Hybrid Accelerator for GCN Inference

Liangsen Wang
224040364

Abstract—Convolutional neural networks(CNNs) have achieved great success on Euclidean data. Inspired by CNNs, graph convolutional networks(GCNs) were proposed to handle non-Euclidean data. As the scale of data continues to grow, the demand for computing GCN inference at a faster speed is increasing. However, due to the inherent characteristics of GCN and graph data structures, there is a huge challenge in computing acceleration on existing architectures, which leads to the necessity of designing dedicated accelerators. According to the computational characteristics of GCN, hybrid architectures are considered a feasible and efficient accelerator structure. In this work, we summarize several different hybrid GCN inference accelerators. We focus on their hardware architecture and software design, respectively introducing their aggregation engine, combination engine, pipeline that links the two engines, and software algorithms. Furthermore, based on experimental data, we compare the design features of each method and their performance on different datasets, discuss the pros and cons of these designs, and propose some possible future research directions.

1 Introduction

Over the past few years, neural network models have witnessed rapid development. In domains such as text and image processing, models represented by convolutional neural networks(CNNs) have achieved remarkable success[13]. However, these models are based on the Euclidean structure of the data. In the real world, a considerable amount of data is organized in non-Euclidean forms, with graphs being a typical example. To address this type of issue, graph neural networks(GNNs) have been put forward[17]. Among numerous GNNs, graph convolutional networks(GCNs)[9], [33] [22], [31], which drew inspiration from the convolutional neural network model, have garnered the attention of many researchers and promptly emerged as one of the most prevalent approaches. GCNs usually obtain the embedding vectors of vertices through two phases, aggregation and combination[25], [7], [33]. In the aggregation phase, each vertex fuses features from its adjacent neighbors in a certain way, such as accumulation, averaging, etc[9]. During this phase, the operations executed by GCN are based on the graph structure. Unfortunately, graph structures typically exhibit a high degree of randomness and sparsity[10]. On the one hand, this results in different numbers of neighbors for each vertex, leading to a load imbalance. On the other hand, random memory access makes it challenging to utilize the locality of memory, and the computing performance will be significantly affected[1]. However, this situation is completely different in the combination stage[19]. The combination stage is more akin to a traditional neural network, where it employs a multi-layer perceptron (MLP) to transform the feature vector of each vertex into a new one.

Apart from the distinctions among the phases, the two phases of GCN also differ from traditional graph analysis. Firstly, in GCN, the feature attributes of vertices tend to be several times more than those in traditional graph analysis. Moreover, in traditional graph analysis, the length of the vertex attribute vector is fixed, but in GCN, the length of the attribute vector in each layer varies. Secondly, the parameters of the traditional MLP-based neural network are not shared, whereas in the combination phase of the GCN, the weight parameters of the MLP are shared among vertices, resulting in a considerable amount of reusable data. Third, the two distinct phases in GCN are executed alternately.

With the advancement of GCN, there are increasing demands for accelerated inference. Nevertheless, due to the aforementioned characteristics, the acceleration of GCN inference confronts substantial challenges[23], [30]. Large graphs with millions of vertices are not amenable to designing efficient and compact GCN accelerators within the confines of limited on-chip memory. The degree distribution of vertices is highly uneven since real-world graphs typically follow a power-law distribution[4]. This leads to extensive memory traffic, irregular memory access, and a workload imbalance. In addition, irregular memory access results in limited data reuse. Existing computing platforms are incapable of effectively coping with these challenges. In CPUs, despite the presence of complex multilevel caches and data prefetching strategies, the unpredictability of GCN's random data access makes it arduous to apply the existing cache strategies efficiently, leading to inefficient data reuse[6]. Moreover, CPUs have difficulty leveraging the inherent parallelism executed in the above two stages. GPUs are designed to employ highly parallel computing for regular data access and processing, which is contrary to the inherent randomness of graph structures. Concurrently, GPUs have difficulties handling the data reused among vertices[26].

Based on the foregoing reasons, the accelerator structure with two phases can be optimized in accordance with different characteristics and is more appropriate for GCN inference. In this work, we initially describe the workload of GCN in the Intel Xeon CPU to elucidate the operational characteristics of GCN. Subsequently, we investigated three distinct hybrid-structured GCN inference accelerators and summarized their aggregation engines, combination engines, cross-engine pipelines, and software algorithms. HyGCN[27] exploits the parallelism within vertices to real-

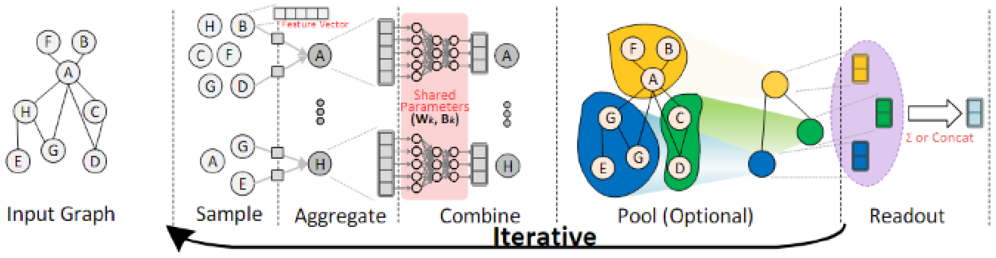


Figure 1: Illustration of the GCN model.

ize load balancing. The structural design based on downstream tasks[14] utilizes pre-trained models to generate load-balanced subgraphs. GPGCN[18], on the contrary, is based on the open-source RISC-V instruction set and takes advantage of its flexibility to design a dedicated extension instruction set for GCN inference, accelerating while maintaining a certain degree of programmability. In conclusion, we enumerate our contributions as follows:

1. We analyzed the disparities between GCN and traditional graph analysis as well as other neural networks with Euclidean structures, and proposed the superiority of the hybrid structure accelerators.
2. Summarized the three existing hybrid-structured GCN inference accelerators, analyzed their design concepts and pros and cons.
3. Put forward some open questions.

The rest of this paper is organized as follows. Section 2 discusses the related work on GCN inference acceleration and RISC-V. In Section 3, we analyze the characteristics of GCN and explain the motivation for the hybrid design. Section 4 describes the detailed structure design. Section 5 introduces the experimental setup and results. Finally, Section 6 concludes the paper.

2 Related Work

2.1 Graph Convolutional Network

Fig. 1 illustrates the execution stages of the GCN model. GCN follow a neighborhood aggregation scheme[33], [8], [24], where the feature vector of each vertex is computed by recursively aggregating and transforming the representation vectors of its neighbor vertices. The notations used in GCNs are listed in Table 1

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 an MLP neural network. Note that the MLP parameters are shared between vertices. Each GCN layer can be foromulated as follows:

$$\alpha_v^l = Aggregate(h_u^{l-1} : u \in N(v) \cup \{v\}) \tag{1}$$

$$h_v^l = Combine(\alpha_v^l) \tag{2}$$

where h_v^l is the embedding vector of vertex v in the layer l.

Typically, given adjacent matrix A and feature matrix X, the detailed graph convolution operation can be generalized as follows:

$$X^{l+1} = \sigma(\widetilde{D}^{-\frac{1}{2}}\widetilde{A}\widetilde{D}^{-\frac{1}{2}}X^l W^l) \tag{3}$$

Notation	Description	Notation	Description
G	Input graph	D_v	Degree of node v
V	Nodes of G	A	Adjacent matrix
v_i	The i^{th} node	h_v	Feature vector of node v
E	Edges set of G	X	Feature matrix
$e_{(ij)}$	Edge from v_i to v_j	W	Shared weight matrix
N	Number of nodes	f	Length of feature vector
$N(v)$	Neighbor set of node v	L	Number of GCN layers
a_v	Aggregation feature of v		

Table 1: Notation and description table

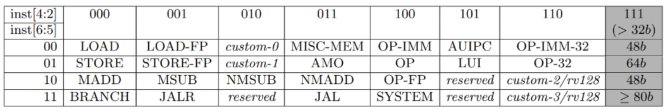
where $\widetilde{A} = A + I$ is the adjacent matrix with self-loop, $D_{ii} = \sum_j \widetilde{A}_{ij}$ is Laplacian matrix. $\sigma(\cdot)$ is the activation function.

To reduce the computational complexity and increase the scalability of GCN, some models apply a sampling function before applying the aggregation function to reduce the number of computed neighbors[8]. Sometimes, the Pool function[28] follows the Combine function to transform the original graph into a smaller graph. For the graph classification, a Readout function[3] is used to obtain the entire graph's representation vector.

2.2 RISC-V Instruction Set Architecture

The open-source RISC-V[21] ISA is based on the RISC philosophy. Its architecture is a basic integer ISA plus a set of optional standard and non-standard extensions to customize and specialize the final instruction set. There are two main base integer ISAs, RV32I and RV64I, which respectively establish the user address space as 32-bit or 64-bit.

