StreamGCN: Accelerating Graph Convolutional Networks with Streaming Processing

Atefeh Sohrabizadeh, Yuze Chi, and Jason Cong
Computer Science Department, UCLA
Los Angeles, USA
{atefehsz, chiyuze, cong} @cs.ucla.edu

***Abstract*—While there have been many studies on hardware acceleration for deep learning on images, there has been a rather limited focus on accelerating deep learning applications involving graphs. The unique characteristics of graphs, such as the irregular memory access and dynamic parallelism, impose several challenges when the algorithm is mapped to a CPU or GPU. To address these challenges while exploiting all the available sparsity, we propose a flexible architecture called StreamGCN for accelerating Graph Convolutional Networks (GCN), the core computation unit in deep learning algorithms on graphs. The architecture is specialized for streaming processing of many small graphs for graph search and similarity computation. The experimental results demonstrate that StreamGCN can deliver a high speedup compared to a multi-core CPU and a GPU implementation, showing the efficiency of our design.**

I. INTRODUCTION

Graphs are the core data structure used in datacenters and have a wide application in different domains such as recommender systems, social networks, and the World Wide Web. Although they are widely used, they are mainly unstructured and have a high dimensionality, making them computationally expensive to process. This problem has motivated researchers to apply deep learning on graphs with the goal of extracting structured, low-dimensional features from it. In this context, Graph Convolutional Networks (GCN) [14] are widely used to assign feature vectors, called *node embeddings*, to nodes of the graph. They consist of multiple layers in which the features of the nodes are propagated within them until a rich information of the input graph is derived. GCNs have shown to be successful in many domains including molecular footprint calculation [7], logic optimization for EDA tools [10], etc.

While some graph data tend to scale rapidly, there are also many graph data that are naturally limited in size, for example, chemical compounds and molecules [3], [4], [19], [28] that have a wide application in different domains including drug development, quantum mechanics, physical chemistry, biophysics, etc [4], [32]; the GREC database consisting of graphs representing symbols from architectural and electronic drawings [6], etc [22], [32]. The average number of nodes for the graphs of these databases ranges from 5 to 50.

Because of the vast application of small graphs, numerous algorithms have been proposed to obtain their information [1], [13], [16], [18], [21]. In particular, SimGNN [1] proposed a GCN-based approach to learn a similarity score for such graphs. SimGNN targets graphs from real-world graph databases, such as AIDS [19], LINUX [29], and IMDB [35]. The target graphs are relatively small, with 10 nodes on average, but the database contains millions of graph pairs, creating many graph matching queries. Although the CPU implementation can finish each SimGNN query in milliseconds, processing millions of queries can take several hours; hence, it requires customized acceleration. Such a workload of graph searching/mining is increasing in importance. For example, searching for antivirus chemical compounds is an important step in drug repurposing for COVID-19.

Despite the popularity and effectiveness of *graph neural networks* (GNN) approaches, there has been limited research on developing an accelerator for them (e.g. [9], [34], [37]) as GNN imposes the following challenges in designing one:

- **Irregular memory access and low data reuse:** As opposed to images, the neighbors of a node in a graph may be stored in any location in memory. This will result in many irregular memory accesses to all levels of the memory hierarchy. Furthermore, GNNs have much lower data reuse compared to Convolutional Neural Networks (CNN). As such, the countless CNN accelerators proposed in the literature (e.g., [24], [25], [31], [39]) are incompatible here. Compared to the traditional graph algorithms such as breadth first search (BFS), the nodes have long feature vectors instead of a single scalar value. Therefore, not only is the access pattern different, but we can also exploit new kinds of parallelism and data reuse, making most graph-based accelerators (e.g., [5], [11], [30], [36]) ill-suited for GNNs.
- **Computation pattern disparity:** Different steps of the GCN algorithm deal with different sparsity rates (see Section IV). Besides, a GNN may include other types of computation patterns, such as neural tensor network in SimGNN (see Section V) to make an end-to-end application. Such variations call for a customized processing unit for each step.
- **Dynamic workload and parallelism:** Since the *number* of neighbors varies across different nodes, there will be a load-imbalance between the graph's nodes.

In addition to the challenges mentioned above, dealing with small graphs requires special design considerations as we will explain in Section IV. To solve these challenges, we present *StreamGCN* as an efficient and flexible GCN accelerator for streaming small graphs - from the different levels of memory and even through the network - and exploiting all the available sparsity. Then, we apply it to accelerate the entire pipeline of SimGNN as an end-to-end application. Since we are facing a *memory-bounded* application, we reduce the global memory transactions to the least amount. To deal with *the irregular memory access*, we utilize a scratchpad memory to store the matrices that need random access. Because of the *computation pattern disparity*, we analyze the requirements of all the steps of the computation pipeline and, accordingly, develop a dedicated architecture for each of them. We further propose an efficient workload distribution mechanism to alleviate the *load-imbalance* problem.

Concisely, we fuse all the stages together and employ a very deep pipeline with three different levels of nested customizable parallelization as listed in Table I. While we use SimGNN for illustrating our approach, the same optimizations can be applied to other GCN-based networks dealing with small graphs such as [2], [13], [21] as well. We implement StreamGCN on three different FPGAs showing its flexibility and adaptivity to different platforms with different global memory bandwidth.

In summary, the key contributions of this paper are:

- We design and develop StreamGCN, a flexible architecture for accelerating GCN specialized for streaming processing of small graphs and exploiting all the available sparsity.
- We adopt StreamGCN to accelerate SimGNN as an end-to-end application, resulting in an efficient architecture with a very deep pipeline and three levels of parallelization.
- We demonstrate the flexibility of our architecture by mapping and customizing it to three different FPGAs with different capacities and memory systems.
- Experimental results suggest that our accelerator can outperform multi-core CPU by 18.2x and GPU by 26.9x, demonstrating the efficiency of our design.

II. BACKGROUND

A. Graph Convolutional Network (GCN)

Layer l of a GCN [14] takes an undirected graph $G(V, E, H^l)$ as the input, where V (E) denotes the nodes (edges) of the graph. $H^l \in \mathbb{R}^{|V| \times f_l}$ is the matrix of the *input node embeddings* for this layer, with each row containing the embedding of one of the nodes where f_l indicates the number of features of each node at layer l. The core computation of a GCN layer to produce the *output node embeddings* is as follows:

978-1-6654-0756-4/22/$31.00 ©2022 IEEE

TABLE I: Our approach compared to state-of-the-art GCN accelerators.

