TGN-PNM: A Near-Memory Architecture for Temporal GNN Inference on 3D-Stacked Memory

Alif Ahmed
alifahmed@virginia.edu
University of Virginia
USA

Felix Lin
felixlin@virginia.edu
University of Virginia
USA

Jundong Li
jl6qk@virginia.edu
University of Virginia
USA

Kevin Skadron
skadron@virginia.edu
University of Virginia
USA

Abstract

This paper introduces TGN-PNM, a Processing-in-Memory (PIM)-based accelerator designed specifically for Temporal Graph Neural Networks (TGNNs). TGNNs are gaining increasing attention due to their ability to capture complex relationships and temporal dynamics in various domains. However, designing accelerators for TGNN workloads poses several challenges, including the lack of a standard model architecture, the absence of distinct execution phases, and the difficulty in maintaining workload balance in evolving graphs. Existing accelerators for static GNNs are not easily extendable to TGNNs. In this work, we propose TGN-PNM, which leverages the concept of vault-level parallelism by placing a Vault Processing Unit (VPU) at each vault in a 3D-stacked memory. The VPU consists of a SIMD unit for memory-intensive operations and a systolic array for compute-intensive operations. By placing compute units at the logic layer, our design achieves near-linear performance improvement with increasing memory stacks and exposes higher internal memory bandwidth. We address the challenges of TGNN workloads by introducing a feature-dimension partitioning scheme that minimizes inter-vault communication and improves workload balance. Our architecture allows compute units in all vaults to work in lock-step, resulting in efficient execution. We evaluate TGN-PNM using various TGNN models, batch sizes, and datasets, demonstrating its effectiveness in handling both memory-bound and compute-bound kernels. Our proposed accelerator offers significant performance improvements over existing approaches and provides flexibility to accommodate future TGNN model variations.

CCS Concepts

• **Hardware → Emerging architectures**; • **Computer systems organization → Systolic arrays**; **Neural networks**; **Data flow architectures**; **Heterogeneous (hybrid) systems**; **Special purpose systems**.

Keywords

graph neural network, temporal graph neural network, processing-in-memory, systolic array

1 Introduction

Graph Neural Networks (GNNs) have gained significant attention in various domains, including social network analysis [1], biology [2–4], and recommendation systems [5, 6], due to their remarkable ability to capture complex relationships and interactions among entities. GNNs typically learns the graph representation via a message-passing mechanism that aggregates the neighborhood information into low-dimensional node embeddings. Afterwards, these embeddings are used to perform various downstream inference tasks on the graph, such as node classification [7, 8], link prediction [9–11], and graph clustering [12].

In user facing production environments, these inference tasks on GNNs are often subjected to stringent latency/throughput constraints. Subsequently, researchers have proposed a plethora of accelerator architectures, focusing primarily on the inference tasks on static GNNs [6, 13–18]. These accelerators leverage the observation that inference tasks on prevailing static GNN models (e.g., GCN) are primarily composed of two distinct execution phases: *aggregation* and *combination* [13]. The *aggregation* phase uses the graph structure and neighbor interactions to recursively update the feature vectors of the nodes. In contrast, the *combination* phase transforms the aggregated features of each node through neural network layers to compute the node embeddings. Among these two phases, the aggregation phase shows the typical pitfalls of graph processing behavior; this phase is data-intensive with highly irregular access patterns, and low compute intensity. On the other hand, the combination phase shows a regular access pattern with a high compute density. Accordingly, existing static GNN accelerators aim to optimize one or both of these phases.

However, many of the real-life applications are dynamic in nature, where the graph topology is no longer static, as the nodes and edges can evolve rapidly over time. For example, on social networks, users are forming new connections, deleting old ones, and groups are constantly evolving. These additions/deletions of connections or changes in user interests can significantly alter the social graph structure over short periods. In telecommunications, the network of calls and messages between users is constantly changing. New subscribers are added, some subscribers might leave, and communication patterns shift frequently, necessitating dynamic graph models to predict the network trajectory. In transportation networks, optimal routes undergo dynamic changes due to traffic conditions, road closures, new road developments, and varying transportation demands. The topology of such graphs changes as traffic lights, signs, and usage alter in real-time. In recommendation systems, as users interact with new items, the underlying graph structure

used to model these interactions changes continuously. By incorporating the timing and sequence of interactions or changes within the graph, *temporal* GNNs (TGNN) can more accurately predict future states of the graphs compared to static GNNs [19–21]. This temporal context helps the model understand not just the structure of the graph but also how that structure evolves, which is essential for accurate predictions in dynamic/evolving graphs. For example, authors in [19] compared the accuracy of temporal GNN approach with several state-of-the-art static GNN approaches on datasets from Wikipedia, Reddit, and Twitter. On edge prediction tasks, temporal GNN improved the accuracy by 7.2% on Wikipedia, 1.3% on Reddit, and 46.6% on Twitter, over the *best* static GNN method. These results highlight the importance of considering temporal contexts.

Despite the importance of TGNNs, it is extremely challenging to design an accelerator targeting TGNN than static GNN, for several reasons: 1) The lack of standard model architecture. Unlike GCN, which is the prevailing model for static GNNs, there is no general consensus about which temporal GNN model is the best, and this often boils down to accuracy-complexity tradeoff. Therefore, it is desirable that the proposed accelerator is flexible to accommodate design choices that might arise in the future. 2) Unlike static GNN's aggregation/combination phases, models for TGNNs often cannot be decomposed into distinct phases of rigid execution patterns. For example, aggregation phase itself may require neural network layers, such as in temporal graph attention [22]. Furthermore, based on the TGNN model architecture, batch size, and dataset, the operational intensity of the execution phases can vary widely and can contain both memory-bound and compute-bound kernels (more discussion on this topic on Section 2.1). The proposed accelerator must handle both types of kernels efficiently to be able to extract maximum performance. 3) Due to the evolving nature of the graph, it is difficult to maintain proper workload balance. This issue is less prominent in static graphs, where pre-processing steps can by applied on the graph to reorganize vertices to ensure balance and ensure maximum locality [23–25]. This pre-processing cost is a one time cost for static graphs and gets amortized over time, but becomes prohibitively expensive on dynamic graphs. Researchers have proposed node migration techniques that can partially solve the workload imbalance on dynamic graphs [26, 27]. These approaches are applicable in distributed memory systems with non-uniform memory access latency, where the nodes migrate towards their neighbors over time to reduce the memory access latency and/or network hops. Unfortunately, migration-based methods introduce additional data movement. This data movement overhead is significant for GNN workloads, as each node is usually associated with a large number of features. Furthermore, as the nodes are constantly moving to minimize latency/hops, migration-based techniques also require additional dictionary lookups to determine the node's current address. Node addresses can be cached to facilitate faster lookup, but in that case maintaining coherence becomes an issue. Because of these aforementioned challenges, we are aware of only one prior accelerator that targets TGNN workloads [28], mapping the TGN framework [19] on an FPGA (referred as tFPGA). However, we observed that this approach can only support a small numbers of compute units due to FPGA resource limitations, and we show this is vastly outperformed by in-/near-memory-processing-based approaches.

In this paper, we propose TGN-PNM, which is a near-memory architecture for accelerating TGNN workloads. TGN-PNM exploits vault-level parallelism by placing one Vault Processing Unit (VPU) at every vault in the logic-layer of a 3D-stacked memory. Each VPU contains a SIMD unit for common memory-intensive operations (e.g., BLAS level 1 and 2 kernels, time encoding, and other elementwise operations) and a systolic array for compute-intensive operations (e.g., BLAS level 3 kernels). Placing the compute units at the logic layer enables obtaining performance that is memory-capacity-proportional (i.e., linear performance improvement can be obtained by increasing the number of memory stacks). Additionally, it also exposes some of the internal memory bandwidth to the processing units, which can be an order of magnitude higher than the bandwidth seen by external I/O links [29]. While it is possible to obtain a finer-grained parallelism by placing the compute units at the banks or subarrays, as proposed by a few prior generic accelerator architectures [30–36], it comes with several pitfalls: i) Any extra logic in the compute units gets multiplied by the number of banks/subarrays. Therefore, these compute unit can only support bare-minimum functionality and cannot accommodate complex logic, such as transcendental time encoding functions needed for TGNN kernels. These functions needs to be handled separately, for example, by the host or by placing compute cores in the logic layer, thereby introducing a lot of data-movement overhead. ii) Placing compute units at the bank-/subarray-level also imposes a stricter requirement on the data layout and moving the intermediate results to conform to this data-layout requirement often times becomes a bottleneck. iii) Finally, gates in the DRAM dies are usually larger and slower than their counterparts in the logic die. These reasons motivate us to place the compute units at the logic layer, rather than in the banks or subarrays.

Data placement is a key factor in the efficiency of a near-data processing solution. In our approach, if we partition the graph traditionally, where each vault holds a subset of nodes, the primary bottleneck arises from significant inter-vault communication during neighbor aggregation. To overcome this limitation and improve workload balance, we introduce a feature-dimension partitioning scheme. The key idea is to allocate each vault with a subset of the features of *all* nodes. This approach takes advantage of the fact that nodes in GNNs typically possess numerous features (ranging from hundreds to thousands). With this partitioning scheme, all element-wise operations, including operand fetch for neighbor aggregation, become localized within each VPU. Consequently, inter-vault communication is only required for dot-product reduction operations during matrix-vector and matrix-matrix multiplications, as each vault produces a portion of the output vector or matrix. This reduction has a regular pattern and can be handled efficiently with a global pipelined tree-adder. Additionally, each vault only needs to contain a subset of the weights and does not need to duplicate the weights across all the vaults. Another advantage of our proposed architecture is that the compute units within all vaults can work in lockstep, leveraging a single instruction queue and a unified fetch/decode unit. We extended our approach with a broadcasting mechanism and hybrid partitioning scheme, both of which helps towards efficiently handling small feature vectors.