The standard supports extending the RISC-V ISA with special extensions to allow for custom accelerators. Non-standard extensions can be added to the code space by utilizing the four main opcode prefixes reserved for custom extensions: custom0, custom1, custom2, and custom3, as shown in Fig. 2.



inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 (> 32b)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	$\geq$ 80b

Figure 2: The custom instruction encoding space of RISC-V.

3 Motivation

In this section, we quantitatively analyzed the execution patterns of the two phases of GCN and explained the motivation behind the hybrid design.

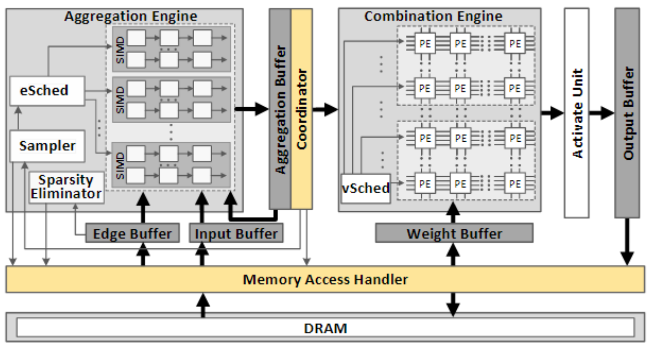


Figure 3: Architecture overview of HyGCN

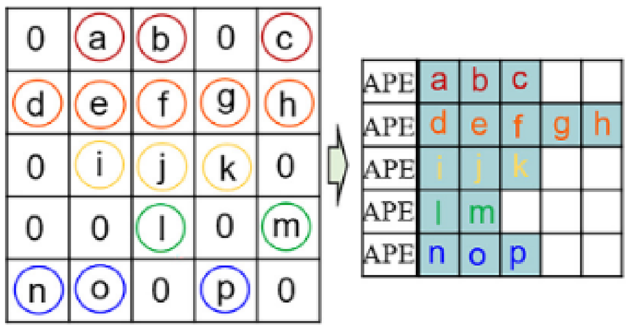


Figure 4: Vertex-concentrated mode workload distribution strategy

3.1 Characteristics of GCN

We use the most advanced GCN software framework PyTorch Geometric (PyG)[7] for quantitative representation on Intel Xeon CPU.

The aggregation phase is highly reliant on the inherent random and sparse graph structure, which gives rise to a considerable amount of dynamic computation and irregular access. Meanwhile, the combination phase incurs significant overhead in data reuse and synchronization. As shown in Table 2, each operation in the aggregation phase demands much more data to be accessed from DRAM than the combination phase, resulting in higher DRAM access energy. Furthermore, the extremely high numbers of misses per kilo-instruction (MPKI) of L2 and L3 caches in the aggregation phase are triggered by the high randomness of neighbor indices of each vertex. Additionally, the indirect and irregular accesses render data prefetching in the aggregation phase ineffective, as it is challenging to predict the data addresses without knowing the indices of neighbors in advance. This leads to numerous ineffective memory accesses for prefetching data.In combination phase, the replication of shared data and thread synchronization accounted for up to 36% of the execution time.

	Aggregation	Combination
DRAM Byte per Ops	11.6	0.06
DRAM Access Energy per Ops	170nJ	0.5nJ
L2 Cache MPKI	11	1.5
L3 Cache MPKI	10	0.9
Ratio of Synchronization Time	-	36%

Table 2: Quantitative characterization on CPU

Based on the above analysis, we summarize the characteristics of the two phases in Table 3 and compare their differences. The aggregation phase exhibits a dynamic and irregular execution pattern that is bounded by memory, whereas the combination phase is static and regular, being bounded by computation.

	Aggregation	Combination
Access Pattern Indirect	Indirect & Irregular	Direct & Regular
Data Reusability	Low	High
Computation Pattern	Dynamic & Irregular	Static & Regular
Computation Intensity	Low	High
Execution Bound	Memory	Compute

Table 3: Different execution patterns of two phases.

3.2 The Inefficiency of Existing Architecture

For the CPU, the irregularity of the aggregation phase makes the existing cache strategies not be effectively utilized. Meanwhile, the data and instruction optimization techniques based on prefetching are not applicable to the discrete and distributed graph data. GPUs are inherently optimized for compute-intensive workloads featuring regular execution patterns, such as neural networks[12]. However, handling the aggregation phase involving irregular memory accesses suffers from low efficiency. Besides, the processing of the combination phase with strong parameter sharing requires costly data copy and thread synchronization. Both CPUs and GPUs fail to have inter-phase optimization for GCN execution due to phase-by-phase execution.

Although specialized accelerators designed for graph analysis or neural networks have achieved significant acceleration and energy savings compared to general-purpose processors[32], [16], their single-format design makes them unsuitable for GCN inference. In addition, the longer and variable length of feature vectors in GCN and the more reusable data also make traditional graph analysis accelerators inefficient in handling GCN tasks.

4 Architecture Design

In this section, based on the above analysis, we describe three different hybrid GCN inference accelerator designs from both hardware and software perspectives.

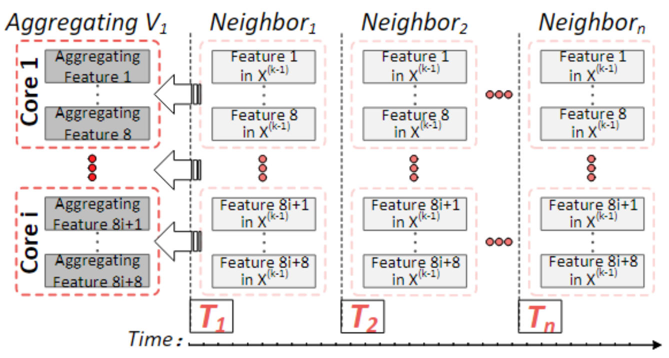


Figure 5: Vertex-disperse mode workload distribution strategy

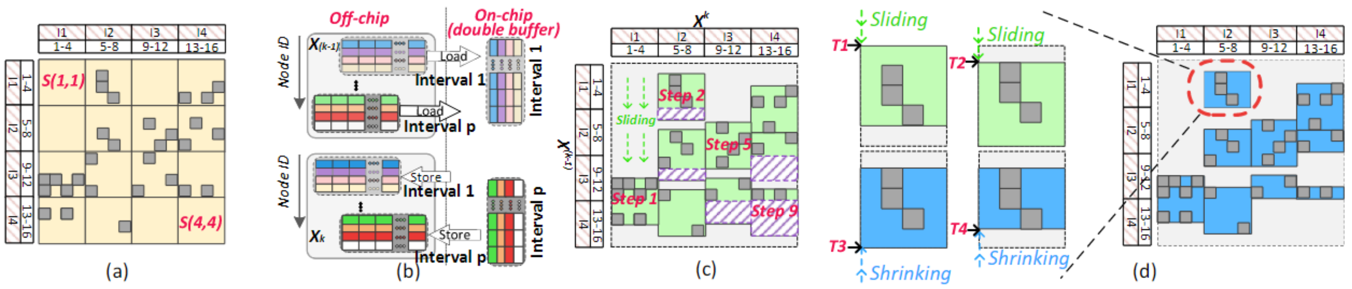


Figure 6: Graph partition, window sliding and window shrinking

4.1 HyGCN

Fig. 3 depicts the architecture of HyGCN which includes two engines (Aggregation Engine and Combination Engine) and one memory access handler. A communication interface (Coordinator) is introduced to bridge these two engines.

4.1.1 *Aggregation Engine*

In the aggregation engine, each SIMD core is an aggregation processing element(APE). We have two ways to utilize these SIMD cores for parallel edge computation. The first one is vertex-concentrated. As Fig. 4 shown, the edges incident to a vertex are successively passed into the APE, and each APE performs a single feature aggregation computation for a single vertex at the same time. This pattern can generate vertex aggregation features in burst mode, i.e. periodically processing a group of vertices. However, the processing delay of a single vertex (called vertex delay) is very long, and fast vertices must wait for slow vertices, leading to an imbalance in the workload. Furthermore, it loses the parallelism that can be executed in parallel for each element's aggregation (i.e. vertex-level parallelism). The second approach is the Vertex-Disperse mode, in which we effectively utilize the inherent parallelism in the node internal aggregation process. The workload in this mode is shown in Fig. 5, where we assign the elements within the vertex feature vector of each vertex to all cores. If a vertex cannot occupy all the cores, idle cores can be assigned to other vertices. Therefore, all cores are always busy, and there is no workload imbalance. Furthermore, since we utilize the intrinsic parallelism within a vertex, the vertex delay per vertex is smaller than that of processing multiple vertices together. Additionally, it can immediately process each vertex in the following combination engine.

