Li JJ, Wang K, Zheng H *et al.* GShuttle: Optimizing memory access efficiency for graph convolutional neural network accelerators. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 38(1): 115-127 Jan. 2023. DOI: 10.1007/s11390-023-2875-9

GShuttle: Optimizing Memory Access Efficiency for Graph Convolutional Neural Network Accelerators

Jia-Jun Li[1] (李家军), Ke Wang[2] (王　可), Hao Zheng[3] (郑　皓), and Ahmed Louri[4], *Fellow, IEEE*

[1] *School of Astronautics, Beihang University, Beijing 100191, China*

[2] *Department of Electrical and Computer Engineering, University of North Carolina at Charlotte, Charlotte, NC 28223 U.S.A.*

[3] *Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL 32816, U.S.A.*

[4] *Department of Electrical and Computer Engineering, George Washington University, Washington, DC 20052, U.S.A.*

E-mail: jiajunli@buaa.edu.cn; ke.wang@uncc.edu; hao.zheng@ucf.edu; louri@gwu.edu

Received September 29, 2022; accepted January 1, 2023.

Abstract Graph convolutional neural networks (GCNs) have emerged as an effective approach to extending deep learning for graph data analytics, but they are computationally challenging given the irregular graphs and the large number of nodes in a graph. GCNs involve chain sparse-dense matrix multiplications with six loops, which results in a large design space for GCN accelerators. Prior work on GCN acceleration either employs limited loop optimization techniques, or determines the design variables based on random sampling, which can hardly exploit data reuse efficiently, thus degrading system efficiency. To overcome this limitation, this paper proposes GShuttle, a GCN acceleration scheme that maximizes memory access efficiency to achieve high performance and energy efficiency. GShuttle systematically explores loop optimization techniques for GCN acceleration, and quantitatively analyzes the design objectives (e.g., required DRAM accesses and SRAM accesses) by analytical calculation based on multiple design variables. GShuttle further employs two approaches, pruned search space sweeping and greedy search, to find the optimal design variables under certain design constraints. We demonstrated the efficacy of GShuttle by evaluation on five widely used graph datasets. The experimental simulations show that GShuttle reduces the number of DRAM accesses by a factor of 1.5 and saves energy by a factor of 1.7 compared with the state-of-the-art approaches.

Keywords graph convolutional neural network, memory access, neural network accelerator

1 Introduction

In recent years, deep learning over graph data has achieved great success in a broad range of applications, such as traffic prediction[1], object detection[2], and disease classification[3]. One of the most successful models is Graph Convolutional Neural Network (GCN)[4] that re-defines the notion of convolution for graph data, and has been widely used in data centers of Google, Alibaba[5], and Facebook[6].

Just like traditional neural networks, training and inference of GCNs are both compute- and memory-intensive, which poses a major challenge to the hardware platforms. Typically, the graph convolutional layers occupy the majority of execution time in GCNs[7, 8] through two primary phases: aggregation and combination. The computation in the combination phase is similar to that in conventional neural networks. However, the aggregation phase relies on the graph structure, which is often sparse and irregular. This difference imposes new requirements on the design of GCN architectures.

Regular Paper

Special Issue in Honor of Professor Kai Hwang's 80th Birthday

A preliminary version of the paper was published in the Proceedings of HPCA 2021.

This work was supported by the U.S. National Science Foundation under Grant Nos. CCF-2131946, CCF-1953980, and CCF-1702980. Part of this work was conducted when Dr. Jia-Jun Li was a post-doctoral researcher at the HPCAT Laboratory, George Washington University.

The key computation pattern in GCNs can be abstracted as chain sparse-dense matrix multiplications (SpMMs)[9]. It involves six loops that result in a large design space for GCN accelerators in terms of parallelism, computation partitioning and scheduling. These problems can be handled by the existing loop optimization techniques[10], such as loop tiling, loop unrolling, loop interchange and loop fusion. Recently, a few customized GCN accelerators[9, 11, 12] have leveraged these techniques to deliver gains in performance and energy efficiency. However, none of them has systematically studied the impact of these techniques on system efficiency in terms of latency, on-chip SRAM accesses and off-chip DRAM accesses. Instead, they either employ limited loop optimization techniques, or determine the design variables based on random sampling. As a result, they can hardly exploit data reuse efficiently, leading to increased memory accesses. As memory access is much more expensive than computation[13], it will significantly degrade the energy efficiency.

To this end, this paper proposes GShuttle, a GCN acceleration scheme that maximizes memory access efficiency to improve energy efficiency. First, we provide an in-depth analysis of the four loop optimization techniques for GCN computation and use corresponding design variables to characterize different acceleration schemes. We then build analytical models to quantitatively estimate the design objectives of GCN accelerators, such as on-chip SRAM accesses and off-chip DRAM accesses. Based on the analytical models, we formulate the accelerator design as an optimization problem, which aims to find the best combination of design variables that maximizes the design objectives under certain design constraints, such as on-chip storage size and the number of processing elements (PEs). We further propose two algorithms, namely pruned search space sweeping (PSSS) and greedy search (GS), to address the problem. PSSS prunes the search space to make the search practically manageable, while GS leverages empirical rules concluded from simulation results to further reduce the time complexity of the search. To demonstrate the efficacy of the proposed framework, we simulate the corresponding hardware accelerators based on the design variables derived from GShuttle. The experimental simulations show that our approach reduces DRAM accesses by a factor of 1.5 and saves energy by a factor of 1.7 on average compared with the state-of-the-art approaches.

In summary, this paper makes the following contributions.

