electronics

MDPI

Article

GPGCN: A General-Purpose Graph Convolution Neural Network Accelerator Based on RISC-V ISA Extension

Wenkai Tang and Peiyong Zhang *

School of Micro-Nano Electronics, Zhejiang University, Hangzhou 310058, China
* Correspondence: zhangpy@zju.edu.cn

Abstract: In the past two years, various graph convolution neural networks (GCNs) accelerators have emerged, each with their own characteristics, but their common disadvantage is that the hardware architecture is not programmable and it is optimized for a specific network and dataset. They may not support acceleration for different GCNs and may not achieve optimal hardware resource utilization for datasets of different sizes. Therefore, given the above shortcomings, and according to the development trend of traditional neural network accelerators, this paper proposes and implements GPGCN: a general-purpose GCNs accelerator architecture based on RISC-V instruction set extension, providing the software programming freedom to support acceleration for various GCNs, and achieving the best acceleration efficiency for different GCNs with different datasets. Compared with traditional CPU, and traditional CPU with vector expansion, GPGCN achieves above 1001×, 267× speedup for GCN with the Cora dataset. Compared with dedicated accelerators, GPGCN has software programmability and supports the acceleration of more GCNs.

Keywords: GCNs; general GCNs accelerator; RISC-V; software programmable

Citation: Tang, W.; Zhang, P. GPGCN: A General-Purpose Graph Convolution Neural Network Accelerator Based on RISC-V ISA Extension. *Electronics* **2022**, *11*, 3833. https://doi.org/10.3390/electronics11223833

Academic Editor: David Defour

Received: 19 October 2022
Accepted: 16 November 2022
Published: 21 November 2022

1. Introduction

Since the GCNs hardware accelerator HYGCN [1] was proposed in 2020, various GCNs accelerators [1–11] have emerged, one after another, that are different in the calculation method, control flow, and scheduling algorithm, with different advantages in accelerating the GCNs [12,13], such as GCN [14], GIN [15], and GSC [16]. HYGCN [1] proposes a GCNs accelerator composed of an aggregation phase and a combination phase. ENGN [2] optimizes computation order for aggregation and combination to improve acceleration efficiency. AWB-GCN [3] optimizes the execution unit scheduling algorithm to balance the workload of each execution unit to improve the overall efficiency. However, they also have hidden downsides. A common disadvantage is that the hardware architecture is not programmable and it is optimized for a specific network and dataset. Their fixed calculation process may have a good acceleration effect for specific sizes and formats of datasets and certain GCNs, but not for other GCNs with different datasets because they do not have freedom of programmability.

Moreover, from the experience of the development history of traditional deep learning neural network accelerators, such as Cambricon [17], Grayskull [18] from Tenstorrent, and RASA [19] from Intel, it is understood that a software-programmable GCNs accelerator architecture based on instruction set is the trend of unified GCNs accelerator architecture in the future.

Traditional GCN hardware accelerators, such as HYGCN [1], ENGN [2], and AWB-GCN [3], which are dedicated accelerators, as shown on the right side of Figure 1, have the advantages of high energy efficiency and high execution rate. However, they also have less programmability or even non-programmability, which leads to the limitation of a fixed acceleration execution mode of the network.

In the general-purpose processor architecture, there are particular SIMD instruction set extensions for control-intensive operations with vector calculations, such as intel's

AVX instruction set extension, ARM's neon instruction set extension, SVE instruction set extension, RISC-V's vector extension, etc. Compute-intensive matrix computing also has particular matrix instruction set extensions, such as Intel's AMX extension, ARM's SME extension, etc. However, because these instruction set extensions of general-purpose processors are designed to cover most application scenarios, they are too general, as shown on the left of Figure 1. Although they have a high degree of programmable freedom, they are not specially customized and optimized for the characteristics of GCNs, and the efficiency of accelerating GCNs will not be as high as dedicated accelerators. Moreover, the large amount of instructions introduces the problem of taking up a lot of memory access bandwidth when fetching the instructions. At the same time, they are limited to the architecture of general-purpose processors, where their scalability is limited.

Therefore, for the programmable GCNs accelerator design in this paper, it is necessary to find an intermediate balance between the general-purpose processor architecture on the left of Figure 1 and the dedicated accelerator architecture on the right to satisfy the high degree of freedom of programmability, high execution efficiency, high energy efficiency, and sufficient scalability requirements at the same time. In this work, we pioneeringly propose the concept of GPGCN and design the GPGCN custom instructions based on RISC-V ISA extension. Then, we propose a general-purpose GCNs hardware accelerator based on the proposed GPGCN custom instructions. We design the general-purpose GCNs hardware accelerator in RTL and evaluate it using cycle-accurate simulation. Compared with a traditional CPU, and a traditional CPU with vector extension, GPGCN achieves above 1001×, 267× speedup for GCN with the Cora dataset. Compared with dedicated accelerators, such as HYGCN [1], GPGCN has software programmability and supports the acceleration of more GCNs.

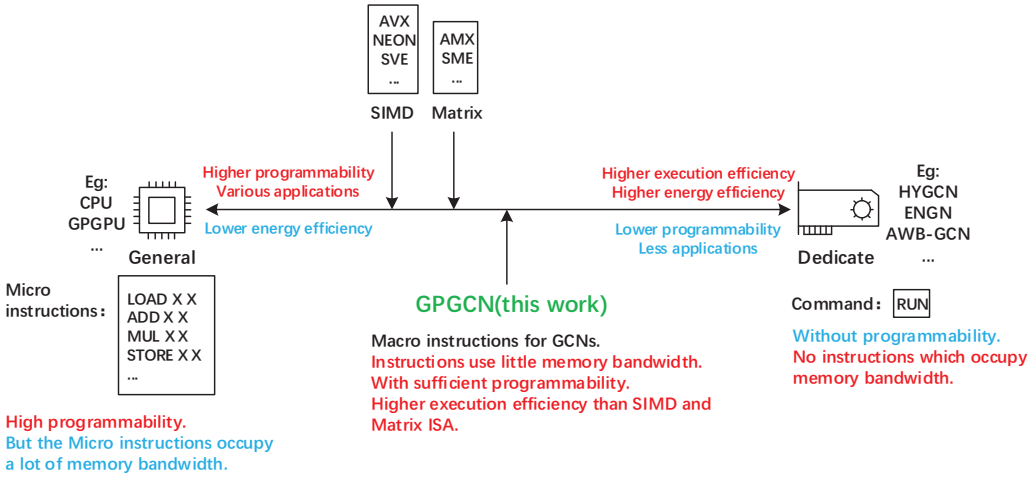

Figure 1. Differences between general-purpose processors, GPGCNs, and dedicated accelerators. Advantages are marked in red, while disadvantages are marked in blue.

2. GCNs Analysis

Different from the traditional convolutional network, which processes data in Euclidean space, GCNs process data in non-Euclidean space, such as connection graph data. The calculation of GCNs generally consists of two phases: aggregation and combination, as shown in Figure 2. Aggregation fuses the feature vectors of each vertex adjacent point in some way. For example, the average sum and weighted sum are evaluated to obtain a new feature vector, as shown in the middle part of Figure 2. After aggregation, the feature vector of the vertex has information about its neighbor vertices. Then, combination uses the aggregated feature vector to perform a fully connected convolution calculation to obtain a low-dimensional feature vector, as shown in the right part of Figure 2. Combination extracts low-dimensional information from the features of the vertex and its neighbor vertices.

Therefore, the unified mathematical expression of most GCNs is as (1), where A is the adjacency matrix, H is the feature matrix, and W is the weight matrix:

$$\text{GCNs networks} = A \cdot H \cdot W \tag{1}$$

The process of aggregation + combination is repeated two or three times, and the final low-dimensional vector is used to complete tasks such as classification.

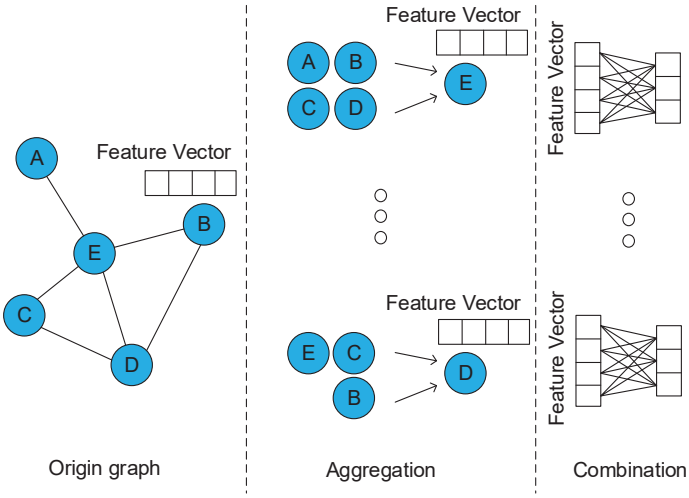

Figure 2. The calculation of most GCNs.

Most of the GCNs aggregation process can be expressed as the multiplication of the weighted adjacency matrix A and the feature matrix H, composed of each vertex's feature vector. However, their respective aggregation characteristics are reflected in the adjacency matrix A difference. The adjacency matrix of GCN is calculated by the degree matrix, while the adjacency matrix of GAT is learned through the training process. Therefore, the aggregation process of these GCNs can be expressed as in Equation (2):

$$\text{Aggregation result} = A \cdot H \tag{2}$$

Although aggregation can be uniformly expressed as matrix multiplication, due to the characteristics of graph datasets, the adjacency matrix A is always a matrix with high sparsity, that is, a large proportion of elements are 0. For example, as shown in Table 1, the adjacency matrix sparsity of the Cora dataset is 99.856%; the adjacency matrix sparsity of the Citeseer dataset is 99.918%, the adjacency matrix sparsity of the Pubmed dataset is 99.977%, and the adjacency matrix sparsity of the Nell dataset is 99.9942%. Therefore, the process of GCNs network aggregation is actually the process of sparse matrix multiplication sparse-GEMM.

