FusedGCN: A Systolic Three-Matrix Multiplication Architecture for Graph Convolutional Networks

Christodoulos Peltekis, Dionysios Filippas
Electrical and Computer Engineering
Democritus University of Thrace, Greece

Chrysostomos Nicopoulos
Electrical and Computer Engineering
University of Cyprus, Cyprus

Giorgos Dimitrakopoulos
Electrical and Computer Engineering
Democritus University of Thrace, Greece

***Abstract*—Machine-learning applications have garnered widespread adoption over the last several years. Graph Neural Networks have been proposed as an extension of machine-learning models to graph-structured data. The training and inference tasks on graph neural networks involve graph convolution operations that can be equivalently expressed as three-matrix multiplications. In this work, we propose FusedGCN, a custom systolic architecture that computes in a *fused*, i.e., combined, manner the product of three matrices. FusedGCN supports compressed sparse representations and tiled computations, which allow the design to adapt to the available input/output bandwidth without losing the regularity of a systolic architecture. The experimental results show that FusedGCN achieves lower execution times than the current best-performing state-of-the-art architecture for computing representative GCN applications. Most importantly, this result is achieved by consuming only marginally more area/power than a traditional systolic array used for two-matrix multiplications.**

I. Introduction

Machine learning applications have become ubiquitous over the last few years. Graph Neural Networks (GNNs) have been widely adopted in applications that use graph-structured data, such as social networks [1] and recommendation systems [2]. GNNs are able to learn graph-related representations, where nodes represent objects and edges represent the relationships between them, thereby solving graph classification, node classification, and link prediction problems [3]. In this work, we focus on Graph Convolutional Networks (GCNs) as some of the first and most effective processing models for GNNs [4].

In GCNs, each node of the graph is accompanied by a feature vector. Updating the features of each node involves the *aggregation* of the features of neighboring nodes and their *combination* to obtain a new feature vector. Each layer of the GCN updates the features based on the nodes that are one hop away. To facilitate updates involving more distant nodes and their connections, a multi-layer GCN is required, as the one shown in Fig. 1.

Aggregation over the entire graph can be expressed as a matrix multiplication $\tilde{H} = S\,H^{k-1}$, where $S = D^{1/2}\tilde{A}D^{1/2}$ represents the normalized adjacency matrix, derived from $\tilde{A} = A + I$, i.e., the adjacency matrix A with added self loops, and D, which is the degree matrix of A [4]. Also, H^{k-1} represents the output features of all nodes of layer $k-1$ of the GCN.

The aggregated features of all nodes $\tilde{H}$ are linearly transformed using the weight matrix W^k of the kth layer, and passed through a non-linear activation function σ, such as ReLU, to compute the output features of the kth layer.

This work was supported by a research grant from Siemens EDA to Democritus University of Thrace for "High Level Synthesis Research for Systems-on-Chip"

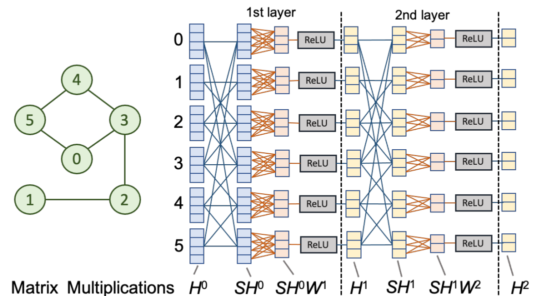


Fig. 1. The dataflow and the computations involved in each layer of a 2-stage GCN for an example 5-node graph.

$$H^k = \sigma\left(\tilde{H}W^k\right) = \sigma\left(SH^{k-1}W^k\right) \tag{1}$$

The last layer of a GCN predicts the labels of the nodes using a softmax classifier on the elements of H of the last layer.

State-of-the-art GCN accelerators compute the double matrix product of (1) in two phases [5]. The first phase (aka *aggregation*) involves a multiplication with the normalized adjacency matrix, while the second phase (aka *combination*) refers to the multiplication with the weight matrix. Aggregation is a sparse$\times$dense matrix multiplication, while combination is a dense$\times$dense matrix multiplication.