• *Problem Definition.* We define an optimization problem for GCN accelerator designs to maximize memory access efficiency based on an in-depth analysis of the loop optimization techniques and analytical models.

• *Search Algorithms.* We develop two search algorithms to solve the optimization problem to find a solution within a reasonable amount of time.

• *Framework Validation.* We build an accelerator simulator based on the design variables obtained from GShuttle, and performed simulation studies to demonstrate the efficacy of the proposed framework.

This paper is a significantly extended and revised version of [12]. In [12], we proposed a flexible and energy-efficient accelerator for GCNs. However, [12] mainly focuses on the dataflow optimization and has not systematically explored the optimization of memory accesses. In this paper, we propose GShutte which maximizes memory access efficiency (including both DRAM and SRAM accesses) to achieve high performance and energy efficiency.

The rest of the paper is organized as follows. Section 2 summarizes the key design variables that characterize the loop optimization techniques for GCN acceleration. Section 3 describes the optimization problem that maximizes the memory access efficiency under certain constraints based on the analytical models that estimate the on-chip SRAM and off-chip DRAM accesses, and presents two approaches to solve the optimization problem to find the best combination design variables within a reasonable amount of time. Section 4 describes the experimental methodology, and presents the experimental results in comparison with prior work. Section 5 introduces related work and Section 6 concludes the paper.

2 GCN Acceleration

2.1 GCN Computation

The typical structure of a graph convolutional layer is illustrated in Fig.1. The main computation of the GCN models[14–16] can be abstracted as a chain SpMM:

$$\boldsymbol{X}^{(k+1)} = \sigma(\hat{\boldsymbol{A}}\boldsymbol{X}^{(k)}\boldsymbol{W}^{(k)}), \tag{1}$$

where $\boldsymbol{X}^{(l)}$ is the matrix of input features in layer l; each column of $\boldsymbol{X}$ represents a feature vector while each row denotes a node. $\boldsymbol{W}^{(l)}$ is the weight matrix of

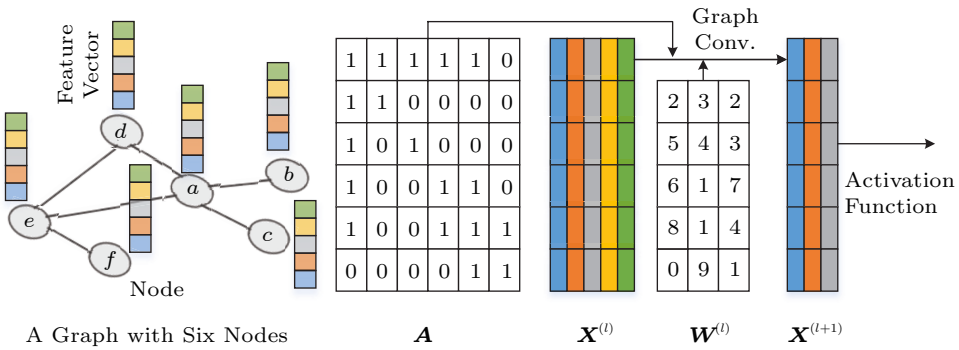

Fig.1. Illustration of a GCN layer. The graph contains six nodes. $\boldsymbol{A}$: adjacency matrix, $\boldsymbol{X}^{(l)}$: feature vectors of layer l, $\boldsymbol{W}^{(l)}$: weight matrix of layer l, and $\boldsymbol{X}^{(l+1)}$: feature vectors of layer $l+1$.

layer l. $\sigma(\cdot)$ denotes the non-linear activation function such as ReLU. $\hat{\boldsymbol{A}}$ is a transformed matrix from the graph adjacency matrix. The transformation function varies across different GCN models. Since $\hat{\boldsymbol{A}}$ can be computed offline from $\boldsymbol{A}$, we hereafter use $\boldsymbol{A}$ to denote the normalized $\hat{\boldsymbol{A}}$.

The chain SpMM in GCNs consists of six loops as shown in the pseudocode in Fig.2. We assume that $\boldsymbol{A} \in \mathbb{R}^{M\times N}$, $\boldsymbol{X} \in \mathbb{R}^{N\times K}$, $\boldsymbol{W} \in \mathbb{R}^{K\times C}$. Matrix $\boldsymbol{B} \in \mathbb{R}^{N\times C}$ is the intermediate result of $\boldsymbol{X}\cdot\boldsymbol{W}$ and $\boldsymbol{O} \in \mathbb{R}^{M\times C}$ is the final output matrix. We assume we use the execution order of $\boldsymbol{A}\cdot(\boldsymbol{X}\cdot\boldsymbol{W})$ as it reduces the number of computations for most graph datasets[9].

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

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

Fig.2. Pseudocode of the chain SpMM in GCNs.

2.2 GCN Accelerators

Recently, a few GCN accelerators have been proposed, delivering substantial gains in performance and energy efficiency compared with generic CPU- and GPU-based solutions. Specifically, HyGCN[11] exploits two dedicated compute engines, i.e., an aggregation engine and a combination engine, to accelerate the aggregation and combination phases, respectively. AWB-GCN[9] is an architecture for accelerating GCNs and Sparse-dense Matrix Multiplication (SpMM) kernels, and addresses the issue of workload imbalance in processing real-world graphs. GCNAX[12] proposes a flexible dataflow for GCNs that simultaneously improves resource utilization and reduces data movement.