Table 1. Sparsity differences of Cora, Citeseer, Pubmed, and Nell datasets.

	Node	Edge	Features	Classes	Feature Matrix Sparsity	Adjacent Matrix Sparsity
Cora	2708	10,556	1433	7	98.73%	99.856%
Citeseer	3327	9104	3703	6	99.15%	99.918%
Pubmed	19,717	88,648	500	3	90.00%	99.977%
Nell	65,755	251,550	61,278	186	99.99%	99.994%

It can be seen from Table 1 that the sparsities of the adjacency matrices of the four typical graph datasets are all above 99%. However, when the sparseness of the matrix is large enough, even using the space-aware sparse matrix multiplication (sp-GEMM) to calculate, it will cause the elements with a value of 0 to occupy unnecessary hardware resources and waste time. Therefore, compared to the matrix operation, the aggregation calculation process is not computationally intensive, but is control-intensive. For control-intensive processing, it is more suitable to use the control logic + vector PE (process element) to complete, as shown in the upper part of Figure 3.

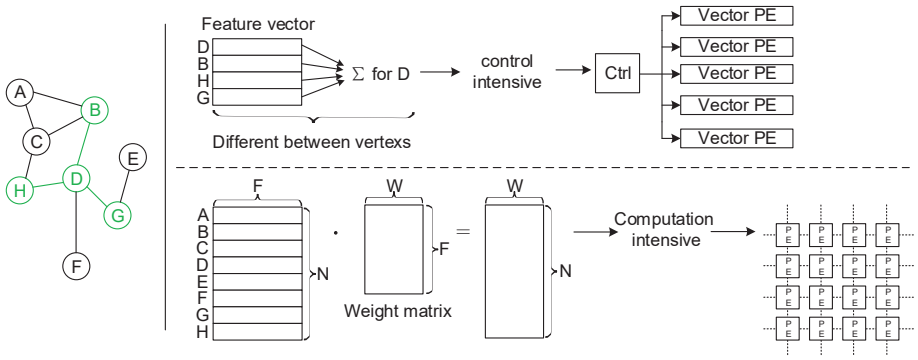

Figure 3. Control-intensive and computation-intensive in GCNs. Vertex D and its adjacent vertexes are marked in green.

For such a sparse matrix operation, the correct approach is shown in Figure 4. The adjacency matrix A with extremely high sparsity is stored in CSR format. The feature matrix H is divided into vector format by row for operation. For the non-zero elements of each row in the adjacency matrix, the column coordinates of the non-zero elements stored in the CSR format are used as indexes, and the feature vector of the corresponding row of the feature vector matrix H is taken out for the aggregation operation.

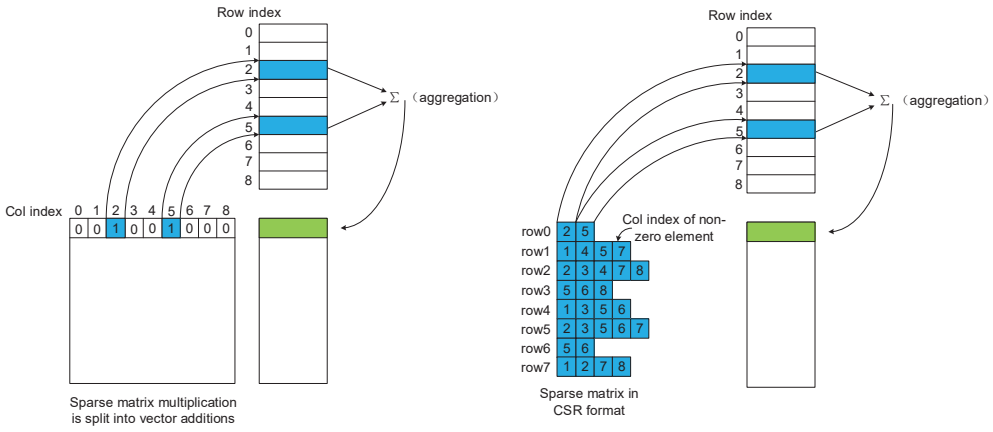

Figure 4. The computation process of a sparse matrix.

The calculation of the combination phase of different networks is similar. Whether a fully connected convolution or a multilayer perceptron, it is a dense matrix multiplication operation (dense-GEMM) or a matrix with a certain degree of sparsity multiplied by a dense matrix. It is computation-intensive and is suitable for computing with a matrix-form computing array, as shown in the lower part of Figure 3.

3. ISA Architecture

3.1. Basic Features of GPGCN Custom Instruction Set Architecture

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. Binding a pair of rs and rd registers together has three advantages:

- We only need to specify the index of one register to operate two registers, which saves a lot of coding space for GPGCN custom instructions to encode other information.
- It is consistent with the computational characteristics of the aggregation process.
- It has a better scalability.

3.2. Custom CSR

Since there are two difficulties in designing macro instructions, one is to encode many pieces of instruction operation information required in the limited RISC instruction encoding space, and the other is to ensure the degree of programming freedom. The first problem is to provide some common auxiliary information between instructions through the custom CSR (current status registers) registers to solve and reduce instruction coding pressure.

Table 2 shows the address space, the register's name, and the specific function description of the custom CSR registers of the GPGCN custom instruction set.

Table 2. The custom CSR registers.

Index	Address	Name	Description
1	0x7c2	Feature matrix base address for aggregation	The starting address of the feature matrix during aggregation calculation.
2	0x7c3	Feature vector length	The number of elements of the feature vector representing a vertex in the feature matrix.
3	0x7c4	Pre-add feature matrix base address	The starting address of another special feature matrix, which will be used when describing redundancy reduction techniques in the next chapter.
4	0x7c5	Result feature matrix base address	The starting address of storing the result matrix during aggregation calculation.
5	0x7c6	Feature matrix base address for combination	The starting address of the feature matrix during combined calculation.
6	0x7c7	Weight matrix base address	The starting address of the weight matrix during combined calculation.
7	0x7c8	Da	A specific dimension of the adjacent matrix and the feature matrix when the result matrix is evaluated in the combined calculation. It will be used when describing the matrix type instruction below.
8	0x7c9	Combination result matrix base address	The starting address of the result matrix of the combinatorial calculation.
9	0x7ca	Number of vector rd	The number of vector register pairs, a fixed value, representing GPGCN hardware size information.
10	0x7cb	Number of matrix rd	The number of matrix registers, a fixed value, representing GPGCN hardware size information.
11	0x7cc	SCM (scratchpad memory) configuration	The current configuration information of scratchpad memory, which will be described in detail when describing the configurable SCM hardware design in the next chapter.

Table 2. *Cont.*

Index	Address	Name	Description
12	0x7cd	Vector 8/16	The number of elements in the vector operated by the vector type instruction: 0 represents 8 elements, 1 represents 16 elements.
13	0x7ce	Float round mode	The rounding mode for floating-point calculations.
14	0x7cf	Matrix/vector mode	Indicates whether the GPGCN hardware is in the matrix instruction mode or the vector instruction mode. The CSR register is used to distinguish which mode the hardware is in in the case of a hardware microarchitecture that integrates the vector and matrix registers and execution unit resources.

3.3. 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 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 Figure 5.

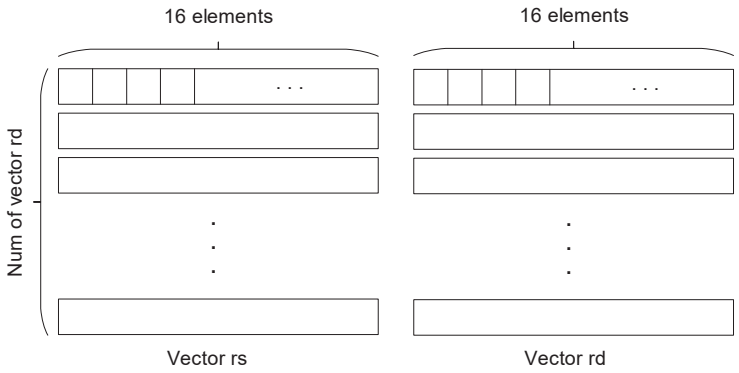

Figure 5. The custom vector registers file.

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 Figure 6. The vector rs register contains 8 32-bit single-precision float point elements. The matrix rs/rd register contains 8×8 32-bit single-precision float point elements to support $8 \times 1 \times 1 \times 8$ outer product operations.

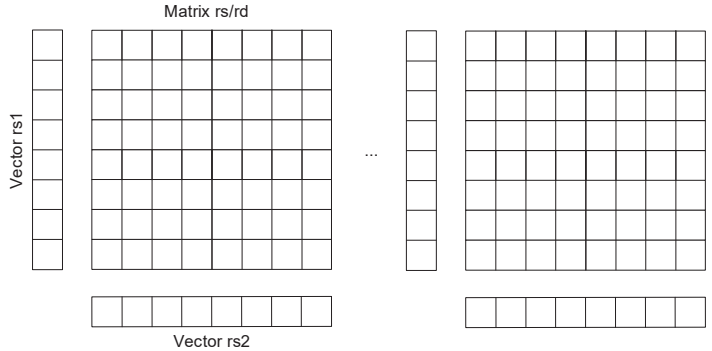

Figure 6. The custom matrix registers file.

