GCNAX: A Flexible and Energy-efficient Accelerator for Graph Convolutional Neural Networks

Jiajun Li*, Ahmed Louri*, Avinash Karanth†, Razvan Bunescu‡
*Department of Electrical and Computer Engineering, George Washington University, Washington, D.C.
†School of Electrical Engineering and Computer Science, Ohio University, Athens, Ohio
‡Department of Computer Science, University of North Carolina at Charlotte, Charlotte, North Carolina
Email: {lijiajun, louri}@gwu.edu, karanth@ohio.edu, rbunescu@uncc.edu

***Abstract*—Graph convolutional neural networks (GCNs) have emerged as an effective approach to extend deep learning for graph data analytics. Given that graphs are usually irregular, as nodes in a graph may have a varying number of neighbors, processing GCNs efficiently pose a significant challenge on the underlying hardware. Although specialized GCN accelerators have been proposed to deliver better performance over generic processors, prior accelerators not only under-utilize the compute engine, but also impose redundant data accesses that reduce throughput and energy efficiency. Therefore, optimizing the overall flow of data between compute engines and memory, i.e., the GCN dataflow, which maximizes utilization and minimizes data movement is crucial for achieving efficient GCN processing.**

In this paper, we propose a flexible and optimized dataflow for GCNs that simultaneously improves resource utilization and reduces data movement. This is realized by fully exploring the design space of GCN dataflows and evaluating the number of execution cycles and DRAM accesses through an analysis framework. Unlike prior GCN dataflows, which employ rigid loop orders and loop fusion strategies, the proposed dataflow can reconfigure the loop order and loop fusion strategy to adapt to different GCN configurations, which results in much improved efficiency. We then introduce a novel accelerator architecture called GCNAX, which tailors the compute engine, buffer structure and size based on the proposed dataflow. Evaluated on five real-world graph datasets, our simulation results show that GCNAX reduces DRAM accesses by a factor of 8.1× and 2.4×, while achieving 8.9×, 1.6× speedup and 9.5×, 2.3× energy savings on average over HyGCN and AWB-GCN, respectively.

***Index Terms*—Graph Convolutional Neural Networks, Dataflow Accelerators, Domain-specific Accelerators**

I. Introduction

Deep learning has achieved great success in a broad range of applications, such as object detection [1], [2] and machine translation [3], [4]. The image and text data in these tasks are typically represented as vectors in the Euclidean space. However, there is an increasing number of applications where data is relational, for which graphs provide a more natural representation. Therefore, many studies have extended deep learning approaches to graphical representations [5]–[8]. Graph Convolutional Neural Networks (GCNs) [8], [9] are one such category of models that re-define the notion of *convolution* for graph data. Recently, GCNs have attracted substantial efforts from both the industrial and academic communities as a model of choice for applications including e-commerce analysis [10], social network analysis [11], and molecular bioactivity identification in medical research [12].

Inference in any modern GCN requires traversing hundreds to thousands of vertices and edges, posing a major challenge to the hardware platforms. Typically, the convolutional layers occupy the majority of execution time in GCNs [13], [14] through two primary phases: *Aggregation* and *Combination*. The computation pattern of the combination phase is similar to that of conventional neural networks. However, the aggregation phase relies on the graph structure, which is often sparse and irregular. This difference imposes new requirements on the design of GCN architectures. Specifically, it requires the GCN accelerators to simultaneously alleviate the irregularity in the aggregation phase and exploit the regularity in the combination phase. Unfortunately, existing graph analytic and neural network accelerators [15], [16] cannot handle the hybrid execution patterns because they are optimized to either alleviate irregularity or exploit regularity in isolation, rather than to simultaneously address both patterns.

To this end, a few dedicated GCN accelerators have been recently proposed in the literature, delivering substantial gains in performance and energy efficiency [17], [18]. HyGCN [17] exploits two dedicated compute engines, i.e., an aggregation engine and a combination engine, to accelerate the *Aggregation* and *Combination* phases, respectively. However, by rigidly following the two phases, HyGCN suffers from two main drawbacks: (i) an inefficient execution order that results in more computations and data accesses, and (ii) under-utilization of compute engines because of workload imbalance between the two engines. As some datasets have more computations during the aggregation phase than the combination phase, the early completing engine has to be idle waiting for the slower engine to complete. On the other hand, AWB-GCN [18] is an architecture for accelerating GCNs and Sparse-dense Matrix Multiplication (SpMM) kernels, and addresses the issue of workload imbalance in processing real-world graphs. However, AWB-GCN incurs its own inefficiency because the loop optimization techniques are not carefully tailored to the GCN processing, resulting in redundant data accesses.

In this paper, we propose GCNAX, a novel accelerator architecture whose dataflow is optimized for throughput and energy efficiency for GCN acceleration. To find the optimal dataflow with maximum utilization and energy efficiency, we perform an extensive design space exploration that enumerates the legal design variants of GCN dataflows. Although prior

GCN accelerators have demonstrated better performance over generic processors, it is hard to compare the efficiency of these implementations because of differences in technology, hardware resources and system setup. To address this problem, we first categorize these implementations into various dataflows according to three optimization techniques: 1) loop interchange; 2) loop tiling; and 3) loop fusion. We then evaluate the throughput and energy efficiency of these dataflows under the same hardware resource constraints such as buffer size and the number of ALUs.

We observed that different GCN configurations require different design choices of the dataflow to achieve optimal efficiency. Therefore, we design a flexible and optimized dataflow that can reconfigure the loop order and loop fusion techniques to adapt to different GCN configurations. We then propose a hardware accelerator called GCNAX to support the flexible dataflow. GCNAX employs an outer-product based method for SpMM [19], which is the key computation pattern in GCNs, to mitigate the workload imbalance caused by the unbalanced distribution of zeros. Furthermore, the compute engine, buffer size and structure are tailored according to the execution order and tile sizes of the dataflow. We implement the GCNAX accelerator in RTL targeting TSMC 40*nm* library. We also build a cycle-accurate simulator that models the microarchitectural behavior of each module while supporting different dataflows. Evaluated on five real-world graph datasets, our simulation results show that GCNAX reduces DRAM accesses by a factor of 8.1× and 2.4× compared to HyGCN and AWB-GCN. On average, GCNAX achieves 8.9×, 1.6× speedup and 9.5×, 2.3× energy savings over HyGCN and AWB-GCN, respectively.

This paper makes the following contributions:

- To find the optimal dataflow with maximum utilization and energy efficiency, we perform an extensive design space exploration that enumerates the legal design variants by considering various optimization techniques.
- We propose a flexible and optimized dataflow that can reconfigure the loop order and loop fusion strategy to adapt to different GCN configurations, and we design a novel accelerator architecture called GCNAX for the proposed dataflow.
- We implement the proposed accelerator in both RTL and a cycle-accurate simulator, and present a thorough evaluation on real-world graph datasets.

II. Background

A. GCN Basics

Recently, there is an increasing interest in extending deep learning approaches for graph data. Graph Neural Networks (GNNs) are deep learning models aiming at addressing graph-related tasks in an end-to-end manner. GNNs can be categorized into recurrent graph neural networks (RecGNNs) [20], graph convolutional neural networks (GCNs), graph autoencoders (GAEs) [21], and spatial-temporal graph neural networks (STGNNs) [22]. Among these, GCNs have seen a rapid development recently due to their attractive efficiency for neural-network-based graph processing.

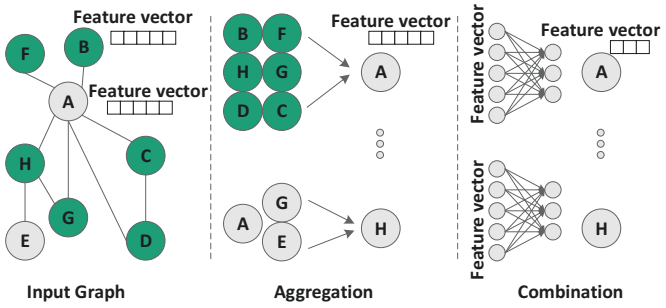


Fig. 1: Illustration of the GCN model.

TABLE I. Notations in GCNs.

Term	Meaning
G	graph $G=(V,E)$
V	vertices of G
E	edges of G
D_v	degree of vertex v
$e_{(i,j)}$	edge between vertex i and j
$N(v)(S(v))$	(sampling subset of) neighbor set of v
$A(A_{ij})$	(element of) adjacency matrix
a_v	aggregation feature vector of v
h_v	feature vector of v
b	combination bias vectors
X	feature matrix composed by feature vectors

GCNs follow the neighborhood aggregation scheme, where the feature vector of each vertex is computed by recursively aggregating and transforming the representation vectors of its neighbor vertices. Fig. 1 illustrates the main execution phases of the convolutional layer in GCN models. Table I lists the notations used in GCNs. In this work, we focus on undirected graphs and the inference stage.

Typically, the layer-wise forward propagation of GCN is formulated as:

$$\begin{aligned} a_v^k &= Aggregate(h_u^{(k-1)} : u \in \{N(v)\} \cup \{v\}), \\ h_v^k &= Combine(a_v^k). \end{aligned} \tag{1}$$

where h_v^k is the representation feature vector of vertex v at the k-th layer. Simply, the Aggregate function aggregates multiple feature vectors from source neighbors to one single feature vector, and the Combine function transforms the feature vector of each vertex to another feature vector using a multi-layer perceptron (MLP) neural network.