These accelerators can be illustrated by the typical architecture shown in Fig.3. It consists of an accel-

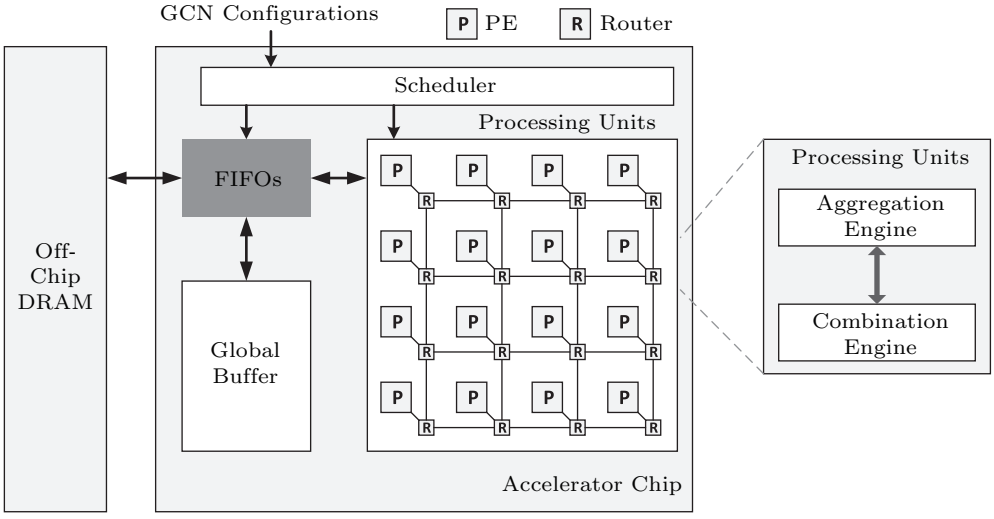

Fig.3. A typical GCN accelerator architecture.

erator chip and off-chip memory (usually DRAM). The accelerator chip is primarily composed of a processing unit (PU), a global buffer (GLB) and a scheduler. The PU can support high compute parallelism to perform the matrix multiplications, consisting of either two separate engines (HyGCN) or one uniform engine (AWB-GCN, GCNAX). The scheduler is used to map the GCNs onto the proposed accelerator using the computation sequences defined by the loop optimization techniques. GLB is usually a uniform software-controlled SRAM scratchpad memory that can be used to exploit input data reuse and hide DRAM access latency, or for the storage of intermediate data. The accelerator provides three levels of the memory hierarchy, including DRAM, GLB and local registers in PEs. Accessing data from a different level implies a different energy cost[13]. In this paper, we focus on the expensive off-chip DRAM accesses (between off-chip DRAM and GLB) and on-chip SRAM accesses (between GLB and registers in PEs).

2.3 Loop Optimization and Design Variables

The chain SpMMs can be transformed in numerous ways to capture different reuse patterns to map the computation to a hardware accelerator implementation. In this paper, we will investigate four loop optimization techniques, namely loop unrolling, loop tiling, loop interchange and loop fusion, to optimize the memory access patterns with the three levels of the memory hierarchy.

2.3.1 *Loop Unrolling*

Loop unrolling determines the parallelization strategy of the GCN loops, which then determines the PE array scale and organization as well as the size of registers in each PE. It can be used to increase the utilization of massive computation resources. Researchers have extensively studied the methods to unroll SpMM for parallel computations. As illustrated in Fig.4 which takes SpMM1 as an example, unrolling different loops directs parallelization of different computations, which affects the optimal PE array architecture with respect to the data reuse opportunities and memory access patterns.

• *Loop*-1 *Unrolled* (*Fig.4*(a)). In this case, a column vector of Pn pixels from $\boldsymbol{X}$ is multiplied with a pixel from $\boldsymbol{W}$ in each cycle, and generates a column vector of P_n pixels which will be accumulated to matrix $\boldsymbol{B}$. If data reuse in local registers is not enabled, it will involve $2 \times P_n + 1$ SRAM reads and P_n SRAM writes in each cycle.

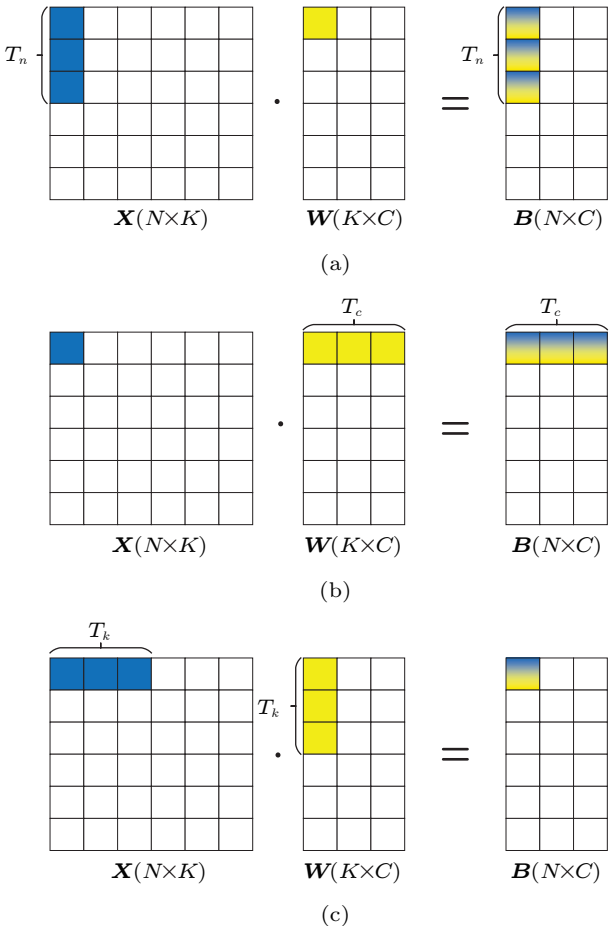

Fig.4. Loop unrolling. (a) Unroll loop-1. (b) Unroll loop-2. (c) Unroll loop-3.