We evaluated the performance of the proposed TGN-PNM architecture in terms of throughput (edges/sec) and latency (μs) for two TGNN models: TGN-attn and TGN-sum, across three datasets: Wikipedia, Reddit, and GDELT. The performance of TGN-PNM is compared against five other platforms: a high-end CPU; an NVIDIA A100 GPU; a subarray-level general purpose PIM architecture, Gearbox [33]; a bank-level AI accelerator, Newton [30]; and the FPGA-based TGNN accelerator tFPGA [28]. By combining vault-level parallelism, hybrid partitioning, and efficient compute units, TGN-PNM demonstrates average throughput gains of 26.8x over CPU, 16.7x over GPU, 5.2x over Gearbox, 4.4x over Newton, and 10.4x over tFPGA. In terms of latency,TGN-PNM provides an average improvement of 26.8x over CPU, 17.2x over GPU, 5.4x over Gearbox, 2.9x over Newton, and 3.8x over tFPGA. Despite having similar processing unit counts, TGN-PNM outperforms the other two general purpose PIM architectures (Gearbox and Newton) due to efficient handling of time encoding and elementwise operations and better data reuse.

2 Background and Motivation

2.1 Temporal Graph Neural Network (TGNN)

Evolving graphs can be expressed in two manners: Discrete-time dynamic graphs (DTDG) and continuous-time dynamic graphs (CTDG). In DTDG, the dynamic graph is represented as a sequence of static graph snapshots. However, DTDG is a coarse-grained representation, where the exact event timestamps between subsequent snapshots are lost. This loss of temporal information can lead to comparatively lower accuracy during inference [20]. On the other hand, CTDG are more fine grained and can represent the dynamic graph as an ordered sequence of timestamped events, $G(t) = \{\delta_1, \delta_2, ..., \delta_k\}$, where each event δ_i denotes addition/deletion of a node or an edge. Typically, CTDG is a multigraph, which means that there can be more than one edge between a pair of nodes, pertaining to multiple interaction events between the nodes at different times. In this paper, we focus on the continuous time representation of the temporal graph.

There are a few prior works that proposed neural network models for learning representations over CTDG [19–21, 38–41]. Among these approaches, TGN [19] proposes a generic message-passing-based modular framework that can be tuned to mimic many of these other approaches. Therefore, we use TGN as the software framework for our accelerator. In TGN, each node is composed of raw node features $\boldsymbol{v_i}$ that denote the static properties of the node, node memory or state $\boldsymbol{s_i}(t)$ that captures the history of temporal interactions, and dynamic node embeddings $\boldsymbol{z_i}(t)$ that combine the node memory with the spatial information (e.g., graph topology). Whenever an interaction event occurs between nodes v_i and v_j, a message function produces two messages that are sent to the mailbox of the corresponding nodes:

$$\boldsymbol{m_i}(t) = \{\boldsymbol{s_i}(t^-)||\boldsymbol{s_j}(t^-)||\Phi(\Delta t_i)||\boldsymbol{e_{ij}}\} \quad (1)$$

$$\boldsymbol{m_j}(t) = \{\boldsymbol{s_j}(t^-)||\boldsymbol{s_i}(t^-)||\Phi(\Delta t_j)||\boldsymbol{e_{ij}}\} \quad (2)$$

Here, $||$ denotes the concatenation operation, $\boldsymbol{s}(t^-)$ denotes the node state just before the event, $\Phi(\Delta t)$ is the vector encoding of elapsed time since the last state update of that particular node,

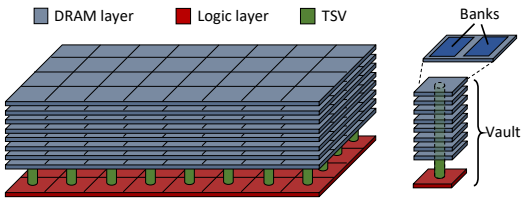


Figure 1: Organization of a Hybrid Memory Cube [44, 45].

and $\boldsymbol{e_{ij}}$ is the edge embedding of that interaction. The time encoding function is usually implemented as $\Phi(\Delta t) = cos(\Delta t \boldsymbol{w} + \boldsymbol{b})$, following prior works such as Time2Vec [42] and TGAT [20]. In batch processing, one node can receive multiple messages within a batch, which is aggregated into a single message per node by a message aggregator function. The message aggregator function can be implemented in various ways. The common implementation simply keeps the most recent message as the aggregated message [19, 28, 41]. The aggregated message is then used to update the node states as follows:

$$\boldsymbol{s_i}(t) = mem(\bar{\boldsymbol{m}}_i(t), \boldsymbol{s_i}(t^-)) \quad (3)$$

$$\boldsymbol{s_j}(t) = mem(\bar{\boldsymbol{m}}_j(t), \boldsymbol{s_j}(t^-)) \quad (4)$$

Here, $\bar{\boldsymbol{m}}(\boldsymbol{t})$ is the aggregated message and $mem()$ is a learnable function for updating the memory, e.g., a recurrent neural network such as LSTM or GRU. Finally, the dynamic embedding is derived by aggregating over the k-hop temporal neighborhood $\mathcal{N}_i^k$:

$$\boldsymbol{z_i}(t) = emb(\boldsymbol{v_i}, \boldsymbol{v_j}, \boldsymbol{s_i}(t), \boldsymbol{s_j}(t), \boldsymbol{e_{ij}}, \Delta t) | \forall j \in \mathcal{N}_i^k \quad (5)$$

Here, $emb()$ is a learnable function and can be realized in several ways. One is to simply use the node's current state, $\boldsymbol{z_i}(t) = \boldsymbol{s_i}(t)$. However, this approach can lead to memory staleness issue if the node's state has not updated in a while [19, 43]. JODIE [21] uses a linear time projection of the node's state to avoid this issue, $\boldsymbol{z_i}(t) = (1+\Delta t \boldsymbol{w}) \circ \boldsymbol{s_i}(t)$. TGAT [20] and TGN [37] uses a multi-head graph attention mechanism over the temporal neighborhood as the embedding function. It is also viable to simply use the average of neighbor states as the embedding while retaining relatively high accuracy, as shown by TGN-sum [37].

Table 1 summarizes the arithmetic intensity of these functions for different datasets, model-architectures, and batch sizes. Here, the arithmetic complexity of the same kernels (e.g., embedding function) can differ vastly among different models. Furthermore, even within the same model and same function, arithmetic complexity can change widely depending on the data reuse opportunity. Therefore, a TGNN accelerator needs to be flexible to accommodate these scenarios efficiently.

2.2 3D-Stacked Memory

In 3D-stacked memory, multiple DRAM dies (e.g., 4 or 8) are stacked vertically and may contain an optional buffer or logic die. This combination of a logic layer interfacing with the DRAM stack is promising in terms of potential PIM architectures. Notably, two distinct variations of 3D-stacked memory technology have emerged [46, 47]: Hybrid Memory Cube (HMC) [44] and High Bandwidth Memory (HBM) [48]. In HBM, the external IO interface of the memory stack is implemented through DDR physical channels. The HBM memory stacks are typically tightly integrated with the host

Table 1: Arithmetic intensity (flops/byte) of various TGNN models.

	Message aggregator function					Memory updater function					Embedding function				
		Batch size = 1		Batch size = 32			Batch size = 1		Batch size = 32			Batch size = 1		Batch size = 32	
	Type	Wiki	GDELT	Wiki	GDELT	Type	Wiki	GDELT	Wiki	GDELT	Type	Wiki	GDELT	Wiki	GDELT
TGN-attn [37]	id	0.07	0.05	0.10	0.06	GRU	0.50	0.50	15.04	15.63	attn (1 layer)	3.11	2.62	83.03	69.68
TGN-sum [37]	id	0.07	0.05	0.10	0.06	GRU	0.50	0.50	15.04	15.63	sum	0.71	0.48	0.77	0.49
TGAT [20]	-	-	-	-	-	-	-	-	-	-	attn (2 layers)	28.09	26.60	116.97	151.61
JODIE [21]	id	0.07	0.05	0.10	0.06	RNN	0.05	0.05	13.47	14.82	time projection	0.50	0.50	0.97	0.97
tFPGA [28]	id	0.07	0.05	0.10	0.06	GRU	0.50	0.50	15.04	15.56	simple attn	1.03	0.74	1.17	0.78