B. GCN Models

Here we introduce several typical GCN models as examples to explain the above operations in detail.

GCN [8] is one of the most successful convolutional networks for graph learning, which bridges the gap between spectral-based convolutions and spatial-based convo-

TABLE II. Structure and data density of the 2-layer GCN for the graph datasets.

Datasets		Cora	Citeseer	Pubmed	Nell	Reddit
Structure	#Vertex	2,708	3,327	19,717	65,755	232,965
	#Edge	10,556	9,104	88,648	266,144	114,615,892
	Feature length	1433-16-7	3703-16-6	500-16-3	61,278-64-186	602-64-41
Data density	A	0.18%	0.11%	0.028%	0.0073%	0.21%
	X	1.27%, 78.0%	0.85%, 89.1%	10.0%, 77.6%	0.011%, 86.4%	51.6%, 60.0%
	W	100%, 100%	100%, 100%	100%, 100%	100%, 100%	100%, 100%

lutions. Its inference model can be described as Equation (2).

$$\begin{cases} a_v^k = \sum_{u \in N(v) \cup \{v\}} \frac{h_u^{(k-1)}}{\sqrt{D_u \cdot D_v}}, \\ h_v^k = ReLU(W^k a_v^k + b^k). \end{cases} \quad (2)$$

$$\begin{cases} a_v^k = \sum_{u \in S(v) \cup \{v\}} h_u^{(k-1)} / r, \\ h_v^k = ReLU(W^k a_v^k + b^k). \end{cases} \quad (3)$$

GraphSage [23] further adopts sampling to obtain a fixed number of neighbors for each node. It performs graph convolution by Equation (3) where r is the number of neighbors sampled for each node.

Graph Isomorphism Network (**GIN**) [24] adjusts the weight of the central node by a learnable parameter ε_k. It performs graph convolution by

$$\begin{aligned} a_v^k &= (1+\varepsilon_k) \cdot h_v^{k-1} + \sum_{u \in N(v)} h_u^{(k-1)}, \\ h_v^k &= MLP^k(a_v^k, W^k, b^k). \end{aligned} \quad (4)$$

From the above analysis, the main computation of the GCN models can be abstracted as:

$$X^{(k+1)} = \sigma(\hat{A} X^{(k)} W^{(k)}) \quad (5)$$

For GCN, $\hat{A}$ is the normalized adjacency matrix: $\hat{A} = D^{-\frac{1}{2}} \times (A+I) \times D^{\frac{1}{2}}$, where I is the identity matrix, and D is a diagonal matrix in which D_{ii} equals the degree of vertex i, X is the feature matrix where each row represents a feature vector of a vertex. For GraphSage, $\hat{A}$ is the sampled adjacency matrix with a scaling factor of $1/r$. For GIN, $\hat{A} = (A + \varepsilon_k \times I)$ where I is the identity matrix. This abstraction will guide us to design efficient GCN accelerators. Note that since $\hat{A}$ can be computed offline from A, we hereafter use A to denote the normalized $\hat{A}$.

C. Graph Datasets

As graph-structured data is ubiquitous, GCNs have a wide variety of applications. Here we provide the benchmark datasets frequently used in the literature [25]. Cora, Citeseer and Pubmed are three popular datasets for paper citation networks [26]. The Cora dataset contains 2708 machine learning publications grouped into seven classes. The Citeseer dataset contains 3327 scientific papers grouped into six classes. The Pubmed dataset [23] contains 19717 diabetes-related publications. In the social network domain, the Reddit dataset is an undirected graph formed by posts collected from the Reddit discussion forum. The Nell dataset [27] is a knowledge graph obtained from the Never-Ending Language Learning project. It consists of facts represented by a triplet which involves two entities and their relation. Table II lists the structure and data density of the datasets, which will be used as benchmarks in this paper.

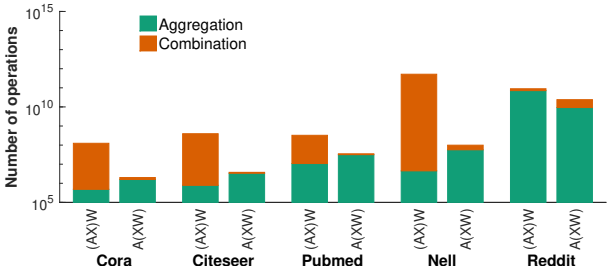


Fig. 2: The number of operations for the five datasets (first layer) using the two execution orders.

```
for(n=0; n<N; n++) {                    SpMM1:B=XW
  for(c=0; c<C; c++) {
    for(k=0; k<K; k++) {
      B[n][c]+=X[n][k]*W[k][c];
}}}

for(m=0; m<M; m++) {                    SpMM2:O=AB
  for(c=0; c<C; c++) {
    for(n=0; n<N; n++) {
      O[m][c]+=A[m][n]*B[n][c];
}}}
```

Fig. 3: Pseudo code of the chain SpMM in GCNs.

III. Accelerator Design Exploration

In this section, we first introduce the design choices and challenges of GCN dataflows, then we present the optimization techniques and explore the design space. Finally, we present the optimized dataflow based on the design space exploration.

A. Execution Order

As introduced in Section II, the key computation pattern in GCNs is a two-phase matrix-matrix multiplication $A \times X \times W$. There are two alternative execution orders: $(A \times X) \times W$ and $A \times (X \times W)$. The difference between the two orders is significant because: 1) the matrix chain problem [28], which indicates that the order in which the product is parenthesized affects the number of simple arithmetic operations needed to compute the product; and 2) the sparsity of the matrices also affects the efficiency of different orders. As shown in Table II, A is ultra-large and sparse, X is moderate sparse, while W is generally small and dense.

To determine which execution order is better, we profiled the number of effective operations for the five graph datasets under the two orders, as shown in Fig. 2. Note that we only count the operations with two non-zero operands. The results show that using $A(XW)$ reduces operations by a factor of 32× on average

Fig. 4: Pseudo code of tiled chain SpMM: (a) without loop fusion (b) with loop fusion. $< T_{n0}, T_{c0}, T_k, T_m, T_{c1}, T_{n1} >$ is the tile size tuple for the loop tiling.

compared to $(AX)W$. The results suggest that performing XW first would potentially lead to higher throughput.

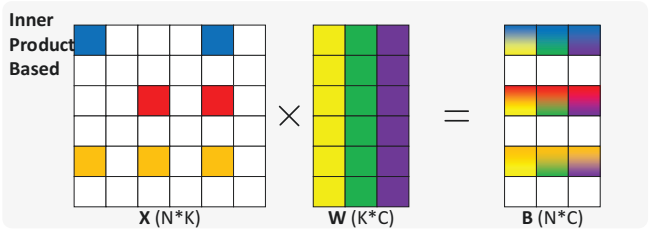
Moreover, performing XW first has another benefit that the matrix-matrix multiplication will be both in sparse-dense format, which enables us to design a unified processing element (PE) architecture for the computations. By contrast, AX requires sparse-sparse matrix multiplication and generates a large and dense matrix, then $(AX)W$ requires dense-dense matrix multiplication. Therefore, it requires two individual PEs for the computation, which is the design paradigm of HyGCN. HyGCN exploits an Aggregation Engine which realizes the efficient execution of irregular data accesses and computations in sparse-sparse matrix multiplication, and a Combination Engine to maximize the efficiency of regular data accesses and computations in dense-dense matrix multiplication. Although such a tandem-engine architecture is intuitive and effective for specific datasets, it will incur under-utilization of either Aggregation Engine or Combination Engine because of the workload imbalance between the two engines. Specifically, some datasets have more computations in Aggregation phase while others have more in the Combination phase. As shown in Fig. 2, when using the $(AX)W$ order, there are more operations in the Combination phase in Citeseer, Cora, Pubmed and Nell than the Aggregation phase, which is opposite in Reddit. This indicates that the former four datasets require a relatively more powerful Aggregation Engine, but Reddit needs a more powerful Combination Engine. Due to the fixed computational capacity of the two engines in tandem-engine architectures, it will inevitably be under-utilized for some datasets. Fortunately, performing XW first and using a unified PE for both phases will naturally avoid this workload imbalance problem.

Therefore, in our accelerator we will perform XW first to 1) reduce the total number of operations; and 2) use a unified PE architecture to avoid under-utilization resulting from inter-engine workload imbalance.

B. Chain Sparse-dense Matrix Multiplication

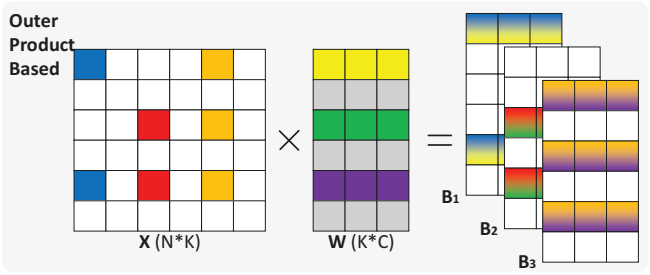
Overall, the key computation pattern of GCNs is abstracted as chain sparse-dense matrix multiplication (chain SpMM). Many previous papers have proposed specialized dataflows for SpMM on various platforms, including GPUs [29], FPGAs [30], and ASICs [19], [31]. However, due to differences in matrix dimensions, sparsity levels, size of the chain, technology, hardware resources and system setup, a direct comparison between different implementations does not provide much insight into the relative efficiency of different dataflows. Therefore, to compare these implementations, we categorize their dataflows based on their data handling characteristics. Then, we build an analysis framework to analyze the throughput and data accesses of these dataflows.