• *Loop*-2 *Unrolled* (*Fig.4*(b)). In this case, a row vector of P_c pixels from $\boldsymbol{W}$ is multiplied with a pixel from $\boldsymbol{X}$ in each cycle, and generates a row vector of P_c pixels which will be accumulated to matrix $\boldsymbol{B}$. If data reuse in local registers is not enabled, it will involve $2 \times P_c + 1$ SRAM reads and P_c SRAM writes in each cycle.

• *Loop*-3 *Unrolled* (*Fig.4*(c)). In this case, the inner product of a row vector of P_k pixels from $\boldsymbol{X}$ and a column vector of the same size from $\boldsymbol{W}$ is computed in each cycle, and generates one pixel which will be accumulated to matrix $\boldsymbol{B}$. If data reuse in local registers is not enabled, it will involve $2 \times P_k + 1$ SRAM reads and 1 SRAM write in each cycle.

These unrolling factors (P_n, P_k, P_c) will determine the total number of parallel multiple-and-accumulation (MAC) operations as well as the number of re-

quired multipliers.

2.3.2 Loop Tiling

Loop tiling can be applied for each SpMM to leverage data locality and it determines the required capacity and the partitioning of GLB. As the on-chip GLB capacity is usually not large enough to hold all the data in GCNs, loop tiling can be used to divide the entire data and only fit a small portion of the data into the on-chip GLB. By properly selecting the loop tile sizes, the data reuse can be maximized to reduce off-chip DRAM access, which will significantly improve the overall energy efficiency as the energy cost of off-chip memory accesses is orders of magnitude higher than that of arithmetic operations. The tile sizes set the lower bound of the required GLB capacity. In other words, the GLB should be sized large enough to hold the data tiles.

2.3.3 Loop Interchange

Loop interchange[17] determines the computation order of the loops and it can be used to enable different types of data reuse to reduce external memory traffic by exchanging the order of the nested loops. There are two types of loop interchange in the GCN loops, namely intra-tiling and inter-tiling loop orders. The intra-tiling loop order determines the pattern of data movements from on-chip GLB to register files. The inter-tiling loop order determines the data movement from external memory to on-chip GLB. Loop interchange along with local memory promotion can reduce the data movements. Specifically, if the innermost loop is irrelevant to a matrix, i.e., the loop iterator does not appear in the access function of the matrix[18], there will be redundant memory operations between different loop iterations which can be eliminated to reduce memory access operations.

2.3.4 Loop Fusion

Loop fusion optimization[19] can be leveraged to reduce data transfer of intermediate data. Specifically, we can fuse the processing of SpMM1 and SpMM2 to reduce the data transfer of matrix $\boldsymbol{B}$ between off-chip DRAM and on-chip GLB. As shown in Fig.2, if the two SpMMs are executed sequentially without fusion, the elements of matrix $\boldsymbol{B}$ are stored back to DRAM in SpMM1, and they are again fetched from DRAM to on-chip in SpMM2. Therefore, we can reduce the data transfer of these intermediate data by fusing the execution of SpMM1 and SpMM2. When SpMM1 finishes the computation of loop k and generates a $\boldsymbol{B}$ chunk, we can pause the execution of SpMM1 and proceed to the execution of SpMM2. By doing so, the data transfer of the intermediate matrix ($\boldsymbol{B}$) is eliminated. Notably, although loop fusion reduces data transfer of intermediate results, it somehow sacrifices the freedom of loop interchange. Specifically, the iteration k in SpMM1 must be the innermost loop to ensure that matrix $\boldsymbol{B}$ finishes all its computations (not a PartialMat) before being forwarded to SpMM2. Moreover, as m becomes the innermost loop in the communication part of SpMM2, matrix $\boldsymbol{O}$ has to be frequently transferred between on-chip and off-chip. Since $\boldsymbol{O}$ is the result matrix, the volume of data transfer is doubled compared with the input matrix such as matrix $\boldsymbol{A}$ because the result matrix has to be written back to the main memory when being replaced, whereas the input matrix can be directly replaced without being written back.

Table 1 lists the parameters in GCNs and the design variables used by the four loop optimization techniques, where variables with a prefix of capital T denotes the tile size, and P for unrolling factors. Since both SpMM1 and SpMM2 contain the loops n and c, we hereafter use $n0, c0$ as the loop iterator in SpMM1, and $n1, c1$ as the loop iterator in SpMM2.

Table 1. GCN Loop Parameters and Design Variables

	GCN Loop	Dimension	Without Loop Fusion		With Loop Fusion	
			Loop Tiling	Loop Unrolling	Loop Tiling	Loop Unrolling
SpMM1 ($\boldsymbol{B}=\boldsymbol{XW}$)	Loop-1	N	T_{n0}	P_{n0}	T_{n0}	P_{n0}
	Loop-2	C	T_{c0}	P_{c0}	T_{c0}	P_{c0}
	Loop-3	K	T_k	P_k	T_k	P_k
SpMM2 ($\boldsymbol{O}=\boldsymbol{AB}$)	Loop-4	M	T_m	P_m	T_m	P_m
	Loop-5	C	T_{c1}	P_{c1}	–(equal to T_{c0})	P_{c1}
	Loop-6	N	T_{n1}	P_{n1}	–(equal to T_{n0})	P_{n1}

3 GShuttle: Optimization Framework

Although we have concluded the key factors that determine the memory accesses, it is not easy to decide which combination of design variables is optimal for a given GCN layer. Simply using static design variables by random sampling for all layers as a lot of

prior work did[9, 11] is far from optimal due to the dimension and sparsity variance across different layers. Therefore, in this section, we introduce how to determine the design variables for a given graph convolutional layer. We first formulate the selection of design variables as an optimization problem, which aims to find the best combination of design variables that maximizes the design objectives (e.g., the number of off-chip DRAM and on-chip SRAM accesses) under certain design constraints (e.g., the on-chip storage size and the number of PEs). We find that it is an NP-hard problem because of the large design space, thus requiring heuristic solutions in practice. Therefore, we further propose two algorithms, namely Pruned Search Space Sweeping, and Greedy Search to address this problem.

