgraph extracted from the DBLP dataset in a manner similar to [3]. Each graph also has a number of output features which are the results of the GNN inference. For Citeseer, Cora, Pubmed, and DBLP, these correspond to node labels. For QM9, these correspond to features which can be used to infer properties about the graph. In the DBLP graphs used for PGNN, no vertex or edge features are present. To accommodate this, the reference implementation uses the vertex degree as a single-element vertex state, a technique we duplicate in our evaluation.

Accelerator Configurations: We explore three configurations (Table VI, Figure 9) of our accelerator to help us compare against our baseline CPU and GPU systems (Table III). The first two configurations, *CPU iso-bandwidth* and *GPU iso-bandwidth*, match the memory bandwidth of our CPU and GPU baseline systems by tiling accelerators in a 2D mesh. The third configuration, *GPU iso-FLOPS*, contains roughly the same number of floating point units as our GPU baseline to help us gauge the utilization of our available compute.

VI. Results

As mentioned above, we evaluate our accelerator architecture against two baselines - an off-the-shelf CPU-based system and a second system that includes both a CPU and a GPU (Table III lists the characteristics of the CPU used in the two baselines and the GPU used in the second baseline). Using reference implementations of the our benchmark sets [10][14][4][1], we measure the inference latency on each baseline system and report results in Table VII. For the GAT baseline measurement, the attention normalization step was removed to match our accelerator implementation in Section IV. For the GPU system measurements, only GPU kernel time is considered

Parameter	Value
Link Delay	1 cycle
Routing Delay	1 cycle
Input buffers	4 flits, 256B
Routing algorithm	min-routing

TABLE IV: Booksim NoC Model Parameters

Dataset	Graphs	Total Nodes	Total Edges	Vertex Features	Edge Feat.	Output Feat.
Cora	1	2708	5429	1433	0	7
Citeseer	1	3327	4732	3703	0	6
Pubmed	1	19717	44338	500	0	3
QM9_1000	1000	12314	12080	13	5	73
DBLP_1	1	547	2654	1	0	3

TABLE V: Input dataset statistics

Configuration	Tiles	Mem. Nodes	ALUs	Mem. BW (GBps)
CPU iso-BW	1	1	198	68
GPU iso-BW	8	8	1584	544
GPU iso-FLOPS	16	8	3168	544

TABLE VI: GNN accelerator configurations used in our evaluations

Benchmark	Input Graph	Inference Latency (ms) CPU System	GPU system
GCN	Cora	3.50	0.366
GCN	Citeseer	3.97	0.391
GCN	Pubmed	30.11	0.893
GAT	Cora	13.60	0.801
MPNN	QM9_1000	2716.00	443.3
PGNN	DBLP_1	15.70	7.50

TABLE VII: Measured inference latencies of the referen implementations of the GNN benchmark applications executing on the CPU and GPU systems.For the GPU system, only the GPU kernel execution times are considered.

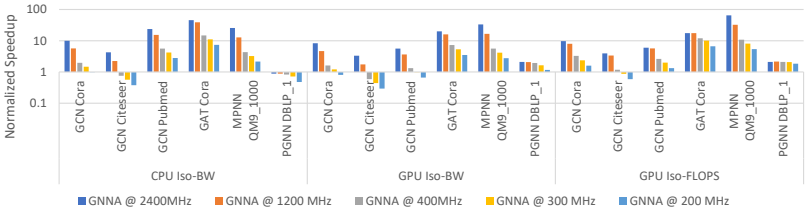

Fig. 8: Normalized speedups of the CPU iso-BW, GPU iso-BW, and multi-tile GNNAs over the CPU, GPU, and GPU iso-FLOPS configuration respectively

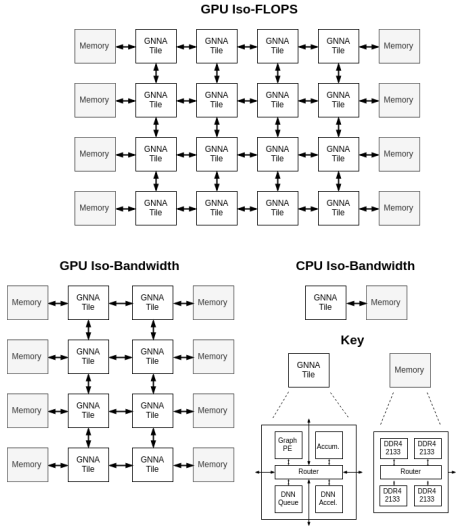

Fig. 9: Topology of GNN accelerator configurations used in our evaluations.

to isolate architectural performance and remove contributions of the OS/GPU runtime. Note that the GCN, GAT, and MPNN, use sparse matrix operations where applicable, so these implementations should be reasonably performant.