In this work, to speedup GCN convolution, we follow a different approach and propose a new systolic architecture, called *FusedGCN*, which computes in a combined/fused manner the product of the three matrices. The operation of FusedGCN can be fully, or partially, unrolled by following a tiled organization that depends on the size of the input and output features of graph nodes, and the available input and output bandwidth. Graph size and sparseness do not affect the hardware complexity, but only the execution time. Sparse blocks of input features found in some applications are also supported.

FusedGCN is evaluated in executing the inference of real-world applications based on two-layer GCNs. In all cases, FusedGCN reduces execution times relative to the current best-performing state-of-the-art accelerator [6] that computes the aggregation and combination phases separately. Most importantly, FusedGCN can achieve these reductions in execution time by incurring only a small hardware overhead (less than 4% in area and around 8% in power), as compared to the classic systolic array that computes a two-matrix product.

II. A Systolic Fused Three-Matrix Multiplier

In this work, our goal is to design an efficient systolic architecture that would compute the triple matrix multiplication $S\,H^{k-1}\,W^{k}$ of Equation (1) in a *fused* manner, while also handling the sparseness of matrix S. For simplicity, we henceforth remove the layer indices from matrices H and W and denote the targeted three-matrix multiplication as $H^{*} = S\,H\,W$.

A. Enabling fused three-matrix multiplication

Let us denote the matrix product $H\,W$ as C. Then, each element c_{jq} of C can be written as the dot product of the jth row of H, $row_j(H)$ and the qth column of W, $col_q(W)$, i.e., $c_{jq} = row_j(H) \cdot col_q(W)$. The final output can be computed using C as $H^{*} = S\,C$. For every element of H^{*}, it follows that $h^{*}_{iq} = \sum_{j=0}^{N-1} s_{ij} c_{jq}$. Substituting the value of c_{jq}, we get

$$h^{*}_{iq} = \sum_{j=0}^{N-1} (s_{ij}\,row_j(H)) \cdot col_q(W) = \sum_{j=0}^{N-1} \sum_{k=0}^{I-1} s_{ij} h_{jk} w_{kq} \quad (2)$$

$$i \in [0, N-1] \text{ and } q \in [0, O-1]$$

To compute Equation (2) we follow two main principles: (a) we completely compute the ith row of the output before moving to the next one; (b) each element of the ith row of S is read only once. To compute all elements h^{*}_{iq} of the ith output row, we need to access all elements of the *same row* of S, i.e., s_{ij}. According to Equation (2), for each element, we should select an associated row of H. The association is performed using the column index of the corresponding element of S, e.g., s_{24} is associated and multiplied with $row_4(H)$. The multiplications between the elements of S and the associated rows of H are the same for all elements of the same output row of H^{*}, and, thus, they can be reused. The only difference between the two outputs of the same row of H^{*} is the selected column of the weight matrix W.

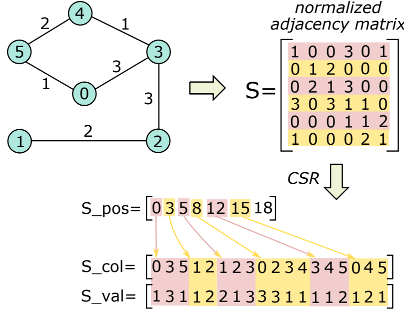


Fig. 2. The CSR format of an example normalized adjacency matrix.

In practice, matrix S is sparse. Therefore, it is stored in a compressed storage format, where only non-zero elements are indexed. Specifically, the Compressed Sparse Row (CSR) format uses three arrays for storing a sparse matrix. Fig. 2 depicts an example graph, its normalized adjacency matrix S, and the CSR representation of said S matrix. Array `S_val` stores the values of the non-zero elements of S in row-wise order. The `S_col` array stores the corresponding column indices of each non-zero value, and `S_pos` contains pointers to the start of each row in `S_col`.

```
for (i = 0; i < N; i++) //temporal
 for (p = 0; p < I; p++) //spatial parallel
  for (q = 0; q < O; q++)  //spatial parallel
   for (j = S_pos[i]; j < S_pos[i+1]; j++) //temporal
    h_star[i][q] += (S_val[j] * h[S_col[j]][p]) * w[p][q];
```

Fig. 3. The fused three-matrix multiplication kernel for a sparse normalized adjacency matrix encoded in CSR format.