3.1 Problem Formulation

3.1.1 Design Objectives

As our aim is to optimize memory access efficiency, we will primarily target at two design objectives: to minimize the following two metrics.

• *Number of Off-Chip DRAM Accesses*. It primarily relies on the size of GLB and the degree of data reuse, which are determined by the tiling size, inter-tiling loop order, and loop fusion strategy.

• *Number of On-Chip SRAM Accesses*. It is determined by the loop unrolling strategies and intra-tiling loop order, since they determine the reuse patterns of the data transfer from GLB to local registers.

To simultaneously achieve both design objectives might be infeasible as the best combination of design variables for off-chip DRAM accesses may not be optimal for on-chip SRAM accesses, and vice versa. Therefore, to optimize the overall memory access efficiency, we combine the two design objectives into one by calculating their weighted total as follows:

$$\begin{aligned} \underset{\mathbb{X}}{Minimize} \quad & V = V_d(\mathcal{X}^t, \mathcal{X}^{oo}, \mathcal{X}^f) + \omega \times V_s(\mathcal{X}^u, \mathcal{X}^{oi}) \\ \text{s.t.} \quad & 0 < T_m \leqslant M, \quad 0 < T_k \leqslant K, \\ & 0 < T_{n0} \leqslant N, \quad 0 < T_{n1} \leqslant N, \\ & 0 < T_{c0} \leqslant C, \quad 0 < T_{c1} \leqslant C, \\ & S_{\boldsymbol{X}} + S_{\boldsymbol{W}} + S_{\boldsymbol{B}1} \leqslant GLBsize, \\ & S_A + S_O + S_{\boldsymbol{B}2} \leqslant GLBsize, \\ & P_{n0} \times P_{c0} \times P_k \leqslant \#PEs, \\ & P_{n1} \times P_{c1} \times P_k \leqslant \#PEs, \end{aligned} \tag{2}$$

where $\mathbb{X} = \mathcal{X}^t \cup \mathcal{X}^{oo} \cup \mathcal{X}^f \cup \mathcal{X}^u \cup \mathcal{X}^{oi}$ denotes the entire search space, and $\mathcal{X}^t, \mathcal{X}^{oo}, \mathcal{X}^f, \mathcal{X}^u, \mathcal{X}^{oi}$ denote the parameter spaces of the tile size, inter-tiling loop order, loop fusion strategy, unrolling factors, and intra-tiling loop order, respectively. V_d and V_s denote the number of off-chip DRAM accesses and on-chip SRAM accesses, respectively. $S_{\boldsymbol{X}}, S_{\boldsymbol{W}}, S_{\boldsymbol{B}1}, S_{\boldsymbol{A}}, S_{\boldsymbol{O}}, S_{\boldsymbol{B}2}$ denote the required on-chip storage sizes of the corresponding matrices, which are determined by the tile size. ω is an adjustment parameter that reflects the difference in the energy cost between basic DRAM access and SRAM access. According to [13], we set $\omega = 0.007\,8$ indicating a basic DRAM access operation consumes 128x more energy than a basic SRAM access does. To solve this optimization problem, we first need to measure V_d and V_s given a combination of design variables and a GCN layer. To this end, we build analytical models for the estimation of V_d and V_s.

3.1.2 Estimation of Off-Chip DRAM Accesses

Since the space of the design variables is polyhedral, we use an example to explain how the analytical models for off-chip DRAM accesses are built. Assuming the inter-tiling loop order is $n_0 \to c_0 \to k, m \to c_1 \to n_1$ and loop fusion is not enabled, the total number of off-chip DRAM accesses is calculated by:

$$\begin{aligned} V_d = \alpha_{\boldsymbol{X}} \times S_{\boldsymbol{X}} + \alpha_{\boldsymbol{W}} \times S_{\boldsymbol{W}} + \alpha_{\boldsymbol{B}1} \times S_{\boldsymbol{B}1} + \\ \alpha_{\boldsymbol{B}2} \times S_{\boldsymbol{B}2} + \alpha_{\boldsymbol{A}} \times S_{\boldsymbol{A}} + \alpha_{\boldsymbol{O}} \times S_{\boldsymbol{O}}, \end{aligned} \tag{3}$$

where

$$\begin{cases} S_{\boldsymbol{X}} = \gamma_{\boldsymbol{X}} \times T_{n0} \times T_k, \\ S_{\boldsymbol{W}} = T_k \times T_{c0}, \\ S_{\boldsymbol{B}1} = T_{n0} \times T_{c0}, \\ S_{\boldsymbol{B}2} = T_{n1} \times T_{c1}, \\ S_{\boldsymbol{A}} = \gamma_{\boldsymbol{A}} \times T_m \times T_{n1}, \\ S_{\boldsymbol{O}} = T_m \times T_{c1}, \end{cases} \tag{4}$$

$$\begin{cases} \alpha_{\boldsymbol{X}} = \alpha_{\boldsymbol{W}} = \dfrac{N}{T_{n0}} \times \dfrac{C}{T_{c0}} \times \dfrac{K}{T_k}, \\ \alpha_{\boldsymbol{B}1} = \dfrac{N}{T_{n0}} \times \dfrac{C}{T_{c0}}, \\ \alpha_{\boldsymbol{B}2} = \alpha_{\boldsymbol{A}} = \dfrac{M}{T_m} \times \dfrac{C}{T_{c1}} \times \dfrac{N}{T_{n1}}, \\ \alpha_{\boldsymbol{O}} = \dfrac{M}{T_m} \times \dfrac{C}{T_{c1}}. \end{cases} \tag{5}$$