Work	Graph Size	Layer Customization	Sparse Engine for Feature Transformation	On-the-fly Sparsity Pruning	Read Each Element Only Once	Parallelization		
						Inter-layer	Feature-level (Sparse Part)	Node-level (Sparse Part)
HyGCN [34]	Large	✗	✗	✗	✗	✗	✓	✓
GraphACT [37]	Small	✗	✗	✗	✗	✗	✓	✗
BoostGCN [38]	Large	✗	✓	✗	✗	✗	✓	✓
AWB-GCN [9]	Large	✓	✓	✓	✗	✓	✗	✓
Ours (StreamGCN)	Small	✓	✓	✓	✓	✓	✓	✓

$$\tilde{A} = A + I_N,$$
$$\tilde{D}_{ii} = \sum_j \tilde{A}_{ij},$$
$$A' = \tilde{D}^{-\frac{1}{2}}.\tilde{A}.\tilde{D}^{-\frac{1}{2}}, \quad (1)$$
$$H^{l+1} = \sigma_{act}(A'.H^l.W^l)$$
$$H^{l+1} \in \mathbb{R}^{|V| \times f_{l+1}}$$

where $\sigma_{act}(.)$ is an activation function which typically is a ReLU and $W^l \in \mathbb{R}^{f_l \times f_{l+1}}$ is a layer-specific *trainable weight matrix*. A' is the *normalized adjacency matrix with added self-connections* that is calculated using A and I_N which are the *adjacency* and the *identity matrix*, respectively. $\tilde{D}$ is a *diagonal matrix* where $\tilde{D}_{ii}$ is the degree of node i plus one.

As Eq. 1 suggests, the first step in the computation $(A'.H^l)$ gathers the neighbors' information for each of the nodes. As A' is a normalized matrix, the computation here is a weighted aggregation. After the Aggregation step, the node embeddings are transformed by applying a pre-trained set of weights and finally passed through a ReLU unit. The time complexity of Layer l can be seen to be $O(|E|f_l f_{l+1})$, where $|E|$ denotes the number of edges including the self-connection ones [14].

III. Previous GCN Accelerators

Because of the popularity of GCN, there is a growing interest in developing an accelerator for it [9], [17], [34], [37], [38]. As summarized in Table I, HyGCN [34], GraphACT [37], and BoostGCN [38] develop a fixed hardware for all the layers of GCN and process them sequentially. This is an undesirable feature particularly when we target small graphs. In fact, in this paper, we first develop a baseline architecture that has the same design principles as these works. Particularly, we reuse the same architecture for all the GCN layers, exploit only the sparsity of the Aggregation step, treat the Feature Transformation step as a regular matrix multiplication, and employ a 2D computation unit for it. Our experimental results in Section VI-C (Table IV) show that not only should we execute the GCN layers in a pipelined fashion, but we should also exploit the sparsity of the node embeddings to enhance both area and performance. In fact, these optimizations bring in 2.27x speedup in the performance and an overall improvement of 3.88x in both performance and computation units' area. While BoostGCN considers the sparsity of node embeddings, it proposes the hardware support only when dealing with ultra-sparsity (more than 90% sparsity). Furthermore, it needs the input to be in the COO format which adds extra overheads. However, *StreamGCN* handles the sparsity by pruning the zeros *on-the-fly* while they are being generated.

AWB-GCN [9] proposes an architecture that supports inter-layer pipelining and considerations for sparsity of the node embeddings for accelerating GCN. However, partitioning the computation by the nodes in their approach complicates the design of the task distributor since the node embeddings are sparse and special consideration is needed to prevent PEs from doing unnecessary operations on the zero elements. On the other hand, feature-level parallelization deals better with workload imbalance as we shall discuss in Section IV. In addition, AWB-GCN is developed for large graphs and adapts the inner-product matrix multiplication (MM), whereas, as we will explain in Section IV-B, the outer-product MM is preferred here. These design decisions distinguish StreamGCN from the previous GCN accelerators as summarized in Table I.

IV. StreamGCN Architecture

We can compute Eq. 1 either as $(A' \times H^l) \times W^l$ or $A' \times (H^l \times W^l)$. We have chosen the latter since it results in a fewer number of operations. Intuitively, this is because both matrices A' and H^l are sparse, but their multiplication creates a dense matrix. As a result, in the former, we end up doing a dense-dense multiplication for the second multiplication. However, if we go with the latter, both multiplications are sparse-dense that as shown in AWB-GCN [9], it reduces the number of operations. Fig. 2 illustrates the high-level view of GCN architecture in StreamGCN. In this section, we employ a bottom-up approach to highlight the optimization opportunities when GCN is applied to small graphs and how we used them to build the GCN accelerator as demonstrated in Fig. 2.

A. StreamGCN Design Principles

StreamGCN is designed:

- To exploit all the available sparsity.
- To reduce the number of times we access the global memory to the least amount possible. In our final architecture, each input element is read only once and there is no need to store any of the intermediate results in the global memory.
- To employ a deep pipeline with varying levels and degrees of parallelization for matching the workload of different stages and maximizing the overall performance.
- To efficiently handle and stream small graphs.

B. Baseline Architecture

In this section, we describe the basic optimizations that can be applied for processing GCNs. Although these optimizations are necessary, they are not enough when dealing with many small graphs. Hence, we propose to apply further optimizations in the subsequent sections.

1) Feature Transformation (FT):

In this step, one must multiply matrices $H^l \in \mathbb{R}^{|V| \times f_{in}}$ and $W^l \in \mathbb{R}^{f_{in} \times f_{out}}$ where f_{in} and f_{out} denote the number of input and output features, respectively. Here, adopting an inner-product-based matrix multiplication results in updating the same output feature in the consecutive iterations which introduces read-after-write (RAW) dependency between them. As a result, our pipeline cannot achieve an initiation interval (II) of one meaning that we cannot schedule new operations in each clock cycle which degrades the efficiency of our design.

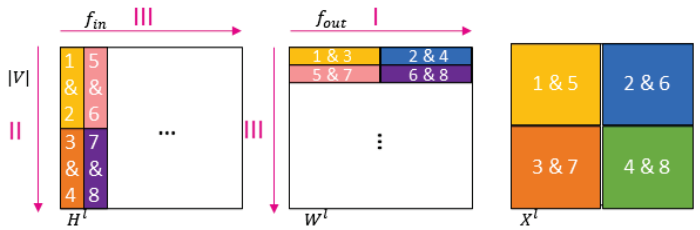

Fig. 1: The overall computation order for FT step. English numbers show the cycle numbers, and the Roman numbers denote the high-level order of the computation.