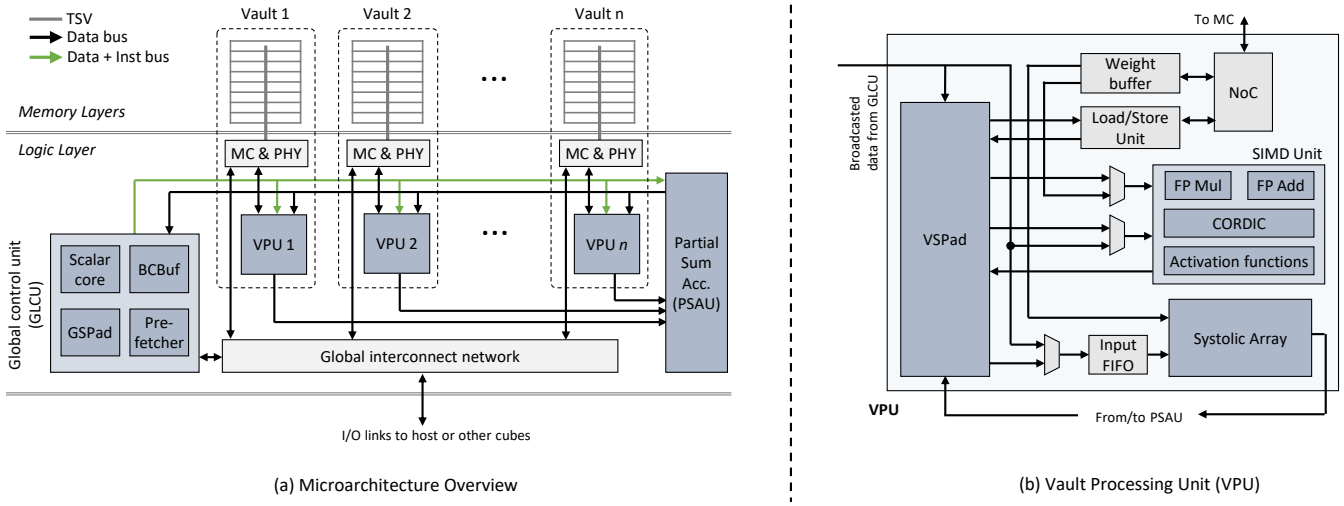


Figure 2: TGN-PNM Architecture.

die using a silicon interposer, with the memory controllers residing on the host die. In contrast, the HMC specification places the memory controllers within a logic die part of the HMC memory stack. Figure 1 shows the organization of HMC. In HMC, memory layers are partitioned vertically to form mostly independent vaults. Each memory layer in each vault contains multiple memory banks. All banks within a vault are connected using through-silicon vias (TSVs) that act as the shared bus to carry DRAM address and command signals to these banks. Each vault also contains a memory controller in its logic layer. DRAM transistors in the memory layers have traditionally been designed for low cost and leakage. The logic die, on the other hand, uses high-performance transistors [49]. HMC 2.1 specification supports up to 32 vaults. Four external SerDes IO links are connected to these memory controllers using a crossbar switch at the logic layer. Note that this crossbar switch only connects the IO links to the vaults, but the vaults themselves do not communicate directly (e.g., send memory access requests) with each other. These IO links can be used to connect to the host or can be used to connect to other HMC cubes to increase capacity. Even though HMC is no longer under development, for our specific application, using an HMC organization offers two advantages compared to HBM: i) Placing the VPUs in the logic layer of HMC results in lower energy for data movement compared to placing them on the host die, as the data do not have to travel through the silicon interposer, and ii) HMC is optimized for random accesses (by using short row buffers, closed page policy, consecutive address to different banks), while HBM is optimized for sequential accesses (wider row buffers, open page policy, consecutive address to the same row). Consequently, HMC is better aligned with the irregular access patterns of graph workloads. In this paper, we will mostly refer to memory specifications and terminologies pertaining to HMC, but the concept can be easily extended to HBM memory by placing the VPUs after the memory controller on the host die, similar to the approach proposed in [50].

3 TGN-PNM Microarchitecture

Figure 2(a) presents the overall architecture of our proposed approach. We primarily add three components in addition to the interconnect network and the memory controllers that are already present in the logic layer of an HMC-like memory[1]: i) Vault-level Processing Units (VPUs) at the logic layer of every vault, ii) one Global Control Unit (GLCU), and iii) one Partial-Sum Accumulation Unit (PSAU). Next, we will describe the contents and connectivity of each of these components.

3.1 Vault-level Processing Unit (VPU)

The VPUs are the primary compute units in our design. Figure 2(b) shows the microarchitecture of a VPU. Each VPU contains an $S_u \times S_u$ systolic array of fused-multiply-accumulators, an S_u-lane SIMD unit, a local scratchpad memory (VSPad), and a weight

[1]We used one of the IO links of the global interconnect network to connect to the GLCU. The interconnect network and the memory controllers are not modified in any other way. In other words, we do not need to design a custom interconnect or memory controllers for our proposed architecture.

buffer. All operations within a VPU is performed in S_u granularity, including access to the VSPad and all the buffers. Selecting S_u poses performance-area trade-off. In our implementation, we used $S_u = 16$, which is the maximum that we could synthesize without going over the available die area (a more detailed area evaluation is given in Section 5.6). For the systolic array, instead of using a single $S_u \times S_u$ array, it is realized as a group of four arrays, each with the dimension of $S_u \times S_a$, where $S_a = S_u/4$. These arrays can be configured to work cooperatively or independently: as a single $S_u \times S_u$ array; two $S_u \times 2S_a$ arrays, or four $S_u \times S_a$ arrays. The concept of such segmented systolic array is borrowed from the work of Yan et al. [13] and is leveraged to efficiently support dense matrix multiplication with small feature dimensions. More on this topic and specific use cases are discussed in Section 4.1. The output of the systolic array is sent to the PSAU for accumulation.

The SIMD unit contains S_u numbers of pipelined multiplier and adders, one 16-stage CORDIC functional unit for time encoding, and one activation function unit. The activation functions are realized using a lookup table. The CORDIC unit consumes S_u operands at a time, but process them sequentially (i.e., initiation interval of S_u cycles). As both the systolic array and the SIMD unit works on S_u data elements at a time, the access granularity to the VSPad is fixed to $size(elem) * S_u$. The operands for these units can come from the VSPad, weight buffer, result bus, or the global broadcast buffer. All the VPUs operate in a lockstep, controlled by the GLCU. VPUs have very limited outside visibility - they can only load/store data from their local vault (by using the address broadcasted by the GLCU) and can send data out only to the PSAU for reduction. VPUs cannot communicate directly with each other.

3.2 Global Control Unit (GLCU)

The primary objective of the GLCU is to drive the VPUs by broadcasting instructions. It also contains a scalar core, a scratchpad memory (GSPad, local to the GLCU), a data broadcast buffer (BCBuf), and a prefetcher. The scalar core can be used to do operations that are not supported by the VPUs or for complex reductions (e.g., softmax in graph-attention). It has access to the full memory stack (via the global interconnect network), the GSPad, and the BCBuf. The function of the broadcast buffer BCBuf is to provide a common operand to all the VPUs. In our mapping scheme (discussed in Section 4.1), the BCBuf is used for time encoding and for handling low dimension GEMM. The BCBuf width is S_u elements. The GLCU also contains a prefetcher, that can fill the GSPad or VSPad by fetching a node's associated data. Using the broadcast mechanism, we can issue memory read/write requests to every vault in every clock cycle. With this approach, we can easily saturate the memory controller queue without resorting to maintaining multiple threads, as done in a few prior PIM-approaches [31, 51].

3.3 Partial-Sum Accumulation Unit (PSAU)

As discussed before, partitioning across the feature dimension entails that we only need inter-vault communication when performing dot-product reductions during GEMV and GEMM operations. This reduction is handled by the PSAU. It contains a pipelined parallel adder tree to reduce the results from all the VPUs' systolic array. A few entry[2] accumulation register stores the partial results. The final output is broadcast to all the VPUs and is stored to either one of the VSPads, or in the GSPad, depending on the destination address. An alternative to running the vaults in lockstep and doing partial sum across vaults is to make the VPUs completely independent. However, it requires either having a copy the model parameters in each vault, or high amount of inter-vault traffic.

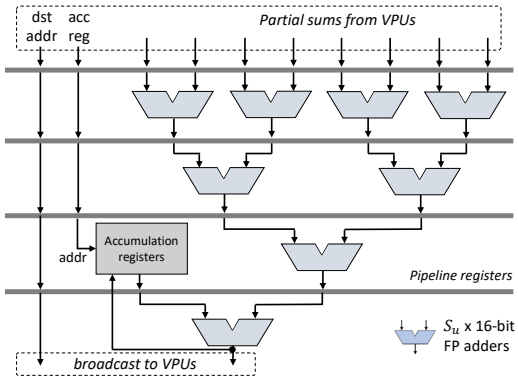


Figure 3: Partial-sum accumulation unit (PSAU). Figure drawn assuming a total of eight VPUs.

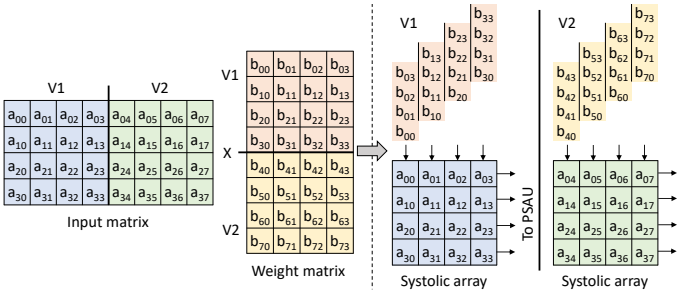
To summarize, the VPUs can get their operands from: i) their local scratchpad memory (VSPad), ii) their local DRAM stack (must load to the VSPad first), or iii) the broadcast buffer (BCBuf). After doing the intended computation, VPUs can send the result to the: i) local scratchpad (VSPad) as the intermediate operands for later stages, ii) local DRAM stack store, or iii) PSAU for reduction. The result of the reduction of the PSAU can be sent back to one of the VPU's scratchpad, or to the GLCU's scratchpad for additional processing.