Here $\alpha_{\boldsymbol{X}}, \alpha_{\boldsymbol{W}}, \alpha_{\boldsymbol{B}}, \alpha_{\boldsymbol{A}}$ and $S_{\boldsymbol{X}}, S_{\boldsymbol{W}}, S_{\boldsymbol{B}}, S_{\boldsymbol{A}}$ denote the trip counts and buffer sizes of memory accesses to $\boldsymbol{X}$, $\boldsymbol{W}$, $\boldsymbol{B}$, $\boldsymbol{A}$, respectively. Note that $\alpha_{\boldsymbol{B}1}, \alpha_{\boldsymbol{B}2}, S_{\boldsymbol{B}1}$, $S_{\boldsymbol{B}2}$ are used to differentiate the accesses in SpMM1 and SpMM2 respectively. As matrices $\boldsymbol{X}$ and $\boldsymbol{A}$ are sparse[9, 12], the off-chip DRAM accesses can be reduced by compressed encoding of the matrices. We

assume that the zeros in matrices $\boldsymbol{X}$ and $\boldsymbol{A}$ are distributed evenly so we use the overall density of $\boldsymbol{X}$ and $\boldsymbol{A}$ $(\gamma_{\boldsymbol{X}}, \gamma_{\boldsymbol{A}})$ to represent the density of each chunk when estimating $S_{\boldsymbol{X}}$ and $S_{\boldsymbol{A}}$, as shown in (4). Although it does not reflect the actual distribution, we make this assumption for simplicity since considering the sparsity distribution would significantly increase the model complexity. Moreover, we find that the estimated values from the model deviate little from the actual values derived from a cycle-level simulation.

When changing the inter-tiling loop order, we only need to modify the equation of $\alpha's$ in (5). The enumeration of the loop orders and the corresponding equation of $\alpha's$ are omitted for brevity. When enabling loop fusion, the total number of off-chip accesses and the buffer size are also calculated by (3) and (4), but the trip counts are estimated as follows.

$$\begin{cases} \alpha_{\boldsymbol{X}} = \alpha_{\boldsymbol{W}} = \dfrac{N}{T_{n0}} \times \dfrac{C}{T_{c0}} \times \dfrac{K}{T_k}, \\ \alpha_{\boldsymbol{B1}} = \alpha_{\boldsymbol{B2}} = 0, \\ \alpha_{\boldsymbol{A}} = \dfrac{M}{T_m} \times \dfrac{C}{T_{c0}} \times \dfrac{N}{T_{n0}}, \\ \alpha_{\boldsymbol{O}} = 2 \times \dfrac{M}{T_m} \times \dfrac{C}{T_{c0}} \times \dfrac{N}{T_{n0}}. \end{cases} \tag{6}$$

We can see that the DRAM accesses of matrix $\boldsymbol{B}$ are eliminated but $\alpha_{\boldsymbol{O}}$ is much larger than that in (5).

3.1.3 *Estimation of On-Chip SRAM Accesses*

If data reuse in local registers is not enabled, the total number of read operations from on-chip GLB for SpMM1 will be calculated as follows:

$$\#sram\_read\_no\_reuse = 3 \times N \times C \times K, \tag{7}$$

since every multiplication needs three SRAM reads. The total number of write operations from PEs to on-chip buffers for SpMM1 will be calculated as follows:

$$\#sram\_read\_no\_reuse = N \times C \times K. \tag{8}$$

As mentioned in Subsection 2.3, the number of on-chip SRAM accesses can be reduced by enabling data reuse under different loop unrolling strategies and intra-tiling loop order. For example, given the intra-tiling loop order $(n0 \rightarrow c0 \rightarrow k)$ and loop-3 unrolled for SpMM1, the data elements in matrix $\boldsymbol{B}$ can be reused for T_k times. Therefore, the total number of SRAM reads and writes is:

$$\begin{aligned} \#sram\_read &= \left(2 + \frac{1}{T_k}\right) \times (N \times C \times K), \\ \#sram\_write &= \frac{1}{T_k} \times N \times C \times K, \end{aligned} \tag{9}$$

respectively.

Similarly, we can calculate the SRAM reads and writes give other combinations of design variables.

3.2 Search Algorithms

Given the analytical models, we need to find out which combination of design variables can minimize the design objective described in (2). Clearly, by sweeping all the combinations of design variables, we can find the optimal solution but it takes a lot of time due to the large search space. According to our experiments, it takes tens of hours to fully explore the entire design space on an Intel I7-8650U@1.90GHz, which is infeasible for practical use.

To simplify the search, we use outer-product-based computation architecture[20] as shown in Fig.4(b) to optimize the SRAM accesses. Although this method would have a negative impact on the reuse of the output matrix, it provides additional input matrix reuse compared with the inner-product-based method. More importantly, it well supports the elimination of zero computations. Since the input pixel from $\boldsymbol{X}$ is the input operand for all the P_{c0} multiplications, these computations can be skipped simultaneously if the input pixel is zero. For DRAM accesses, we tackle the optimization problem by developing two approaches, pruned search space sweeping and greedy search.