***Optimized Scheduling**: Read the Weight Matrix Row-wise; Stream the Embeddings Matrix Column-wise.* To alleviate the RAW dependency problem, we perform Cartesian product as in [20].

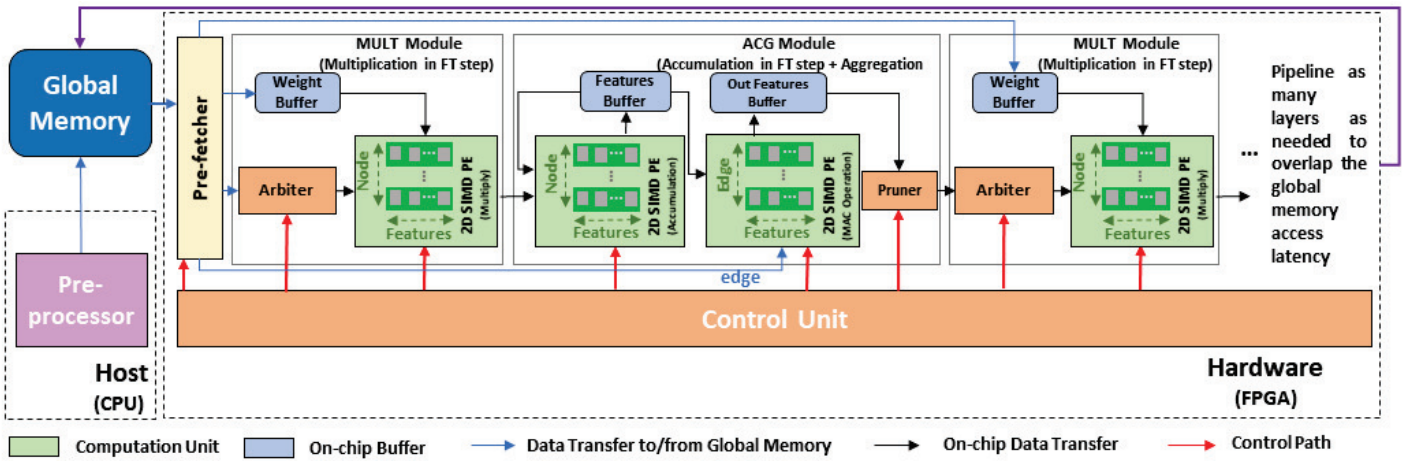

Fig. 2: High-level overview of the GCN accelerator architecture in StreamGCN

Meaning that we design a processing element (PE) consisting of *SIMD* multiplication and accumulation (MAC) units. At each cycle, we update different output locations by taking an element from H^l (read as a stream) and broadcasting that to parallel MAC units while each MAC unit reads different elements of the W^l matrix. To read each element only once and increase data reuse, for each fetched element of H^l, we schedule all the operations it is involved with before its eviction. We add a second level of parallelization by duplicating the SIMD PE by a *duplication factor (DF)* which parallelizes the node dimension. To avoid RAW dependencies between the PEs, we read H^l in the column order. Note that if we read it rather in row order, we update the same location every $\frac{f_{out}}{SIMD}$ iterations instead of every $\frac{|V|}{DF} \times \frac{f_{out}}{SIMD}$ iterations. Reading in column order also lets us cache and reuse the corresponding row of the weight matrix. Fig. 1 illustrates the final execution order of this step. The arrows denote the high-level ordering of traversing different dimensions, and the numbers show the elements that are accessed at their respective cycles. It is important to traverse the input feature dimension (f_{in}) last (*arrow III*) since it is the dimension causing the dependencies.

2) Aggregation:

In this step, we must multiply matrices $A' \in \mathbb{R}^{|V| \times |V|}$ and $X^l \in \mathbb{R}^{|V| \times f_{out}}$ where X^l is the result of the FT step. Due to the highly irregular access to the matrix X^l to aggregate features of the neighbors, we cache it in a scratchpad memory. Matrix A', is often ultra sparse [9]. To reduce the number of both transferred elements and operations, we prune this matrix and only pass its non-zero elements, which represent edges, to the FPGA. Instead of dedicating an on-chip memory for storing the edges, we read them as a stream and update all the features of the destination node, before retiring the edge. It helps us with freeing up the storage for caching X^l which is the same matrix that needs to be cached for the FT step. We further re-arrange the edges, as a step of pre-processing, before sending them to the FPGA, so that the ones with the same destination node are at least *L* (the latency of the functional unit causing the dependency) locations apart to make sure there is not more than one update to the same node within the window of *L* cycles. As edge-level parallelism can result in bank conflicts since they update random nodes, we only make use of feature-level parallelism to distribute the workload here. Nevertheless, one can include that with adding another level of pre-processing by further re-ordering the edges.

3) Intra-layer Pipelining:

To further boost the performance, we add intra-layer pipelining by connecting the modules as a dataflow architecture. As a result, the overall latency will be close to the latency of the slowest module. In addition, we can avoid global memory accesses in between these modules. The MULT module, depicted in Fig. 2, is responsible for doing all the multiplications of the FT step. It has a local buffer to store the weights and streams the elements of H^l from the input FIFO. Each entry of this FIFO is a concatenation of DF elements. Once the multiplication results are ready, they are packed and sent in a FIFO to the ACG module (Fig. 2). In this module, we merge the ACC unit of the FT step and the Aggregation step to save memory resources since they share the matrix X^l. After fetching the output of the *MULT* module, the *ACG* module unpacks the data based on the same DF, and dispatches SIMD elements to each *SIMD ACC Unit* with the same SIMD factor. Once the additions are done, it will store the partial results to the local buffer *features buffer*. After all updates are committed to the *features buffer*, the matrix X^l is computed and the Aggregation step can start. The SIMD factor of this step is higher than the one in FT step since we only exploit feature-level parallelization here. After this step is finished, the elements of the *out features buffer* are added with a bias, passed through a ReLU unit ($max(0,\cdot)$), and stored into the global memory. Note that in the baseline architecture, we reuse the same modules for all the GCN layers.