4 Mapping TGNN Frameworks on TGN-PNM

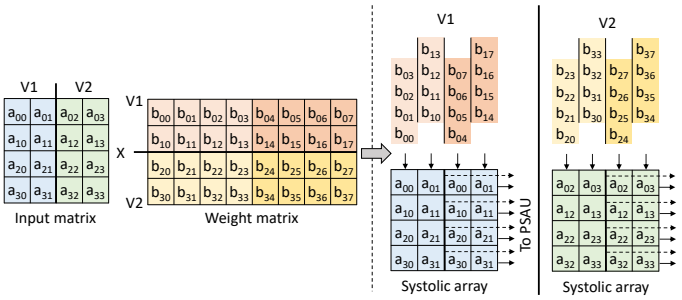
4.1 Mapping of common operations

4.1.1 Dense matrix multiplication. We always process dense matrix multiplication using the systolic array. Figure 4 demonstrates our mapping scheme. Depending on the current input location and input's feature dimension, we map the operation into one of the three possible scenarios: i) In the first scenario, input is currently stored in VSPad (i.e., using feature-based partitioning) and the number of input-features within the vault is $\geq S_u$. In this scenario, all lanes of the systolic array will be occupied without resorting to any special strategy. We use input-stationary dataflow on the systolic array for this case. If the input matrix is $(m \times n)$ and the weight matrix is $(n \times k)$, then m is usually the batch size (or batch size * the number of neighbors when performing neighbor aggregation) or the partitioned output of an intermediate matrix. Dimension n is partitioned across vaults, making the effective dimension per vault $n/(\#\ of\ vaults)$. Therefore, these two dimensions are relatively small when compared to k. In this case, using input-stationary dataflow enables streaming the weights along k, thereby providing higher reuse of the inputs. Furthermore, we use two sets of registers

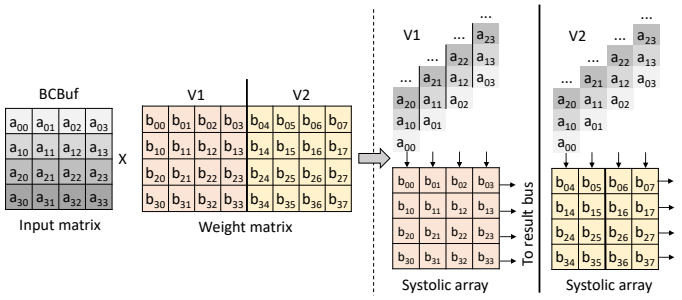
[2]We used 4096 entry accumulation register in our experiments.



(a) Scenario 1: Input stored in VSPad, with per-vault feature-dimension $\geq S_u$



(b) Scenario 2: Input stored in VSPad, with per-vault feature-dimension $< S_u$



(c) Scenario 3: Input stored in the broadcast buffer (BCBuf).

Figure 4: Mapping of dense matrix multiplication on TGN-PNM.

to enable loading the next set of inputs and biases while processing the current set. Outputs of the systolic arrays are sent to the PSAU for reduction, after which the final result can be stored in the VSPad or BCBuf, depending on the output dimension. ii) In the second scenario, input is stored in the VSPad; however, the per-vault feature dimension is smaller than S_u. In this case, using the former scheme will leave some of the systolic array lanes unused. To avoid this, we use a smaller segment of the systolic array[3]. To fully utilize the available lanes, the inputs are duplicated P times, and the weight matrix is also folded P times. This approach fully occupies all the MAC units, maximizing the available compute bandwidth. The maximum value of P is the number of segments. While increasing the number of segments means that we can support narrower input

[3]The idea of a segmented systolic array is borrowed from [13], where a systolic module is formed by combining several narrower arrays. Refer to Section 3.1) for more detail.

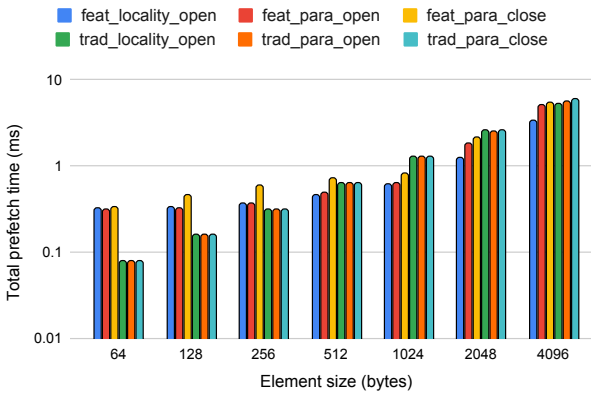


Figure 5: Total operand load time for the Wikipedia dataset with different memory address mapping and page policy configurations. *Lower is better.*

matrices efficiently, it also means that the output of the systolic array produces a higher number of outputs per clock (i.e., $P.S_u$), and the PSAU had to be widened accordingly. In our implementation, we support four segments. Any higher value resulted in unacceptable area overhead for PSAU[4]. iii) In this scenario, the input is stored in the BCBuf and broadcast to all VPUs, where the corresponding results are calculated and stored back on the same vault. As we are broadcasting the inputs, we use weight stationary dataflow in this scenario.

4.1.2 Time encoding. As mentioned earlier in Section 2, time encoding is usually implemented as $\Phi(\Delta t) = cos(\Delta t \boldsymbol{w_t} + \boldsymbol{b_t})$. Depending on the task, Δt denotes either the time elapsed since the last update of a node or the incident time of an edge. $\boldsymbol{w_t}$ and $\boldsymbol{b_t}$ are the learnable weights and biases. Each VPU contains a subset of these weights and biases in its weight buffer. Timestamps are stored in the memory in a traditional fashion (discussed in Section 4.2) and fetched in the GSPad for processing. The scalar core computes the Δt of a batch and then puts the results in the BCBuf. Then, the Δt of S_u elements are broadcasted to the VPUs. The SIMD units on the VPUs will multiply the broadcasted value with the stored weights, add bias, and then send it to the CORDIC unit to calculate the resultant time encoding.

Elementwise operations are performed using the SIMD unit. Note that using the broadcasting mechanism for elementwise operations is not efficient. This is because the same value gets broadcast to every VPU, but we do not have any reuse opportunity in elementwise operations. Therefore, we store both operands in the VSPad for elementwise operations. One potential shortcoming of our approach is that, if the dimension of the operands is small, then the SIMD lanes will be underutilized.

4.2 Graph Storage Format

Static GNN accelerators typically use compressed formats such as CSR to improve locality characteristics. However, such formats are unsuitable for dynamic graphs, as any change in the graph

[4]Eight or higher number of segments resulted the PSAU occupying more than $4mm^2$ of logic-die area when synthesized on a 14nm node.

topology (e.g., addition of nodes or edges) requires reconstructing the whole graph from scratch [52]. To support this dynamic behavior efficiently, we store the graph in an adjacency list format and limit the maximum number of neighbors a node can have[5]. This constraint is fairly common in existing TGNN frameworks and accelerators [19, 28]. To maintain a fixed number of neighbors, the adjacency list of each node is realized using a circular queue. More specifically, we stored the following metadata for each node: $\{head, tail, ts, eid_0, eid_1, ..., eid_K, ets_0, ets_1, ..., ets_K\}$. Here, *head* and *tail* are used to determine the location for new edge insertion in the circular queue. *ts* is the timestamp of the last update. eid_i and ets_i are i-th edge's neighbor node id and incident timestamp, respectively. Note that partitioning along the feature dimension is impossible for these graph metadata, as these are mostly scalar values and are also shared by all the vaults. It is possible to partition these graph metadata along the node dimension so that each vault contains all metadata associated with a subset of the nodes. However, real-world graphs usually demonstrate power-law degree distribution, indicating that some of the vaults will get a disproportionately high number of accesses if we partition along the node dimension, causing severe load imbalance. Therefore, these metadata are stored in a traditional fashion (i.e., consecutive memory address is mapped to either in the same DRAM row to maximize row buffer locality or in different banks to maximize parallelism) and fetched to the global scratchpad (GSPad) instead of the vault's local scratchpad (VSPad).

On the other hand, vector data, such as weights and the states of nodes and edges, can be partitioned along their feature dimension so that each vault contains a subset of features for all nodes. However, if the feature dimension is small, it may be beneficial to use traditional memory address mapping for these vector data as well. Figure 5 shows the operand prefetch time with different partitioning and address mapping schemes. We used an 8 GB HMC 2.1 memory stack for this experiment with 32 vaults, 32B DRAM column width, and 256 B wide row buffers. Here, the x-axis shows the data size associated with each node. We tried three paging-policy/address-mapping-scheme combinations: i) optimized for row buffer locality, where consecutive addresses are mapped to the same row (*x_locality_open*), ii) optimized for parallelism, where consecutive addresses are mapped to different banks, keeping the row buffer open (*x_para_open*), and iii) optimized for parallelism with closed-page policy (*x_para_closed*). Figure 5 suggests that traditional mapping performs best for per-node data size $\leq 256B$. For larger data, the feature-based scheme $feat\_locality\_open$ performs best. One interesting observation is that feature-based partitioning performs better in some cases, even when each vault's portion of the feature may not fully occupy a DRAM column width (i.e., for an element size of 512B, translating to a per-vault element size of 16B). One reason is that in the case of feature-based partitioning, the memory requests in every vault are perfectly balanced. Another factor impacting the traditional scheme was the additional latency of traversing the global interconnect. Therefore, we propose a *hybrid* partitioning where the data is stored using a traditional scheme if the associated data size $\leq 256B$ (feature-dimension ≤ 128 with FP16) and using a feature-based scheme otherwise. An additional consideration is the intended operation on the data. If it is an elementwise operation, then using the traditional scheme is not viable because broadcasting is not optimal for elementwise operations. In this case, the data is stored using the feature-based scheme.