In addition to load balancing, data reuse is also very important during the aggregation phase. Because the feature vectors at each vertex are usually very large, we use block-wise computation to increase locality of features. As exampled in Fig. 6(a), the vertices are divided into several intervals and edges are organized as shards. We group the vertices within the same interval together (e.g. I_i), and then process the aggregation of their source neighbors also interval by interval(i.e. traverse I_j) as shown in Fig. 6(b). In this way, we can increase the reuse of feature vectors and intermediate results.

However, merely through the aforementioned approaches, we still have not addressed the issue of sparsity. Subsequently, through a window sliding method, we have mitigated the sparsity.

As shown in Fig. 6(c), we create a window of interval size and slide it downwards until the first row with a border appears. After the window sliding optimization, although we have eliminated most of the sparsity at the top, the sparsity at the bottom is still retained. To further optimize this part of the sparsity, we further use the window shrinking method as shown in Fig. 6(d). Specifically, after the window sliding, we fix the top of the window, and then move the bottom of the window upwards until there is at least one border at the bottom. In this way, only the feature data of remaining neighbor vertices when performing the aggregation operation for each interval are loaded, which eliminates plenty of redundant accesses.

4.1.2 *Combination Engine*

In the combination engine, our design is based on the well-known systolic array. Corresponding to the two modes of the aggregation engine, the composition engine possesses two execution modes, namely the independent mode and the cooperative mode.

In the independent mode, each systolic module operates independently. As depicted in Fig. 7(a), under such circumstances, each module processes a small group of vertices. While in cooperative mode, these systolic modules can be further assembled together to simultaneously process more vertices, as shown in 7(b).

Besides, in independent mode, the weight parameters for each module in this case are directly accessed from the weight buffer and just reused within module(see Fig. 8(a)). The advantage of this mode is the lower vertex latency because we can process the combination operations of this small group of vertices immediately once their aggregated features are ready, without waiting for more vertices. This

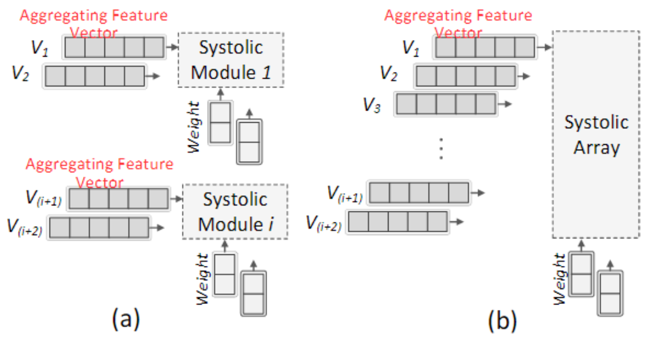


Figure 7: Different use of the systolic arrays

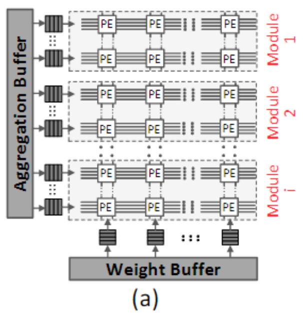


Figure 8: Combination engine design

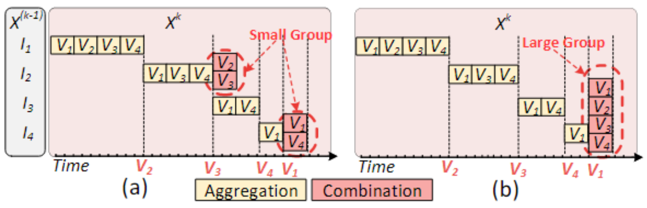


Figure 9: Timing illustration of different pipeline modes

mode matches well with the vertex-disperse processing mode of aggregation engine, where the aggregated features are produced quickly but sequentially.

Different from the immediate processing of vertices, cooperative mode requires to assemble the aggregated features of a large group of vertices together before performing their combination operations. The advantage is that, the weight parameters can flow from the weight buffer to the downstream systolic modules and then gradually to the upstream ones which are greatly reused by all systolic arrays(as Fig. 8(b) show). This helps reduce the energy consumption.

4.1.3 *Inter-Engine Optimization*

To efficiently fuse the phase-by-phase execution a coordinator is designed and provide two pipeline modes, latency-aware and energy-aware.

In Latency-Aware Pipeline, the combination engine works in independent mode. The aggregated features are produced vertex by vertex in the aggregation engine. The following combination will be processed immediately once the aggregated features of a small group of vertices are ready like Fig. 9(a) shows.

The Energy-Aware Pipeline use cooperative mode in the combination engine. The vertex-by-vertex processing changes to a burst mode, where a large group of vertices will be processed like Fig. 9(b) shows.

4.2 Downstream Task Based Design

This work propose another framework to achieve efficient GCN inference acceleration which based on the downstream task. The overview of the system hardware architecture is shown in Fig. 10. Similar to HyGCN, it has an aggregation engine, a combination engine, and a buffer connecting the two engines. The difference is that before computing the GCN, the AM-based graph sparsification approach is first adopted to balance the degree distribution of the original graph and compress the GCN model at the algorithm level.

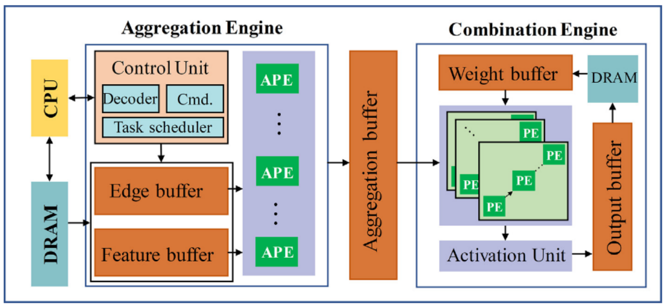


Figure 10: Overview of downstream task based design system hardware architecture.

4.2.1 *Attention-Mechanism Based Graph Sparsification*

Despite the extreme sparsity of graph, there are still many redundant edges in the graph structure. These redundant edges are irrelevant to the downstream task and harm the model performance in turn. From hardware perspective, more edges require more memory access. This inspired us to propose graph sparsification as preprocessing in the GCN inference.

The key idea of the attention-mechanism based graph sparsification is to take the attention coefficients as the importance indicator for the down-stream task. The attention coefficients can be calculated similar to graph attention network (GAT)[20], [5], and be utilized to determine whether that edge shall be used for neighbor aggregation through a predefined threshold. The attention coefficients of all the neighbors of the target node are normalized using the Softmax function, which can be employed to represent the relative significance of the neighboring nodes. Consequently, the edges corresponding to the top k attention coefficients can be retained from the input graph, while the remaining edges are eliminated, thereby obtaining a sparse k-neighbor subgraph. The advantage of this approach is twofold: firstly, from an algorithmic perspective, the number of edges relevant to the task in the graph can be adjusted by optimizing the hyperparameter k, and different k values can be chosen to best fit graphs with different features and downstream tasks. Secondly, from a hardware perspective, the k-neighbor subgraph is very friendly to parallel comput-

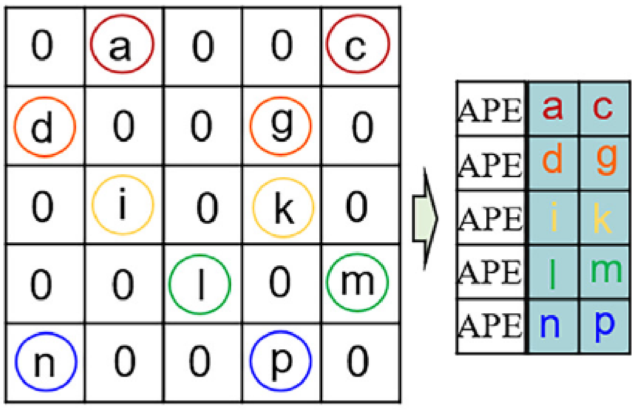


Figure 11: Workload of k-neighbor subgraph

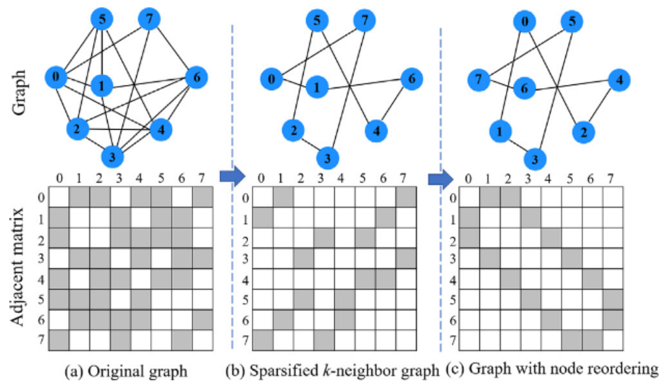

Graph
Adjacent matrix
(a) Original graph
(b) Sparsified k-neighbor graph
(c) Graph with node reordering

Figure 12: A toy example of graph sparsification and node reordering without self-loop.