We start the analysis of the dataflows from the pseudo code shown in Fig. 3. We assume that $A \in \mathbb{R}^{M \times N}$, $X \in \mathbb{R}^{N \times K}$, $W \in \mathbb{R}^{K \times C}$. Matrix $B \in \mathbb{R}^{N \times C}$ is the intermediate result of $X \times W$ and $O \in \mathbb{R}^{M \times C}$ is the final output matrix. The code in Fig. 3 can be optimized using three techniques. First, loop tiling can be applied for each SpMM to leverage data locality (Fig. 4), where $< T_{n0}, T_{c0}, T_k, T_m, T_{c1}, T_{n1} >$ is the tile size tuple. Second, loop interchange [32] enables different types of data reuse to reduce external memory traffic by exchanging the order of the nested loops. Third, loop fusion optimization [33] can be leveraged




```
// on-chip computation
for(tn0=n0; tn0<min(n0+Tn0,N); tn0++) {
  for(tc0=c0; tc0<min(c0+Tc0,C); tc0++) {
    for(tk=k; tk<min(k+Tk,K); tk++) {
#pragma HLS UNROLL
   L:  B[tn0][tc0]+=X[tn0][tk]*W[tk][tc0];
}}}
```

(a)




```
// on-chip computation
for(tn0=n0; tn0<min(n0+Tn0,N); tn0++) {
  for(tk=k; tk<min(k+Tk,K); tk++) {
  if(X[tn0][tk]==0)
    continue;
  else
    for(tc0=c0; tc0<min(c0+Tc0,C); tc0++) {
#pragma HLS UNROLL
      L:  B[tn0][tc0]+=X[tn0][tk]*W[tk][tc0];
    }
  endif
}}
```

(b)

Fig. 5: On-chip computation optimization: (a) Inner-product based method (b) Outer-product based method.

to reduce data transfer of intermediate data. Specifically, since the result of SpMM1 is then used as the input of SpMM2, we can fuse the processing of SpMM1 and SpMM2 to enable caching of the intermediate data and eliminate transferring the intermediate data to off-chip DRAM. According to these techniques, we present the taxonomy of dataflows according to: 1) loop orders; 2) selection of tile size tuple; and 3) loop fusion strategies of the chain SpMM. Notably, as shown in Fig. 4(b), the tile size tuple when enabling loop fusion is $<T_{n0}, T_{c0}, T_k, T_m>$, which doesn't contain T_{n1} and T_{c1} because of loop fusion. To keep consistency, we virtualize the tile sizes T_{n1} and T_{c1} for loop fusion with the constraint $T_{n1} = T_{n0}, T_{c1} = T_{c0}$, which facilitates the analysis for the dataflows.

C. Computation Optimization

Loop unrolling can be used to increase the utilization of massive computation resources. Researchers have extensively studied the methods to unroll SpMM for parallel computations. Fig. 5 shows two of these methods taking the SpMM1 in Fig. 4(a) as an example. The first one is inner-product based methods [34], which uses the loop nest $t_{n0} \rightarrow t_{c0} \rightarrow t_k$ (the leftmost iterator denotes the outermost loop) that maximizes the reuse of output matrix. The t_k dimension is unrolled for parallel computation. Inner-product based method takes advantage of data locality for large dense matrices, however, it incurs inefficiency for SpMM since the run-time is dominated by irregular memory accesses and index-matching in order to perform the inner product operations. Furthermore, inner-product based methods would also incur workload imbalance problem with the increasing of multiplier numbers [31].

In this paper, we use outer-product based method [19] as shown in Fig. 5(b). It employs the loop nest $t_{n0} \rightarrow t_k \rightarrow t_{c0}$ and unrolls the t_{c0} dimension. Although this method would ruin the reuse of the output matrix, it has better input matrix reuse compared to the inner-product based method. More importantly, it well supports the elimination of zero computations and avoids the workload imbalance problem. We store the sparse matrix in Compressed Sparse Column (CSC) format, while the input DenseMat is stored in dense format in row-major order. Since $X[t_{n0}][t_k]$ is the input operand for all the T_{c0} multiplications, these computations can be skipped simultaneously if $X[t_{n0}][t_k]$ is zero. Note that we keep the redundant "if" statement in Fig. 4(b) only for clarity, and we don't need to implement the "compare" logic in hardware because the zeros are already removed by CSC format. Similarly, the corresponding loop order for the computation int SpMM2 is $t_m \rightarrow t_{n1} \rightarrow t_{c1}$.

Performance Modeling. Given a specific tile size tuple, the total number of execution cycles for the chain SpMM is modeled as follows.

稀疏矩阵用CSC存储，密集矩阵用Row-major Dense format存储

$$N_{cycles} = N_{cycle1} + N_{cycle2} \tag{6}$$

where

$$\begin{aligned} N_{cycles1} &= \gamma_X \times \lceil \frac{N}{T_{n0}} \rceil \times \lceil \frac{C}{T_{c0}} \rceil \times \lceil \frac{K}{T_k} \rceil \times (T_{n0} \times T_k) \\ N_{cycles2} &= \gamma_A \times \lceil \frac{M}{T_m} \rceil \times \lceil \frac{C}{T_{c1}} \rceil \times \lceil \frac{N}{T_{n1}} \rceil \times (T_m \times T_{n1}) \end{aligned} \tag{7}$$

γ_X and γ_A denote the density of the matrix X and A, respectively.

D. Memory Access Optimization

Fig. 4 also illustrates the external data transfer operations of the chain SpMM. The data elements of input/output matrices in each SpMM are loaded before the compute engine starts computation and the generated output partial sums are written back to main memory. There are two methods to minimize the external data transfer. The first method is local memory promotion. Specifically, if the innermost loop in the communication part is irrelevant to a matrix, i.e., the loop iterator does not appear in the access function of the matrix [35], there will be redundant memory operations between different

loop iterations. Local memory promotion can be used to reduce redundant operations. For example, in SpMM1 in Fig. 4(a), the innermost loop dimension k in SpMM1 is irrelevant to matrix B, so the memory accesses to matrix B can be promoted to outer loops. Since the loops can be permuted, we can change the loop order to reduce the memory accesses of the corresponding matrices.

The second method to reduce memory transfer operations is to fuse the processing of SpMM1 and SpMM2. As shown in Fig. 4(a), the elements of matrix B are stored back to DRAM in SpMM1 (line 14), and they are again fetched from DRAM in SpMM2 (line 20). Therefore, we can reduce the data transfer of these intermediate data by fusing the execution of SpMM1 and SpMM2. As illustrated in Fig. 4(b), when SpMM1 finishes the computation of loop k and generates a B chunk, we pause the execution of SpMM1 and proceed to the execution of SpMM2 (line 17). By doing so, the data transfer of the intermediate matrix (B) is eliminated. Notably, although loop fusion reduces data transfer of intermediate results, it somehow sacrifices the flexibility of loop order. Specifically, the iteration k in SpMM1 must be the innermost loop to ensure that matrix B finishes all its computations (not a PartialMat) before being forwarded to SpMM2. Moreover, as m becomes the innermost loop in the communication part of SpMM2, matrix O has to be frequently transferred between on-chip and off-chip. Since O is the result matrix, the volume of data transfer is doubled compared to the input matrix such as matrix A because the result matrix has to be written back to the main memory when being replaced, whereas the input matrix can be directly replaced without being written back.

Off-chip Data Access Modeling. Since off-chip data accesses usually dominate the energy consumption of accelerators [17], [36], reducing the number of data accesses will improve energy efficiency. The number of data accesses is analytically modeled using the design choices including the tile size tuple, loop order and loop fusion strategy. Since the design space is polyhedral, we use the design choices shown in Fig. 4(a) as an example to illustrate our model. The total number of off-chip accesses is calculated by:

$$N_d = \alpha_X \cdot S_X + \alpha_W \cdot S_W + \alpha_{B1} \cdot S_{B1} + \alpha_{B2} \cdot S_{B2} + \alpha_A \cdot S_A + \alpha_O \cdot S_O \tag{8}$$

where

$$\begin{cases} S_X = \gamma_X \times T_{n0} \times T_k \\ S_W = T_k \times T_{c0} \\ S_{B1} = T_{n0} \times T_{c0} \\ S_{B2} = T_{n1} \times T_{c1} \\ S_A = \gamma_A \times T_m \times T_{n1} \\ S_O = T_m \times T_{c1} \end{cases} \tag{9}$$

$$\begin{cases} \alpha_X = \alpha_W = \dfrac{N}{T_{n0}} \times \dfrac{C}{T_{c0}} \times \dfrac{K}{T_k} \\ \alpha_{B1} = \dfrac{N}{T_{n0}} \times \dfrac{C}{T_{c0}} \\ \alpha_{B2} = \alpha_A = \dfrac{M}{T_m} \times \dfrac{C}{T_{c1}} \times \dfrac{N}{T_{n1}} \\ \alpha_O = \dfrac{M}{T_m} \times \dfrac{C}{T_{c1}} \end{cases} \tag{10}$$

Here $\alpha_X, \alpha_W, \alpha_B, \alpha_A$ and S_X, S_W, S_B, S_A denote the trip counts and buffer sizes of memory accesses to matrix $X/W/B/A$ respectively. Note that $\alpha_{B1}, \alpha_{B2}, S_{B1}, S_{B2}$ are used to differentiate the accesses in SpMM1 and SpMM2 respectively. In this model, we assume that the zeros in matrix X and A are distributed evenly so we use the overall density of X and A (γ_X, γ_A) to represent the density of each chunk when estimating S_X and S_A, as shown in Equation (9). Although it does not reflect the actual distribution, we make this assumption for simplicity since considering the sparsity distribution would significantly increase the model complexity. Moreover, we found that the estimated values from the model deviate little from the actual values derived from a cycle-accurate simulation.