Eight vector rs/rd registers can be combined into a matrix rs/rd register for use so that vector and matrix operations multiplex register resources and improve hardware utilization efficiency. The GPGCN hardware microarchitecture in the next chapter is also designed in this way.

3.4. Instruction Extension

According to the common characteristics of aggregation calculation and combination calculation in different GCNs, we designed four types of instructions extensions: vector type instructions, matrix type instructions, memory-access-related instructions, and special instructions, which correspond to aggregation, combination, memory access, and synchronization in the forward inferring process of different GCNs.

RISC-V provides four customizable instruction encoding spaces: custom0, custom1, custom2, and custom3, as shown in Figure 7. The GPGCN custom instruction set is implemented in these encoding spaces.

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 7. The custom instruction encoding space of RISC-V.

3.5. Vector Instruction Extension

It can be seen from Table 3 that the vector type instructions are subdivided into three categories: the basic category, the fix-rd category, and the fixed-rs category. The following describes the basic functions and design principles of vector-type instructions based on these three categories.

Table 3. The vector instructions extension.

Macro op	Instructions	Code Class
basic		
00001	loadvec8/16 vector_rd (idx) (CSR1,CSR2)	custom0_rs1
00001	loadvec8/16 vector_rs (idx) (CSR1,CSR2)	custom0_rs1
00001	loadvec8/16 all_vector_rs (idx) (CSR1,CSR2)	custom0_rs1
00010	storevec8/16 vector_rd (idx) (CSR3,CSR2) [relu]	custom0_rd_rs1, rd = 0
00010	storevec8/16 vector_rd (idx) (CSR4,CSR2) [relu]	custom0_rd_rs1, rd = 0
00011	addvec8/16 vector_rd vector_rs	custom0
00100	mov vector_rd 0	custom3
fixed-rd		
01000	load-rs-add-rd-vec8/16 vector_rd (idx) (CSR1,CSR2)	custom1_rs1
01000	load-rs-add-rd-vec8/16 vector_rd (idx) (CSR3,CSR2)	custom1_rs1
01001	load-rs-add-rd-vec8/16 vector_rd (idx1) (idx2) (CSR1,CSR2)	custom0_rs1_rs2
01010	load-rs-add-rd-vec8/16 vector_rd (idx) (aij) (CSR1,CSR2)	custom1_rs1_rs2
fixed-rs		
01100	load-rd-add-rs-store-rd-vec8/16 (idx) (CSR4,CSR2)	custom1_rd_rs1, rd = 0
01101	load-rd-add-rs-store-rd-vec8/16 (idx) (aij) (CSR4,CSR2)	custom2_rs1_rs2

3.5.1. 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 as the function complement of the fixed-rd and fixed-rs vector instructions.

Unlike the traditional SIMD instruction, the basic load/store instruction specifies the memory access address through the index stored in the RISC-V integer register. The index means the row coordinates of the feature vector to be accessed in the entire feature matrix.

The hardware will calculate the final memory access address through custom CSR1 (base address of feature matrix) and custom CSR2 (feature vector length) using Formula (3).

$$\text{load/store address} = \text{base\_address(CSR1)} + \text{idx} * \text{vector\_length(CSR2)} \quad (3)$$

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 Figure 8.

Figure 8. The basic vector load/store instructions equivalent.

3.5.2. Fixed-Rd Vector Instructions

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

The fixed-rd class vector instruction represents the fixed vector rd calculation mode in the aggregation calculation. As shown in Figure 9, the fixed vector rd calculation mode represents the aggregation calculation process of the feature vectors of all the neighbors of a vertex. 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.

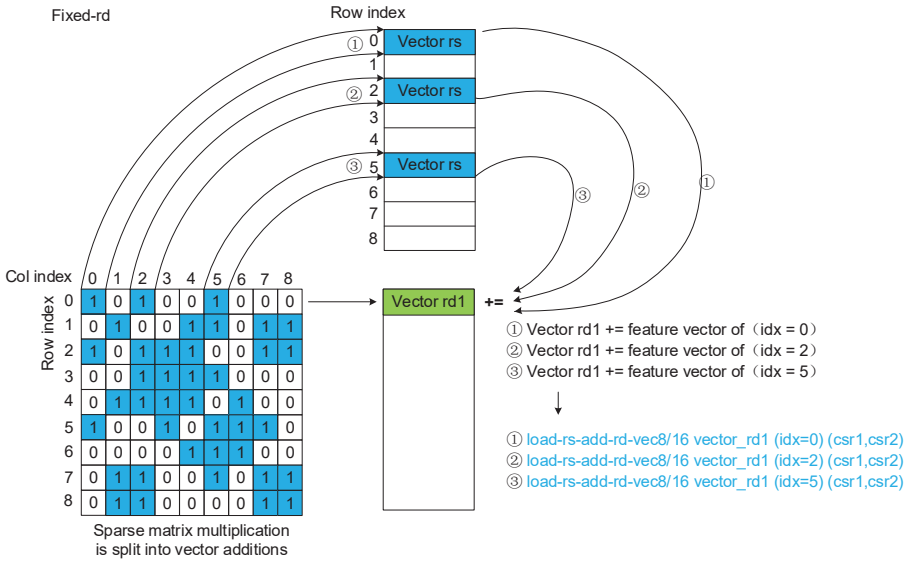

Figure 9. The computation process of fixed-rd mode.

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, so there is no need to specify the index of the vector rs register. The vector rd is bound to the corresponding vector rs, which is also the theoretical basis for the paired definition of vector rs/rd described in the GPGCN custom instruction set architecture features above.

Then, the instruction encoding only needs to specify the index of vector rd, which saves a lot of other encoding space for the RISC-V instruction encoding. Therefore, the remaining encoding space can be used 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.

Therefore, a fixed-rd vector instruction is equivalent to the combined operation of multiple traditional scalar and traditional vector instructions. As shown in Figure 10, 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.

Figure 10. The load-rs-add-rd-vec8/16 instruction equivalent.

Even though the load-rs-add-rd-vec8/16 instruction already consists of multiple operations, due to the excellent mechanism of binding vector rd and vector rs, 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 that a load-rs-add-rd-vec8/16 instruction can calculate the aggregation process of two feature vectors, and further increase instruction density, as shown in Figure 11.

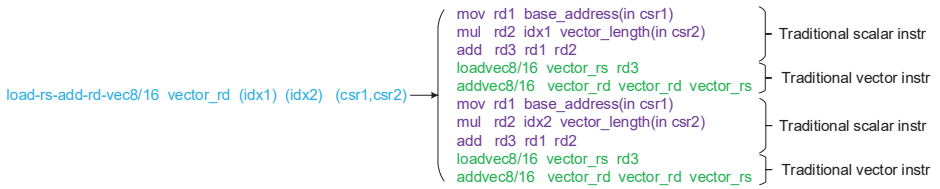

Figure 11. Another load-rs-add-rd-vec8/16 instruction equivalent.

In the same way as the above principle, for the aggregation calculation process with weight, such as the GAT network, the abovementioned extra free coding space can be used as the index of the floating-point register, and the value of the weight is stored in this floating-point register. There is one more floating-point multiplication operation of vector multiplication by a scalar (weight) in the calculation process, as shown in Figure 12.

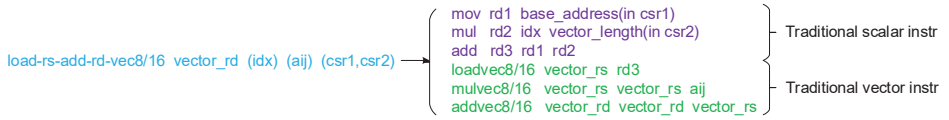

Figure 12. The load-rs-add-rd-vec8/16 instruction for GAT equivalent.

The fixed-rd vector instruction also has the advantage in that the method of indexing the feature vector corresponds to the sparse storage format of CSR, as shown in Figure 13.

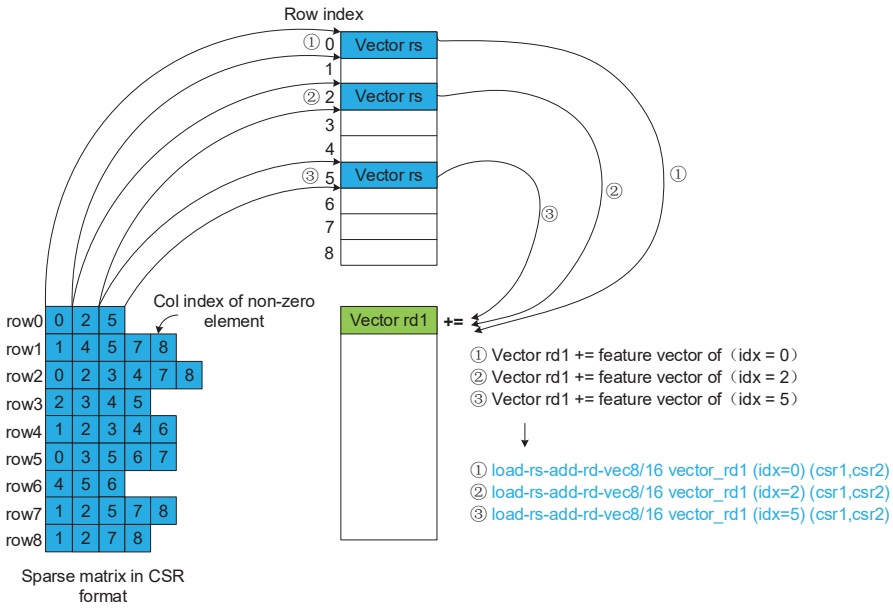

Figure 13. The computation process of fixed-rd mode with CSR data format.