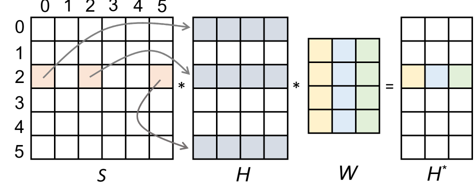


Fig. 4. The elements involved in the fused three matrix multiplication. Computing the 2nd output row requires all non-zero elements of the 2nd row of S, the corresponding columns of H, and all columns of W.

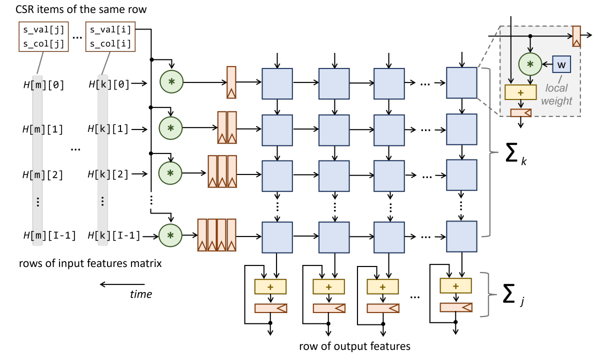


Fig. 5. The weight-stationary FusedGCN systolic array that facilitates fused three-matrix multiplication. The datapath is reused for many cycles until all rows of the output have been computed. The weights of W are preloaded into the systolic array and reused for all elements of S and H.

To transform the fused three-matrix multiplication to operate directly on CSR representation, we need to replace the instances of s_{ij} with the corresponding CSR representation of the non-zero elements of S. The fused three-matrix multiplication kernel for a CSR-encoded matrix S is shown Fig. 3. The elements associated for computing one output row of the result are highlighted in Fig. 4.

The outer loop passes through all the rows of matrix S (each node of the graph is visited once). The non-zero elements of a selected row of S (row 2 in the example of Fig. 4) with value `S_val[j]` are multiplied with all the elements of the row of H that corresponds to the column `S_col[j]`. The product derived for each non-zero element is multiplied with all columns of W to produce a complete row of the output. Since S is sparse, only selected rows of H are fetched: the ones that correspond to the columns of the non-zero elements of S. The same steps are performed in parallel for all elements that belong to the same output row, using the systolic array of Fig. 5.

B. Supporting Arbitrary Input Feature Sizes

The basic FusedGCN systolic architecture of Fig. 5 assumes that a row of matrix H (consisting of I elements) can be fetched as a single entity. This approach is practical and feasible for small values of I, but it does not scale with

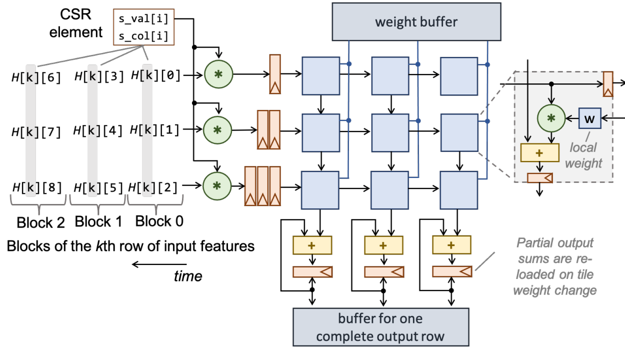


Fig. 6. A tiled version of FusedGCN using a 3 × 3 systolic array for the computation of the tiled fused three-matrix multiplication. The non-zero elements of S arrive serially and index the corresponding row of H. The row is fetched in blocks of three words each. The output row is also computed in blocks using an appropriate tile of weights in each cycle (recycled from the weight buffer).

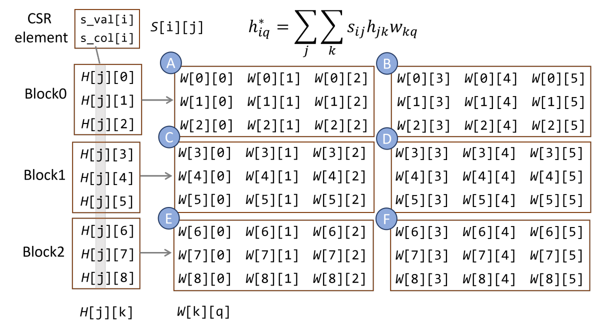


Fig. 7. The non-zero elements of S and the blocks of its associated row of H are multiplied with the tiles of the weight matrix to produce the corresponding partial output result. The example here assumes 9 input features and 6 output features per node, organized in blocks of three elements.

increasing numbers of input features. More realistically, a row of H would be fetched in multiple segments/partitions, over consecutive clock cycles. The same limitation exists on the output side, i.e., when trying to write in memory a computed row of the output H^*. In most practical cases, the computed row would be stored in multiple segments.