[5]One reason that limiting the number of neighbors does not hamper the model accuracy by a large margin is that the impact of temporal interactions of the discarded neighbors is already captured and summarized in the node's memory.

5 Evaluation

In this section, we compare the performance of our method against five other architectures: (i) a high-end CPU server as the baseline, (ii) NVIDIA A100 tensor core GPU, (iii) subarray-level general purpose PIM architecture Gearbox [33], (iv) bank-level PIM-based machine learning accelerator Newton [30], and (v) an FPGA-based TGNN accelerator (we refer to this approach as *tFPGA* in our paper) [28]. A summary of the configuration of these platforms is provided in Table 3. Due to the lack of TGNN specific accelerators except for tFPGA, we choose to compare against semi-general purpose PIM accelerators Gearbox and Newton. As one of these is subarray-level, and the other one is bank-level, it highlights the trade-offs of the proposed vault-level accelerator.

5.1 Methodology

We evaluated our approach by mapping two variants of the TGNN model architecture proposed in [19]: TGN-attn and TGN-sum. Both of these models use GRU as the memory updater function. TGN-attn uses a single layer multi-head graph-attention mechanism for neighbor aggregation, while TGN-sum uses simple average of the neighbor states. As a result, TGN-attn has both memory intensive (collecting neighbor state data) and compute intensive (creating key, value, and query matrices for calculating self-attention weights) stages. On the other hand, TGN-sum's neighbor aggregation is memory-bound (refer to Section 2.1 for more details). We set the maximum number of neighbors to 10 and batch size to 200, following the original model implementation in [19]. However, as discussed later in Section 5.3, Gearbox and Newton only supports GEMV operation and performs GEMM by repeating GEMV on every input columns. We set batch size to one on these two approaches as they do not benefit from batching.

We implemented our approach on an HMC 2.1-like memory stack with 32 vaults and 16 banks per vault. The TSV width is 64-bits, clocked at 2 GHz, attaining per-vault memory bandwidth of 16 GB/s (total 512 GB/s for a single stack). The logic layer is clocked at 1.2 GHz. S_u is set to 16 (i.e., 16×16 systolic array and 16-lane SIMD units in each VPU). In our implementation, we use 16-bit brain floating-point (BF16) format for all the operands. Integer arithmetic is not supported. BF16 has the same dynamic range as IEEE 754 32-bit FP numbers (both uses eight bits for storing exponent), but the number of bits to store fraction is reduced from 23 to 7. Compared to IEEE 16-bit FP, BF16 format provides ~16% better energy/op while consuming ~15% less area [31]. To further reduce the area required by the FP units, we dropped the support for denormalization. Also, overflow is rounded to positive infinity and underflow is rounded to negative infinity. Furthermore, rounding is done by truncation to avoid multiple normalization iterations. The loss of precision caused by these changes are tolerable because machine learning applications are usually not very sensitive to the precision of floating point operations.

The performance measures are collected by building cycle-accurate simulation models for the VPUs and the GLPU and then feeding the resulting memory request trace to a modified version of DRAMSim3 [53]. Area of processing elements and control logic are derived by synthesizing RTL models on SAED 14 nm node using Synopsys Design Compiler. Area of the buffers and register files are modeled using CACTI-3DD [54] on a 32 nm node and then scaled to 14 nm. Memory controller and interconnects are modeled using McPAT [55].

As the performance metrics, we measured batch processing latency and throughput. Batch processing latency is defined as the elapsed time between receiving a batch to process and writing back the updated node embeddings to memory. For throughput measurement, we use the number of processed events per unit time. The objective is to maximize the throughput and minimize the latency for TGNN inference task.

5.2 Datasets

We conduct the experiments on three real-world datasets: Wikipedia [21, 56], Reddit [21, 56], and GDELT [38]. Details of these datasets are given in Table 2. Wikipedia and Reddit are bipartite graphs consisting of user edits on wikipedia pages and posts made by users on subreddits. GDELT is a reduced version of a temporal knowledge graph dataset introduced in [41]. Among these datasets, Wikipedia and Reddit do not have any raw node features. The time encoding and the nodes' memory dimensions are configurable hyper parameters. We used 100 as the time encoder dimension. On the Wikipedia and Reddit datasets, we used 100 as the memory dimension following the prior works [19–21]. On GDELT, the memory dimension is set equal to the raw node feature dimension of 483.

Table 2: Characteristics of the used datasets. Time encoder dimension is fixed to 100 for all datasets.

Dataset	$\|V\|$	$\|E\|$	$\|v_i\|$	$\|e_{ij}\|$	$\|s_i(t)\|$	Max weight dim	Total weight (MB)
Wikipedia	9K	157K	-	100	100	516 x 944	5.95
Reddit	11K	672K	-	100	100	516 x 944	5.95
GDELT	9K	1913K	413	186	413	1242 x 1680	19.4

Table 3: Configuration of the evaluated architectures.

Architecture	Parameters
TGN-PNM	8GB HMC 2.1, 32 vaults, 256B row buffers, TSV BW 16GB/s, 16-lane bfloat16 SIMD and 16x16 systolic array per vault, 32KB VPU SPad, 1MB global SPad, logic layer freq 1.2GHz.
CPU	Server with two AMD EPYC 7742 64-core @ 2.25GHz (total 256 hardware threads), 1024GB DDR4, 8 memory channels, peak memory BW 409.6GB/s.
GPU	NVIDIA A100 SXM, 80GB HBM2e, peak memory BW 2039GB/s, peak compute rate for 16-bit FP is 624 TFLOPs. Host is the same as the baseline CPU server.
Gearbox	8GB HMC 2.1, 32 vaults, 256 subarrays per vault, 256B row buffers, 49ns row activation time, 8192 subarray-level ALUs @ 164MHz, TSV BW 16GB/s per vault, logic layer cores ARM Cortex-A35 @ 600MHz, 128KB scratchpad memory shared by the cores.
Newton	8GB HBM2e-like, 16 pseudo channels, 16 banks per channel, 1024B row buffers, 49ns row activation time, 16 MAC units per bank.
tFPGA	Xilinx U200 FPGA @ 250MHz, 77GB/s DDR4 memory, two CUs, four 8x8 systolic arrays and one 16-lane multiply-add tree per CU.

5.3 Mapping on evaluated architectures

As mentioned earlier, we evaluated our approach against five other architectures. This section goes into the implementation details and mapping schemes on these architectures.

5.3.1 CPU and GPU. For performance evaluation on the CPU and GPU platforms, we profiled the open-source implementation of the TGN model architecture [57], which is written using the PyTorch Geometric library. However, on the GPU platform, we have used FP16 instead of BF16 because the GRU cell of PyTorch does not support BF16 on CUDA as of version 2.0.1. To provide a fair comparison, we have not included the host-GPU data transfer times in the measurements. The reported results are an average of five runs.

5.3.2 Gearbox [33]. Gearbox places scalar processing units at the subarray-level of a 3D-stacked memory. These processing units support word-level arithmetic and logic operations, as well as control flow instructions. Three latched row buffers (called Walkers) in each subarray acts as the source/destination registers. Gearbox also contains an ARM core in each vault's logic layer, primarily for reduction operations. Although the processing units of Gearbox can operate only on a single word per cycle, Gearbox can attain high performance by leveraging massive subarray-level parallelism.

The Gearbox authors implemented GEMV by mapping each row of the matrix to a subarray and then broadcasting the input vector elements one-by-one to all subarrays. The input vector itself is stored in a shared buffer at the logic layer. The subarray-level processing units (APLUs) perform the MAC operation. This approach does not require partial sum accumulation across subarrays. However, given the large number of subarrays (8192 subarrays in an 8GB HMC 2.1 stack), this approach only makes sense for very large matrices. For example, the authors used a matrix of dimension 25600x19200 for evaluating performance. The matrix size is often times much smaller in practical TGNN datasets (refer to Table 2) and causes extreme under-utilization of the processing units. In our evaluation, we used an alternative scheme, wherea every [*subarrays_in_vault X elems_in_dram_row*] slice of the matrix will be mapped to a vault. A full matrix is thus potentially distributed across multiple vaults. This approach can process subset of columns in parallel while maintaining DRAM row buffer locality. Furthermore, all the vaults are cooperatively processing a single event at a time, thus minimizing the processing latency of events. Additionally, this scheme does not require duplication of matrices and also maintains proper event ordering. One concern with this approach is that it requires accumulating partial sums across vaults. Our evaluation shows that the inter-vault partial sum accumulation adds moderate overhead (25% - 37%). But even with this overhead, the latency improvement over the original Gearbox implementation is substantial.

As for mapping the rest of the operations, the GEMM kernel is implemented by repeating GEMV for each column of the input matrix. Although this approach appears inefficient as it does not utilize any form of cache blocking to leverage the reuse opportunities, it will not negatively impact the attainable throughput. This is because the machine balance of Gearbox is extremely low (~0.03 flop/byte with 164MHz ALUs and 49ns row activation time). As a result, Gearbox is fundamentally compute bound even for GEMV kernels, and by

extension on GEMM kernels. Tiling for increasing reuse will not provide any improvement in throughput unless we increase the number of processing units. Therefore, GEMM is implemented simply by repeating GEMV. As a side note, TGNN framework does not require GEMM operations unless we are batching the queries. Since GEMM in Gearbox does not provide any advantages over multiple individual GEMV operations, query batching is in fact undesirable as it increases the query latency without providing any throughput benefit.