3.2.1 *Pruned Search Space Sweeping*

We discovered that the main reason for the large search space is that $\mathcal{X}^t$ (the search space of tiling size) contains every integer value between 1 and the dimension size. Therefore, pruning the search space $\mathcal{X}^t$ can significantly shrink the entire search space. Most tile size values cannot be fully divided by the dimension size. In such cases, we need to pad the data block to simplify data movements[21], which will degrade the GLB space utilization. Clearly, the tile size that causes less padding will utilize the GLB space efficiently, thus reducing unnecessary off-chip DRAM accesses. Among a set of the tile sizes that results in the same number of iterations, the one requires minimum padding has the least data padding. Therefore, from 1 to the dimension size, we only need to consider the smallest tile size values that yield a new number of loop iterations. For example, assuming a value of 10 for dimension N, the candidate tile size will be

$\{1, 2, 3, 4, 5, 10\}$ since they yield the number of loop iterations of $\{10, 5, 4, 3, 2, 1\}$. The number of points to sweep in dimension N will be reduced from $O(N)$ to $O(2\sqrt{N})$, which will significantly shrink the search space, thus reducing the search time from tens of hours to several minutes.

3.2.2 *Greedy Search*

Alternatively, we also provide a greedy search algorithm that further reduces the search time to several seconds. Table 2 shows how to determine the design variables for off-chip DRAM accesses ($\mathcal{X}^t, \mathcal{X}^{oo}, \mathcal{X}^f$). This greedy algorithm leverages the empirical rules concluded from many simulation results.

Table 2. Greedy Search Algorithm to Determine the Design Variables

Condition	Loop Fusion	Inter-Tiling Loop Order	Tile Size Setting Priority
$N \times C \geqslant GLBsize$	No	$n_0 \to c_0 \to k$, $m \to c_1 \to n_1$	①T_{n0}, T_m ②T_{c0}, T_{c1} ③T_{n1}, T_k
$N \times C < GLBsize$	Yes	$n_0 \to c_0 \to k \to m$	①T_{n0}, T_{n1} ②T_{c0}, T_{c1} ③T_m, T_k

The tile size setting priority indicates which tiling factor has the priority for larger number settings. For example, if $N \times C \geqslant GLBsize$, T_{n0} and T_m have the highest priority for larger number settings, which means they will be set to the maximal number while satisfying other constraints. T_{c0}, T_{c1} have the second highest priority. When T_{n0} and T_m are already set as the largest number, then T_{c0}, T_{c1} will be set as large as possible.

To better understand the greedy search, Table 3 presents the tile size, loop order and loop fusion choices for five datasets: Cora[22], CiteSeer[22], PubMed[15], Nell[23] and Reddit[15]. We constrain *GLBsize* at 128 KB.

4 Evaluation

4.1 Experimental Setup

Accelerator Implementation. To evaluate GShuttle, we built a cycle-level simulator in C++ to model the behavior of the hardware described in Section 2 when using the design variables derived from GShuttle. The simulator models the microarchitectural behaviors of each module, and supports different combinations of design variables. The simulator counts the exact amount of DRAM reads and writes, on-chip SRAM reads and writes and the number of execution cycles, which is used for estimation of performance and energy efficiency.

Design Constraints. We constrain the number of PEs no more than 128, and the on-chip GLB is no more than 128 KB. The off-chip DRAM bandwidth is 128 GB/s.

Baselines. We compare GShuttle with three GCN accelerators (HyGCN[11], AWB-GCN[9] and GCNAX[12]). GShuttle has two variants, one is GShuttle-PSSS which uses pruned search space sweeping to find the combination of design variables, and the other is GShuttle-GS which uses greedy search. Table 4 summarizes the characteristics of the baselines and GShuttle. The baseline accelerators are scaled to be equipped with the same number of multipliers and DRAM bandwidth as GShuttle. Since HyGCN and AWB-GCN use single-precision floating-point numbers (32-bit) whereas GCNAX uses double-precision (64-bit), we uniformly use double-precision numbers for all accelerators to provide a fair comparison. We also resize the baseline accelerators to be equipped

Table 3. Design Variables (Tile Size, Inter-Tiling Loop Order and Loop Fusion) Derived from Greedy Search

Dataset	Layer	(*M-N-K-C*)	γ_A	γ_X	Loop Fusion	Inter-Tiling Loop Order	Tile Size Tuple $(T_{n0}, T_{c0}, T_k, T_{n1}, T_{c1}, T_m)$
Cora	L1	(2 708-2 708-1 433-16)	0.001 800	0.012 70	Yes	$n_0 \to c_0 \to k \to m$	(2 708, 16, 1, 2 708, 16, 1)
	L2	(2 708-2 708-16-7)	0.001 800	0.780 00	Yes	$n_0 \to c_0 \to k \to m$	(2 708, 7, 1, 2 708, 7, 1)
Citeseer	L1	(3 327-3 327-3 703-16)	0.001 100	0.008 50	Yes	$n_0 \to c_0 \to k \to m$	(3 000, 16, 5, 3 000, 16, 1)
	L2	(3 327-3 327-16-6)	0.001 100	0.008 50	Yes	$n_0 \to c_0 \to k \to m$	(3 000, 6, 1, 3 000, 6, 1)
Pubmed	L1	(19 717-19 717-500-16)	0.000 280	0.100 00	No	$n_0 \to c_0 \to k$, $m \to c_1 \to n_1$	(3 073, 16, 1, 1, 16, 3 073)
	L2	(19 717-19 717-16-3)	0.000 280	0.776 00	No	$n_0 \to c_0 \to k$, $m \to c_1 \to n_1$	(3 000, 3, 1, 1 025, 3, 3 000)
Nell	L1	(65 755-65 755-61 278-64)	0.000 073	0.000 11	No	$n_0 \to c_0 \to k$, $m \to c_1 \to n_1$	(4 096, 1, 33, 1, 1, 4 096)
	L2	(65 755-65 755-64-186)	0.000 073	0.864 00	No	$n_0 \to c_0 \to k$, $m \to c_1 \to n_1$	(257, 186, 1, 1, 17, 2 817)
Reddit	L1	(232 965-232 965-602-64)	0.002 100	0.516 00	No	$n_0 \to c_0 \to k$, $m \to c_1 \to n_1$	(641, 64, 1, 1, 9, 4 096)
	L2	(232 965-232 965-64-41)	0.002 100	0.600 00	No	$n_0 \to c_0 \to k$, $m \to c_1 \to n_1$	(1 153, 41, 1, 1, 17, 2 817)