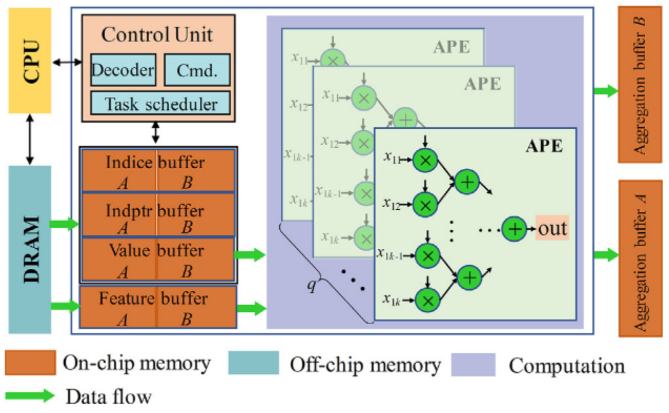


Figure 13: Overview and data flow of aggregation engine.

ing during inference. Since each node has the same number of neighbors, it is possible to effectively avoid workload imbalance in the hardware architecture, providing convenience for improving computational parallelism(like Fig. 11 shows).

4.2.2 Node Reorder

Due to the random node layout, there will be a lot of memory accesses during the GCN inference process, which limits the data reuse[29]. In this design, we reorder the nodes and obtain a new adjacency matrix, where the non-zero elements tend to be more compact near the diagonal. Fig. 12(b)(c) graphically describe our graph sparsification and node reordering. By the above processing, we can further improve the data locality by grouping adjacent nodes, thereby reducing external memory accesses and improving data reuse.

4.2.3 Aggregation Engine

For a large graph it is difficult to implement the entire graph onto the chip. We use common approach to split the graph into partitions, which fit to limited on-chip resources and can be computed separately, this also improve data reuse.

Fig. 13 shows the overview of the architecture. We store adjacent matrix in CSR format. Each edge is stored as a three element tuple src, dst, val representing the source indices, destination pointers, and edge values respectively. So we design 3 buffer to cache them, indice buffer, indptr buffer and value buffer. The control unit transmits a data loading command to the buffer, loading the adjacency matrix in CSR format into the buffer and loading the feature vectors from DRAM into the feature buffer. Subsequently, the decoder generates the destination address to index the corresponding node feature vectors. Then, the indexed feature vectors are transferred from the feature buffer to the APEs, and the corresponding edge weights are passed from the value buffer to the APEs to conduct multiplication and accumulation operations in parallel. Subsequently, the aggregated results are written to the aggregation buffer. A double-buffering strategy is applied within the aggregation engine to conceal the data loading latency.

4.2.4 Combination Engine

Well-known one-dimensional systolic arrays are employed as the core architecture to carry out the multiplication between a batch of feature vector blocks and a weight matrix.

All the on-chip buffers in the combination engine, including the weight buffer and output buffer, are double-buffered to conceal the data transmission latency. The aggregation and weight buffers supply the systolic arrays, and the output of the systolic arrays is sent to the activation unit; the results are written into the output buffer. The results in the output buffer can either be written back to the DRAM or be reused, as this buffer is capable of caching partial results, which will be utilized for accumulation.

4.3 GPGCN

Different from the two aforementioned methods, GPGCN is dedicated to designing a more programmable architecture and optimizing it based on the computational characteristics of GCN while maintaining the software's degree of freedom. GPGCN compresses the encoding of macro instruction with many operations into RISC-V instruction, which has only 32 bits of encoding space. The architecture registers of the vector/matrix are divided into source register (or rs register) and destination register (or rd register), and the rs register corresponds to the rd register one-to-one and is used together; that is to say, as long as the index of the rd register is specified in the instruction, the rs register index is also specified.

4.3.1 Custom CSR

Owing to the two difficulties in designing macro instructions, namely encoding the numerous instruction operation information necessary in the limited RISC instruction encoding space and guaranteeing the degree of programming freedom. The first issue is addressed by customizing the CSR (Current Status Register) register to provide some common auxiliary information among instructions, thereby reducing the instruction encoding pressure.

4.3.2 Register Extension

The GPGCN instruction set architecture contains two types of register group: the vector register group and the matrix register group. The ninth custom CSR register specifies the

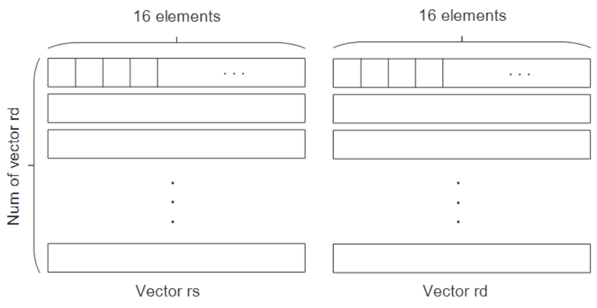


Figure 14: The custom vector registers file.

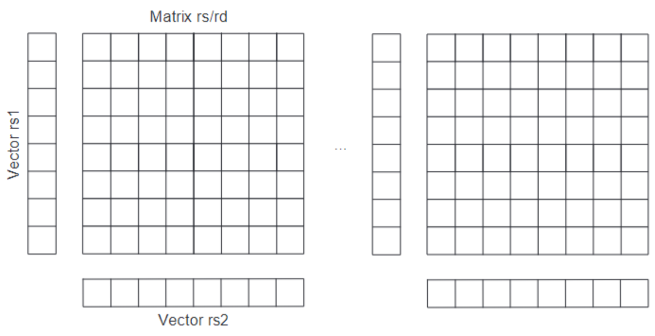


Figure 15: The custom matrix registers file.

number of vector register pairs in the vector register group. Each pair of vector registers contains one vector rs register and one vector rd register, and each vector register contains 16 32-bit single-precision floating-point elements, as shown in Fig. 14.

The 10th custom CSR register specifies the number of matrix registers in the matrix register group. Each pair of matrix registers contains two vector rs registers, one matrix rs register, and one matrix rd register, as shown in Fig. 15. The vector rs register contains 8 32-bit single-precision float point elements. The matrix rs/rd register contains 8 $\times$ 8 32-bit single-precision float point elements to support outer product operations.

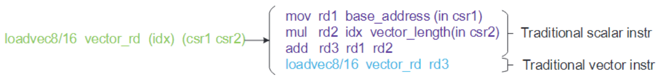
Eight vector rs/rd registers can be combined into a matrix rs/rd register, so that vector and matrix operations can reuse register resources and improve hardware utilization efficiency.

4.3.3 *Basic Vector Instructions*

The basic vector instruction is similar to the traditional SIMD instruction, and defines some basic vector load, store, add, and mov operations. We take load vector instruction as an example, The hardware will calculate the final memory access address through custom CSR1 (base address of feature matrix) and custom CSR2 (feature vector length) using the formula:

$$\begin{aligned} &load/store\ address = \\ &base\_address(CSR1) + idx \times vector\_length(CSR2) \end{aligned} \tag{4}$$

Thus, a basic vector load/store instruction is equivalent to a combination of three traditional scalar instructions and one traditional SIMD instruction, as shown in Fig. 16.



```
loadvec8/16  vector_rd  (idx)  (csr1 csr2)
mov  rd1  base_address (in csr1)
mul  rd2  idx  vector_length(in csr2)
add  rd3  rd1  rd2
loadvec8/16  vector_rd  rd3
```

Figure 16: The basic vector load/store instructions equivalent.

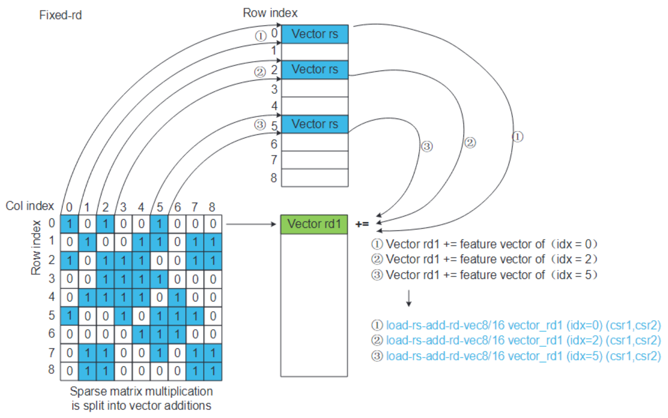
4.3.4 *Fixed-Rd Vector Instructions*

The fixed-rd and fixed-rs instructions are designed according to GCNs network aggregation calculation characteristics. They complete the primary process of aggregation calculation in the GCNs network and provide a certain degree of programming freedom for different software algorithms.