C. Extension 1: Multi-layer Support and Inter-layer Pipelining

As it is commonly practiced ([17], [34], [37]), in the baseline architecture, we only exploit intra-layer pipelining and reuse the modules for all the GCN layers. However, this is not sufficient when we are dealing with small graphs. The off-chip communication is a serious burden for this application since it deals with small-sized inputs. To alleviate this problem, we intend to reduce the number of accesses to the off-chip (global) memory as much as possible. The baseline architecture is inefficient with this regard since, at the end of each layer, the output should be stored to the global memory and read back again for the next layer. To avoid these redundant accesses, we extend the dataflow architecture described in Section IV-B3 to all the layers of GCN. To realize this, we instantiate new modules for each layer and connect them with FIFOs as depicted in Fig. 2. Fusing the computation for all the layers by enabling dataflow architecture has several benefits such as: 1) we can avoid writing the intermediate results to the global memory by forwarding them to the next layer through FIFOs. 2) The operations will be dynamically scheduled since each module can perform its operation whenever it has a data available. 3) Since we are instantiating different modules for each layer, we can customize the parallelization factors of each module based on the workload of their respective GCN layer. 4) As the adjacency matrix of a graph does not change across different layers, we can read the edges from the global memory only once for the first layer and reuse them for the subsequent ones by transferring them through the on-chip FIFOs.

D. Extension 2: In Situ Sparsity Support in FT Step

The input node embeddings to the first layer of GCN usually contain many zero elements since they often adopt one-hot encoding for assigning initial vectors to the nodes. Furthermore, since there is a ReLU unit at the end of each GCN layer, the matrix generated by each layer, which is the input to the next layer, is sparse. In fact, we saw 52% and 47% sparsity on average for the input to the second and the third layers of GCN in SimGNN for randomly drawn graphs from our target dataset. Therefore, the FT step also needs to have the support for sparse computation. To reduce the number of operations, we prune the zero elements and only pass the non-zero ones to the next layer. As a result, the updates to the output buffer may come in random cycles; thus, it is necessary to store the buffer containing the partial results on-chip to enable random access. For the same reason, we pack the node features with their address which includes their row and column ID. Packing the elements with their address helps to make the dispatch unit simpler since each SIMD PE is free to work with any data and knows which partial result should be updated; hence, there is no need to take special considerations to navigate the data to the correct PE. We only need to make sure that at all the times, each SIMD PE is working with a different memory bank. We employ an arbiter for this matter, as explained below.

As mentioned in Section IV-B, to reduce the number of RAW dependencies, we chose to stream the node embedding matrix and broadcast the elements to different computation units (CU) which read the weight matrix as a batch. Since the node embedding is a sparse matrix, reading it as a stream facilitates the pruning mechanism we employ and enables us to distribute the workload more efficiently. Fig. 3 demonstrates a toy example illustrating this. The colored squares show the non-zero elements of the node embedding matrix. By mapping the weights, which are non-zero, to the SIMD dimension, all the CUs in the PE would execute useful operations and we can skip all the operations involving a zero node embedding.

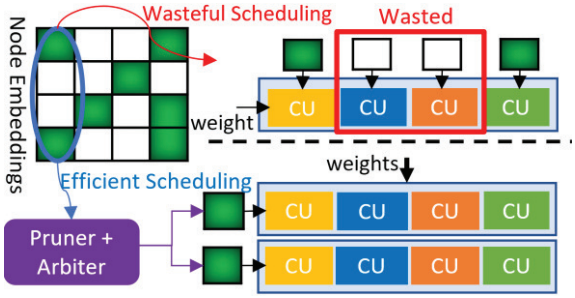

Fig. 3: The benefit of streaming the node embeddings and mapping the weights to the SIMD dimension.

When skipping the zero node embeddings, the dependency distance for output elements may change dynamically since the number of non-zero inputs between the updates to the same location can be different. Even though the scheduling discussed in Section IV-B increases the dependency distance as much as possible by doing all the operations when a nonzero input is encountered (each non-zero element would fill $\frac{f_{out}}{SIMD_{FT}}$ cycles of the dependency window), there still may be some cases where the dependency distance is less than *L* after this optimization. Instead of setting the II to *L* to ensure the correctness, we first insert *L* registers to store the partial results of CU at the end of each of its pipeline stages; hence, we can schedule a new set of operations at each clock cycle (II=1). There may be cases where the new scheduled operations want to update a location whose old value is still in the registers and have not updated the buffer. To ensure the correctness, we add a control unit which keeps track of the last cycle that each of the output locations was updated. If the number of cycles between two updates to the same location is less than *L*, the control unit will insert bubbles into the pipeline until the previous update is committed.

We insert a unit for pruning zeros at the end of the *ACG* module. As Fig. 4 demonstrates, at each cycle, we evaluate *P* elements of the node embeddings and pass each to a FIFO if it is not zero. The *MULT* module of the next layer takes the *P* FIFOs as the input and uses an arbiter to fetch, at most, *DF* of them (*DF* <= *P*) for passing to *DF* SIMD PEs. An arbiter keeps track of the FIFO whose turn it is to be read first in the next cycle. It then uses a round-robin ordering for dispatching the elements from the non-empty FIFOs. After dispatching the inputs, it checks for the RAW dependency by scanning the *prev iter* buffer which contains the last cycle when each element was seen as the input. If the distance was less than *L*, it will insert bubbles in the pipeline until the previous input has committed its update. If there is no dependency, for each memory bank at most one element from the dispatched inputs will be issued to a SIMD PE and the current cycle number will be stored in *prev iter* buffer for that input.

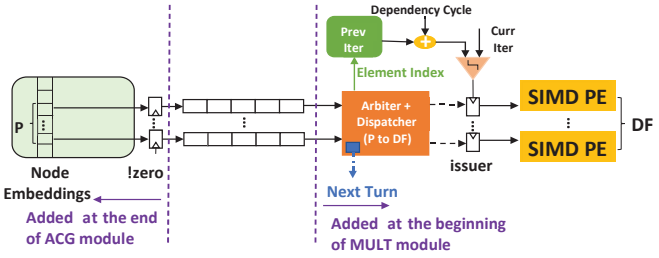

Fig. 4: Architecture support for sparse computation in Feature Transformation step.

The StreamGCN architecture provides a flexibility in choosing the parallelization factors. Table II lists the parameters that can be tuned for each GCN layer based on its workload. The SIMD factors correspond to feature-level parallelization, while *DF* and *P* map to node dimension.

TABLE II: Summary of the architecture parameters for the accelerator of each GCN layer in StreamGCN.