3.5.3. 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. Therefore, the feature vector of this vertex is reusable in the aggregation calculation of different neighbors. The process of aggregation calculation can fix the feature vector of this vertex to the vector rs register and then load the intermediate results of the aggregation calculation of different neighbors to vector rd, and perform the aggregation calculation of these neighbors in turn, until all neighbors use the feature vector of the vertex to calculate one round, as shown in Figure 14. Therefore, the fixed-rs vector instruction can reuse the feature vector fixed in the vector rs register instead of reading from memory or cache every time.

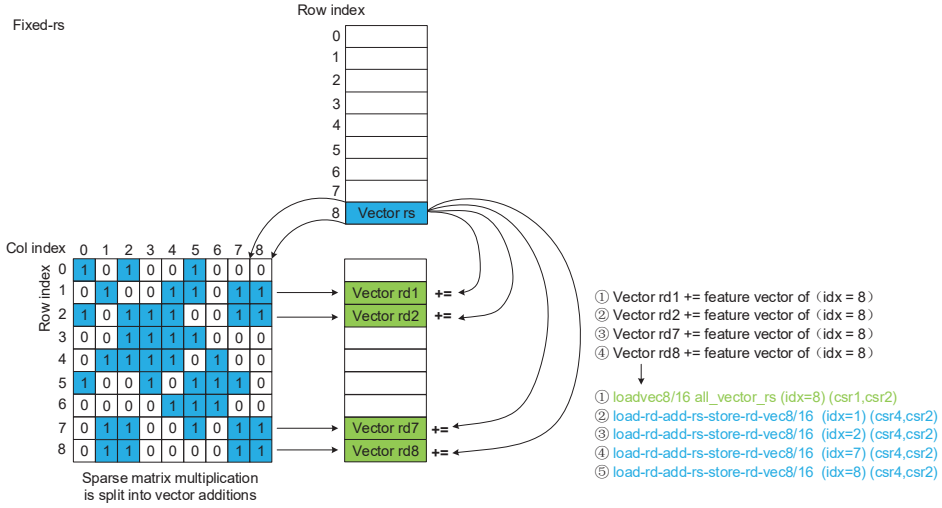

Figure 14. The computation process of fixed-rs mode.

The fixed-rs vector instruction has one characteristic that differs from the fixed-rd instruction, which is that the vector rd instruction is no longer reused after it is calculated, and it needs to be stored back to the original address where it was loaded. This store does not need the extra information that occupies the coding space, so the fixed-rs instruction directly integrates the store operation into the instruction, which further increases the density of the instruction, as shown in Figures 15 and 16.

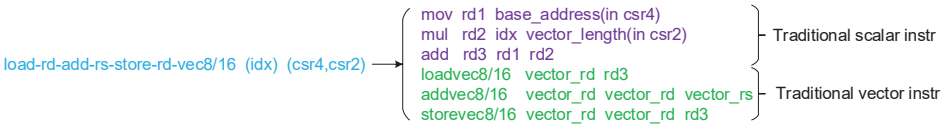

Figure 15. The load-rd-add-rs-store-rd-vec8/16 instruction equivalent.

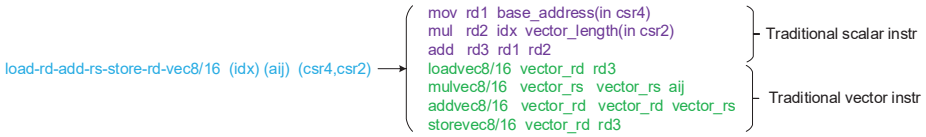

Figure 16. The load-rd-add-rs-store-rd-vec8/16 instruction for GAT equivalent.

Corresponding to the last advantage of the fixed-rd vector instructions, the fixed-rs instructions also have the advantage in that the method of indexing the intermediate results of different vertices and loading them into vector rd matches the CSC sparse storage format, as shown in Figure 17.

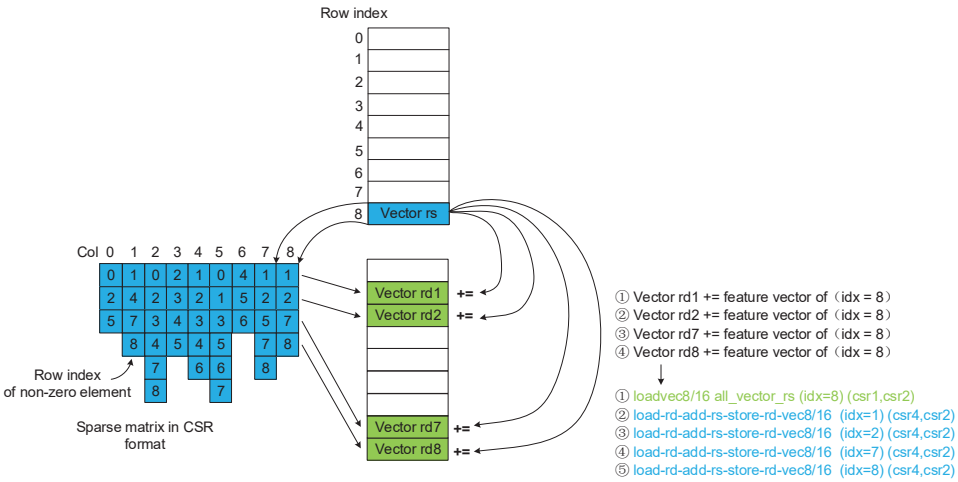

Figure 17. The computation process of fixed-rd mode with CSC data format.

3.6. Matrix Instruction Extension

The execution of the matrix-type instructions is closely related to the storage format of the matrix. Here, the storage format of the matrix is introduced first. As shown in Figure 18, the input and output matrices are divided into blocks in the combination operation of the GCNs. The block is indexed according to the index number. For the feature matrix of the size of a $\times$ b and the weight matrix of the size of b $\times$ c, which are also input matrices, the size of a block is 8 $\times$ Da elements, where Da is specified by custom CSR7; for the result matrix (output matrix) of the size of a $\times$ c, the size of a block is 8 $\times$ 8 elements.

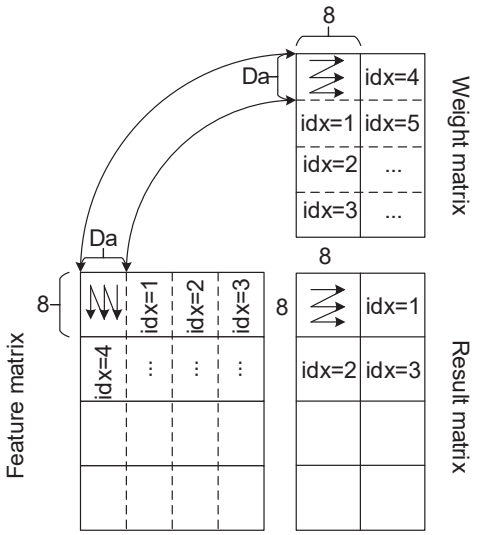

Figure 18. The computation process of matrix mode.

The traditional matrix instructions, such as ARM's SME instruction set extension, are to take the outer product of two vectors (assuming the lengths are n and m, respectively) to produce an n × m result matrix, that is, the instruction completes n × 1 × 1 × m = n × m vector outer product operation. In order to further increase the instruction density of the matrix instructions, the first matrix instruction in Table 4 specifies the value of Da through the custom CSR7 register and completes the matrix multiplication operation of the feature matrix of 8 × Da size with the weight matrix of Da × 8 size, that is, the matrix multiplication operation of 8 × Da × Da × 8 = 8 × 8, as shown in Figure 19.

Table 4. The matrix instructions extension.

Macro Op	Instructions	Code Class
Basic		
10000	load-outerproduct-add-8*8 matrix_rd (idx1) (idx2) (CSR5,CSR6,CSR7)	custom3_rs1_rs2
10001	storematrix8*8 matrix_rd8*8 (idx) (CSR8) [relu]	custom2_rd_rs1, rd=0
10001	storematrix8*8 matrix_rs8*8 (idx) (CSR8) [relu]	custom2_rd_rs1, rd=0
10010	loadmatrix8*8 matrix_rd (idx) (CSR8)	custom2_rs1
10010	loadmatrix8*8 matrix_rs (idx) (CSR8)	custom2_rs1
10011	addMatrix matrix_rd matrix_rs	custom1

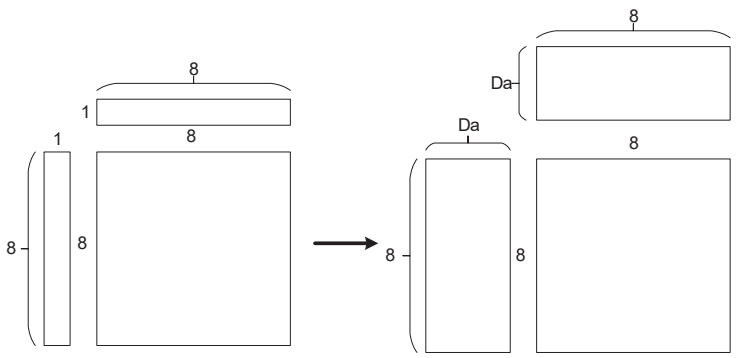

Figure 19. The difference between traditional matrix instructions and GPGCN matrix instructions.