The fixed vector rd calculation mode represents the aggregation calculation process of the feature vectors of all the neighbors of a vertex, as shown in Fig. 17. Because it will always reuse a vector rd to store the intermediate results of the aggregation calculation, it needs to continuously load the feature vectors of different neighbors to the corresponding vector rs for accumulation until the final aggregation result of this vertex is calculated. Hence, the fixed vector rd is for multiplexing data (aggregated intermediate results) in vector rd. In the fixed-rd calculation mode, the data in the vector rd register are multiplexed. In contrast, the data in the vector rs register are not multiplexed, each time we just need to load data to rs register. This saves encoding space to fuse the loadvec instruction and the addvec instruction making the vector instruction of GPGCN more macro and increasing the instruction information density, and the instruction bandwidth is improved.

As shown in Fig. 18 the operations performed by the load-rs-add-rd-vec8/16 instruction include the following: calculate the address of the specified feature vector according to the index, then load the feature vector from memory to the corresponding vector rs according to this address, and then sum vector rd and vector rs and store the result into vector rd.

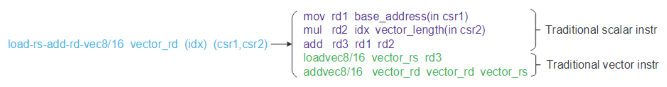
Also since there is no need to specify the index of the vector rs register, we bind rd to rs as described before, there is additional free coding space available, so this coding space can be used as the index of another integer register, which stores the row index of another feature vector so



① Vector rd1 += feature vector of (idx = 0)
② Vector rd1 += feature vector of (idx = 2)
③ Vector rd1 += feature vector of (idx = 5)

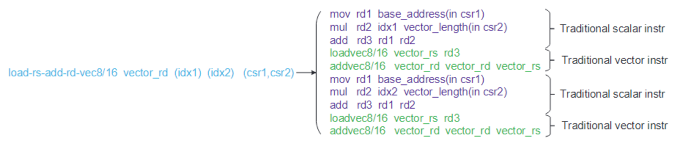
```
① load-rs-add-rd-vec8/16 vector_rd1 (idx=0) (csr1,csr2)
② load-rs-add-rd-vec8/16 vector_rd1 (idx=2) (csr1,csr2)
③ load-rs-add-rd-vec8/16 vector_rd1 (idx=5) (csr1,csr2)
```

Figure 17: The computation process of fixed-rd mode.



```
load-rs-add-rd-vec8/16  vector_rd  (idx)  (csr1,csr2) →
    mov  rd1  base_address(in csr1)
    mul  rd2  idx  vector_length(in csr2)        Traditional scalar instr
    add  rd3  rd1  rd2
    loadvec8/16  vector_rs  rd3
    addvec8/16  vector_rd  vector_rd  vector_rs  Traditional vector instr
```

Figure 18: The load-rs-add-rd-vec8/16 instruction equivalent.



```
load-rs-add-rd-vec8/16  vector_rd  (idx1)  (idx2)  (csr1,csr2) →
    mov  rd1  base_address(in csr1)
    mul  rd2  idx1  vector_length(in csr2)       Traditional scalar instr
    add  rd3  rd1  rd2
    loadvec8/16  vector_rs  rd3
    addvec8/16  vector_rd  vector_rd  vector_rs  Traditional vector instr
    mov  rd1  base_address(in csr1)
    mul  rd2  idx2  vector_length(in csr2)       Traditional scalar instr
    add  rd3  rd1  rd2
    loadvec8/16  vector_rs  rd3
    addvec8/16  vector_rd  vector_rd  vector_rs  Traditional vector instr
```

Figure 19: Another load-rs-add-rd-vec8/16 instruction equivalent.

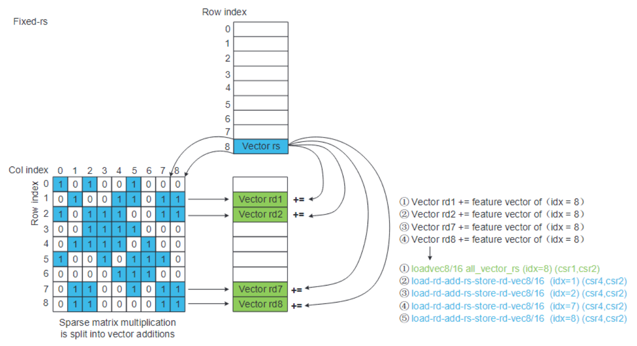
that a load-rs-add-rd-vec8/16 instruction can calculate the aggregation process of two feature vectors, and further increase instruction density(like Fig. 19 shows).

4.3.5 Fixed-Rs Vector Instructions

The fixed-rs vector instruction represents the calculation mode of fixed vector rs in the aggregation calculation. Because a vertex may be a neighbor of multiple vertices, the feature vector of this vertex will be shared by the aggregation calculation of these neighbors, as shown in Fig. 20. Therefore, the feature vector of this vertex is reusable in the aggregation calculation of different neighbors. The instruction is shown in Fig. 21.

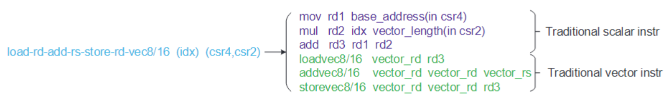
4.3.6 Matrix Instruction Extension

The design of matrix-type instructions is based on the storage format of the matrix referred to previously. As shown in Fig. 22, the input and output matrices are divided into blocks in the combination operation of the GCNs. The block



```
① Vector rd1 += feature vector of（idx = 8）
② Vector rd2 += feature vector of（idx = 8）
③ Vector rd7 += feature vector of（idx = 8）
④ Vector rd8 += feature vector of（idx = 8）
↓
① loadvec8/16 all_vector_rs (idx=8) (csr1,csr2)
② load-rd-add-rs-store-rd-vec8/16  (idx=1) (csr4,csr2)
③ load-rd-add-rs-store-rd-vec8/16  (idx=2) (csr4,csr2)
④ load-rd-add-rs-store-rd-vec8/16  (idx=7) (csr4,csr2)
⑤ load-rd-add-rs-store-rd-vec8/16  (idx=8) (csr4,csr2)
```

Figure 20: The computation process of fixed-rs mode.



```
load-rd-add-rs-store-rd-vec8/16  (idx)  (csr4,csr2) →
    mov  rd1  base_address(in csr4)
    mul  rd2  idx  vector_length(in csr2)        Traditional scalar instr
    add  rd3  rd1  rd2
    loadvec8/16  vector_rd  rd3
    addvec8/16  vector_rd  vector_rd  vector_rs  Traditional vector instr
    storevec8/16  vector_rd  vector_rd  rd3
```

Figure 21: The load-rd-add-rs-store-rd-vec8/16 instruction equivalent.

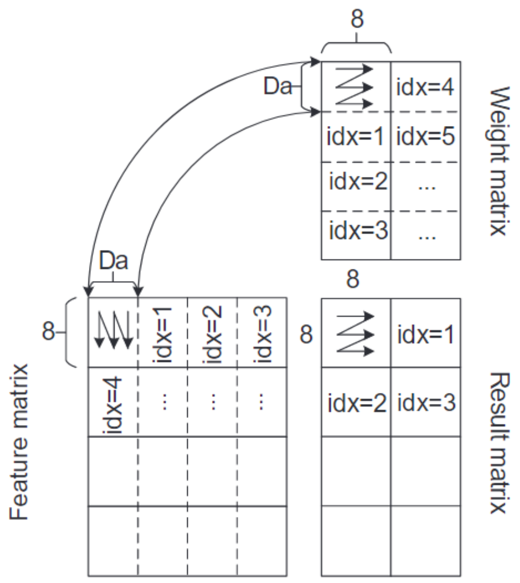


Figure 22: The computation process of matrix mode.

is indexed according to the index number. For input matrix, the size of a block is 8 × Da elements where Da is specified by custom CSR7; for output matrix is 8 × 8.

In order to further increase the instruction density of the matrix instructions, the matrix multiplication operation of $8 \times Da \times Da \times 8 = 8 \times 8$ Therefore, the instrction can be designed as:

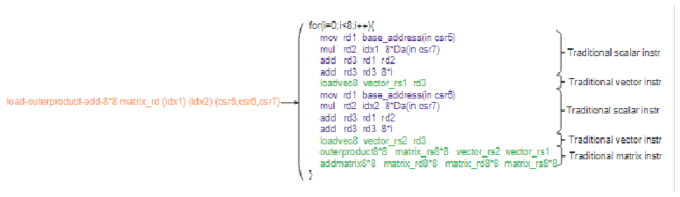
The (idx1) and (idx2) of this instruction specify the block index of the feature matrix and weight matrix, respectively, which are used to calculate the starting address of the block data to be accessed by hardware using Formulas.

The load-outerproduct-add-8*8 matrix_rd (idx1) (idx2) instruction is split into Da times as Fig. 23. Load one column of the feature matrix block (eight elements), load the corresponding row of the weight matrix block (eight elements), and then perform the outer product of $8 \times 1 \times 1 \times 8$ to obtain the 8 × 8 result matrix, sum with matrix rd, and then store them in matrix rd.