It is instructive to compare the measured latency numbers for GCN on the CPU and the GPU baselines (Table VII) against those modeled on a DNN accelerator (Table II). Recall that the DNN accelerator contains 182 compute units while the CPU system only has 14. Despite the 13x more compute, the DNN accelerator is outperformed by the CPU on the sparsest graph in our dataset, Pubmed. This can be explained by considering that, for Pubmed, the accelerator wastes 97% of its compute due to sparsity, giving an effective throughput of only 5.4 operations per cycle. For graphs with lower sparsity (Cora and Citeseer), fewer operations are wasted due to sparsity and the higher compute throughput causes the accelerator to outperform the CPU baseline. This comparison shows that, in order to scale performance to sparse graphs, the ability to execute sparse operations is as important as compute throughput.

To compare our GNN accelerator against our baseline systems, we perform three experiments, each using a different configuration for the GNN accelerator, depicted in Figure 9 and described in Table VI.

A. CPU Iso-Bandwidth

We measure the inference latency of our benchmark GNN models for various input graphs on the CPU iso-bandwidth configuration of our accelerator. We sweep several clock frequencies to determine the sensitivity to clock scaling. The left third of Figure 8 shows that our accelerator provides speedups for all benchmarks except PGNN. To help us interpret these results, we also plot several performance metrics in Figure 10. For the GCN benchmarks, a bandwidth utilization of 79%, 70%, and 54% is observed for the Cora, Citeseer, and Pubmed inputs. Compared to the useful bandwidth requested by a similarly-sized DNN accelerator (Figure 2, these represent utilization improvements of +61% +39% +53%, respectively. The improved utilization of available memory bandwidth directly translates to performance improvements.

We see that the benchmarks with larger portions of DNA computations see the largest speedups. GAT, MPNN are two of the best performers, and have the most computation being executed on the DNA. This is due to the roughly 14x compute bandwidth compared to the CPU baseline.

PGNN does not see any benefit from our accelerator, and sees a 12% increase in inference latency at the maximum clock of 2.4 GHz. This is likely due to a combination of factors. First, the input graph on the PGNN model is relatively small, as there is only a single feature per vertex and the number of vertices are relatively small (see Table V). For a working set this small, the CPU caches and prefetching do well at hiding the memory latency, whereas we assume a fixed 20ns latency to memory in our accelerator. Figure 10 shows that the PGNN benchmark shows very little DNA utilization. This is because of the complicated graph traversal required largely outweighs the simple dense matrix vector operations executing on the DNA. As such, the GPE becomes the bottleneck, preventing saturation of DNNA compute.

B. GPU Iso-Bandwidth and Iso-FLOPS

We use the GPU iso-bandwidth configuration to evaluate our accelerator against a GPU-like system with 547 GB/s of memory bandwidth. The middle of Figure 8 shows the inference latencies normalized to the GPU baseline latencies given in Table VII. We see that even when comparing our architecture to a similarly equipped GPU system, there are speed-ups across all benchmarks at 1.2 GHz clocks and above.

For GCN Pubmed and GAT Cora, speedups are slightly reduced to those in the CPU iso-bandwidth evaluation, which is due to the

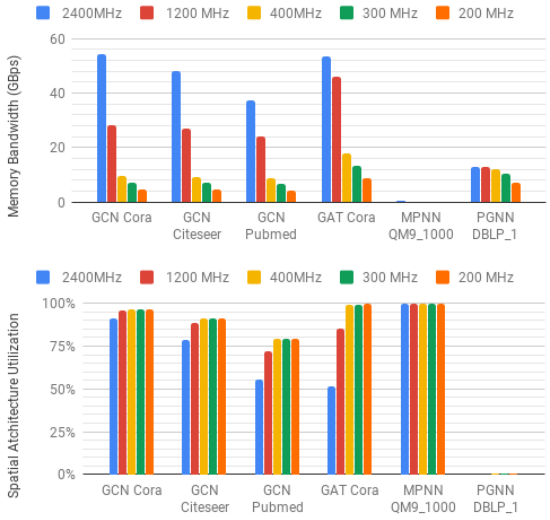

Fig. 10: Observed mean memory bandwidth and DNA utilization of all benchmarks in the CPU iso-bandwidth configuration

greater compute throughput available in the GPU baseline. Interetsingly, the speed-ups for the GCN Cora and Citeseer benchmarks see marginal reductions in speedups compared to the CPU iso-bandwidth comparisons. This is due to the fact that these baselines have relatively low compute requirement, and are thus bottlenecked in both cases by the bandwidth to memory.