The computation engine should match this read/write throughput limits to avoid underutilization. Fig. 6 depicts a 3×3 systolic array used to compute in *tiles* the fused product of a GCN layer with I=9 input features and O=6 output features per node, respectively. Even if the computation evolves in more steps to facilitate the limited input-output bandwidth, the basic rule of the operation of FusedGCN does not change: one output row is first fully computed before moving to the next one.

1) Dense Input Features: Each non-zero element of S indexes the associated row of H, based on its column index. The selected row arrives in blocks of K words and the computation associated with it evolves in I/K phases. In each phase, we compute a partial result for the same output row using the blocks of the input features H for multiple cycles, and by recycling the tiles of the weight matrix. Fig. 7 highlights which elements of $row(H)$ and the weight matrix are involved in each computation step, assuming that $row(H)$ is partitioned in three blocks/segments. Following the fused matrix multiplication principle of Equation (2) to compute the jth output row of H^*, we need to fetch all non-zero elements of the same row of S, i.e., s_{ij}, together with their associated $row_j(H)$. Each s_{ij} is fetched once and waits to be multiplied with all blocks of $row_j(H)$. Each element $s_{ij}h_{jk}$ of this intermediate vector should be multiplied with all w_{kq} for all columns of the weight matrix.

In the example shown in Fig. 7, the intermediate vector produced for 'Block0' should be multiplied (inner product) with all elements of the first three rows of W. Since the systolic array does not have more than three columns, the weights of the first three rows are partitioned in tiles A and B. The systolic array first uses the weights of tile A, and then the weights of tile B, both for 'Block0' of the input features. The partial result computed for the columns of tile A at the output accumulators should be kept and reloaded when computation is performed in weight tiles C and E. To enable this functionality, each column of W has its own dedicated buffered accumulator. When the corresponding column is activated, its buffered accumulator value is loaded into the appropriate accumulators of the systolic array. Once the partial computation is finished, the partial result of the running accumulator is stored back into the buffered accumulator. Similar to the rewind of the partial output results, the tiles of weights are reloaded from the local weight buffer.

2) Sparse Input Features: In many applications, as the ones used in the experimental results [7], the input features of the first layer of the GCN, H^0, are also highly sparse. The remaining layers are mostly dense. To take advantage of this characteristic – and without altering the fundamental operation of FusedGCN – we would like the computation process to skip the blocks of a row of H that consist of only zero elements.

For instance, following the example of Fig. 7, if we know beforehand that 'Block1' of $row_j(H)$ is an all-zero vector, i.e., `H[j][3]=H[j][4]=H[j][5]=0`, then we can completely skip that block and let the systolic array operate only on'Block0' and 'Block2'. To completely skip an all-zero block, we need to encode the blocks of the input features matrix in a CSR format, as the one shown in Fig. 8.

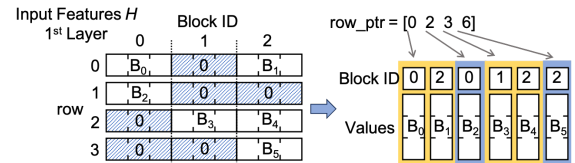
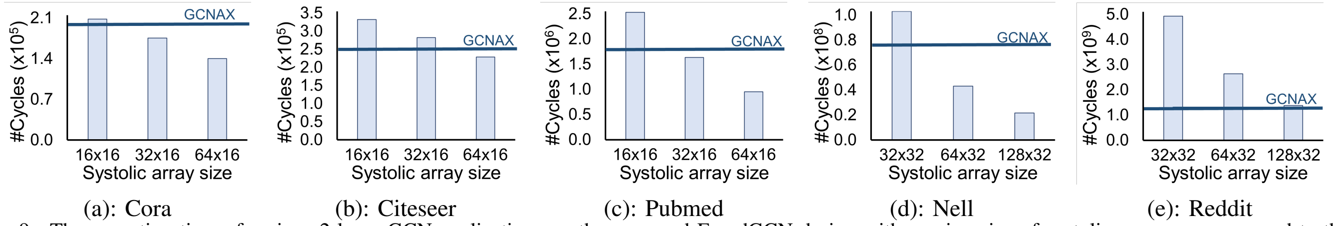


Fig. 8. Encoding in CSR format the non-zero blocks of the input features matrix H of the first layer.