Design Parameter	Explanation
$SIMD_{FT}$	SIMD factor of the FT step
$SIMD_{Agg}$	SIMD factor of the Aggregation step
DF	Duplication factor of the PEs in FT step
P	Number of input FIFOs to the arbiter

FT: Feature Transformation

V. StreamGCN Application to Graph Matching

In Section IV, we proposed an architecture for GCN specialized for small graphs. In this section, we extend our architecture to accelerate an end-to-end application, SimGNN, which introduces new computation patterns beyond GCN.

A. SimGNN

Bai et al. [1] proposed a neural-network-based approach to assign a similarity score to two graphs. Its computation pipeline consists of four major stages. The first stage has three layers of GCN to extract the *node embeddings* $H \in \mathbb{R}^{|V| \times F}$ where F is the number of features of the last layer. In the second stage, it uses a *Global Context-Aware Attention layer (Att)* to combine the node embeddings and generate a single embedding per graph $h_G \in \mathbb{R}^F$. For this matter, it adapts an attention mechanism to find out the importance of each of the nodes. The graph embedding, then, can be calculated by taking a weighted sum of the node embeddings using the attention weights. The following formula summarizes the computation in this stage:

$$h_G = \sum_{n=1}^{|V|} \sigma\left(h_n^T . \tanh\left(\frac{1}{|V|} W_{Att} \sum_{n=1}^{|V|} h_n\right)\right) . h_n \qquad (2)$$

where $\sigma(\cdot)$ denotes the sigmoid function to produce the attention weights and $W_{Att} \in \mathbb{R}^{F \times F}$ is a learnable *weight*. The time complexity of this stage can be seen to be $O(|V|F)$. The third stage is a *Neural Tensor Network (NTN)* that calculates a vector of similarity scores between the two graphs:

$$s(h_{G_1}, h_{G_2}) = \sigma(h_{G_1}^T W_{NTN}^{[1:K]} h_{G_2} + V.concat(h_{G_1}, h_{G_2}) + b) \quad (3)$$

Where $W_{NTN}^{[1:K]} \in \mathbb{R}^{F \times F \times K}$, $V \in \mathbb{R}^{K \times 2F}$, and $b \in \mathbb{R}^K$ are learnable *weight tensor*, *weight matrix*, and *bias vector*, respectively. K is a hyper-parameter that controls the number of similarity scores. The time complexity of this stage is $O(F^2K)$. The last stage uses a fully connected network (FCN) to gradually reduce the similarity vector to only one score.

The non-GCN stages make use of *exp* and *tanh* functions which are expensive to have on FPGA that can limit their parallelism rate. On the other hand, the computation complexity of the different stages shows that the GCN step is the most computation-intensive one; hence, when pipelining all the stages together, the accelerator will be bottlenecked by the GCN step. Therefore, we do not aggressively parallelize the rest of the steps and rather focus on reducing their resource utilization.

B. Att Architecture

The SimGNN pipeline applies the GCN stage to two graphs for each comparison query. Instead of duplicating the architecture in Fig. 2, we process the graphs serially and reuse the GCN module for the two input graphs in the query. Reusing the GCN module enables us to map the design to small FPGAs as well. We improve the performance of processing one query by overlapping the *GCN* computation of one graph with the *Att* computation of the other one. Thus, the total performance will be bottlenecked with the performance of GCN, and we can focus on reducing the area and reusing the resources for *Att*. In computing $v = W_{Att} \sum_{n=1}^{|V|} h_n$, we first must add h_n vectors and then do a matrix-vector multiplication (MVM). Instead of instantiating separate adders for the first additions and the ones in MVM, we rewrite the equation as follows to reuse the adders:

$$v = W_{Att} \sum_{n=1}^{|V|} h_n = \sum_{n=1}^{|V|} W_{Att} h_n = sum(H.W_{Att}, 2) \quad (4)$$

where $sum(H.W_{Att}, 2)$ denotes the reduction of the resulting matrix across its second dimension (columns), meaning that all the multiplications associated with a column of H should be added together. Fig. 5 demonstrates an overview of the *Att* module. As in the GCN stage, we divide the MAC operations on the matrices to two different modules, one responsible for multiplications and the other for additions. Again, we use SIMD PEs to implement these modules. However, the SIMD factor here can be set to a different value compared to the GCN stage since they have different computation complexities. The *Repack* module is responsible for adjusting the output of GCN with the SIMD factor of this stage. For *tanh* and *exp* functions, we adopt their implementation from the *Xilinx HLS Math* library. Note that the last summation in Eq. 2 can be seen as $H^T \times a$ where $a \in \mathbb{R}^{|V|}$ contains the sigmoid results. Hence, we use a matrix vector multiply (MVM) unit at the end.

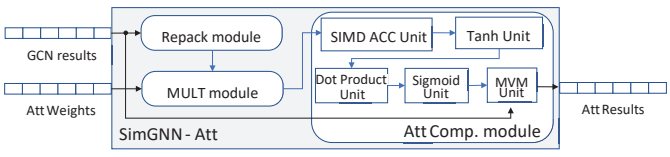

Fig. 5: Architecture overview of the second stage of SimGNN in StreamGCN: Att

C. NTN + Fully Connected Network (FCN) Architecture

The computation in the NTN stage is rather simple since it is a series of fixed-size MVMs followed by a bias addition and an activation function. Furthermore, the layers of the FCN in the last stage either need an MVM unit or a reduction tree to lower a vector to a scalar. Like the previous stages, we implement all the sub-modules of these two stages in a dataflow-manner. Fig. 6 depicts the architecture of these two steps.

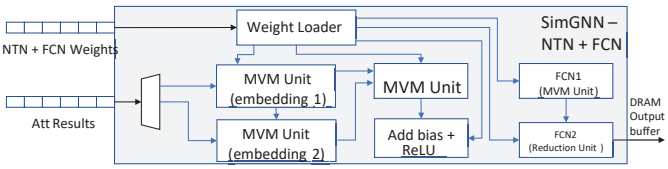

Fig. 6: Architecture overview of the last two stages of SimGNN in StreamGCN: NTN and FCN.