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 (4) and (5):

$$\text{feature matrix block addr} = \text{feature\_matrix\_base\_addr}(\text{CSR5}) + \text{idx1} * 8 * \text{Da}(\text{CSR7}) \quad (4)$$

$$\text{weight matrix block addr} = \text{weight\_matrix\_base\_addr}(\text{CSR6}) + \text{idx2} * 8 * \text{Da}(\text{CSR7}) \quad (5)$$

The load-outerproduct-add-8*8 matrix_rd (idx1) (idx2) instruction is split into Da times as follows: 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 and sum with matrix rd, and then store them in matrix rd, as shown in Figure 20.

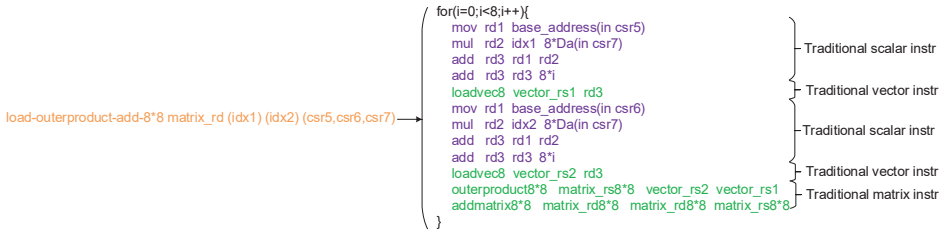

Figure 20. The matrix instruction equivalent.

3.7. Memory Access Extension

The scratchpad memory-related operation instructions in Table 5 include three 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.

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.

Table 5. The memory access instructions extension.

Macro Op	Instructions	Code Class
10101	preload block0-3 (idx) (CSR1,CSR2) (CSR3,CSR2) (CSR5,CSR7) (CSR6,CSR7)	custom3_rs1
10110	sync-preload block0-3	custom2
10111	storescmback (idx)(CSR4,CSR2)[relu]	custom3_rd_rs1, rd = 0

3.8. Fence Extension

Although the CPU fetches and sends the instructions of the GPGCN accelerator, the CPU does not know whether the GPGCN accelerator instructions have been executed in the accelerator, nor can it detect the address correlation between the CPU load/store instruction and the GPGCN memory access instruction. Therefore, in order to synchronize with the CPU instruction stream, the GPGCN-fence instruction in Table 6 is specially defined here to complete the synchronization operation between the GPGCN instruction stream and the CPU instruction stream:

Table 6. The fence instruction extension.

Macro Op	Instructions	Code Class
10100	gpgcn-fence	custom2 rd = 0

The gpgcn-fence instruction is usually followed by an RISC-V fence instruction. When all the older GPGCN instructions before gpgcn-fence are executed, the gpgcn-fence instruction is executed and can be committed in the CPU reorder buffer, otherwise the gpgcn-fence instruction will block at the commit head of the CPU reorder buffer, preventing the commit of subsequent CPU instructions. After the gpgcn-fence instruction is committed, the next RISC-V fence instruction can be committed, and the subsequent load/store memory access instructions can be executed. The memory access synchronization operation between the GPGCN instruction and the RISC-V CPU instructions is completed.

3.9. Encoding of GPGCN Instruction Set

Figure 21 shows the encoding of all GPGCN instructions.

macro op	instr	code class	31-25	24-20	19-15	14-12	11-7	6-0
base								
00001	loadvec8/16 vector_rd (idx) (csr1,csr2)	custom0_rs1	vector_rd[5:0]+1'b0	5'bxxxxx	reg for (idx	010	5'bxxxxx	0001011
00001	loadvec8/16 vector_rs (idx) (csr1,csr2)		vector_rs[5:0]+1'b1	5'b0xxxx	reg for (idx	010	5'bxxxxx	0001011
00001	loadvec8/16 all_vector_rs (idx) (csr1,csr2)		vector_rs[5:0]+1'b1	5'b1xxxx	reg for (idx	010	5'bxxxxx	0001011
00010	storevec8/16 vector_rd (idx) (csr3,csr2) [relu]	custom0_rd_rs1 ,rd=0	vector_rd[5:0]+1'bx	1'b0/1+4'b0xxx	reg for (idx	110	5'b00000	0001011
00010	storevec8/16 vector_rd (idx) (csr4,csr2) [relu]		vector_rd[5:0]+1'bx	1'b0/1+4'b1xxx	reg for (idx	110	5'b00000	0001011
00011	addvec8/16 vector_rd vector_rs	custom0	vector_rd/rs[5:0]+1'bx	5'bxxxxx	5'bxxxxx	000	5'bxxxxx	0001011
00100	mov vector_rd 0	custom3	vector_rd[5:0]+1'bx	5'bxxxxx	5'bxxxxx	000	5'bxxxxx	1111011
00101	mul vector_rd (D^-1/2)	custom0_rd_rs1_rs2 ,rd=0,rs1	vector_rd[5:0]+1'bx	reg for (D)	5'b00000	111	5'b00000	0001011
fixed-rd								
01000	load-rs-add-rd-vec8/16 vector_rd (idx) (csr1,csr2)	custom1_rs1	vector_rd[5:0]+1'bx	5'bx0xxx	reg for (idx	010	5'bxxxxx	0101011
01000	load-rs-add-rd-vec8/16 vector_rd (idx) (csr3,csr2)		vector_rd[5:0]+1'bx	5'bx1xxx	reg for (idx	010	5'bxxxxx	0101011
01001	load-rs-add-rd-vec8/16 vector_rd (idx1) (idx2) (csr1,csr2)	custom0_rs1_rs2	vector_rd[5:0]+1'bx	reg for (idx2)	reg for (idx	011	5'bxxxxx	0001011
01010	load-rs-add-rd-vec8/16 vector_rd (idx) (aij) (csr1,csr2)	custom1_rs1_rs2	vector_rd[5:0]+1'bx	reg for (aij)	reg for (idx	011	5'bxxxxx	0101011
fixed-rs								
01100	load-rd-add-rs-store-rd-vec8/16 (idx) (csr4,csr2)	custom1_rd_rs1, rd=0	7'bxxxxxxx	5'bxxxxx	reg for (idx	110	5'b00000	0101011
01101	load-rd-add-rs-store-rd-vec8/16 (idx) (aij) (csr4,csr2)	custom2_rs1_rs2	7'bxxxxxxx	reg for (aij)	reg for (idx	011	5'bxxxxx	1011011
matrix								
10000	load-outerproduct-add-8*8 matrix_rd (idx1) (idx2) (csr5,csr6,csr7)	custom3_rs1_rs2	matrix_rd[5:0]+1'bx	reg for (idx2)	reg for (idx	011	5'bxxxxx	1111011
10001	storematrix8*8 matrix_rd8*8 (idx) (csr8) [relu]	custom2_rd_rs1, rd=0	matrix_rd[5:0]+1'b0	1'b0/1+4'bxxxx	reg for (idx	110	5'b00000	1011011
10001	storematrix8*8 matrix_rs8*8 (idx) (csr8) [relu]		matrix_rs[5:0]+1'b1	1'b0/1+4'bxxxx	reg for (idx	110	5'b00000	1011011
10010	loadmatrix8*8 matrix_rd (idx) (csr8)	custom2_rs1	matrix_rd[5:0]+1'b0	5'bxxxxx	reg for (idx	010	5'bxxxxx	1011011
10010	loadmatrix8*8 matrix_rs (idx) (csr8)		matrix_rs[5:0]+1'b1	5'bxxxxx	reg for (idx	010	5'bxxxxx	1011011
10011	addMatrix matrix_rd matrix_rs	custom1	matrix_rd[5:0]+matrix_rs[5:0]		5'bxxxxx	000	5'bxxxxx	0101011
fence								
10100	gpgcn-fence (use with riscv fence)	custom2 rd=0	7'bxxxxxxx	5'bxxxxx	5'bxxxxx	100	5'b00000	1011011
scm								
10101	preload block0-3 (idx) (csr1,csr2) (csr3,csr2) (csr5,csr7) (csr6,csr7	custom3_rs1	block[1:0]+5'bxxxxx	1'bx+csr[1:0]+2'bxx	reg for (idx	010	5'bxxxxx	1111011
10110	sync-preload block0-3 (finish read)	custom2	block[1:0]+5'bxxxxx	5'bxxxxx	5'bxxxxx	000	5'bxxxxx	1011011
10111	storescmback (idx) (csr4,csr2) [relu] for fixed-rs	custom3_rd_rs1, rd=0	7'bxxxxxxx	1'b0/1+4'bxxxx	reg for (idx	110	5'b00000	1111011

Figure 21. The encoding of GPGCN instruction set.

4. Hardware Architecture

4.1. Overall Microarchitecture

The GPGCN hardware accelerator is coupled with the boom RISC-V cpu core through the rocc interface, as shown in Figure 22. The GPGCN instructions are pushed to the GPGCN accelerator for execution through the boom pipeline.