If changing the loop order in Fig. 4(a), we just need to modify the equation of $\alpha's$ in Equation (10). The enumeration of the loop orders and the corresponding equation of $\alpha's$ are omitted for brevity. When enabling loop fusion as shown in Fig. 4(b), the total number of off-chip accesses and the buffer size are also calculated by Equation (8) and Equation (9), but the trip counts are estimated as Equation (11).

$$\begin{cases} \alpha_X = \alpha_W = \dfrac{N}{T_{n0}} \times \dfrac{C}{T_{c0}} \times \dfrac{K}{T_k} \\ \alpha_{B1} = \alpha_{B2} = 0 \\ \alpha_A = \dfrac{M}{T_m} \times \dfrac{C}{T_{c0}} \times \dfrac{N}{T_{n0}} \\ \alpha_O = 2 \times \dfrac{M}{T_m} \times \dfrac{C}{T_{c0}} \times \dfrac{N}{T_{n0}} \end{cases} \tag{11}$$

$$\begin{cases} 0 < T_k \le (\#\ of\ MACs) \\ 0 < T_{c1} \le (\#\ of\ MACs) \\ 0 < T_m \le M, \quad 0 < T_k \le K \\ 0 < T_{n0} \le N, \quad 0 < T_{n1} \le N \\ 0 < T_{c0} \le C, \quad 0 < T_{c1} \le C \\ S_X + S_W + S_{B1} <= GLBsize \\ S_A + S_O + S_{B2} <= GLBsize \end{cases} \tag{12}$$

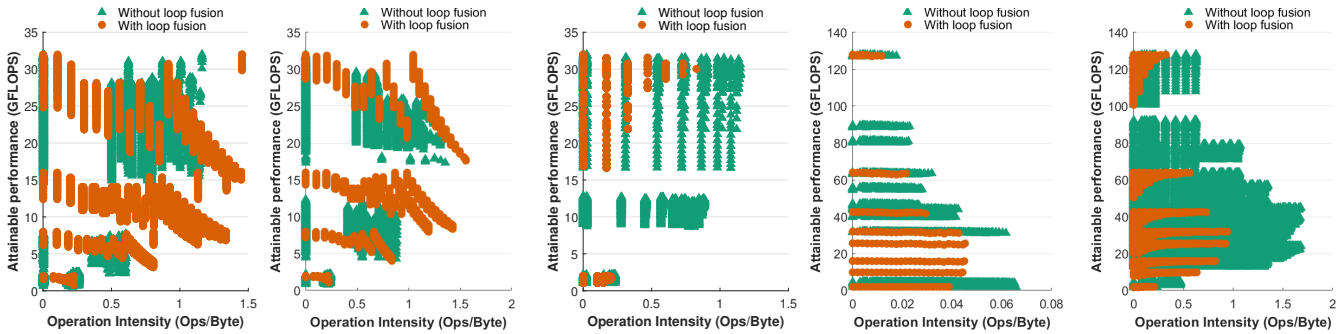
We can see that the DRAM accesses of matrix B are eliminated but α_O is much larger than that in Equation (10).

E. Design Space Exploration

It is a non-trivial task to choose the best combination of the design variables ($T_m, T_{n0}, T_{n1}, T_{c0}, T_{c1}, T_k$) and design choices (loop order and loop fusion strategy) that maximize the performance and minimize the number of off-chip data accesses, which needs a systematic design space exploration methodology. We use the performance and DRAM access models to build an optimization framework for fast design space exploration. We enumerate all possible loop orders, tile sizes, and fusion strategies to generate a series of computational performance and operation intensity pairs. Specifically, the space of all legal tile sizes is constrained by Equation (12) where *GBLsize* denotes the global buffer size constraint.

Fig. 6 depicts the legal solutions for the first layer of the five datasets using the roofline model coordination system [37], where *GBLsize* is set as 512 KB and the computational roof is set as 32 *GFLOPS*. The "x" axis denotes the operation intensity, or the ratio of floating point operation per DRAM byte access. The "y" axis denotes the computational performance measured by GFLOPS. The slope of the line between any point and the origin point (0,0) denotes the minimum bandwidth requirement for this combination of design choices.

As illustrated in Fig. 6, different datasets prefer different design choices. For Cora, enabling loop fusion would achieve optimal performance as well as operation intensity. By contrast, Reddit prefers not to enable loop fusion. Table III shows the optimal tile sizes, loop order and loop fusion strategy for the



(a) Cora (b) Citeseer (c) Pubmed (d) Nell (e) Reddit

Fig. 6: Design space exploration for the first layer of the graph datasets: (a) Cora; (b) Citeseer; (c) Pubmed; (d) Nell; (e) Reddit.

TABLE III. Dataset specific optimal solution and cross-dataset optimization.

Dataset	Layer	Loop fusion	Loop order	Optimal Tile Size Tuple $(T_{n0},T_{c0},T_k,T_{n1},T_{c1},T_m)$	DRAM Accesses	Cross-dataset Optimization $(T_{n0},T_{c0},T_k,T_{n1},T_{c1},T_m)$	Loop fusion	Loop order	DRAM Accesses
Cora	L1	Yes	$n_0 \to c_0 \to k \to m$	(2708,16,1,2708,16,1)	172131	(2048,16,16,2048,16,16)	Yes	$n_0 \to c_0 \to k \to m$	207446
	L2	Yes	$n_0 \to c_0 \to k \to m$	(2708,7,1,2708,7,1)	85084		Yes	$n_0 \to c_0 \to k \to m$	97338
Citeseer	L1	Yes	$n_0 \to c_0 \to k \to m$	(3000,16,5,3000,16,1)	300925	(2048,16,16,2048,16,16)	Yes	$n_0 \to c_0 \to k \to m$	386351
	L2	Yes	$n_0 \to c_0 \to k \to m$	(3000,6,1,3000,6,1)	104243		Yes	$n_0 \to c_0 \to k \to m$	124874
Pubmed	L1	No	$n_0 \to c_0 \to k$, $m \to c_1 \to n_1$	(3073,16,1,1,16,3073)	3800622	(2048,16,16,16,16,2048)	No	$n_0 \to c_0 \to k$, $m \to c_1 \to n_1$	4839367
	L2	No	$n_0 \to c_0 \to k$, $m \to c_1 \to n_1$	(3000,3,1,1025,3,3000)	860549		No	$n_0 \to c_0 \to k$, $m \to c_1 \to n_1$	1041408
Nell	L1	No	$n_0 \to c_0 \to k$, $m \to c_1 \to n_1$	(4096,1,33,1,1,4096)	188541177	(2048,16,16,16,16,2048)	No	$n_0 \to c_0 \to k$, $m \to c_1 \to n_1$	272550109
	L2	No	$n_0 \to c_0 \to k$, $m \to c_1 \to n_1$	(257,186,1,1,17,2817)	320259165		No	$n_0 \to c_0 \to k$, $m \to c_1 \to n_1$	463651357
Reddit	L1	No	$n_0 \to c_0 \to k$, $m \to c_1 \to n_1$	(641,64,1,1,9,4096)	1780902301	(2048,16,16,16,16,2048)	No	$n_0 \to c_0 \to k$, $m \to c_1 \to n_1$	2479084738
	L2	No	$n_0 \to c_0 \to k$, $m \to c_1 \to n_1$	(1153,41,1,1,17,2817)	1095478962		No	$n_0 \to c_0 \to k$, $m \to c_1 \to n_1$	1423139406

five datasets. The dataflows in existing designs, such as HyGCN and AWB-GCN, stick to static loop order and loop fusion strategy, which would inevitably incur inefficiency because of the GCN configuration variance. To this end, we propose a flexible dataflow to adapt to different GCN configurations. The dataflow tailors the tile sizes based on cross-dataset optimization, and can reconfigure the loop order and loop fusion technique for given GCN configurations. Enabling this flexibility for the dataflow will significantly improve the overall efficiency across different GCN configurations.