D. Putting It All Together

The whole computation pipeline of SimGNN is implemented as a three-level dataflow architecture. The first two levels resemble an inter-stage pipelining while the last one is for intra-stage pipelining. The first level enables a task-level parallelization by grouping the graph-related steps, the *GCN* (Section IV) and *Att* (Section V-B) modules, and overlapping them with the rest, *NTN_FCN module* (Section V-C). The second level of the dataflow architecture overlaps the *GCN* stage with the *Att*. Finally, the last level applies dataflow architecture to each of the *GCN*, *Att*, and *NTN_FCN* modules as shown in Fig. 2, 5, and 6, respectively. We apply three optimizations for reducing the off-chip communication latency: 1) each input buffer can be mapped to a different DRAM bank or HBM channel to enable parallel access to them, 2) the available global memory bandwidth is fully utilized by applying memory coalescing. Memory burst is also applied to amortize the initialization overhead, 3) the modules that access the global memory are overlapped by the computation modules by implementing the accelerator as a dataflow architecture.

VI. Experimental Results

A. Benchmark

We consider a real-life graph dataset, AIDS [19], for benchmarking our design. AIDS contains 42,687 antivirus chemical compounds gathered by the Developmental Therapeutics Program at NCI/NIH. The graphs in AIDS have 25.6 (27.6) nodes (edges) on average. We randomly form 10,000 queries of them for testing. The kernel time and end-to-end (E2E) time reported in this section are the average of all queries.

B. Experimental Setup

The StreamGCN architecture is described using Vivado HLS C++ [33]. The design is synthesized and implemented using Xilinx Vitis 2019.2 on three different target platforms: Xilinx Alveo U50, Xilinx Alveo U280, and Xilinx Kintex UltraScale+ KU15P. The first two are equipped with HBM2 and, ideally, can achieve a bandwidth of 316 GB/s (460 GB/s) with a TDP of 75W (225W); while the last one utilizes DDR4 as the global memory. Table III compares the hardware resources of these boards. For comparison to CPU and GPU, the PyTorch-based implementation of SimGNN from [23] is used that is built using the state-of-the art PyTorch Geometric (PyG) library [8] which is commonly used as a baseline by previous works [9], [17], [34]. For the Aggregation step, PyG exploits sparsity and edge-level parallelism by adapting the PyTorch Scatter library. For the Feature Transformation step, it uses Intel MKL [40] and NVIDIA cuBLAS library [41] for CPU and GPU respectively, making it a reasonable and optimized baseline. The target CPU in our experiments is Intel(R) Xeon(R) CPU E5-2699 v4 running at 2.2 GHz. For testing on GPU, we use an AWS p3.2xlarge instance which has an NVIDIA V100 GPU.

TABLE IV: Impact of GCN architecture optimizations on U280. The meaning of design parameters is summarized in Table II. Baseline design shows a single set of design parameters because it uses the same hardware for all the layers.

Architecture	Design Parameters (L1 / L2 / L3)				LUT / FF / DSP / BRAM / URAM (%)	Freq. (MHz)	Kernel (ms)	Kernel × DSP
	$SIMD_{FT}$	$SIMD_{Agg}$	DF	P				
Baseline	16	32	8	—	9.8 / 7.7 / 7.4 / 6.8 / 0	265	0.599 (1×)	4.46 (1×)
+Inter-Layer Pipeline	32/16/16	32/32/16	8/8/8	—	14 / 12 / 18 / 3.6 / 2.5	271	0.383 (1.56×)	6.74 (0.66×)
+Extended Sparsity	32/32/16	32/32/16	2/1/1	8/2/2	4.8 / 6.0 / 4.4 / 4.8 / 3.1	300	0.264 (**2.27**×)	1.15 (**3.88**×)

TABLE III: Properties of the FPGAs used in this paper.

Platform	BRAM (Mb)	LUT (K)	FF (K)	DSP	URAM (Mb)	Max BW (GB/s)
KU15P	34.6	523	1045	1968	36	19.2
U50	47.3	872	1743	5952	180	316
U280	70.9	1304	2607	9024	270	420

C. Impact of GCN Architecture Optimizations

1) Inter-Layer Pipelining:

Table IV shows the resource usage and performance of the StreamGCN architecture when accelerating three GCN layers of SimGNN on the U280 FPGA. The baseline uses the same hardware for all the GCN layers. With inter-layer pipeline added, all 3 GCN layers run in parallel as a coarse-grained pipeline. Since each layer utilizes different pieces of hardware, we can customize the design parameters to match the throughput of each layer. As a result, the 3 layers require 2.4x more DSPs compared with the baseline. We distribute the storage units needed between BRAM, URAM, and LUT to obtain a better frequency. The GCN kernel time is reduced by 36% with inter-layer pipelining added to the baseline. However, if we look at the latency-area product metric, i.e., KernelxDSP, we can see that the performance improvement does not catch up with the computation units (DSP) increment, suggesting the potential for further optimizations.

2) In Situ Sparsity for Feature Transformation Step:

Although using *P* queues (Section IV-D) helps the arbiter fetch non-zero elements more frequently, it may still not be enough to dispatch data to all the *DF* PEs. Furthermore, by increasing the *DF*, we may need to insert more bubbles in the pipeline to avoid RAW dependency since it reduces the number of cycles between the updates to the same location. As a result, there is a trade-off in choosing the right *DF* for each layer. The best parallelization factors are summarized in Table IV. When DF is set to 1, we no longer need to have separate banks in the row dimension of the buffers which can lessen the number of needed memory blocks. This makes it more efficient to use dense memory blocks (BRAM and URAM) as opposed to LUTs for the buffers. As Table IV shows, extending sparsity to Feature Transformation step over inter-layer pipeline has further reduced the kernel time by 31%, while decreasing the DSP usage by 4.09x. The results clearly suggest that, since this is a memory-bounded application, throwing more resources to the architecture is not helpful. Instead, the memory access latency should be reduced, and the computation units shall be used more efficiently. Since a large number of zero elements and the required DSPs are excluded, there is a 2.27x speedup over the baseline and the latency-area metric (KernelxDSP) is greatly improved by 3.88x.

D. End-to-end Acceleration of SimGNN

1) Flexibility of Mapping to Different FPGAs:

We implement the whole pipeline of SimGNN on 2 HBM FPGAs and KU15P that uses DDR memory. Fig. 7 compares the resource breakdown of the modules at the top hierarchy of our design when mapped to U280. We allocate most of the resources to the GCN stage as it is the computation-intensive part of the network. Table V shows the resource usage and performance for the three FPGA platforms. We can see that the kernel runs faster on HBM FPGAs compared to KU15P. This is mainly due to the fact that HBM FPGAs can achieve a better frequency as they have more resources and the Vitis tool has more freedom in placement and routing (PnR) to optimize the timing. The fact that the multiplication and addition units have different latencies on these boards further increases this difference. In fact, the cycle count of the same kernel when it uses different types and number of banks for global memory is almost the same. This suggests that after our optimizations the bottleneck is no longer at the memory level.