The time encoding and most of the activations used by LSTM/-GRU and GAT requires calculating transcendental functions. As Gearbox ALPU does not support these functions, these operations are handled by the cores at the logic layer. ReLU activations are mapped to ALPUs as they are simple comparisons. Elementwise operations, such as Hadamard product, matrix-matrix/vector-vector additions, are cooperatively handled by the cores in the logic layer as well.

Performance of Gearbox is estimated by leveraging the simulation framework of Pulley [58]. Here, we use analytical model for regular operations (e.g., GEMV) and simulation otherwise (e.g., feature aggregation of neighbors). We assumed that all operands, except for the weight matrices, are loaded into the shared scratchpad memory before processing. We also assume that the update/predictions events are stored in a queue in the shared scratchpad memory, from where it gets scheduled in-order.

5.3.3 Newton [30]. Newton is a near memory accelerator proposed by SK Hynix and primarily targets acceleration of the GEMV operations of machine learning workloads. Newton puts several MAC units in a SIMD fashion (number of MAC units matched with DRAM columns) in every bank of an HMB2E-like memory stack. The weight matrix is stored in a chunk-interleaved manner, where the first matrix row's first chunk is followed by the second matrix row's first chunk, and so on. The input vector is stored in a global buffer and is broadcast to the bank-level compute units a single chunk at a time, where the result gets reduced by a parallel adder tree. The width of the chunk is made the same as the DRAM row width to take advantage of the spatial locality. The first chunk of all the matrix rows are processed first, following by the second chunk of all matrix rows, etc. This approach provides maximum reuse of the input vectors. Similarly to Gearbox, GEMM operations are performed with repeated GEMV. Elementwise operations, activations, and time encodings are performed by the host. Performance estimation is derived using the performance model provided by the authors in the original paper [30].

5.3.4 tFPGA [28]. This approach proposes a model-architecture co-design for accelerating the TGN framework [37] on FPGA. The design consists of multiple independent compute units. Each compute units supports memory state update using GRU and embedding using a simplified version of graph-attention mechanism. The GRU is implemented using three 8×8 systolic arrays[6] for efficient GEMM operations, corresponding to the update, reset, and memory gates of GRU. For the embedding, unlike the multi-head attention mechanism used by TGN that involves computing the key, value, and query matrices, tFPGA uses a simplified approach that only considers the temporal separation of the neighbors for calculating the attention weights, thereby eliminating a major portion of the computations. This embedding function is realized using a 16-lane multiply-adder tree for feature aggregation and one 8×8 systolic array for feature transformation. For neighbor sampling, tFPGA samples a fixed number of most recent neighbors, similar to our approach. Time encoding is realized by a coarse grained loop-up table. Further optimizations are done by pipelining all the stages and using a dedicated edge prefetcher. The HLS code of this approach is open-sourced [59]. However, we faced compilation issues when trying to generate the FPGA bitstream using the published code, and therefore opted to use the performance model provided by the authors in their paper.

[6]Authors have evaluated tFPGA on two FPGAs: Xilinx Alveo U200 and Xilinx ZCU104, with different number of compute elements due to FPGA resource constraint. We use the configuration of the more powerful Xilinx Alveo U200 for evaluation.

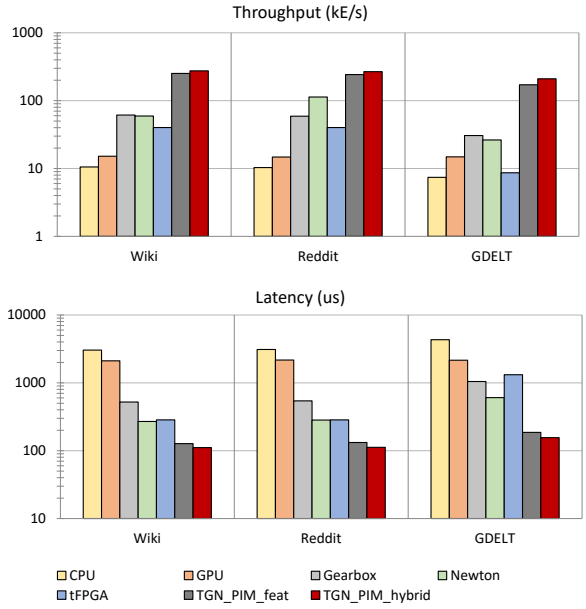


Figure 6: Throughput and batch processing latency for TGN-attn.

5.4 Throughput and latency results

Figure 6 presents the throughput and latency results for the TGN-attn model and Figure 7 presents the results for the TGN-sum model. Here, TGN_PIM_feat uses only the feature-based partitioning scheme, while TGN_PIM_hybrid uses the hybrid partitioning scheme. In both models, TGN_PIM_hybrid provides the best throughput and latency across the benchmarks. Table 4 summarizes the average throughput gain and latency reduction observed by the TGN_PIM_hybrid across the datasets.

Our results show that CPU and GPU perform worst, both in terms of latency and throughput. High latency in the case of GPU is expected as GPU architecture is optimized primarily for throughput and not latency. We attribute the low throughput of GPU for this particular workload to the small batch size. Increasing the batch size

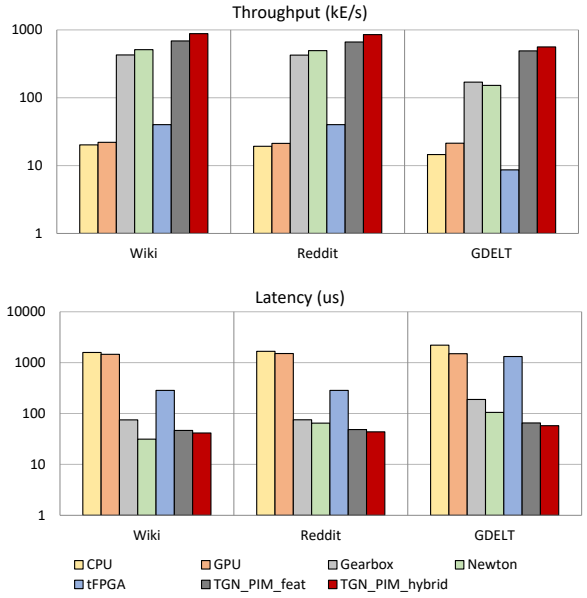


Figure 7: Throughput and batch processing latency for TGN-sum.

Table 4: Average throughput gain and latency reduction of TGN-PNM-hybrid approach across the datasets.

	Avg. througput gain		Avg. latency reduction	
	TGN-attn	TGN-sum	TGN-attn	TGN-sum
CPU	26.8	42.1	27.6	38.2
GPU	16.7	34.8	17.2	31.6
Gearbox	5.2	2.4	5.4	2.2
Newton	4.4	2.2	2.9	1.3
tFPGA	10.3	31.1	3.8	10.1
TGN_PIM_feat	1.14	1.23	1.17	1.12

from 200 to 1000 increased the throughput of GPU by 3.7x on average while having a moderate impact on latency, which is increased by 1.4x. However, increasing batch size to improve throughput may not be feasible in practical scenarios, as user-facing interactive applications tend to have strict latency constraints. Furthermore, increasing the batch size also means that the graph state will update less frequently, and therefore, embedding will be performed using stale data and can negatively impact accuracy. Besides, the maximum batch size is limited by the capacity of the on-chip buffers for tFPGA and our approach.

TGN_PNM_hybrid provides a substantial performance gain over the subarray-level and bank-level PIM architectures. Note that all these three PIM architectures have almost the same number of MAC units: TGN-PNM has $(16 * 16 + 16) * 32 = 8704$ MACs, Gearbox has 8192 ALUs, and Newton also has 8192 MACs. Despite having similar number of MAC units, for the TGN-attn model, our approach has 5.2x higher throughput than subarray-level Gearbox and 4.4x higher throughput than bank-level Newton. There are a

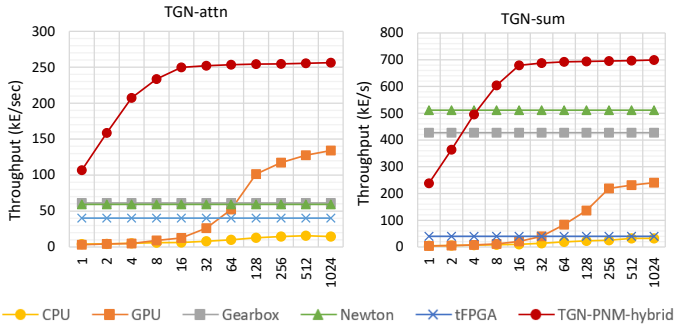


Figure 8: Throughput with varying batch sizes on the Wikipedia dataset.

few key advantages of our approach that enable this throughput gain: i) VPUs of TGN-PNM run at a much higher clock frequency than both Gearbox and Newton's compute units. This is becuase DRAM transistors in the memory layers are designed for low cost and leakage. The logic die uses high-performance transistors [49]. Although, note that we have used 164 MHz for Gearbox, which the authors reported for a 32-bit ALU, not 16-bit. Therefore, the attainable frequency of Gearbox could be higher for 16-bit FP. ii) Another advantage of TGN-PNM over the other two approaches is handling time encoding. In case of Gearbox and Newton, their processing elements have to be extremely simple to meet the strict area/power overhead budget of only about 20% [30] or otherwise lose capacity [31]. Thus, those two approaches cannot accommodate units for transcendental functions needed by time encoding. As a result, time encoding needs to be performed by the host in case of Newton and at the logic layer in case of Gearbox. The time encoding itself is not a bottleneck, however, as time encoding sits at the intermediate stages, data have to move frequently in/out of their subarrays/banks. iii) As Newton only supports broadcast (one of the operands in their bank-level SIMD unit always comes from the broadcast buffer), it cannot handle elementwise operations efficiently. In case of Gearbox, elementwise operations may require re-layout of the data in the intermediate steps. iv) In case of Newton, underutilization can occur if the matrix dimension is less than the total number of banks [30], which is often the case for the node states. Despite these shortcomings, Gearbox and Newton perform fairly closely to our approach for the more memory-intensive workload of TGN-sum, only trailing by 2.4x and 2.2x, respectively. Finally, our experiment shows that using hybrid storage scheme (i.e., feature-based partitioning for large matrix/vectors and traditional for others) improves the throughput slightly by 12-23% over the feature-based only approach of TGN_PNM_feat.