Since only the non-zero blocks of each row of H would be fetched, each block should use its block ID to determine which tiles of the weight matrix would be activated. For a dense matrix H, all the tiles of weights are used and always used in the same order (e.g., $A \rightarrow B \rightarrow \cdots \rightarrow E \rightarrow F$, for the example of Fig. 7). On the contrary, for a sparse matrix H, only the tiles that correspond to non-zero blocks are needed. The block's ID determines which groups of rows of W are used per block.



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

Fig. 9. The execution time of various 2-layer GCN applications on the proposed FusedGCN design with varying size of systolic arrays, as compared to the performance of the GCNAX state-of-the-art architecture [6].

III. Evaluation

Having presented and analyzed the underlying concepts and the salient micro-architectural details of the FusedGCN design, we now proceed with a thorough evaluation of its performance and hardware cost.

A. Application characteristics

Table I summarizes the key characteristics of the 2-layer GCN datasets used in our evaluations. The first notable observation is that, in all applications, the number of input and output features between the first and the second (hidden) layer of the GCN are highly asymmetric. The input features of the first layer of the GCN (denoted as '#Inp. Features' in the table) are two, or even three, orders of magnitude larger than the input features of the second layer (i.e., the outputs of the first layer, denoted as '#Output Features').

TABLE I
The key characteristics of the 2-layer GCN applications used in the evaluations.

	Cora	Citeseer	PubMed	Nell	Reddit
#Nodes	2708	3327	19717	65755	232965
nnz	13264	12431	108365	331899	114848857
Density of S Matrix	0.18%	0.11%	0.028%	0.0077%	0.21%
#Inp. Features H^0	1433	3703	500	61278	602
Density	1.27%	0.85%	10%	0.11%	51.6%
#Hidden Features H^1	16	16	16	64	64
#Output Features H^2	7	6	3	186	41

Secondly, in all applications, the normalized adjacency matrices (i.e., S matrices) are very sparse, with their density ranging roughly from 0.0077% (Nell) to 0.21% (Reddit). Sparseness is also high in the input features of the first layer of all GCNs. For instance, in the Cora and CiteSeer datasets, the number of input features of the first GCN layer is 1433 and 3703, respectively. From those input features, only 1.27% and 0.85% in each case, respectively, are non zero.

B. Execution time comparisons

The proposed FusedGCN architecture is compared against GCNAX [6], which is the most recent state-of-the-art GCN accelerator that exhibits the lowest execution time relative to other highly efficient approaches [8], [9]. The GCNAX design computes aggregation and combination in two separate phases, which take advantage of the sparseness of the normalized adjacency matrix and the possible sparseness of the input features of the first layer of the GCN. GCNAX takes advantage of this attribute in a *fine-grained* manner, whereas FusedGCN employs a *coarser-grained* approach; it only skips *blocks* of the input rows that consist of all zero elements, as previously described in Section II-B2.

The FusedGCN architecture was fully implemented in C++ and synthesized to Verilog RTL using Catapult HLS. The resulting RTL Verilog implementation was validated and verified to ensure functional correctness against the C++ testbench. The execution times reported here were derived after cycle-accurate RTL Verilog simulations of the GCN applications under investigation. The execution times for GCNAX are taken verbatim from [6].

Fig. 9 reports the obtained execution times for the investigated 2-layer GCN applications of Table I. For FusedGCN, execution times are provided for varying sizes of systolic arrays. Only the first dimension of the systolic array is varied, because the performance of FusedGCN is sensitive to the number of blocks with all-zero elements in the *input* features. For Nell and Reddit, we use larger array sizes, because these applications have very large datasets with larger feature sizes in the hidden layer (as shown in Table I). For each application, the execution latency of GCNAX – as reported in [6] – is illustrated as a straight line.