TABLE V: Performance and resource utilization of a StreamGCN design accelerating SimGNN on different target FPGAs.

FPGA	LUT / FF / DSP /BRAM/ URAM (%)	Freq. (MHz)	Kernel (ms)	E2E (ms)	E2E (query /s)
KU15P	34 / 29 / 35 / 30 / 23	201	0.786	1.135	881
U50	17 / 16 / 12 / 16 / 4.7	279	0.423	0.538	1858
U280	11 / 10 / 7.7 / 10 / 3.1	290	0.327	0.509	1965

E2E: End-to-End

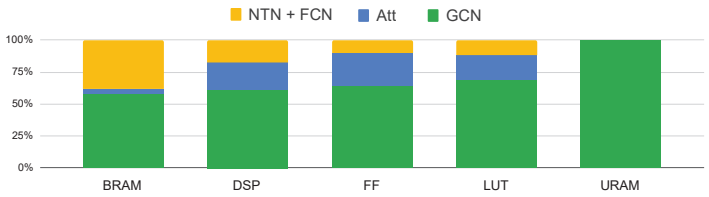

Fig. 7: Resource breakdown of the SimGNN accelerator on U280.

2) StreamGCN vs CPU and GPU:

We test the performance of the whole pipeline of SimGNN on the CPU and GPU described in Section VI-B. In this section, we are assuming that the inputs are already stored in the host memory, and we want to offload the graph comparison queries to either of the target platforms. The goal is to compare the performance of these platforms for processing a graph matching query. Table VI summarizes the average runtime per query. The queries are started sequentially, and the end-to-end time of all the platforms is the time interval between two consecutive queries are started. This contains the runtime for any pre-processing steps as well. For FPGA and GPU, it also includes the host-kernel communication via the PCIe link, writing data to FPGA/GPU's global memory, reading the results from that, and the overheads for using the APIs (OpenCL for FPGA and PyTorch for CPU/GPU). We use the end-to-end time for comparison since these overheads are inevitable and should be

accounted for. The kernel time on CPU/GPU is measured with the PyTorch profiler.

The results demonstrate that our FPGA solution can outperform both CPU and GPU significantly. As discussed in Section I, this is partly because of the dynamic load balance and the irregular memory access of the graph structure. Furthermore, since we target small graphs, it results in extreme underutilization of GPU. In fact, the profiling results indicate that the GPU utilization does not go higher than 6% and, for the most part, the PyG-GPU only uses 1 streaming multiprocessor (SM) since the matrices are small. Because of this and the fact that GPU runs at a lower frequency (1.3GHz) compared to CPU (2.2 GHz), the GPU version of this application is even slower than the CPU. The *nvprof* profiling results show that PyG-GPU runs 225 kernels for accelerating this application that on average have 4.6 *KFLOPs*. With this low computation intensity, the overhead of running the kernel (such as *cudaLaunchKernel*) is larger than the actual kernel runtime that greatly impacts the GPU performance. Designing the GPU kernel manually can alleviate some of these shortcomings, but the underlying problem still exists due to the coarse-grained execution model of GPU. In contrast, our FPGA solution suffers from the kernel initialization overheads only once since we develop a deep pipeline across all stages of the computation by fusing them in one kernel. This pipelining has several other benefits as explained in Section IV-C. Note that both FPGA and GPU have enough resources left for batch processing, so it is meaningful to compare their single query execution.

TABLE VI: Performance comparison of running SimGNN on different hardware platforms.

Platform	Max BW (GB/s)	Kernel (ms)	E2E (ms)	Speedup (vs. CPU)	Speedup (vs. GPU)
KU15P	19.2	0.786	1.135	8.2	12.1
U50	316	0.423	0.538	17.2	25.5
U280	460	**0.327**	**0.509**	**18.2**	**26.9**
PyG-CPU	76.8	5.85	9.27	1	1.5
PyG-GPU	900	9.68	13.7	0.68	1

BW: Bandwidth, E2E: End-to-End

3) Discussion on Scalability:
Table V illustrates that the available resources allow us to instantiate 6 StreamGCN pipelines with U280 before hitting the 80% resource usage upper-bound. 80% is an empirical threshold that beyond that the Xilinx tool would have a hard time mapping the design to the FPGA. Since U280 is equipped with HBM which makes use of pseudo channels (PCs) that can be accessed *independently*, this batch processing can be done completely in parallel. This does not change the latency of each graph query, but it would increase the throughput by 6x. In addition, although the graphs in our target benchmark have 25.6 nodes on average and we designed our accelerator for them, we can use the unused resources for increasing the target graph size or processing more GCN layers. Obviously, increasing either of the batch number, graph size, and number of GCN layers limits the other values. If all the other options are fixed, we can increase these three parameters by 6, 150, and 20, respectively, when targeting the U280 board for SimGNN.

VII. Other Related Works

SpMM and SCNN Accelerators: We reviewed the related works which propose an accelerator for GCN in Section III. Apart from the works focusing on GCN, there has been a lot of research on sparse MM either for pruned CNNs or normal MM [12], [15], [20], [26], [27]. They all rely on the fact that the sparse matrix is known offline, and they can pre-process it. For example, EIE [12] propose a sparse matrix vector multiplier for the fully connected layers. It reorganizes the sparse matrix in compressed sparse column (CSC) format and pre-loads that into on-chip memory. As another example, Kung et al. [15] pre-process the data by merging multiple sparse columns of the weight matrix into one and pruning all the weights except for the most-significant ones. resulting in some accuracy loss. These approaches are not feasible for GCN in which the sparse matrix (i.e., the node embeddings) is generated *while running* the algorithm; whereas we proposed a technique to prune the zeros *on-the-fly*.