The exploration results also show that the tile sizes should be adaptive to different GCN configurations to further improve efficiency. However, to support multiple dataflows with different tile sizes would be challenging. Complex hardware structures are required to reconfigure compute engines and interconnects. Besides, different tile sizes may also require different buffer sizes and structures, which further increases the complexity. An alternative approach is to design a hardware architecture with uniform tile sizes for all the datasets. We enumerate all legal solutions to select the optimal tile sizes for cross-dataset optimization, as shown in Table III. Notably, since SpMM1 and SpMM2 use the same compute engine, the tile sizes in the two SpMMs share the same constraint of the hardware characteristics. Therefore, the tile sizes in the two SpMMs should be co-optimized to match the hardware characteristics. Specifically, if loop fusion is not enabled, we constrain $T_m = T_{n0}$, $T_{c1} = T_{c0}$, $T_{n1} = T_k$ so that the buffer requirements of the two SpMMs are the same. For the dataflow with loop fusion, we constrain $T_m = T_k$ so that the buffer requirements of the two SpMMs are the same. Therefore, for cross-dataset optimization, the search space are significantly simplified as the independent variables are reduced from six to three (T_{n0}, T_{c0}, T_k). Although we use uniform tile sizes, we have two sets of tile size combinations, as shown in Table III, one set for the case when loop fusion is enabled, and the other for the case when loop fusion is disabled. GCN dataflow with uniform tile sizes is simple to implement, but might be sub-optimal for some datasets. Table III shows that using uniform tile sizes increases the DRAM accesses by less than 50% compared to that of dedicated tile sizes for each dataset, which is acceptable considering the low complexity of implementation.

IV. GCNAX Architecture

This section introduces the proposed accelerator architecture called GCNAX to support the flexible dataflow, GCNAX employs an outer-product based method for SpMMs to avoid workload imbalance, and tailors buffer size and structures according to the tile sizes.

A. Accelerator Overview

The system architecture of GCNAX is shown in Fig. 7. It consists of a SparseMat Buffer (SMB), Input/Output DenseMat Buffers (IDMB/ODMB), a Look-Ahead FIFO, a DenseRow Prefetcher (DRP), a MAC Array, and a Control Unit. The Software Scheduler is used to reconfigure the loop order and loop fusion strategy for different datasets, and generates corresponding commands to the Control Unit. The accelerator's state machine operates on the SpMM in the order defined by the received commands.

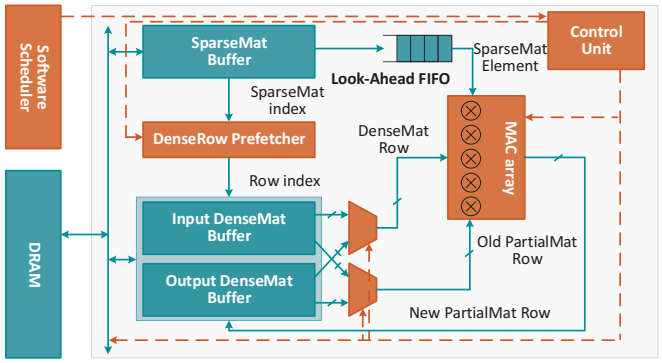


Fig. 7: The full GCNAX architecture.

TABLE IV. GCNAX configurations.

Design parameters	Value
Multiplier width	64 bits
MAC Array	1×16
Look-ahead FIFO	16 entries
SMB	320 KB
IDMB	4 KB
ODMB	256 KB
DRAM	HBM@128GB/s

TABLE V. Hardware characteristics of GCNAX.

GCNAX	Area (mm^2)	Power(mW)
Total	6.51 (100%)	365.0 (100%)
MAC Array	0.46 (7.1%)	86.2 (23.6%)
SMB	2.60 (39.9%)	83.4 (22.8%)
IDMB	0.10 (1.5%)	62.5 (17.1%)
ODMB	2.70 (41.5%)	74.3 (20.4%)
DRP	0.41 (6.3%)	46.2 (12.7%)
Control Unit	0.24 (3.7%)	12.4 (3.4%)

To process the SpMM, a portion of the sparse matrix is fetched from DRAM into SMB in CSC format, and a portion of the input/output matrix is fetched into IDMB and ODMB in dense format. First, an old PartialMat row is fetched from ODMB to the MAC array waiting for accumulation. Meanwhile, an element value from SMB is sent to the FIFO, while its row index in the CSC format is sent to DRP. Then, the DRP fetches the corresponding input DenseMat row from IDMB according to the received index and the row index of the PartialMat row. Since the required input DenseMat row is not known until receiving the index of the SparseMat element, there is a latency between the arriving time of the SparseMat element and the input DenseMat row to the MAC Array. The look-ahead FIFO is used to hide this latency. Instead of directly sending the SparseMat element to MAC Array, it is sent to the FIFO. At the same time, the DRP calculates the required row index and prefetches rows to MAC Array. The MAC Array will then conduct the outer product between the SparseMat element and the DenseMat row and the generated row will be accumulated with the Old PartialMat row.

When possible, the PartialMat row is held consistently in the MAC Array until its related computations are finished. Upon completion of the current PartialMat row, the generated new PartialMat row is then flushed to ODMB, and it proceeds to the next PartialMat row according to the execution order defined by the dataflow. When the output DenseMat can serve as the input DenseMat for the following SpMM, which is the case when we enable loop fusion, the IDMB and ODMB are logically swapped to the two SpMMs' computation sequence.

B. GCNAX Architecture Configuration

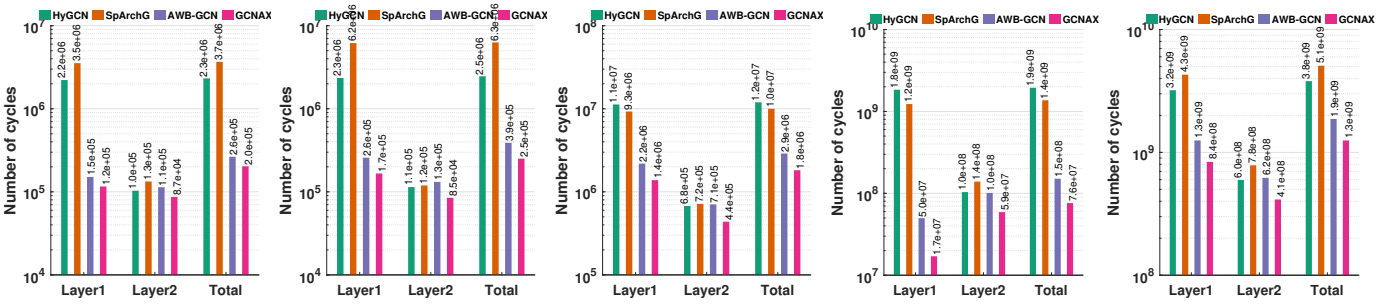
Table IV lists the key parameters of the GCNAX design according to the cross-dataset optimization shown in Section III. The design employs a 1×16 MAC Array using double-precision floating point multipliers. The SMB/IDMB/ODMB are sized according to the tile sizes. For example, since SMB is used to store the matrix X and A in the two SpMMs, the size of SMB should be large enough to hold the data chunk of X and A defined by the tile sizes. Specifically, since the chunk size of X is $T_{n0} \times T_{c0}$ (32K) and the density of X varies from 0.011% to 89.1%, the storage size of the non-zero values in each chunk will not exceed $32KB * 8 = 256KB$ as we use double precision-floating point numbers (8 Bytes per data element). Besides, the chunk size of A when not enabling loop fusion is $T_{n0} \times T_m$ (4M) but the density of A does not exceed 0.21%, so the storage size of the non-zero values in each chunk will not exceed 256 KB. Considering that the CSC format for the sparse matrix introduces overhead for storage of row indices and column pointers, and the non-zeros are not evenly distributed, we offer the SMB an additional $64KB$ to cover this overhead, which results in $320KB$ for the SMB. IDMB is used to store the data chunk of W in SpMM1 ($T_k \times T_{c0} \times 8B = 2KB$). If not enabling loop fusion, IDMB stores the data chunk of B in SpMM2 ($T_{n1} \times T_{c1} \times 8B = 2KB$). If enabling loop fusion, IDMB stores the data chunk of O ($T_m \times T_{c1} \times 8B = 2KB$) in SpMM2. Therefore, the storage requirement for IDMB is $2KB$, which is the largest of the above three values. Since the DRP will prefetch DenseMat rows, we doubled the IDMB size to be $4KB$. Similarly, the size of ODMB is $T_{n0} \times T_{c0} \times 8B = 256KB$.

V. Experimental Methodology

Hardware simulator. To evaluate the performance of our design, we built a cycle-accurate simulator in C++ to model the behavior of the hardware. The simulator models the microarchitectural behaviors of each module, and supports the dataflows shown in Fig. 4. The simulator counts the exact amount of DRAM read and write, which is used to estimate the DRAM access energy consumption according to [38].

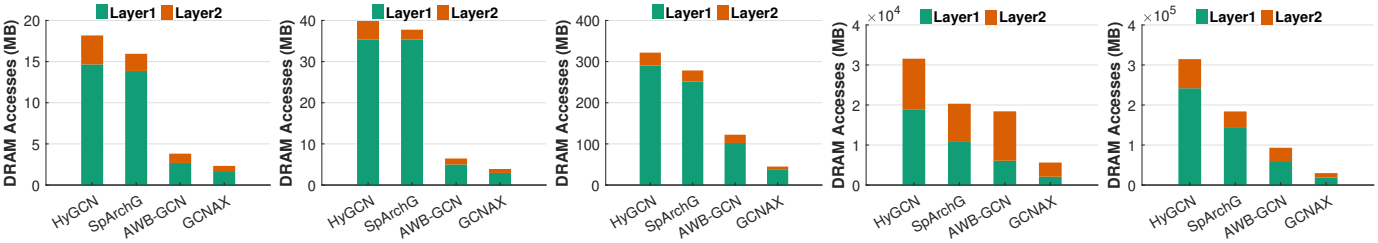
ASIC synthesis. To measure the area and power consumption, we model all the logic including the MAC Array, FIFOs, DRP, and DRAM. We use the Synopsys Design Compiler with the TSMC 40nm library for the synthesis, and estimate the power using Synopsys PrimeTime PX. We set the clock frequency at 1 GHz. We use Cacti [39] to estimate the area, power, and access latency of the on-chip buffers and FIFOs.