4.3.7 Memory Access Extension

There are three memory-related operation instructions as auxiliary instructions:

The *preload instruction* preloads the feature matrix or weight matrix we want to access to the corresponding storage block of the scratchpad memory in advance.



```
load-outerproduct-add-8*8 matrix_rd (idx1) (idx2) (csr5,csr6,csr7) →
for(i=0;i<8;i++){
    mov  rd1  base_address(in csr5)
    mul  rd2  idx1  8*Da(in csr7)                Traditional scalar instr
    add  rd3  rd1  rd2
    add  rd3  rd3  8*i
    loadvec8  vector_rs1  rd3                    Traditional vector instr
    mov  rd1  base_address(in csr6)
    mul  rd2  idx2  8*Da(in csr7)                Traditional scalar instr
    add  rd3  rd1  rd2
    add  rd3  rd3  8*i
    loadvec8  vector_rs2  rd3                    Traditional vector instr
    outerproduct8*8  matrix_rs8*8  vector_rs2  vector_rs1
    addmatrix8*8  matrix_rd8*8  matrix_rd8*8  matrix_rs8*8   Traditional matrix instr
}
```

Figure 23: The matrix instruction equivalent.

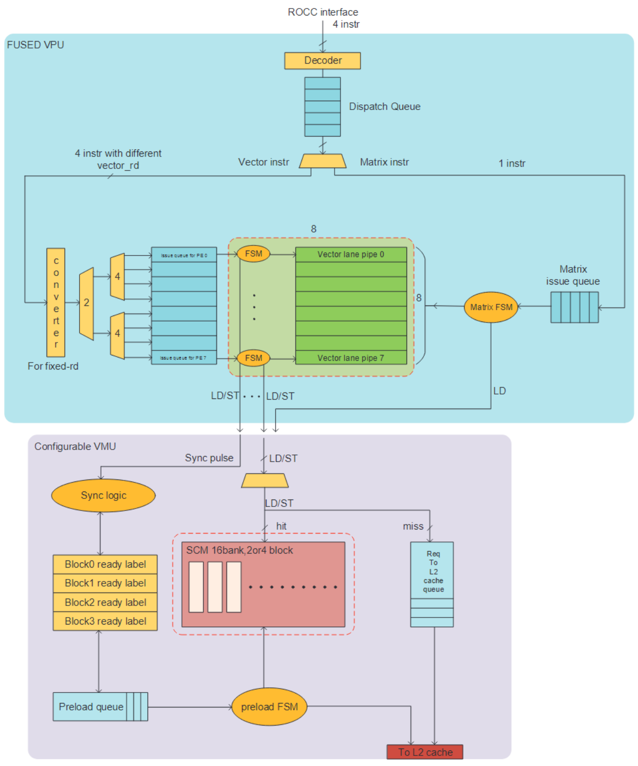


Figure 24: The hardware architecture of GPGCN.

The *sync-preload instruction* is used to indicate that the data in a block of scratchpad memory are no longer used and can be replaced.

The *storescmback instruction* is used to write a block of scratchpad memory back to the main memory. For example, after the fixed-rs instructions calculate the final result, the final result in scratchpad memory is written back to the main memory.

4.3.8 Hardware Architecture

The GPGCN hardware accelerator is coupled with the boom RISC-V cpu core through the rocc interface.

As shown in Fig. 24, the hardware micro architecture of the GPGCN accelerator consists of two parts: fused VPU (vector process unit) and configurable VMU (vector memory unit). The fused VPU combines the execution units of vector instructions with the execution units of matrix instruction, that is, the execution unit array in the VPU can be configured as N SIMD8 vector pipelines to execute vector instructions or can be configured as M 88 array to calculate the matrix instructions, which improves the utilization efficiency of the execution units.

The VMU can be configured into three different modes according to the execution of different instruction streams: the matrix mode, which supports the memory access mode of matrix instructions, the fixed-rd mode that supports the memory access mode of fixed-rd instructions in vector instructions, and the fixed-rs mode that supports the memory access mode of fixed-rs instructions.

4.3.9 Redundant Computation Reduction

There are many hidden redundant calculations in the aggregation calculation process of the GCNs. Different vertices may need to accumulate the same feature vectors. In fact, these redundant accumulation calculations only need to be calculated once. In this design, we simply precompute the sum of the feature vectors of two consecutive rows in the feature matrix and store to the pre-add feature matrix address space (address space specified by custom CSR3).

We use the hardware logic named converter in hardware architecture to identify the fixed-rd vector instruction: load-rs-add-rd-vec8/16 vector_rd (idx1) (idx2) (CSR1,CSR2). When the converter recognizes this instruction and judges that idx2 = idx1+1 the converter will convert load-rs-add-rd-vec8/16 vector_rd (idx1) (idx2) (CSR1,CSR2) instruction to load-rs-add-rd-vec8/16 vector_rd (idx) (CSR3,CSR2) instruction.

4.3.10 Memory Access Optimization

In the fixed-rd mode, different vector lane pipelines load feature vectors from the feature matrix, and may load the same feature vector simultaneously. There may be data locality between load requests of different vector lane pipelines. The load accumulate buffer uses a certain memory access delay in exchange for the overall memory access bandwidth.

The buffer contains four queues, each of which corresponds to the read port of the corresponding bank of the SCM block. The eight load requests from the eight vector lane pipelines enter different queues for temporary storage according to the least significant 2-bit addresses. Each queue judges whether the memory access addresses of up to n load requests at the head of the queue are equal, and merges load requests.

5 PERFORMANCE EVALUATION AND EXPERIMENTAL RESULTS

In this section, we first introduce the experimental setup and present the evaluation results of the three methods. We compare the optimization effects of these three methods relative to traditional architectures and compare their advantages and disadvantages on different models and datasets.

5.1 Experimental Setup

For HyGCN[27], to compare the performance and energy consumption of HyGCN with state-of-the-art works, we evaluate PyTorch Geometric (PyG)[7] on a Linux workstation equipped with two Intel Xeon E5-2680 v3 CPUs and 378 GB DDR4 memory and on an NVIDIA V100 GPU, denoted as PyG-CPU and PyG-GPU, respectively. HyGCN run in a GHz compute unit with 32 SIMD16 cores and 8 systolic modules (each with 4 $\times$128 arrays).

For downstream task based design, we implemented a prototype on the Xilinx Alveo U200 platform using the Verilog-HDL and Vivado 2019.2 was used for synthesis. The platform had 64 GB off-chip DRAM (77 GB/s peak bandwidth) and 35 MB on-chip SRAM and ran at 250 MHz.

GPGCN hardware accelerator is designed and implemented using chisel language under the chipyard[2] soc integration framework, and all performance data are obtained by Verilog simulation accurate to the clock cycle, in which the behavior of DDR is simulated using the dramsim2[15] model and Micron's DDR3 timing model.

Dataset	#Vertex	Feature Length	#Edge
IMDB-BIN (IB)	2,647	136	28,624
Cora (CR)	2,708	1,433	10,556
Citeseer (CS)	3,327	3,703	9,104
COLLAB (CL)	12,087	492	1,446,010
Pubmed (PB)	19,717	500	88,648
Reddit (RD)	232,965	602	114,615,892
Flickr (FI)	89,250	500	899,756
Yelp (YL)	716,847	300	6,977,410

Table 4: Dataset Characteristics

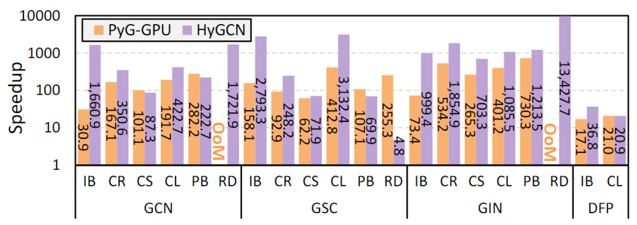


Figure 25: Speedup of HyGCN.

Dataset	Baseline GCN model	Sparsified GCN model	Edge removed
PubMed	0.792 ± 0.004	0.795 ± 0.001	33.7%
Flickr	0.503 ± 0.001	0.504 ± 0.003	40.6%
Reddit	0.945 ± 0.002	0.945 ± 0.003	65.8%
Yelp	0.378 ± 0.001	0.377 ± 0.002	67.7%

Table 5: Node classification performance.

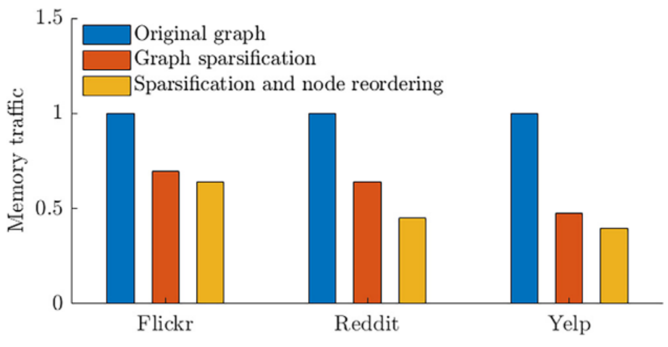


Figure 27: Memory traffic reduction on three large datasets.