However, models with small input graphs (PGNN and MPNN) see improved speedups comparing to the CPU platform. We believe this is because the GPU in our baseline system can only perform relatively wide accesses, so the smaller graphs use this bandwidth inefficiently.

As another comparison against a GPU system, we measured speed-ups against our GPU iso-FLOPS configuration, which has a comparable number of ALUs as the GPU baseline. The simulated speed-ups are given in the right third of Figure 8 relative to the measured GPU latencies in Table VII.

Models with very high compute requirement, such as MPNN, see the greatest speedups at over 60x compared to the GPU baseline system. However, all other models see speedups similar to that in the GPU iso-bandwidth evaluations. This shows that the remaining models are not compute bound, and are instead bound by memory bandwidth or latency. This can also be seen by comparing the speedups at 2.4GHz vs those at 1.2GHz. In both cases, the NoC and memory bandwidth are identical, but at 1.2GHz, the DNA and GPE have half of the throughput. In Figure 8, we see that for GCN, GAT, and PGNN, there is little change in speedup between 2.4 and 1.2GHz, which supports our claim that the system is memory-bound.

VII. Conclusion

Graph neural networks (GNNs) have been shown to extend the power of machine learning to problems dealing with graph-structured inputs. In this paper, we showed that existing execution platforms do not perform well for GNNs given their unique memory and data movement requirements. We analyzed several popular GNN algorithms to uncover the fundamental building blocks of an accelerator for GNNs. The proposed GNN accelerator connects dedicated hardware units for graph traversals, dense matrix operations, and graph aggregations using a flexible NoC to enable efficient execution of the irregular data movements required for graph computation. We showed that our architecture outperforms existing execution platforms in terms inference latency on several key GNN benchmarks (e.g., 7.5x higher performance than GPUs and 18x higher performance than CPUs at iso-bandwidth).

Acknowledgments

The authors would like to thank Patrick McGrady for his initial studies, anonymous reviewers for their feedback, and Samsung for its partial support of the work.

References

[1] afansi. Implementation of the paper "Community Detection with Graph Neural Networks", [1] in Pytorch: afansi/multiscalegnn, August 2019. original-date: 2018-01-05T21:34:29Z.

[2] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 367–379. IEEE, 2016.

[3] Zhengdao Chen, Xiang Li, and Joan Bruna. Supervised Community Detection with Line Graph Neural Networks. *arXiv:1705.08415 [stat]*, May 2017. arXiv: 1705.08415.

[4] Fei Ding. Graph Neural Networks for Quantum Chemistry. Contribute to ifding/graph-neural-networks development by creating an account on GitHub, July 2019. original-date: 2018-03-06T14:29:27Z.

[5] X. Du, Y. Cai, S. Wang, and L. Zhang. Overview of deep learning. In *2016 31st Youth Academic Annual Conference of Chinese Association of Automation (YAC)*, pages 159–164, November 2016.

[6] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. TANGRAM: Optimized Coarse-Grained Dataflow for Scalable NN Accelerators. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '19*, pages 807–820, Providence, RI, USA, 2019. ACM Press.

[7] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural Message Passing for Quantum Chemistry. *arXiv:1704.01212 [cs]*, April 2017. arXiv: 1704.01212.

[8] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, October 2016.

[9] Song Han, Huizi Mao, and William J. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In *ICLR*, 2016. arXiv: 1510.00149.

[10] Thomas Kipf. Implementation of Graph Convolutional Networks in TensorFlow: tkipf/gcn, August 2019. original-date: 2016-11-11T10:59:21Z.

[11] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. *arXiv:1609.02907 [cs, stat]*, September 2016. arXiv: 1609.02907.

[12] Nan Jiang, D. U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. E. Shaw, J. Kim, and W. J. Dally. A detailed and flexible cycle-accurate Network-on-Chip simulator. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 86–96, April 2013.

[13] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, 20(1):61–80, January 2009.

[14] Petar Velikovi. Graph Attention Networks (https://arxiv.org/abs/1710.10903): PetarV-/GAT, August 2019. original-date: 2018-02-01T02:17:22Z.

[15] Petar Velikovi, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Li, and Yoshua Bengio. Graph Attention Networks. *arXiv:1710.10903 [cs, stat]*, October 2017. arXiv: 1710.10903.

[16] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A Comprehensive Survey on Graph Neural Networks. *arXiv:1901.00596 [cs, stat]*, January 2019. arXiv: 1901.00596.

[17] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 544–557, Vienna, February 2018. IEEE.

[18] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, and Maosong Sun. Graph Neural Networks: A Review of Methods and Applications. *arXiv:1812.08434 [cs, stat]*, December 2018. arXiv: 1812.08434.