Baselines. We compare GCNAX with two GCN accelerators (HyGCN and AWB-GCN) and an SpMM accelerator (SpArch). To evaluate the efficiency of the flexible dataflow of GCNAX, we also compare it with GCNAX-F/GCNAX-NF, which always/never enable loop fusion. Table VI summarizes the characteristics of these baselines and GCNAX.



(a) Cora (b) Citeseer (c) Pubmed (d) Nell (e) Reddit

Fig. 8: Number of execution cycles for GCNAX and the baseline accelerators.



(a) Cora (b) Citeseer (c) Pubmed (d) Nell (e) Reddit

Fig. 9: DRAM accesses of GCNAX and the baseline accelerators.

TABLE VI. Characteristics of the accelerators.

Accelerator	Execution order	Compute engine	Loop fusion	Loop order	Inner Spatial Dataflow
HyGCN	$(AX)W$ †	Tandem engine	Yes	Static	Inner product
AWB-GCN	$A(XW)$	Uniform engine	Yes	Static	Inner product
SpArchG §	$(AX)W$	Uniform engine	No	Static	Outer product
GCNAX-F	$A(XW)$	Uniform engine	Yes	Static	Outer product
GCNAX-NF	$A(XW)$	Uniform engine	No	Static	Outer product
GCNAX	$A(XW)$	Uniform engine	Adaptive	Adaptive	Outer product

† HyGCN uses edge-centric programming model for the aggregation phase, so their computation in the aggregation phase is not matrix multiplication. Nevertheless, the result of the aggregation phase is a large matrix that is used as the input to perform matrix multiplication in the combination phase.

§ SpArchG uses SpArch [31] (an SpGEMM accelerator) to process matrix multiplications in GCNs.

The baseline accelerators are scaled to be equipped with the same number of multipliers and DRAM bandwidth as GCNAX. Since HyGCN and AWB-GCN use single-precision floating point numbers (32-bit) whereas SpArch uses double-precision (64-bit), we uniformly use double-precision for all accelerators to provide a fair comparison. As HyGCN uses a tandem-engine architecture consisting of SIMD cores for the aggregation phase and systolic modules for the combination phase, the multipliers are divided to two groups in a ratio of 1:8 for the two engines according to its original configuration. We also resized the baseline accelerators to be equipped with the on-chip storage capacity. For example, we simulated the HyGCN accelerator with 580 KB on-chip storage rather than the original 16 MB. The DRAM bandwidth for all the accelerators is scaled to 128GB/s. Note that as HyGCN uses edge-centric programming model for the aggregation phase, their computation in the aggregation phase is not matrix multiplication. Our simulator takes this into account and estimates the execution cycles and DRAM accesses according to HyGCN's dataflow. Although SpArch is not customized for GCNs, it is still selected as a baseline since it supports the key computations in GCNs. As SpArch does not mention how to support chain SpMM, we assume that it processes the chain SpMM sequentially without loop fusion. Hereafter we denote SpArchG as our simulated accelerator that uses SpArch to process chain SpMM.

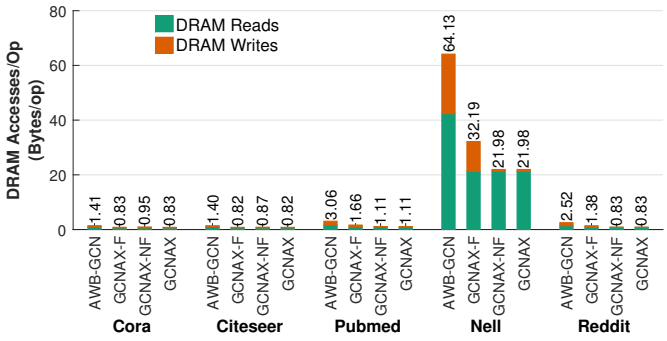
VI. Experimental Results

A. Area and Power

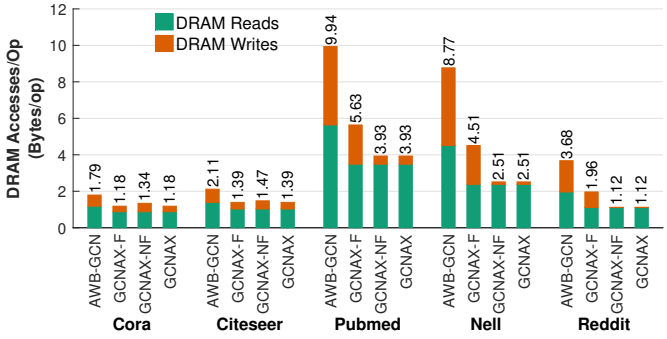
We obtain the area and power consumption of GCNAX accelerator under TSMC 40nm technology. Table V summarizes the area and power of the major components. A significant fraction of the PE area is contributed by memories (SMB, IDMB and ODMB), which consume 82.9% of the total area, while the MAC array only consumes 7.1%. IDMB and ODMB heavily banked for parallelization so they consume more area than SMB. As for power, the MAC Array and the buffers consume 23.6% and 60.3% of the total power, respectively.

B. Performance

Fig. 8 compares the performance of GCNAX and the baselines measured by the total number of execution cycles.



(a) Layer 1



(b) Layer 2

Fig. 10: DRAM access breakdown by read and write of AWB-GCN, GCNAX-F, GCNAX-NF and GCNAX.

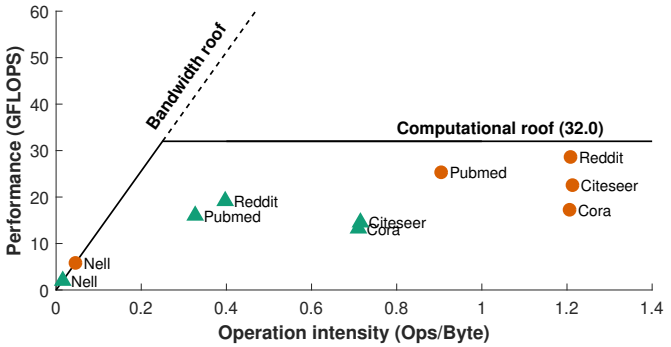


Fig. 11: Roofline Model for the AWB-GCN (triangles) and GCNAX (circles) evaluated on the first layer of the five datasets.

On average, GCNAX is 8.9×, 11.3×, and 1.6× faster than HyGCN, SpArchG and AWB-GCN, respectively. GCNAX outperforms the baselines on all the five datasets. The reasons for the high performance of GCNAX are threefold. First, the execution order of the chain-SpMM of GCNAX reduces the number of operations compared to that of HyGCN. Second, GCNAX uses a uniform-engine architecture that avoids the workload imbalance incurred by tandem-engine architectures. HyGCN can only achieve optimal performance by carefully orchestrating the computational capacity of the combination and aggregation engines for a given dataset, but it inevitably incurs performance loss when accommodating different datasets with different computational requirements for the aggregation and combination engines. Finally, GCNAX achieves the lowest DRAM accesses by adaptively configuring the dataflow for different datasets, which also explains why GCNAX outperforms AWB-GCN and SpArch. The number of DRAM accesses has a strong impact on performance since it might be the system bottleneck. AWB-GCN uses the inner-product based method for SpMM which incurs workload imbalance between PEs thereby degrading the performance. AWB-GCN addresses this inefficiency by a software scheduler and additional hardware modules that increase hardware complexity and introduce extra overhead. Since SpArchG is customized for sparse-sparse matrix multiplication, it achieves high performance when performing AX. However, the performance gain of SpArchG is hindered because 1) its processing order results in larger computation volume; 2) SpArchG is not good at processing dense-dense matrix multiplication.

As for the speedup for specific datasets, GCNAX performs 3.0–25.6× better over HyGCN, 4.0–25.1× better over SpArchG, and 1.3–2.0× better over AWB-GCN. The performance gain on the Reddit dataset is not so significant because the execution order reduces computations by only 2.9× which is not that much compared to other datasets. Besides, the density of feature vectors in Reddit (larger than 50%) is higher than that of other datasets, which hinders the performance gains of GCNAX because it still performs SpMM even though the input matrix is not that sparse.

C. DRAM Accesses

Fig. 9 shows the number of DRAM accesses of the four accelerators. Overall, GCNAX achieves on average 8.1×, 6.2× and 2.4× reduction on DRAM accesses over HyGCN, SpArchG and AWB-GCN, respectively. This benefits from the optimal tile size tuple, the data reuse optimization and the adaptive loop fusion strategy. The DRAM access reduction varies across the datasets. Specifically, GCNAX reduces DRAM accesses by a factor of 5.6–10.6× over HyGCN, 3.6–10.7× over SpArchG, and 1.6–3.3× over AWB-GCN. Since HyGCN and SpArchG use inefficient execution order, they involve more computations that result in more DRAM accesses. AWB-GCN optimizes the reuse of the intermediate matrix. However, it sacrifices the reuse of the output matrix due to the limited on-chip storage size. Moreover, the tile sizes are not carefully tailored in the AWB-GCN accelerator.