Benchmark Graph Datasets and GCN Models Table 4 presents the information of the dataset employed. All GCNs use a two-layer structure, the feature vector length of the hidden layer is set to 128, and the forward calculation of all GCNs uses the calculation order of combination first and then aggregation. The software adaptation method of the GCNs network under the GPGCN accelerator is that each SIMD vector lane calculates a corresponding vertex aggregation.

5.2 Results of HyGCN

- **Speedup.** Fig. 25 depicts that HyGCN achieves average 1509× and 6.5× speedup compared with PyG-CPU and PyG-GPU, respectively. The performance improvement comes from the individual optimizations in Aggregation Engine & Combination Engine, and the inter-engine pipeline & coordination. First, the parallel processing in SIMD cores and systolic arrays speed up the computations. Second, the graph partition and sparsity elimination increase the feature reuse and decrease redundant accesses in Aggregation Engine, which saves DRAM bandwidth. Third, the weight parameters are reused efficiently in Combination Engine, which also helps better utilize the bandwidth. Finally, the inter-engine pipeline further optimizes the parallelism and the off-chip memory access coordination improves the DRAM access efficiency.
- **Energy Consumption.** As Fig. 26 shows, HyGCN consumes only 0.04% and 10% energy on average compared to PyG-CPU and PyG-GPU, respectively. The energy consumption of all platforms includes the off-chip memory. Note that, although the results of PyG-CPU and PyG-GPU do not include the overhead of the Sampling operation, they are still costly. For example, the Sampling energy of GSC is 2715J on the RD dataset. In contrast, our work consumes only 1.79J compared to the total 2716J in PyG-GPU.

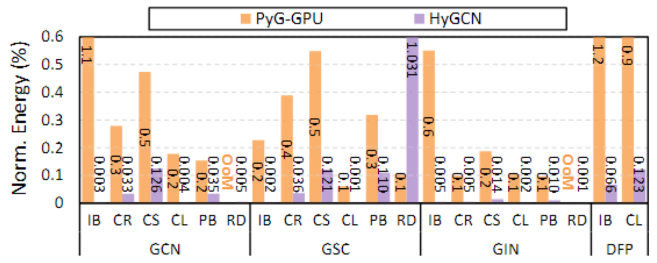


Figure 26: Normalized energy over PyG-CPU.

- **DRAM Access.** Although the 16MB on-chip memory is much smaller than the 60MB L3 cache on CPU and 34MB on GPU, HyGCN accesses only 21% and 33% of off-chip data compared with PyG-CPU and PyG-GPU on average, respectively. This benefits from our data reuse optimizations, sparsity elimination, and the immediate processing between two engines. On the CL dataset for GCN, GSC[8], and GIN[24], multiple graphs are assembled to form a larger one before being processed, which results in intensive sparsity. HyGCN can efficiently eliminate the sparsity via window sliding and shrinking, thus avoiding unnecessary data accesses.

5.3 Results of Downstream Task Based Design

- **Evaluation for Graph Sparsification.** We implemented the proposed graph sparsification algorithm in PyG framework for efficient GPU computation. Cross-entropy was employed to formulate the loss function, and Adam optimizer was applied for the training. The initial learning rate of the optimizer was set to be 0.01. Based on the original graph, pretraining models were obtained on all datasets, as the baseline GCN models. For PubMed, Flickr, Reddit and Yelp, hyperparameter k was set to be 7, 7, 40 and 10, respectively. From Table 5, it can be seen that compared with the baseline GCN model, the proposed AM-based graph sparsification approach consistently achieved comparable or superior performance. The proposed sparsification approach could effectively filter the most important edges by leveraging the guidance from downstream tasks.

	PubMed	Flickr	Reddit	Yelp	DSP	Frequency
PyG-CPU	229 ms	3.3 s	81 s	56 s	–	–
PyG-GPU	3.4 ms	337 ms	OoM*	OoM*	–	–
ASAP2020	9.5 ms	48.7 ms	598.7 ms	306.3 ms	5312	250 MHz
HyGCN	0.64 ms	N/A*	289 ms	N/A*	N/A*	1 GHz
BoostGCN	1.14 ms	20.1 ms	98.1 ms	193 ms	3840	250 MHz
AWB-GCN	**0.23 ms**	N/A*	49.7	N/A*	4096	330 MHz
LW-GCN	8.56 ms	N/A*	N/A*	N/A*	512	250 MHz
DTB-GCN	2.18 ms	**13.1 ms**	**42.26 ms**	**81.56 ms**	2304	250 MHz

Table 6: Performance comparison of various models across datasets. *OoM indicates Out of Memory; N/A indicates data not available.

GCN Cycles	Boom[31] Cpu	Boom with Hawacha[11]	Boom with GPGCN	HYGCN
	209,380,505 cycles/comb1	57,996,022 cycles/comb1	293,060 cycles/comb1	
	395,401,881 cycles/agg1	101,583,962 cycles/agg1	229,414 cycles/agg1	
Cora	1,395,689 cycles/comb2	799,722 cycles/comb2	138,711 cycles/comb2	21,000 cycles/total
	197,663,198 cycles/agg2	53,797,383 cycles/agg2	141,820 cycles/agg2	
	803,841,273 cycles/total	214,177,089 cycles/total	803,005 cycles/total	
	928,041,011 cycles/comb1	203,898,801 cycles/comb1	621,287 cycles/comb1	
	832,942,404 cycles/agg1	196,764,007 cycles/agg1	204,178 cycles/agg1	
Citeseer	2,393,692 cycles/comb2	1,269,727 cycles/comb2	174,775 cycles/comb2	300,000 cycles/total
	416,406,467 cycles/agg2	106,222,113 cycles/agg2	124,801 cycles/agg2	
	2,179,783,574 cycles/total	508,154,648 cycles/total	1,125,041 cycles/total	

Table 7: The GCN execution latency of different hardware with different datasets.

GAT Cycles	Boom Cpu	Boom with Hawacha	Boom with GPGCN	HYGCN
	268,836,121 cycles/comb1	64,912,288 cycles/comb1	310,857 cycles/comb1	
	611,028,937 cycles/agg1	129,569,840 cycles/agg1	254,199 cycles/agg1	
Cora	2,028,210 cycles/comb2	1,142,922 cycles/comb2	141,559 cycles/comb2	Not Support
	287,572,614 cycles/agg2	69,073,928 cycles/agg2	165,580 cycles/agg2	
	1,169,465,882 cycles/total	264,698,978 cycles/total	872,195 cycles/total	
	920,576,720 cycles/comb1	188,827,953 cycles/comb1	620,022 cycles/comb1	
	997,887,623 cycles/agg1	199,138,394 cycles/agg1	219,149 cycles/agg1	
Citeseer	2,313,190 cycles/comb2	1,222,387 cycles/comb2	180,613 cycles/comb2	Not Support
	474,441,246 cycles/agg2	105,822,637 cycles/agg2	142,387 cycles/agg2	
	2,395,218,779 cycles/total	495,011,371 cycles/total	1,162,171 cycles/total	

Table 8: The GAT execution latency of different hardware with different datasets.

- **Memory Traffic of DRAM Access.** Fig. 27 proves the effectiveness of graph preprocessing in improving data reuse and reducing memory traffic. Thanks to preprocessing, external memory accesses for the three large datasets were reduced by 30.6%, 36.1%, and 52.64% respectively through graph sparsification. After integrating node reordering, external memory accesses for the three datasets were reduced by 35.9%, 54.9%, and 60.5% respectively.
- **Speed Comparison.** We compared our work with several of the most advanced GCN acceleration frameworks currently available, including HyGCN mentioned in this paper. The inference times and DSP utilization rates on four graph datasets are listed in Table 6. It can be seen that DTB-GCN exhibits excellent performance on almost all of the datasets. It is worth noting that our work shows a performance drop for the PubMed dataset because the node feature matrix in the graph is very sparse. Previous works designed an additional sparse computation module to handle sparse feature matrices. However, this superior performance comes at the cost of logic resources and control logic complexity on the chip. HyGCN is based on advanced ASIC technology and operates at a frequency of 1 GHz. Although there are clock frequency differences, our proposed framework achieves 6.8x acceleration on the large Reddit dataset. The peak performance of the HyGCN architecture is 4.6 TOPS, the peak performance of the BoostGCN architecture is 0.89 TOPS; at the same time, the peak performance of DTB-GCN is 1.136 TOPS.

  Despite the fact that DTB-GCN shows better performance than HyGCN, the design based on downstream tasks limits its scalability. Compared to HyGCN, the acceleration of the inference stage in DTB-GCN is partly based on more complex training and preprocessing, which is also a disadvantage.