It is evident in Fig. 9 that GCNAX outperforms the proposed FusedGCN design in systolic arrays with a small-size first dimension (corresponding to the input features). This is due to the above-mentioned fine-grained capitalization of sparseness enjoyed by GCNAX. Instead, the block-based approach of FusedGCN is not as effective in taking advantage of sparseness when the input-feature dimension of the systolic array is small. However, as the size of the first dimension of the systolic array increases, FusedGCN overtakes GCNAX in performance.

C. Hardware complexity analysis

To evaluate the hardware complexity of FusedGCN, we compare it against a regular weight-stationary systolic array [10] that computes a standard matrix multiplication of two dense matrices. Structurally, FusedGCN consists of a regular systolic array and an extra column of multipliers at its west input (the green multipliers on the left side of Figs. 5 and 6). To quantify the area/power overhead of the extra multipliers for various configurations, we mapped the Verilog RTL of both designs to a 45 nm standard-cell library using the Oasys logic synthesis tool, assuming a target clock frequency of 1 GHz.

Power was estimated after logic synthesis using the PowerPro power analysis and optimization tool. Switching activity information was gathered after simulating each design using the same random input matrices H and W. To have a fair comparison, we set matrix S equal to the identity matrix, on purpose. Hence, even though FusedGCN computes SHW, it effectively produces the two-matrix product HW.

For each implementation, we examined two different systolic array sizes (16×16 and 32×32), and two different floating point datatypes: 'fp32' for single-precision floating point format and 'bfloat16'. In both designs under evaluation, Catapult HLS derived efficient pipelined floating-point datapaths.

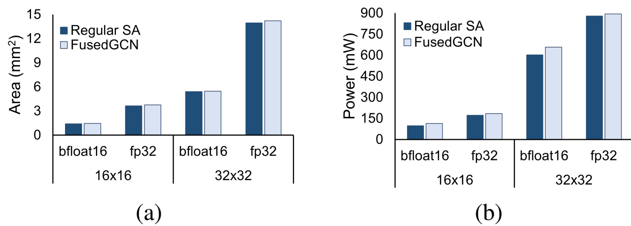


Fig. 10. The (a) area and (b) power cost of FusedGCN, as compared to an equal-size regular Systolic Array (SA), for two different floating point datatypes and two different array sizes.

The goal is to highlight the logic overhead of the extra multipliers in the proposed FusedGCN architecture. Therefore, for both designs, we assume that each PE stores one weight of the same width as the input data using a local register. The obtained area/power results are depicted in Fig. 10. In all examined cases, the complexity of the fused systolic array closely tracks the complexity of the regular systolic array. Due to the extra multipliers, FusedGCN introduces a small area overhead, in the range of 1.2% to 3.6%, as compared to a weight-stationary regular systolic array (see Fig. 10(a)). The average power overhead of FusedGCN, as compared to the regular systolic array, is 8% in all examined cases (Fig. 10(b)).

IV. Related Work

One of the earliest works on GCN acceleration, GraphACT [11] implements a GNN accelerator in a hybrid CPU-FPGA platform. The forward and backward passes of the GNN are computed on the FPGA, while the loss gradients and activation functions are computed on the CPU. Similarly, HyGCN [8], consists of separate aggregation and combination modules. Aggregation utilizes a set of SIMD cores in combination with a sampler and a sparsity eliminator, whereas combination is implemented with a standard systolic array. GNNerator [12] follows a similar approach and optimizes the data flow between the dense matrix engine and the graph engine. Zhang *et al.* in [13] focus on accelerating GCN training on FPGAs. By sampling parts of the adjacency matrix and input features in each training epoch, they show that they can achieve similar quality of results with lower computational complexity.

EnGN [14], inspired by CNN accelerators, treats graph convolution as a concatenated matrix multiplication of feature vectors, adjacency matrices, and weights. Similarly, Auten *et al.* in [15] combine existing accelerator modules designed for DNNs with graph-specific function accelerators.