We further look into the DRAM access breakdown by read and write as shown in Fig. 10. We added GCNAX-F and GCNAX-NF into comparison to evaluate the effects of the adaptive loop fusion strategy. On the Cora and Citeseer datasets, enabling loop fusion would lead to fewer DRAM accesses, while the other three datasets prefer not enabling loop fusion for the given tile sizes and hardware characteristics. The layer 1 in the Nell dataset induces more DRAM accesses than other datasets because both the A and X matrix are ultra sparse in this layer, which leads to irregular data accesses.

To understand how far our design is from the theoretically optimal solution, we use the roofline model to analyze

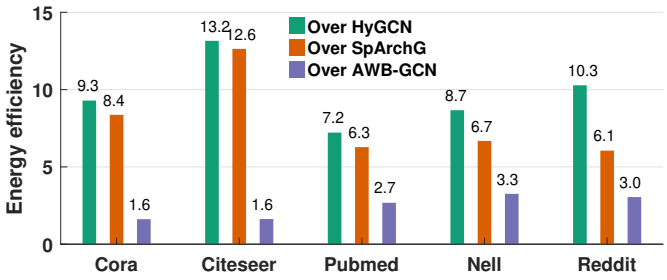


Fig. 12: Energy savings of GCNAX over the baselines.

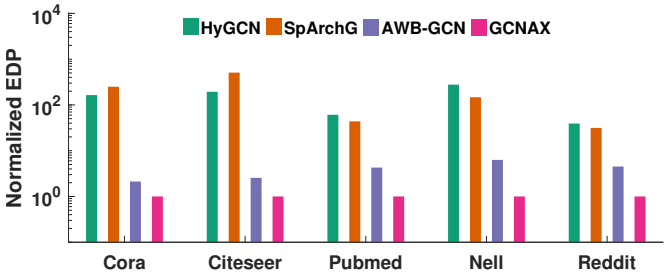


Fig. 13: Energy delay product (EDP) of the four accelerators. It is normalized to the EDP of GCNAX.

our performance. Fig. 11 shows the result of the roofline analysis. The theoretical computation roof is 32 $GFlops$ in our design since we use 16 MACs running in $1GHz$. The peak multiplication performance is 16 $GFlops/s$, and the overall peak performance (multiplication and addition) is 32 $GFlops/s$. It is observed that GCNAX is in the right and upper region compared to AWB-GCN, which indicates that GCNAX achieves better performance with fewer DRAM accesses. Notably, the performance on the Nell dataset is bounded by the bandwidth roof.

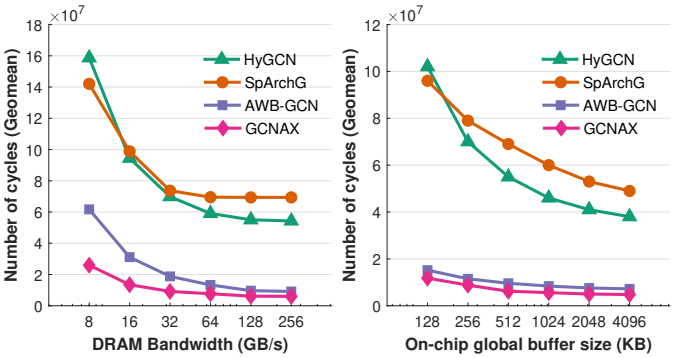
D. Energy Consumption

Fig. 12 shows the normalized energy consumption of the four accelerators. Overall, GCNAX achieves 9.5×, 7.7×, and 2.3× energy savings compared to HyGCN, SpArchG and AWB-GCN. This is because our proposed accelerator has fewer DRAM accesses and better utilization of the computing resource.

Energy-delay product: Energy-delay product is used to verify that a dataflow does not achieve high energy efficiency by sacrificing processing parallelism. Fig. 13 shows the normalized EDP of the four accelerators. The delay is calculated as the reciprocal of the number of execution cycles. Compared with the baseline accelerators, GCNAX is 115.9×, 120.7×, and 3.7× better in EDP averaged on the five datasets.

E. Sensitivity to Hardware Parameters

As mentioned in Section V, we scaled the baseline accelerators to be equipped with the same hardware resources as GCNAX, which would potentially hurt the efficiency of the baseline accelerators. Therefore, we conduct a sensitivity analysis to quantify the effects of the hardware parameter variations on the accelerator performance. In Fig. 14(a), we



(a) Sensitivity to DRAM bandwidth (b) Sensitivity to on-chip buffer size

Fig. 14: Performance of the accelerators w.r.t the variation of: (a) DRAM bandwidth; (b) on-chip buffer size. The number of cycles is the geometric mean of the five graph datasets.

sweep the DRAM bandwidth provisioning from 8 GB/s up to 256 GB/s while fixing the other hardware parameters. Clearly, GCNAX performs consistently better than the baseline accelerators under different bandwidth provisioning because its dataflow drastically reduces the DRAM traffic. Nevertheless, the performance of GCNAX also decreases with the reduced DRAM bandwidth provisioning. Specifically, GCNAX suffers a 24.8% performance degradation when changing the DRAM bandwidth from 128GB/s to 64 GB/s. In Fig. 14(b), we sweep the on-chip buffer size from 128 KB up to 4 MB to investigate how it influences the performance. HyGCN and SpArchG are more likely to suffer from performance degradation with reduced buffer size because their execution order of the chain SpMMs generates more intermediate data, which requires extra on-chip storage.

F. Comparison with GPUs

As GPUs are inherently optimized for compute-intensive workloads, we also compare GCNAX with GPUs in terms of speedup and energy consumption. We evaluate PyTorch Geometric on an NVIDIA Tesla P40 GPU (denoted as *PyG-GPU*), which has 3840 cores operating @1.3GHz with a memory bandwidth of 480.4 GB/s. GCNAX achieves 18.3× speedup and 25.9× energy savings over *PyG-GPU*.

VII. Related Work

The most relevant works to ours are the two GCN inference accelerators, HyGCN and AWB-GCN, which have been discussed in the paper. Beides, we also discuss other relevant accelerator designs.

Graph analytics accelerators. With the emergence of applications on graph analytics, many accelerators are proposed to efficiently support these workloads [15], [40]–[53]. Hong *et al.* [40], [54] propose a warp centric execution model for graph applications. Ozdal *et al.* [15] propose a configurable architecture template that is specifically optimized for iterative vertex-centric graph applications with irregular access patterns and asymmetric convergence. Graphicionado [44] exploits not

only data structure-centric datapath specialization, but also memory subsystem specialization for efficient graph analytics processing. Medusa [45] is a parallel graph processing system on GPUs that enables developers to leverage the massive parallelism and other hardware features. GraphR [46] is a ReRAM-based graph processing accelerator that leverages the near-data processing and explores the opportunity of performing massive parallel analog operations with low hardware and energy cost. GraphABCD [47] is an asynchronous heterogeneous graph analytic framework that offers algorithm and architectural supports for asynchronous execution, without undermining its fast convergence properties. Yan *et al.* [48] propose a hardware/software co-design with decoupled datapath and data-aware dynamic scheduling to alleviate irregularity in graph analytics accelerators. Although these accelerators deliver considerable performance and energy efficiency improvement, they are inefficient when handling GCNs because even though they are designed to alleviate irregularity of graph data, they do not leverage the regularity in GCNs.

Neural Network Accelerators. There have been many works devoted to accelerating neural networks [16], [55]–[60]. For dense neural networks, the accelerators mainly focus on leveraging the massive parallelism to improve performance and utilization, such as TPU [55] and Eyeriss [16]. Due to the intrinsic sparsity structure, many accelerator [56]–[59] have been proposed to reduce operations from sparsity. However, GCNs contain two-phase matrix multiplications that enable new kinds of parallelisms and data reuse patterns that are not exploited in these neural network accelerators. Although we can extend CNN accelerators to run SpMMs by equalizing the input and filter dimensions, it weakens the advantages of CNN accelerators since they are specialized for convolutions rather than matrix multiplications.

GeMM, SpMV and SpMM Accelerators. Since the key computation pattern in GCNs is matrix multiplication, we would also like to mention the matrix multiplication accelerators. They are categorized as general matrix matrix multiplication (GeMM), sparse matrix-vector multiplication (SpMV) and sparse matrix-matrix multiplciation (SpMM), with implementations on GPUs [29], [40], [61], FPGAs [30] and ASICs [19], [31], [62]. Lin *et al.* [30] presented the design and implementation of a sparse matrix-matrix multiplication architecture on FPGAs. OuterSpace [19] proposed outer product based SpGEMM, which has perfect input reuse compared to inner product based method. However, these works provide little insight in how to efficiently support chain SpMMs.