5.5 Impact of batch size

Figure 8 shows the throughput results with varying batch sizes on the Wikipedia dataset. Other datasets are not shown to avoid clutter, as they demonstrate a similar pattern. As mentioned earlier, Gearbox and Newton are optimized for matrix-vector multiplication and does not benefit from reuse. As a result, their throughput remains same irrespective of the batch size. On the other hand, TGN-PNM

Component	Count	Per unit area (mm^2)	Total area (mm^2)
Scalar core (ARM Cortex A-35)	1	0.68	0.68
Partial Sum Acc. Unit	1	2.25	2.25
Global NoC	1	1.43	1.43
GSPad	1	1.03	1.03
Memory controller and DDR PHY	32	0.18	5.90
Systolic array	32	0.25	8.08
SIMD: FP add & mul	32	0.02	0.51
SIMD: CORDIC	32	0.03	0.96
SIMD: Activation	32	0.15	4.68
VSPad	32	0.32	10.31
Total:			**35.83**

Table 5: Area estimation of TGN-PNM.

benefits from increasing batch size up to 16 (= S_u). At this point TGN-PNM leverages maximum reuse within its systolic arrays. Increasing the batch size further does not improve throughput. Similarly, CPU and GPU gain throughput with batch size increment up to a certain point, after which the benefit flattens. This plateau occurs at around batch size of 256 for both. With large batches, GPU outperforms other PIM-based approaches on compute intensive TGN-attn model. However, using large batch can negatively impact the batch processing latency and the model accuracy.

5.6 Area estimation

The area of processing elements and control logic are derived by synthesizing RTL models on the SAED 14 nm node using the Synopsys Design Compiler. The area of the SRAM buffers and scratchpad memories are modeled using CACTI-3DD [54] on a 32 nm node and then scaled to 14 nm. The memory controller and interconnect areas are modeled using McPAT [55]. The resulting area estimation for the major components are given in Table 5. Note that although the total amount of memory is the same for GSPad and VSPad (32kB * 32 = 1MB), VSPad requires a much larger area as it has to accommodate many read/write ports. The total estimated area of these components is 35.83mm^2, which is 53% of the total logic-die area of 68mm^2 of an HMC stack [49]. This leaves around 32mm^2 for the components that we haven't accounted for, such as I/O circuits, memory built-in self-test (MBIST), and features to support testing and debugging. In comparison, Gearbox requires an estimated die area of 80.64mm^2 (area of Newton is unavailable).

6 Related Work

With the emergence of machine learning workloads, a lot of hardware accelerators have been proposed by researchers targeting either the compute-intensive [60–64] or memory-intensive [30, 31] kernels of neural networks. Unfortunately, these approaches are not specifically designed for irregular access patterns exhibited by the neighborhood aggregation of graph neural networks. On the other hand, there are many hardware accelerators tailored for graph analytics workloads [29, 65–69]. However, these approaches cannot handle the compute-intensive portion of the temporal GNN workloads efficiently.

A few works cater to the unique hybrid nature of the GNN workloads. HyGCN [13] proposed an ASIC accelerator for static GCN, where the aggregation is scheduled on a series of SIMD units and node embeddings are processed by a collection of configurable systolic arrays. AWB-GCN [14] improved upon HyGCN by adding workload balancing mechanism for power-law graphs by distribution smoothing and row remapping. GCoD [15] proposed an algorithm/hardware co-design with separate micro-architectures for dense and sparse matrix. StreamGCN [17] targets streaming processing of many small graphs. FlowGNN [16] introduced support for edge embeddings. Recently, a PIM-based GNN accelerator has been proposed that accelerates the memory-bound kernels on PIM side and delegates compute-bound kernels on GPU [70]. A few general purpose PIM architectures can handle the GNN workloads efficiently if the graph is stored in specific sparse formats [32, 33]. However, these aforementioned approaches are only applicable to *static* GNNs where the graph topology does not change over time. On the other hand, Mint [71] proposes an accelerator architecture for mining small motifs in temporal graphs. However, Mint is specifically designed for mining motifs and cannot process the neural network portion of the TGNN workloads.

There is only one prior accelerator of which we are aware specifically targeting temporal GNN [28]. Authors in this work proposed an algorithm-hardware co-optimization, where they mapped the TGN framework [19] on an HBM-enabled FPGA. Optimizations proposed by this approach include hardware pipeline stages, look-up table based time-encoding function, double buffering and prefetching mechanisms. However, this approach can only accommodate a small number of MAC units due to FPGA resource constraint, limiting the potential speedup. We evaluated against this approach in Section 5 and observed vastly superior performance.

7 Conclusions

In this paper, we proposed TGN-PNM, a near-memory architecture for accelerating TGNN workloads. In our approach, we placed a SIMD unit for memory-intensive operations and a systolic array for GEMM operations at the vault level. The potential bottleneck arising from inter-vault communication during neighbor aggregation is avoided by partitioning the graph along the feature dimension, facilitating near perfect workload balance as well, which is very difficult to achieve on evolving graphs. Out evaluation against a few other architectures revealed that near-/in-memory approaches perform the best for TGNN-type workloads.

Acknowledgments

This work was supported in part by PRISM, one of seven centers in JUMP 2.0, an SRC program sponsored by DARPA.

References

[1] Federico Monti, Fabrizio Frasca, Davide Eynard, Damon Mannion, and Michael M Bronstein. Fake news detection on social media using geometric deep learning. *arXiv preprint arXiv:1902.06673*, 2019.

[2] Emanuele Rossi, Federico Monti, Michael Bronstein, and Pietro Liò. ncrna classification with graph convolutional networks. *arXiv preprint arXiv:1905.06515*, 2019.

[3] Marinka Zitnik, Monica Agrawal, and Jure Leskovec. Modeling polypharmacy side effects with graph convolutional networks. *Bioinformatics*, 34(13):i457–i466, 2018.

[4] Kirill Veselkov, Guadalupe Gonzalez, Shahad Aljifri, Dieter Galea, Reza Mirnezami, Jozef Youssef, Michael Bronstein, and Ivan Laponogov. Hyperfoods: Machine intelligent mapping of cancer-beating molecules in foods. *Scientific reports*, 9(1):9237, 2019.

[5] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 974–983, 2018.

[6] Hongxia Yang. Aligraph: A comprehensive graph neural network platform. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 3165–3166, 2019.

[7] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[8] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[9] David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. Convolutional networks on graphs for learning molecular fingerprints. *Advances in neural information processing systems*, 28, 2015.

[10] Alex Fout, Jonathon Byrd, Basir Shariat, and Asa Ben-Hur. Protein interface prediction using graph convolutional networks. *Advances in neural information processing systems*, 30, 2017.

[11] Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. *Advances in neural information processing systems*, 31, 2018.

[12] Zhitao Ying, Jiaxuan You, Christopher Morris, Xiang Ren, Will Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. *Advances in neural information processing systems*, 31, 2018.

[13] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. Hygcn: A gcn accelerator with hybrid architecture. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 15–29. IEEE, 2020.

[14] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, et al. Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 922–936. IEEE, 2020.

[15] Haoran You, Tong Geng, Yongan Zhang, Ang Li, and Yingyan Lin. Gcod: Graph convolutional network acceleration via dedicated algorithm and accelerator co-design. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 460–474. IEEE, 2022.

[16] Rishov Sarkar, Stefan Abi-Karam, Yuqi He, Lakshmi Sathidevi, and Cong Hao. Flowgnn: A dataflow architecture for real-time workload-agnostic graph neural network inference. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1099–1112. IEEE, 2023.

[17] Atefeh Sohrabizadeh, Yuze Chi, and Jason Cong. Streamgcn: Accelerating graph convolutional networks with streaming processing. In *2022 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–8. IEEE, 2022.

[18] Mingi Yoo, Jaeyong Song, Jounghoo Lee, Namhyung Kim, Youngsok Kim, and Jinho Lee. Sgcn: Exploiting compressed-sparse features in deep graph convolutional network accelerators. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–14. IEEE, 2023.

[19] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. Temporal graph networks for deep learning on dynamic graphs. *arXiv preprint arXiv:2006.10637*, 2020.

[20] Da Xu, Chuanwei Ruan, Evren Korpeoglu, Sushant Kumar, and Kannan Achan. Inductive representation learning on temporal graphs. *arXiv preprint arXiv:2002.07962*, 2020.