As shown in Figure 23, the hardware microarchitecture 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 8 $\times$ 8 array to calculate the matrix instructions, which improves the utilization efficiency of the execution units. Among them, N is specified by the custom CSR9 register, and M is specified by the custom CSR10 register. In implementing the GPGCN hardware microarchitecture of this design, considering the IPC and memory access bandwidth that the single-core CPU rocc interface can provide, the above parameters are designed as n = 8, m = 1.

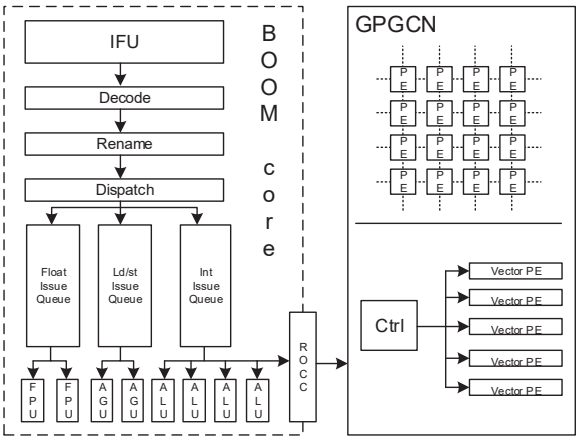

Figure 22. The overall architecture of GPGCN with boom cpu.

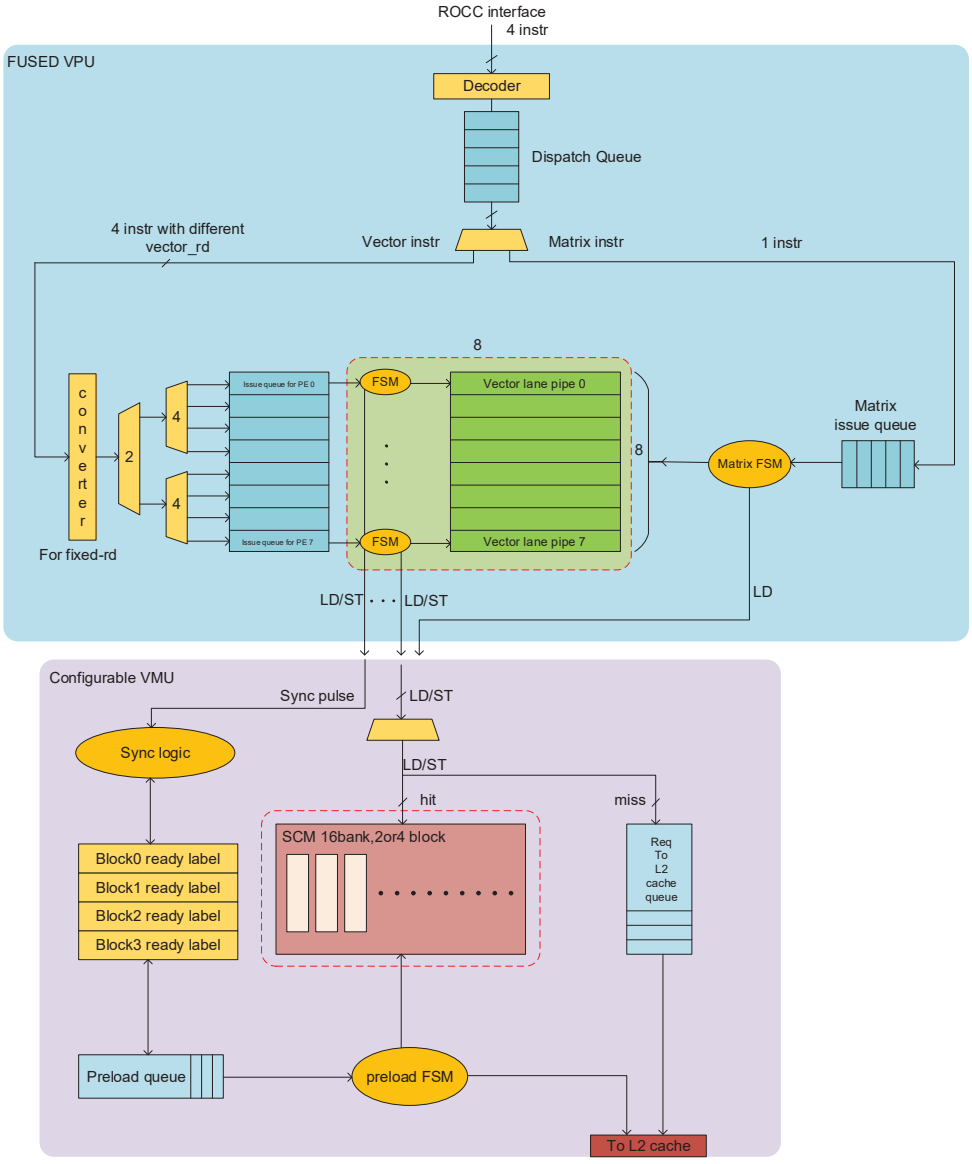

Figure 23. The hardware architecture of GPGCN.

The fused VPU includes the GPGCN custom instruction decoder (decoder), dispatch queue (dispatch queue), vector instruction issue queue (vector issue queue), matrix instruction issue queue (matrix issue queue), and an execution unit array that can be configured as eight SIMD8 vector lane pipelines or one 8×8 matrix pipeline.

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.

4.2. Microarchitecture Design Features

4.2.1. Redundant Computation Reduction

There are many hidden redundant calculations in the aggregation calculation process of the GCNs. As shown in the dotted line box in Figure 24, vertices 1 and 2 need to accumulate the feature vectors of vertices 7 and 8; vertices 2, 3, and 4 need to accumulate the feature vectors of vertices 2, 3, and 4; vertices 5 and 6 need to accumulate the eigenvectors of vertices 5 and 6; and vertices 7 and 8 need to accumulate the feature vectors of vertices 1, 2 and vertices 7, 8.

In fact, these redundant accumulation calculations only need to be calculated once: for example, the feature vectors of vertices 7 and 8 are added in advance, and then the pre-addition result can be directly used when the aggregation of vertices 1, 2, 7, and 8 are calculated, which saves three vector addition operations and four vector load operations.

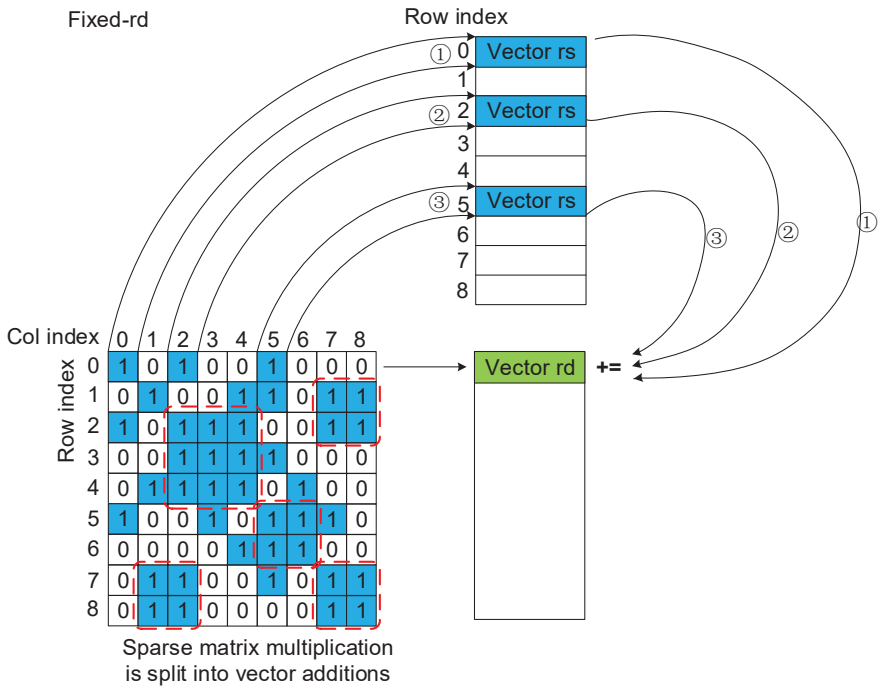

Figure 24. The redundant computations in GCNs. The numbers in circle indicate the computation order.

The scheme to achieve redundant calculation reduction in this design is to perform redundant calculation reduction of two consecutive vertices in the feature matrix for the accumulation operation: first, the sum of the feature vectors of two consecutive rows in the feature matrix is precomputed and stored in the pre-add feature matrix address space (address space specified by custom CSR3), as shown in Figure 25.

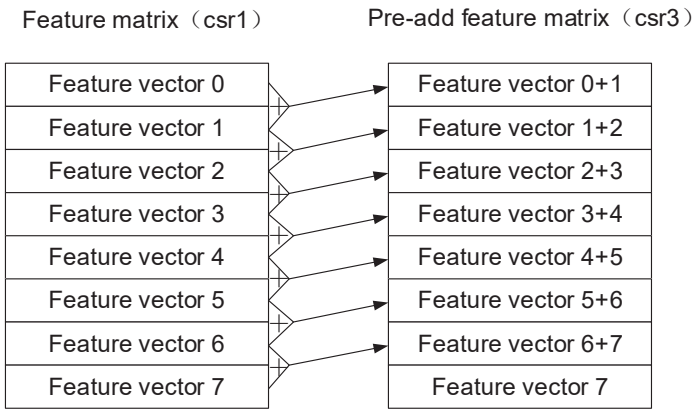

Figure 25. Pre-add for redundant computations reduction.