We believe that the proposed approach can benefit other applications that utilize matrix chain multiplications such as the finite-element simulation [63]. Furthermore, the proposed approach can also be extended to support other types of graph neural networks such as Dense Graph Propagation Networks [64] operating on directed graphs and previously used for zero-shot learning, or Dual Graph Convolutional Networks [65] used for semi-supervised classification. To support other applications, e.g. finite-element simulation, the hardware parameters may need to be changed to adapt to the corresponding matrix dimensions and dataflow characteristics.

VIII. Conclusion

In this paper, we propose an energy-efficient and optimized accelerator architecture for GCNs called GCNAX. The salient features of the proposed architecture is that the dataflow can reconfigure the loop order and loop fusion strategy to adapt to different GCN configurations, and the tile sizes are carefully tailored based on cross-dataset optimization, which simultaneously improves resource utilization and reduces data movement. The GCNAX accelerator tailors the compute engine, buffer structure and size to support the optimized dataflow. We evaluated our proposed architecture on five real-world graph datasets, and the simulation results show that GCNAX reduces DRAM accesses by a factor of 8.1× and 2.4×, while achieving 8.9×, 1.6× speedup and 9.5×,2.3× energy savings over HyGCN and AWB-GCN, respectively.

Acknowledgment

This research was partially supported by NSF grants CCF-1702980, CCF-1812495, CCF-1901165, CCF-1703013 and CCF-1936794. We sincerely thank the anonymous reviewers for their excellent and constructive feedback.

References

[1] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788.

[2] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, pp. 91–99.

[3] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," *arXiv preprint arXiv:1508.04025*, 2015.

[4] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, and K. Macherey, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[5] Z. Zhang, P. Cui, and W. Zhu, "Deep learning on graphs: A survey," *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–1, 2020.

[6] M. Henaff, J. Bruna, and Y. LeCun, "Deep convolutional networks on graph-structured data," *arXiv preprint arXiv:1506.05163*, 2015.

[7] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *Advances in neural information processing systems*, pp. 3844–3852.

[8] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[9] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, "Spectral networks and locally connected networks on graphs," *arXiv preprint arXiv:1312.6203*, 2013.

[10] H. Yang, "Aligraph: A comprehensive graph neural network platform," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery Data Mining*, pp. 3165–3166.

[11] A. Lerer, L. Wu, J. Shen, T. Lacroix, L. Wehrstedt, A. Bose, and A. Peysakhovich, "Pytorch-biggraph: A large-scale graph embedding system," *arXiv preprint arXiv:1903.12287*, 2019.

[12] N. De Cao and T. Kipf, "Molgan: An implicit generative model for small molecular graphs," *arXiv preprint arXiv:1805.11973*, 2018.

[13] M. Yan, Z. Chen, L. Deng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Characterizing and understanding gcns on gpu," *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 22–25, 2020.

[14] Z. Zhang, J. Leng, L. Ma, Y. Miao, C. Li, and M. Guo, "Architectural implications of graph neural networks," *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 59–62, 2020.

[15] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 166–177, 2016.

[16] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pp. 367–379, IEEE.

[17] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Hygcn: A gcn accelerator with hybrid architecture," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 15–29.

[18] T. Geng, A. Li, T. Wang, C. Wu, Y. Li, A. Tumeo, and M. Herbordt, "Uwb-gcn: Hardware acceleration of graph-convolution-network through runtime workload rebalancing," *arXiv preprint arXiv:1908.10834*, 2019.

[19] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 724–736, IEEE.

[20] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2008.

[21] T. N. Kipf and M. Welling, "Variational graph auto-encoders," *arXiv preprint arXiv:1611.07308*, 2016.

[22] Y. Seo, M. Defferrard, P. Vandergheynst, and X. Bresson, "Structured sequence modeling with graph convolutional recurrent networks," in *International Conference on Neural Information Processing*, pp. 362–373, Springer.

[23] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in neural information processing systems*, pp. 1024–1034.

[24] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?," *arXiv preprint arXiv:1810.00826*, 2018.

[25] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *arXiv preprint arXiv:1901.00596*, 2019.

[26] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, and T. Eliassi-Rad, "Collective classification in network data," *AI magazine*, vol. 29, no. 3, pp. 93–93, 2008.

[27] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka, and T. M. Mitchell, "Toward an architecture for never-ending language learning," in *Twenty-Fourth AAAI conference on artificial intelligence*.

[28] S. S. Godbole, "On efficient computation of matrix chain products," *IEEE Transactions on Computers*, vol. 100, no. 9, pp. 864–866, 1973.

[29] Y. Nagasaka, A. Nukada, R. Kojima, and S. Matsuoka, "Batched sparse matrix multiplication for accelerating graph convolutional networks," *arXiv preprint arXiv:1903.11409*, 2019.

[30] C. Y. Lin, N. Wong, and H. K. So, "Design space exploration for sparse matrix-matrix multiplication on fpgas," *International Journal of Circuit Theory and Applications*, vol. 41, no. 2, pp. 205–219, 2013.

[31] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "Sparch: Efficient architecture for sparse matrix multiplication," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 261–274, IEEE.

[32] J. R. Allen and K. Kennedy, "Automatic loop interchange," in *Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pp. 233–246.

[33] W. Pugh, "Uniform techniques for loop optimization," in *Proceedings of the 5th international conference on Supercomputing*, pp. 341–352.

[34] S. Winograd, "A new algorithm for inner product," *IEEE Transactions on Computers*, vol. 100, no. 7, pp. 693–694, 1968.

[35] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *field programmable gate arrays*, pp. 161–170.

[36] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *architectural support for programming languages and operating systems*, vol. 49, no. 4, pp. 269–284, 2014.

[37] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for floating-point programs and multicore architectures," *Communications of the Association for Computing Machinery*, 2009.

[38] M. Horowitz, "Energy table for 45nm process," 2012.

[39] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to understand large caches," *HP Laboratories*, 2009.

[40] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating cuda graph algorithms at maximum warp," *Acm Sigplan Notices*, vol. 46, no. 8, pp. 267–276, 2011.

[41] V. Balaji and B. Lucia, "Combining data duplication and graph reordering to accelerate parallel graph processing," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pp. 133–144.

[42] G. Dai, Y. Chi, Y. Wang, and H. Yang, "Fpgp: Graph processing framework on fpga a case study of breadth-first search," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 105–110.

[43] T. Oguntebi and K. Olukotun, "Graphops: A dataflow library for graph analytics acceleration," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 111–117.

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

[45] J. Zhong and B. He, "Medusa: A parallel graph processing system on graphics processors," *ACM SIGMOD Record*, vol. 43, no. 2, pp. 35–40, 2014.

[46] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Graphr: Accelerating graph processing using reram," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 531–543, IEEE.

[47] Y. Yang, Z. Li, Y. Deng, Z. Liu, S. Yin, S. Wei, and L. Liu, "Graphabcd: Scaling out graph analytics with asynchronous block coordinate descent,"

[48] M. Yan, X. Hu, S. Li, A. Basak, H. Li, X. Ma, I. Akgun, Y. Feng, P. Gu, and L. Deng, "Alleviating irregularity in graph analytics acceleration: a hardware/software co-design approach," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 615–628.

[49] A. Abdolrashidi, D. Tripathy, M. E. Belviranli, L. N. Bhuyan, and D. Wong, "Wireframe: Supporting data-dependent parallelism through dependency graph execution in gpus," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 600–611.

[50] X. Ma, D. Zhang, and D. Chiou, "Fpga-accelerated transactional execution of graph workloads," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 227–236.

[51] F. Sadi, J. Sweeney, S. McMillan, T. M. Low, J. C. Hoe, L. Pileggi, and F. Franchetti, "Pagerank acceleration for large graphs with scalable hardware and two-step spmv," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pp. 1–7, IEEE.

[52] Y. Wang, J. C. Hoe, and E. Nurvitadhi, "Processor assisted worklist scheduling for fpga accelerated graph processing on a shared-memory platform," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 136–144, IEEE.

[53] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, "Graphq: Scalable pim-based graph processing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 712–725.

[54] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "Graphp: Reducing communication for pim-based graph processing with efficient data partition," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 544–557, IEEE.

[55] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance

analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, (3080246), pp. 1–12, ACM.

[56] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *ACM SIGARCH Computer Architecture News*, vol. 44, pp. 1–13, IEEE Press.

[57] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-X: An accelerator for sparse neural networks," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE.

[58] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 27–40, ACM.

[59] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. N. Vijaykumar, "SparTen: A sparse tensor accelerator for convolutional neural networks," 2019.

[60] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 243–254, IEEE Press.

[61] R. Nath, S. Tomov, T. T. Dong, and J. Dongarra, "Optimizing symmetric dense matrix-vector multiplication on gpus," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–10.

[62] E. Montagne and R. Surós, "Systolic sparse matrix vector multiply in the age of tpus and accelerators," in *2019 Spring Simulation Conference (SpringSim)*, pp. 1–10.

[63] G. F. Pinder and W. G. Gray, *Finite element simulation in surface and subsurface hydrology*. Elsevier, 2013.

[64] M. Kampffmeyer, Y. Chen, X. Liang, H. Wang, Y. Zhang, and E. P. Xing, "Rethinking knowledge graph propagation for zero-shot learning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 11487–11496.

[65] C. Zhuang and Q. Ma, "Dual graph convolutional networks for graph-based semi-supervised classification," in *Proceedings of the 2018 World Wide Web Conference*, pp. 499–508.