[21] Srijan Kumar, Xikun Zhang, and Jure Leskovec. Predicting dynamic embedding trajectory in temporal interaction networks. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 1269–1278, 2019.

[22] Da Xu, Chuanwei Ruan, Evren Korpeoglu, Sushant Kumar, and Kannan Achan. Inductive representation learning on temporal graphs. *arXiv preprint arXiv:2002.07962*, 2020.

[23] Priyank Faldu, Jeff Diamond, and Boris Grot. A closer look at lightweight graph reordering. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–13. IEEE, 2019.

[24] Vignesh Balaji and Brandon Lucia. When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 203–214. IEEE, 2018.

[25] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 22–31. IEEE, 2016.

[26] Andrew McCrabb and Valeria Bertacco. Optimizing vertex pressure dynamic graph partitioning in many-core systems. *IEEE Transactions on Computers*, 70(6):936–949, 2021.

[27] Andrew McCrabb, Eric Winsor, and Valeria Bertacco. Dredge: Dynamic repartitioning during dynamic graph execution. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[28] Hongkuan Zhou, Bingyi Zhang, Rajgopal Kannan, Viktor Prasanna, and Carl Busart. Model-architecture co-design for high performance temporal gnn inference on fpga. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1108–1117. IEEE, 2022.

[29] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 105–117, 2015.

[30] Mingxuan He, Choungki Song, Ilkon Kim, Chunseok Jeong, Seho Kim, Il Park, Mithuna Thottethodi, and TN Vijaykumar. Newton: A dram-maker's accelerator-in-memory (aim) architecture for machine learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 372–385. IEEE, 2020.

[31] Jin Hyun Kim, Shin-haeng Kang, Sukhan Lee, Hyeonsu Kim, Woongjae Song, Yuhwan Ro, Seungwon Lee, David Wang, Hyunsung Shin, Bengseng Phuah, et al. Aquabolt-xl: Samsung hbm2-pim with in-memory processing for ml accelerators and beyond. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, pages 1–26. IEEE, 2021.

[32] Marzieh Lenjani, Patricia Gonzalez, Elaheh Sadredini, Shuangchen Li, Yuan Xie, Ameen Akel, Sean Eilert, Mircea R. Stan, and Kevin Skadron. Fulcrum: a Simplified Control and Access Mechanism toward Flexible and Practical in-situ Accelerators. In *HPCA*, 2020.

[33] Marzieh Lenjani, Ahmed Alif, Mircea R. Stan, and Kevin Skadron. Gearbox: A Case for Supporting Accumulation Dispatching and Hybrid Partitioning in PIM-based Accelerators. In *'To Appear in ISCA*, 2022.

[34] Shuangchen Li, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. DRISA: A DRAM-based reconfigurable in-situ accelerator. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

[35] Nastaran Hajinazar, Geraldo F Oliveira, Sven Gregorio, João Dinis Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gómez-Luna, and Onur Mutlu. SIMDRAM: a framework for bit-serial SIMD processing using DRAM. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[36] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A Kozuch, Onur Mutlu, Phillip B Gibbons, and Todd C Mowry. Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 273–287, 2017.

[37] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. Temporal graph networks for deep learning on dynamic graphs. *arXiv preprint arXiv:2006.10637*, 2006.

[38] Weilin Cong, Si Zhang, Jian Kang, Baichuan Yuan, Hao Wu, Xin Zhou, Hanghang Tong, and Mehrdad Mahdavi. Do we really need complicated model architectures for temporal networks? *arXiv preprint arXiv:2302.11636*, 2023.

[39] Xuhong Wang, Ding Lyu, Mengjian Li, Yang Xia, Qi Yang, Xinwen Wang, Xinguang Wang, Ping Cui, Yupu Yang, Bowen Sun, et al. Apan: Asynchronous propagation attention network for real-time temporal graph embedding. In *Proceedings of the 2021 international conference on management of data*, pages 2628–2638, 2021.

[40] Rakshit Trivedi, Mehrdad Farajtabar, Prasenjeet Biswal, and Hongyuan Zha. Dyrep: Learning representations over dynamic graphs. In *International conference on learning representations*, 2019.

[41] Hongkuan Zhou, Da Zheng, Israt Nisa, Vasileios Ioannidis, Xiang Song, and George Karypis. Tgl: A general framework for temporal gnn training on billion-scale graphs. *arXiv preprint arXiv:2203.14883*, 2022.

[42] Seyed Mehran Kazemi, Rishab Goel, Sepehr Eghbali, Janahan Ramanan, Jaspreet Sahota, Sanjay Thakur, Stella Wu, Cathal Smyth, Pascal Poupart, and Marcus Brubaker. Time2vec: Learning a vector representation of time. *arXiv preprint arXiv:1907.05321*, 2019.

[43] Seyed Mehran Kazemi, Rishab Goel, Kshitij Jain, Ivan Kobyzev, Akshay Sethi, Peter Forsyth, and Pascal Poupart. Representation learning for dynamic graphs: A survey. *The Journal of Machine Learning Research*, 21(1):2648–2720, 2020.

[44] Hybrid Memory Cube Consortium. Hybrid memory cube specification 2.1. https://www.hybridmemorycube.org/, 2015.

[45] Ramyad Hadidi, Bahar Asgari, Burhan Ahmad Mudassar, Saibal Mukhopadhyay, Sudhakar Yalamanchili, and Hyesoon Kim. Demystifying the characteristics of 3d-stacked memories: A case study for hybrid memory cube. In *Proceedings of the IEEE International Symposium on Workload Characterization* , pages 66–75, 2017.

[46] Christian Weis, Norbert Wehn, Loi Igor, and Luca Benini. Design space exploration for 3d-stacked drams. In *2011 Design, Automation & Test in Europe*, pages 1–6. IEEE, 2011.

[47] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. Spacea: Sparse matrix vector multiplication on processing-in-memory accelerator. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 570–583. IEEE, 2021.

[48] JEDEC. High bandwidth memory 3 specification. https://www.jedec.org/standards-documents/docs/jesd238a, 2023.

[49] Joe Jeddeloh and Brent Keeth. Hybrid memory cube new DRAM architecture increases density and performance. In *2012 symposium on VLSI technology (VLSIT)*, pages 87–88. IEEE, 2012.

[50] Ivan Fernandez, Ricardo Quislant, Eladio Gutiérrez, Oscar Plata, Christina Giannoula, Mohammed Alser, Juan Gómez-Luna, and Onur Mutlu. Natsa: a near-data processing accelerator for time series analysis. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 120–129. IEEE, 2020.

[51] UPMEM. https://www.upmem.com/.

[52] Alif Ahmed, Farzana A Siddique, and Kevin Skadron. Graphtango: A hybrid representation format for efficient streaming graph updates and analysis. In *IPDPS, (under submission)*, 2023.

[53] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. Dramsim3: A cycle-accurate, thermal-capable dram simulator. *IEEE Computer Architecture Letters*, 19(2):106–109, 2020.

[54] Ke Chen, Sheng Li, Naveen Muralimanohar, Jung Ho Ahn, Jay B Brockman, and Norman P Jouppi. Cacti-3dd: Architecture-level modeling for 3d die-stacked dram main memory. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 33–38. IEEE, 2012.

[55] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd annual ieee/acm international symposium on microarchitecture*, pages 469–480, 2009.

[56] Srijan Kumar, Xikun Zhang, and Jure Leskovec. Snap dataset collection: Jodie. https://snap.stanford.edu/jodie/, 2019.

[57] Rossi et al. TGN github repository. *URL: https://github.com/twitter-research/tgn*, 2020.

[58] FulcumV3. https://github.com/MarziehLenjani/FulcrumV3.

[59] Hongkuan Zhou, Bingyi Zhang, Rajgopal Kannan, Viktor Prasanna, and Carl Busart. Github repository, model-architecture co-design for high performance temporal gnn inference on fpga. https://github.com/zjjzby/TGNN-FPGA-IPDPS2022, 2022.

[60] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH computer architecture news*, 44(3):367–379, 2016.

[61] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):292–308, 2019.

[62] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[63] Norman P Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, et al. Ten lessons from three generations shaped google's tpuv4i: Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14. IEEE, 2021.

[64] Md Aamir Raihan, Negar Goli, and Tor M Aamodt. Modeling deep learning accelerator enabled gpus. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 79–92. IEEE, 2019.

[65] Ham et al. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *MICRO*, pages 1–13, 2016.

[66] Hu et al. Graphlily: Accelerating graph linear algebra on hbm-equipped fpgas. In *ICCAD*, pages 1–9, 2021.

[67] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. Graphr: Accelerating graph processing using reram. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 531–543. IEEE, 2018.

[68] 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. IEEE, 2018.

[69] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. In *2017 IEEE International symposium on high performance computer architecture (HPCA)*, pages 457–468. IEEE, 2017.

[70] Hai Jin, Dan Chen, Long Zheng, Yu Huang, Pengcheng Yao, Jin Zhao, Xiaofei Liao, and Wenbin Jiang. Accelerating graph convolutional networks through a pim-accelerated approach. *IEEE Transactions on Computers*, 2023.

[71] Nishil Talati, Haojie Ye, Sanketh Vedula, Kuan-Yu Chen, Yuhan Chen, Daniel Liu, Yichao Yuan, David Blaauw, Alex Bronstein, Trevor Mudge, et al. Mint: An accelerator for mining temporal motifs. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1270–1287. IEEE, 2022.