Then, we use the hardware logic named converter in Figure 23 to identify the fixed-rd vector instruction: load-rs-add-rd-vec8/16 vector_rd (idx1) (idx2) (CSR1,CSR2). The original execution step of this instruction is to retrieve the feature vector with the number of rows in the feature matrix equal to idx1, accumulate it into the vector rd register, and then retrieve the feature vector with the number of rows equal to idx2, and accumulate it into the vector rd register.

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, where idx = idx1. This means that when two feature vectors that this instruction needs to accumulate are in two consecutive rows in the feature matrix, it only needs to retrieve and accumulate the feature vector specified by idx1 in the pre-add feature matrix to the vector rd register.

This way, the original two loads and two accumulation calculations are reduced to one load and one accumulation calculation.

4.2.2. Memory Access Optimization

In the VMU configuration in fixed-rd mode, there is a module that is unavailable in other modes: load accumulate buffer.

Since the SCM in fixed-rd mode is configured as four blocks, each block can only provide four read ports with overlapping bank addresses, while in fixed-rd mode, eight vector lane pipelines may access the VMU at the same time. It is possible that at the same time, there are eight load requests to access the same block with only four read ports, so there must be some load requests that must wait until the next cycle to successfully access.

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, as Figure 26 shows.

The load accumulate buffer uses the data locality hidden between load requests in the fixed-rd mode and uses a certain memory access delay in exchange for the overall memory access bandwidth.

The schematic diagram of the load accumulate buffer in Figure 27 is as follows:

- It 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 has gather logic, which judges whether the memory access addresses of up to n load requests at the head of the queue are equal, and merges load requests

with equal memory access addresses into one load request access, then enters this load request into the read port of the bank corresponding to the SCM block.

- When the SCM block returns the read result of the load request, it returns the result to the vector lane pipeline corresponding to all load requests before gathering. This process is called scatter.

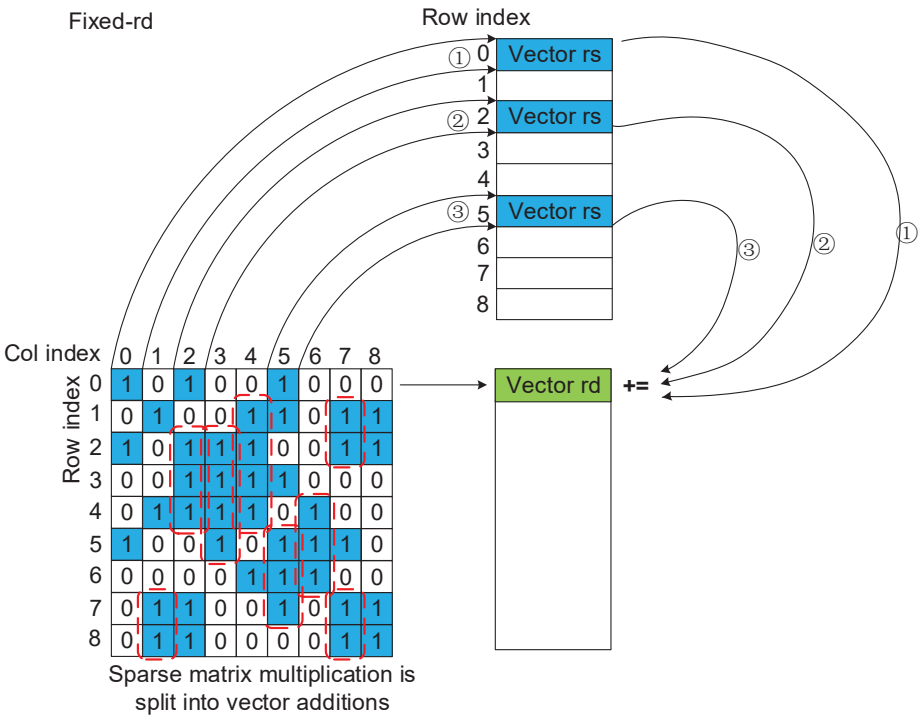

Figure 26. The data locality of aggregation in GCNs. The numbers in circle indicate the computation order.

The load accumulate buffer not only converts eight load requests into four load requests but also utilizes the locality of the access data between load requests due to the gather mechanism so that the overall memory access bandwidth is not reduced.

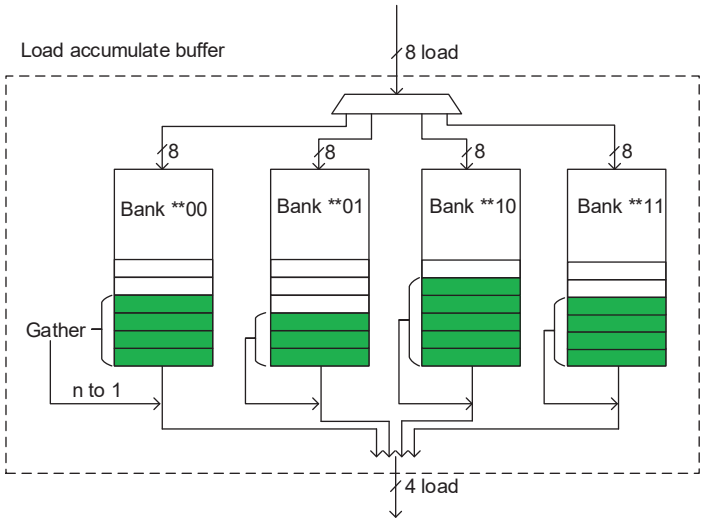

Figure 27. The schematic diagram of the load accumulate buffer. "*" in this image represents any binary value.

5. Evaluation

Experimental environment: A GPGCN hardware accelerator is designed and implemented using chisel language under the chipyard [20] 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 [21] model and Micron's DDR3 timing model.

All GCNs use a two-layer structure, the feature vector length of the hidden layer is set to 16, 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. The information of the dataset used is shown in Table 1, and the parameter configuration of the hardware is shown in Table 7.

Table 7. The parameter configuration of hardware.

	Boom [22] Cpu	**Boom with Hawacha [23]**	**Boom with GPGCN**	**HYGCN [1]**
Compute unit	2 GHZ @ 1 core	2 GHz @ 4xSIMD4	2 GHz @ 8xSIMD8 vector lane(=8 × 8 matrix array)	1 GHz @ 32 SIMD16 cores, 8 systolic modules (each with 4 × 128 arrays)
On-chip memory	32 KB icache + 32 KB dcache	32 KB icache + 32 KB dcache	32 KB icache+ 32 KB dcache + 128 KB SCM	128 KB (Input), 2 MB (Edge), 2 MB (Weight)
Off-chip memory	666 MHz @ 512 MB DDR3 with 5.3 GB/s bandwidth	666 MHz @ 512 MB DDR3 with 5.3 GB/s bandwidth	666 MHz @ 512 MB DDR3 with 5.3 GB/s bandwidth	256 GB/s @ HBM1.0

The execution latency of GCN with the Cora dataset and Citeseer dataset under different hardware is shown in Tables 8 and 9. All latencies are normalized to cycles to remove the effects of different frequencies.

Table 8. The GCN execution latency of different hardware with different datasets.

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

The execution latency of GAT with the Cora dataset and the Citeseer dataset under different hardware is shown in Tables 10 and 11.

Table 9. The GCN execution speedup of different hardware with different datasets.

GCN Speedup	Boom Cpu	Boom with Hawacha	Boom with GPGCN	HYGCN
Cora	1×/comb1 1×/agg1 1×/comb2 1×/agg2 1×/total	3.61×/comb1 3.89×/agg1 1.74×/comb2 3.67×/agg2 3.75×/total	714×/comb1 1723×/agg1 10×/comb2 1393×/agg2 1001×/total	38,278×/total
Citeseer	1×/comb1 1×/agg1 1×/comb2 1×/agg2 1×/total	4.55×/comb1 4.23×/agg1 1.88×/comb2 3.92×/agg2 4.29×/total	1493×/comb1 4079×/agg1 13×/comb2 3336×/agg2 1937×/total	7265×/total

Table 10. The GAT execution latency of different hardware with different datasets.

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

Table 11. The GAT execution speedup of different hardware with different datasets.

GAT Speedup	Boom Cpu	Boom with Hawacha	Boom with GPGCN	HYGCN
Cora	1×/comb1 1×/agg1 1×/comb2 1×/agg2 1×/total	4.14×/comb1 4.71×/agg1 1.77×/comb2 4.16×/agg2 4.42×/total	864×/comb1 2403×/agg1 14×/comb2 1736×/agg2 1340×/total	Not Support
Citeseer	1×/comb1 1×/agg1 1×/comb2 1×/agg2 1×/total	4.87×/comb1 5.01×/agg1 1.89×/comb2 4.48×/agg2 4.84×/total	1484×/comb1 4553×/agg1 12×/comb2 3332×/agg2 2060×/total	Not Support

Each layer of the GCNs is divided into combination and aggregation for comparison. Compared with the traditional CPU, the execution efficiency of the GPGCN accelerator is significantly improved in the rest of the calculation process:

- The acceleration ratio of aggregation is above 1300× for the Cora dataset and above 3300× for the Citeseer dataset.
- The acceleration ratio of combination of the first layer network is about 700× for the Cora dataset and about 1500× for the Citeseer dataset.
- The acceleration ratio of comb2 (the combination stage of the second layer network) is about 10× for both the Cora dataset and the Citeseer dataset.
- The total acceleration ratio is about 1001× for the Cora dataset and 1937× for the Citeseer dataset.