Other approaches focused more on restructuring the computation involved in GCNs. AWB-GCN [9] removes the workload imbalance introduced by the irregular non-zero distribution in the adjacency matrix, using three workload balancing functions. Also, combination is computed first as a means to reduce the operations needed during the aggregation phase. BoostGCN [16] relies on data reorganization and tiling across all dimensions, i.e., vertices, edges, and input features, to improve memory accessing and increase utilization. Finally, DyGNN [17] dynamically skips vertices and edges that are considered redundant, thus saving computation time without reducing the quality of results.

V. Conclusions

This work introduces for the first time – to the best of our knowledge – a systolic array architecture that can *multiply three matrices in a fused manner* to accelerate graph convolutions. The proposed FusedGCN architecture can readily support the inherent sparseness of the graph adjacency matrix and the possible sparseness of the first layer of input features. The structure of the systolic array and the corresponding flow of data can be fully, or partially, unrolled to adapt to the input and output bandwidth constraints.

Most importantly, even though FusedGCN has targeted the acceleration of GCN-related operations, it is, fundamentally, a generic three-matrix multiplier. Hence, by appropriately placing the data in matrices S, H, and W, or by setting any of the three matrices equal to the unitary matrix, FusedGCN can implement arbitrary two-matrix multiplications, covering both sparse×dense and dense×dense cases.

References

[1] B. Perozzi and et al., "Deepwalk: Online learning of social representations," in *ACM Int. Conf. on Knowledge Discovery and Data Mining (KDD)*, 2014, p. 701–710.

[2] R. Ying and et al., "Graph convolutional neural networks for web-scale recommender systems," in *ACM Int. Conf. on Knowledge Discovery and Data Mining (KDD)*, 2018, p. 974–983.

[3] W. L. Hamilton, *Graph Representation Learning*. Morgan and Claypool, Synthesis Lectures on Artificial Intelligence and Machine Learning, 2020.

[4] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *ICLR*, 2017.

[5] R. Garg and et al., "A taxonomy for classification and comparison of dataflows for gnn accelerators," in *IEEE Int. Parallel and Distributed Processing Symp.*, May 2022.

[6] J. Li and et al., "GCNAX: A flexible and energy-efficient accelerator for graph convolutional neural networks," in *IEEE Int. Symp. on High Perf. Comp. Arch. (HPCA)*, 2021, pp. 775–788.

[7] P. Sen and et al., "Collective classification in network data," *AI Magazine*, vol. 29, no. 3, Sep. 2008.

[8] M. Yan and et al., "HyGCN: A GCN accelerator with hybrid architecture," in *IEEE HPCA*, Feb. 2020, pp. 15–29.

[9] T. Geng and et al., "AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing," in *IEEE Int. Symp. on Microarchitecture*, 2020, pp. 922–936.

[10] A. Samajdar and et al., "A systematic methodology for characterizing scalability of DNN accelerators using SCALE-Sim," in *IEEE Int. Symp. on Performance Analysis of Systems and Software*, 2020, pp. 58–68.

[11] H. Zeng and V. K. Prasanna, "Graphact: Accelerating GCN training on CPU-FPGA heterogeneous platforms," *CoRR*, vol. abs/2001.02498, 2020. [Online]. Available: http://arxiv.org/abs/2001.02498

[12] J. R. Stevens and et al., "GNNerator: A hardware/software framework for accelerating graph neural networks," in *Design Automation Conf. (DAC)*, 2021, pp. 955–960.

[13] B. Zhang and et al., "Hardware acceleration of large scale GCN inference," in *IEEE Int. Conf. on Application-specific Systems, Architectures and Processors (ASAP)*, 2020, pp. 61–68.

[14] S. Liang and et al., "EnGN: A high-throughput and energy-efficient accelerator for large graph neural networks," *IEEE Trans. on Comp.*, vol. 70, no. 9, pp. 1511–1525, 2021.

[15] A. Auten and et al., "Hardware acceleration of graph neural networks," *Design Automation Conf. (DAC)*, 2020.

[16] B. Zhang and et al., "BoostGCN: A framework for optimizing GCN inference on FPGA," in *IEEE Int. Symp. on Field-Programmable Custom Comp. Machines*, 2021, pp. 29–39.

[17] C. Chen and et al., "Dygnn: Algorithm and architecture support of dynamic pruning for graph neural networks," in *Design Automation Conf. (DAC)*, 2021, pp. 1201–1206.