VIII. Conclusion

In this paper, we analyzed and examined the optimization opportunities when GCN is applied to small graphs. We presented an efficient architecture, StreamGCN, and developed an accelerator for SimGNN based on that as an end-to-end application, which demonstrated significant speedup over the CPU and GPU results. StreamGCN is ideal for real-time or near real-time graph search and similarity computation for many biological, chemical, or pharmaceutical applications. The computation disparity existing in the network calls for a customized accelerator. Besides, since the GPU has coarse-grained execution, we cannot have improvement beyond the optimizations applied for each phase since different phases are executed separately. However, on the FPGA side, we can exploit a deep pipeline across the phases by enabling a dataflow architecture. Not only does it help us reduce the global memory transactions, but we can also eliminate the overhead of running different kernels. Furthermore, we showed that since this is a memory-bounded application, instantiating many computation units (as in GPU) is not beneficial. Due to these optimizations, the experimental results demonstrate that StreamGCN can outperform CPU and GPU by 18.2x and 26.9x, respectively.

Acknowledgments

The majority of this work was conducted while the first author was interning at Samsung Semiconductor Inc. It is also supported by the CAPA award jointly funded by NSF (CCF1723773) and Intel (36888881), the RTML award funded by NSF (CCF-1937599), and CDSC industrial partners[1].

References

[1] Y. Bai *et al.*, "SimGNN: A neural network approach to fast graph similarity computation," in *WSDM*, 2019.
[2] Y. Bai *et al.*, "Learning-based efficient graph similarity computation via multi-scale convolutional set matching," in *AAAI*, 2020.
[3] E. E. Bolton *et al.*, "PubChem: integrated platform of small molecules and biological activities," in *Annual reports in computational chemistry*. Elsevier, 2008.
[4] G. Chen *et al.*, "Alchemy: A quantum chemistry dataset for benchmarking ai models," *arXiv preprint arXiv:1906.09427*, 2019.
[5] G. Dai *et al.*, "ForeGraph: Exploring large-scale graph processing on multi-FPGA architecture," in *FPGA*, 2017.
[6] P. Dosch *et al.*, "Report on the second symbol recognition contest," in *International Workshop on Graphics Recognition*. Springer, 2005.
[7] D. K. Duvenaud *et al.*, "Convolutional networks on graphs for learning molecular fingerprints," in *NeurIPS*, 2015.

[1] https://cdsc.ucla.edu/partners/

[8] M. Fey *et al.*, "Fast graph representation learning with PyTorch geometric," *arXiv preprint arXiv:1903.02428*, 2019.

[9] T. Geng *et al.*, "AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing," in *MICRO*. IEEE, 2020.

[10] W. Haaswijk *et al.*, "Deep learning for logic optimization algorithms," in *ISCAS*. IEEE, 2018.

[11] T. J. Ham *et al.*, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *MICRO*, 2016.

[12] S. Han *et al.*, "EIE: Efficient inference engine on compressed deep neural network," *ACM SIGARCH Computer Architecture News*, 2016.

[13] S. Ishida *et al.*, "Graph neural networks with multiple feature extraction paths for chemical property estimation," *Molecules*, 2021.

[14] T. N. Kipf *et al.*, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[15] H. Kung *et al.*, "Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization," in *ASPLOS*, 2019.

[16] Y. Li *et al.*, "Graph matching networks for learning the similarity of graph structured objects," in *ICML*, 2019.

[17] S. Liang *et al.*, "ENGN: A high-throughput and energy-efficient accelerator for large graph neural networks," *IEEE TC*, 2020.

[18] H. Ma *et al.*, "Multi-view graph neural networks for molecular property prediction," *arXiv preprint arXiv:2005.13607*, 2020.

[19] NCI/NIH, "Aids antiviral screen data." 2004. [Online]. Available: https://wiki.nci.nih.gov/display/NCIDTPdata/AIDS+Antiviral+Screen+Data

[20] A. Parashar *et al.*, "SCNN: An accelerator for compressed-sparse convolutional neural networks," *ACM SIGARCH Computer Architecture News*, 2017.

[21] Z. Qin *et al.*, "GHashing: Semantic graph hashing for approximate similarity search in graph databases," in *KDD*, 2020.

[22] K. Riesen *et al.*, "IAM graph database repository for graph based pattern recognition and machine learning," in *Joint SPR and SSPR*, 2008.

[23] B. Rozemberczki, "A PyTorch Implementation of SimGNN." 2022. [Online]. Available: https://github.com/benedekrozemberczki/SimGNN

[24] Y. Shen *et al.*, "Escher: A CNN accelerator with flexible buffering to minimize off-chip transfer," in *FCCM*, 2017.

[25] A. Sohrabizadeh *et al.*, "End-to-end optimization of deep learning applications," in *FPGA*, 2020.

[26] N. Srivastava *et al.*, "MatRaptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *MICRO*, 2020.

[27] N. Srivastava *et al.*, "Tensaurus: A Versatile Accelerator for Mixed Sparse-Dense Tensor Computations," in *HPCA*, 2020.

[28] T. Sterling *et al.*, "Zinc 15–ligand discovery for everyone," *Journal of chemical information and modeling*, 2015.

[29] X. Wang *et al.*, "An efficient graph indexing method," in *ICDE*, 2012.

[30] Y. Wang *et al.*, "Processor assisted worklist scheduling for FPGA accelerated graph processing on a shared-memory platform," in *FCCM*, 2019.

[31] X. Wei *et al.*, "TGPA: tile-grained pipeline architecture for low latency CNN inference," in *ICCAD*, 2018.

[32] Z. Wu *et al.*, "MoleculeNet: a benchmark for molecular machine learning," *Chemical science*, 2018.

[33] Xilinx, "Vivado design suite user guide: high-level synthesis," 2021. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf

[34] M. Yan *et al.*, "HyGCN: A GCN accelerator with hybrid architecture," in *HPCA*. IEEE, 2020.

[35] P. Yanardag *et al.*, "Deep graph kernels," in *SIGKDD*, 2015.

[36] Y. Yang *et al.*, "GraphABCD: Scaling Out Graph Analytics with Asynchronous Block Coordinate Descent," in *ISCA*, 2020.

[37] H. Zeng *et al.*, "GraphACT: Accelerating GCN training on CPU-FPGA heterogeneous platforms," in *FPGA*, 2020.

[38] B. Zhang *et al.*, "BoostGCN: A framework for optimizing GCN inference on FPGA," in *2021 IEEE International Symposium on FCCM*, 2021.

[39] X. Zhang *et al.*, "DnnBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs," in *ICCAD*, 2018.

[40] Intel, "Intel MKL" 2022. [Online]. Available: https://software.intel.com/en-us/mkl

[41] NVIDIA, "NVIDIA cuBLAS" 2022. [Online]. Available: https://developer.nvidia.com/cublas