The relatively smaller acceleration ratio of comb2 is due to the fact that the data in the combination stage of the second-layer network are relatively small, which is a dense matrix multiplication operation of n $\times$ 16 $\times$ 16 $\times$ 8, and no sparsity can be used to bring about a significant acceleration effeciency.

When comparing cpu with the traditional vector expansion, such as hwacha [23], although the computing resources of GPGCN are four times those of hwacha, the acceleration ratio is much more than 4$\times$, which indicates that the GPGCN's acceleration efficiency is much higher:

- The total acceleration ratio is about 267$\times$ for GCN with the Cora dataset and 460$\times$ for GCN with the Citeseer dataset.

Comparing the acceleration effects of different datasets under the same GCNs, the acceleration ratio of GPGCN on the Citeseer dataset is higher than that of the Cora dataset, because the Citeseer dataset has a higher sparsity that can be utilized, as shown in Table 1.

However, compared with dedicated accelerators such as HYGCN, the acceleration efficiency of GPGCN accelerators is relatively lower because HYGCN uses a lot more computing resources (about 72$\times$) and larger on-chip cache and off-chip main memory bandwidth (about 50$\times$). However, the speedup ratio of HYGCN for the Citeseer dataset is lower than that of the Cora dataset, indicating that HYGCN does not full use the sparsity of the dataset as does GPGCN, and because GPGCN has software programmability, it can accelerate the GAT network that HYGCN does not support.

6. Conclusions and Future Works

In this work, we pioneeringly propose the concept of GPGCN and design the GPGCN custom instructions based on RISC-V ISA extension. Then, we propose a general-purpose GCNs hardware accelerator based on the proposed GPGCN custom instructions with various optimized designs such as redundant computation reduction and load accumulate buffer.

The acceleration efficiency of the GPGCN accelerator based on RISC-V instruction extension is higher than that of CPU with traditional vector units. Compared with traditional CPU, GPGCN achieves above 1001$\times$ speedup for GCN with the Cora dataset and 1937$\times$ speedup for the Citeseer dataset. Compared with CPU with traditional vector units, GPGCN achieves above 267$\times$ speedup for GCN with the Cora dataset and 460$\times$ speedup for the Citeseer dataset.

Compared with dedicated accelerators, since GPGCN has better programmability and generality, it supports accelerating the GAT network, while HYGCN [1] does not support it.

Moreover, since GPGCN provides software programmability, our future work is to use the reorder algorithm to mine data locality in graph datasets to break the limitation of the memory wall and further improve the acceleration efficiency of the GPGCN accelerator.

Author Contributions: Conceptualization, W.T. and P.Z.; methodology, W.T.; software, W.T.; validation, W.T.; formal analysis, W.T.; investigation, W.T.; resources, W.T.; data curation, W.T.; writing—original draft preparation, W.T.; writing—review and editing, P.Z.; visualization, W.T.; supervision, W.T.; project administration, W.T. All authors have read and agreed to the published version of the manuscript.

Funding: This work is supported by the National Key R&D Program of China (2021YFB2206200).

Data Availability Statement: The data presented in this study are available on request from the corresponding author after publication. The data are not publicly available due to privacy or ethical restrictions.

Acknowledgments: Thanks to Z.J.U. for providing the high-performance server to support our research.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Yan, M.; Deng, L.; Hu, X.; Liang, L.; Feng, Y.; Ye, X.; Zhang, Z.; Fan, D.; Xie, Y. Hygcn: A gcn accelerator with hybrid architecture. In Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), San Diego, CA, USA, 22–26 February 2020; pp. 15–29.
2. Liang, S.; Wang, Y.; Liu, C.; He, L.; Huawei, L.; Xu, D.; Li, X. Engn: A high-throughput and energy-efficient accelerator for large graph neural networks. *IEEE Trans. Comput.* **2020**, *70*, 1511–1525. [CrossRef]
3. Geng, T.; Li, A.; Shi, R.; Wu, C.; Wang, T.; Li, Y.; Haghi, P.; Tumeo, A.; Che, S.; Reinhardt, S.; et al. AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing. In Proceedings of the 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Athens, Greece, 17–21 October 2020; pp. 922–936.
4. Li, J.; Louri, A.; Karanth, A.; Bunescu, R. Gcnax: A flexible and energy-efficient accelerator for graph convolutional neural networks. In Proceedings of the 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), Seoul, Republic of Korea, 27 February–3 March 2021; pp. 775–788.
5. You, H.; Geng, T.; Zhang, Y.; Li, A.; Lin, Y. Gcod: Graph convolutional network acceleration via dedicated algorithm and accelerator co-design. In Proceedings of the 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA), Seoul, Republic of Korea, 12–16 February 2022; pp. 460–474.
6. Kiningham, K.; Levis, P.; Ré, C. GRIP: A graph neural network accelerator architecture. *IEEE Trans. Comput.* **2022**, 1–12. *Early Access*. [CrossRef]
7. Kang, M.; Hwang, R.; Lee, J.; Kam, D.; Lee, Y.; Rhu, M. GROW: A Row-Stationary Sparse-Dense GEMM Accelerator for Memory-Efficient Graph Convolutional Neural Networks. *arXiv* **2022**, arXiv:2203.00158.
8. Tao, Z.; Wu, C.; Liang, Y.; He, L. LW-GCN: A Lightweight FPGA-based Graph Convolutional Network Accelerator. *arXiv* **2021**, arXiv:2111.03184.
9. Romero Hung, J.; Li, C.; Wang, P.; Shao, C.; Guo, J.; Wang, J.; Shi, G. ACE-GCN: A Fast data-driven FPGA accelerator for GCN embedding. *ACM Trans. Reconfigurable Technol. Syst. (TRETS)* **2021**, *14*, 1–23. [CrossRef]
10. Stevens, J.R.; Das, D.; Avancha, S.; Kaul, B.; Raghunathan, A. Gnnerator: A hardware/software framework for accelerating graph neural networks. In Proceedings of the 2021 58th ACM/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 5–9 December 2021; pp. 955–960.
11. Chen, C.; Li, K.; Zou, X.; Li, Y. DyGNN: Algorithm and Architecture Support of Dynamic Pruning for Graph Neural Networks. In Proceedings of the 2021 58th ACM/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 5–9 December 2021; pp. 1201–1206.
12. Veličković, P.; Cucurull, G.; Casanova, A.; Romero, A.; Lio, P.; Bengio, Y. Graph attention networks. *arXiv* **2017**, arXiv:1710.10903.
13. Miao, S. A Review on Important Issues in GCN Accelerator Design. In Proceedings of the 2021 International Conference on Public Art and Human Development (ICPAHD 2021), Kunming, China, 24–26 December 2021; pp. 1158–1162.
14. Kipf, T.N.; Welling, M. Semi-supervised classification with graph convolutional networks. *arXiv* **2016**, arXiv:1609.02907.
15. Xu, K.; Hu, W.; Leskovec, J.; Jegelka, S. How powerful are graph neural networks? *arXiv* **2018**, arXiv:1810.00826.
16. Hamilton, W.; Ying, Z.; Leskovec, J. Inductive representation learning on large graphs. In Proceedings of the Proceedings of the 31st International Conference on Neural Information Processing Systems, Long Beach, CA, USA, 4–9 December 2017.
17. Liu, S.; Du, Z.; Tao, J.; Han, D.; Luo, T.; Xie, Y.; Chen, Y.; Chen, T. Cambricon: An instruction set architecture for neural networks. In Proceedings of the 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), Seoul, Republic of Korea, 18–22 June 2016; pp. 393–405.
18. Vasiljevic, J.; Bajic, L.; Capalija, D.; Sokorac, S.; Ignjatovic, D.; Bajic, L.; Trajkovic, M.; Hamer, I.; Matosevic, I.; Cejkov, A.; et al. Compute substrate for Software 2.0. *IEEE Micro* **2021**, *41*, 50–55. [CrossRef]
19. Jeong, G.; Qin, E.; Samajdar, A.; Hughes, C.J.; Subramoney, S.; Kim, H.; Krishna, T. RASA: Efficient Register-Aware Systolic Array Matrix Engine for CPU. In Proceedings of the 2021 58th ACM/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 5–9 December 2021; pp. 253–258.
20. Amid, A.; Biancolin, D.; Gonzalez, A.; Grubb, D.; Karandikar, S.; Liew, H.; Magyar, A.; Mao, H.; Ou, A.; Pemberton, N.; et al. Chipyard: Integrated design, simulation, and implementation framework for custom socs. *IEEE Micro* **2020**, *40*, 10–21. [CrossRef]
21. Rosenfeld, P.; Cooper-Balis, E.; Jacob, B. DRAMSim2: A cycle accurate memory system simulator. *IEEE Comput. Archit. Lett.* **2011**, *10*, 16–19. [CrossRef]
22. Zhao, J.; Korpan, B.; Gonzalez, A.; Asanovic, K. Sonicboom: The 3rd generation berkeley out-of-order machine. In Proceedings of the Fourth Workshop on Computer Architecture Research with RISC-V, online, 29 May 2020; Volume 5.
23. Lee, Y.; Waterman, A.; Avizienis, R.; Cook, H.; Sun, C.; Stojanović, V.; Asanović, K. A 45nm 1.3 GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators. In Proceedings of the ESSCIRC 2014-40th European Solid State Circuits Conference (ESSCIRC), Venice, Italy, 22–26 September 2014; pp. 199–202.