5.4 Results of GPGCN

The execution latency of GCN and GATwith the Cora dataset and Citeseer dataset under different hardware is shown in Tables 7 and 8. All latencies are normalized to cycles to remove the effects of different frequencies.

The evaluation reveals significant performance improvements in both aggregation and combination stages, with notable differences across datasets. For the Cora dataset, the aggregation stage achieves an acceleration ratio exceeding 1300×, while for the Citeseer dataset, it surpasses 3300×, demonstrating the ability of accelerators to exploit sparsity during aggregation. In the combination stage, the first-layer network shows acceleration ratios of approximately 700× for the Cora dataset and 1500× for the Citeseer dataset. However, the second-layer combination stage exhibits relatively smaller acceleration ratios due to the nature of its computation, which involves dense matrix multiplications where sparsity cannot be utilized for significant acceleration.

Overall, the total acceleration ratios reach about 1001× for the Cora dataset and 1937× for the Citeseer dataset. When comparing GPGCN with traditional vector expansion methods, although GPGCN employs computing resources four times greater than Hawacha, its acceleration ratio far exceeds 4×, highlighting its efficient design.

Nonetheless, compared to dedicated accelerators such as HyGCN, the acceleration efficiency of GPGCN is relatively lower because HyGCN leverages significantly larger computational resources (approximately 72×) and superior on-chip cache and off-chip memory bandwidth (approximately 50×). However, HtGCN's speedup ratio for the Citeseer dataset is lower than that for the Cora dataset, indicating its inability to fully exploit the dataset's sparsity, as GPGCN does. Moreover, GPGCN's software programmability offers additional flexibility, enabling it to accelerate architectures like GAT networks, which HyGCN cannot support. These findings underline the trade-offs between efficiency, adaptability, and hardware resource utilization in GCN accelerator designs.

6 CONCLUSIONS

In this work, we presented an analysis of hybrid accelerator architectures designed for efficient Graph Convolutional Network (GCN) inference. Through our evaluation of three distinct designs—HyGCN, Downstream Task-Based Design (DTB-GCN), and GPGCN—we highlighted their unique architectural features, strengths, and limitations across different datasets and workloads. The hybrid design approach, which tailors the aggregation and combination phases to their respective computational characteristics, demonstrates significant performance improvements compared to traditional CPU- and GPU-based frameworks. By optimizing each phase independently and coordinating their execution, hybrid architectures address the irregularity of the aggregation phase and the computational intensity of the combination phase.

HyGCN achieved exceptional acceleration ratios and outstanding energy efficiency through parallelism, sparsity elimination, and data reuse optimizations. DTB-GCN, with its focus on graph sparsification and downstream task optimization, achieved competitive performance while reducing external memory traffic. However, its reliance on preprocessing and training stages limits scalability. GPGCN, leveraging programmable RISC-V-based instructions, achieved remarkable acceleration ratios in both aggregation and combination stages. Its programmability makes it uniquely capable of supporting diverse network architectures, including GAT, which other accelerators like HyGCN cannot support.

We contend that programmable accelerators represent the future development trajectory, and GPGCN offers a fundamental paradigm for this direction. Despite having achieved a general GCN accelerator that fully exploits graph sparsity, numerous characteristics of GCN have yet to be effectively harnessed, such as two-stage collaboration, more efficient aggregation approaches, and locality utilization, among others. Striking a balance between acceleration and generalization in these aspects awaits future exploration.

REFERENCES

[1] S. Abadal, A. Jain, R. Guirado, J. López-Alonso, and E. Alarcon, "Computing graph neural networks: A survey from algorithms to accelerators," *ACM Computing Surveys*, Sep 2020.

[2] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton *et al.*, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.

[3] J. Binkowski, A. Sawczyn, D. Janiak, P. Bielak, and T. Kajdanowicz, "Graph-level representations using ensemble-based readout functions," in *International Conference on Computational Science*. Springer, 2023, pp. 393–405.

[4] M. Boguñá, "The structure and dynamics of networks," *Journal of Statistical Physics*, p. 419–421, Jan 2007. [Online]. Available: http://dx.doi.org/10.1007/s10955-006-9267-8

[5] S. Brody, U. Alon, and E. Yahav, "How attentive are graph attention networks?" *arXiv preprint arXiv:2105.14491*, 2021.

[6] I.-H. Chung, C. Kim, H.-F. Wen, and G. Cong, "Application data prefetching on the ibm blue gene/q supercomputer," *IEEE International Conference on High Performance Computing, Data, and Analytics*, Nov 2012.

[7] M. Fey and J. Lenssen, "Fast graph representation learning with pytorch geometric," *Cornell University - arXiv*, Mar 2019.

[8] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Neural Information Processing Systems*, Jun 2017.

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

[10] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval, "How much parallelism is there in irregular applications?" *ACM SIGPLAN Notices*, vol. 44, no. 4, p. 3–14, Feb 2009. [Online]. Available: https://doi.org/10.1145/1594835.1504181

[11] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanovic, and K. Asanovic, "A 45nm 1.3ghz 16.7 double-precision gflops/w risc-v processor with vector accelerators," in *ESSCIRC 2014 - 40th European Solid State Circuits Conference (ESSCIRC)*, Sep 2014. [Online]. Available: https://doi.org/10.1109/esscirc.2014.6942056

[12] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, p. 39–55, Mar 2008. [Online]. Available: http://dx.doi.org/10.1109/mm.2008.31

[13] K. O'Shea, "An introduction to convolutional neural networks," *arXiv preprint arXiv:1511.08458*, 2015.

[14] S. Ran, B. Zhao, X. Dai, C. Cheng, and Y. Zhang, "Software-hardware co-design for accelerating large-scale graph convolutional network inference on fpga," *Neurocomputing*, vol. 532, pp. 129–140, 2023.

[15] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE Computer Architecture Letters*, p. 16–19, Jan 2011. [Online]. Available: http://dx.doi.org/10.1109/l-ca.2011.4

[16] F. Sadi, J. Sweeney, T. M. Low, J. C. Hoe, L. Pileggi, and F. Franchetti, "Efficient spmv operation for large and highly sparse matrices using scalable multi-way merge parallelization," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 347–358. [Online]. Available: https://doi.org/10.1145/3352460.3358330

[17] 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.

[18] W. Tang and P. Zhang, "Gpgcn: A general-purpose graph convolution neural network accelerator based on risc-v isa extension," *Electronics*, vol. 11, no. 22, p. 3833, 2022.

[19] C. Tian, L. Ma, Y. Zhi, and Y. Dai, "Pcgcn: Partition-centric processing for accelerating graph convolutional network," *IEEE Conference Proceedings*, Jan 2020.

[20] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[21] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The risc-v instruction set manual, volume i: User-level isa, version 2.0," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54*, p. 4, 2014.

[22] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 1, pp. 4–24, 2020.

[23] Z. M. Xiong, "A survey of fpga based on graph convolutional neural network accelerator," in *2020 International Conference on Computer Engineering and Intelligent Control (ICCEIC)*. IEEE, 2020, pp. 92–96.

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

[25] K. Xu, C. Li, Y. Tian, T. Sonobe, K.-i. Kawarabayashi, and S. Jegelka, "Representation learning on graphs with jumping knowledge networks," *International Conference on Machine Learning*, Jul 2018.

[26] 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*, p. 22–25, Jan 2020. [Online]. Available: http://dx.doi.org/10.1109/lca.2020.2970395

[27] 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)*. IEEE, 2020, pp. 15–29.

[28] R. Ying, J. You, C. Morris, X. Ren, W. Hamilton, and J. Leskovec, "Hierarchical graph representation learning with differentiable pooling," *Cornell University - arXiv*, Jun 2018.

[29] B. Zhang, H. Zeng, and V. Prasanna, "Hardware acceleration of large scale gcn inference," in *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Jul 2020. [Online]. Available: http://dx.doi.org/10.1109/asap49362.2020.00019

[30] Z. Zhang, J. Leng, L. Ma, Y. Miao, C. Li, and M. Guo, "Architectural implications of graph neural networks," *IEEE Computer Architecture Letters*, p. 1–1, Jan 2020. [Online]. Available: http://dx.doi.org/10.1109/lca.2020.2988991

[31] Z. Zhang, P. Cui, and W. Zhu, "Deep learning on graphs: A survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 1, pp. 249–270, 2020.

[32] S. Zhou, R. Kannan, V. K. Prasanna, G. Seetharaman, and Q. Wu, "Hitgraph: High-throughput graph processing framework on fpga," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 10, pp. 2249–2264, 2019.

[33] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou, "Aligraph: a comprehensive graph neural network platform," *Proc. VLDB Endow.*, vol. 12, no. 12, p. 2094–2105, Aug. 2019. [Online]. Available: https://doi.org/10.14778/3352063.3352127