with the on-chip storage capacity. For example, we simulate the HyGCN accelerator with 128 KB on-chip storage rather than the original 16 MB. The DRAM bandwidth for all the accelerators is scaled to 128 GB/s. Note that as HyGCN uses an edge-centric programming model for the aggregation phase, their computation in the aggregation phase is not matrix multiplication. Our simulator takes this into account and estimates the execution cycles and DRAM accesses according to HyGCN's dataflow.

Table 4. Characteristics of Baselines

Baseline	Execution Order	Loop Fusion	Inter-Tiling Loop Order	Tile Size
HyGCN[11]	$(\boldsymbol{AX})\boldsymbol{W}$	Yes	$n_0 \to c_0 \to k \to m$	Static
AWB-GCN[9]	$\boldsymbol{A}(\boldsymbol{XW})$	Yes	$n_0 \to c_0 \to k \to m$	Static
GCNAX[12]	$\boldsymbol{A}(\boldsymbol{XW})$	Adaptive	Adaptive	Static
GShuttle	$\boldsymbol{A}(\boldsymbol{XW})$	Adaptive	Adaptive	Adaptive

Benchmarks. We use the graph datasets listed in Table 3 as the benchmarks to test the efficiency of GShuttle and the baselines. These datasets vary in the number of edges, vertices and sparsity levels, which provides sufficient diversity to evaluate the effectiveness of these methods.

4.2 Experimental Results

4.2.1 Number of DRAM Accesses

We firstly make the comparison on the total number of DRAM accesses, as shown in Fig.5. GShuttle provides a tremendous DRAM access advantage over the baselines. Specifically, GShuttle-GS reduces DRAM accesses by a factor of 11.1x, 3.3x and 1.4x compared with HyGCN, AWB-GCN and GCNAX, respectively. Moreover, GShuttle-PSSS reduces DRAM accesses by a factor of 11.7x, 3.4x and 1.5x over the three baselines. The high DRAM access efficiency of GShuttle mainly stems from the followings. 1) GShuttle uses adaptive loop order and fusion strategy for different graph datasets. 2) The adaptive tiling sizes of GShuttle can make the best use of the global buffer. Since HyGCN uses inefficient execution order, it involves more computations that result in more DRAM accesses. AWB-GCN optimizes the reuse of the intermediate matrix. However, it sacrifices the reuse of the output matrix due to the limited on-chip storage size. Moreover, the tile sizes are not carefully tailored in the AWB-GCN accelerator.

4.2.2 Energy Consumption

For energy evaluation, we primarily focus on three parts: 1) energy consumed by arithmetic operations, 2) energy consumed by on-chip SRAM accesses, and 3) energy consumed by off-chip DRAM accesses. As the simulator counts the number of arithmetic operations and data accesses at different levels, we use these numbers to estimate the total energy consumption by multiplying them with the energy cost of basic arithmetic and memory operations in a 45 nm CMOS process[13]. Fig.6 shows the normalized energy consumption of these accelerator designs. Overall, GShuttle-GS achieves 9.5x, 3.1x, and 1.4x energy savings on average over HyGCN, AWB-GCN, and GCNAX, respectively, while GShuttle-PSSS saves energy by a factor of 12.1x, 3.9x, and 1.7x compared with the three baseline accelerators respectively.

4.2.3 Sensitivity to Hardware Parameters

The optimization framework in Section 3 indicates that *GLBsize* is an important specification for the number of off-chip DRAM accesses. Moreover, as we scale the baseline accelerators to be equipped with the same global buffer size, which would potentially hurt the efficiency of the baseline accelerators, we conduct a sensitivity analysis on the Reddit dataset to quantify the effects of global buffer size. In Fig.7, we sweep the on-chip buffer size from 128 KB up to 8 MB to investigate how it influences the number of DRAM accesses. The y axis denotes the total number of DRAM accesses of the two layers. All schemes achieve DRAM access reduction with larger GLB provisioning, especially when *GLBsize* is small ($GLBsize < 1$ MB). With *GLBsize* going larger, the benefit gains slower. For specific schemes, GShuttle-GS and GShuttle-PSSS outperform the baselines in all GLB settings.

5 Related Work

Besides the GCN accelerators mentioned in Section 2, there are also a few other GNN (graph neural network) accelerators in the literature. Auten *et al.*[24] proposed a GNN accelerator to efficiently execute the irregular data movement required for graph computation in GNNs, while also providing a high compute throughput required by GNN models. EnGN[25] is designed to accelerate the three key stages of GNN propagation, which is abstracted as common comput-

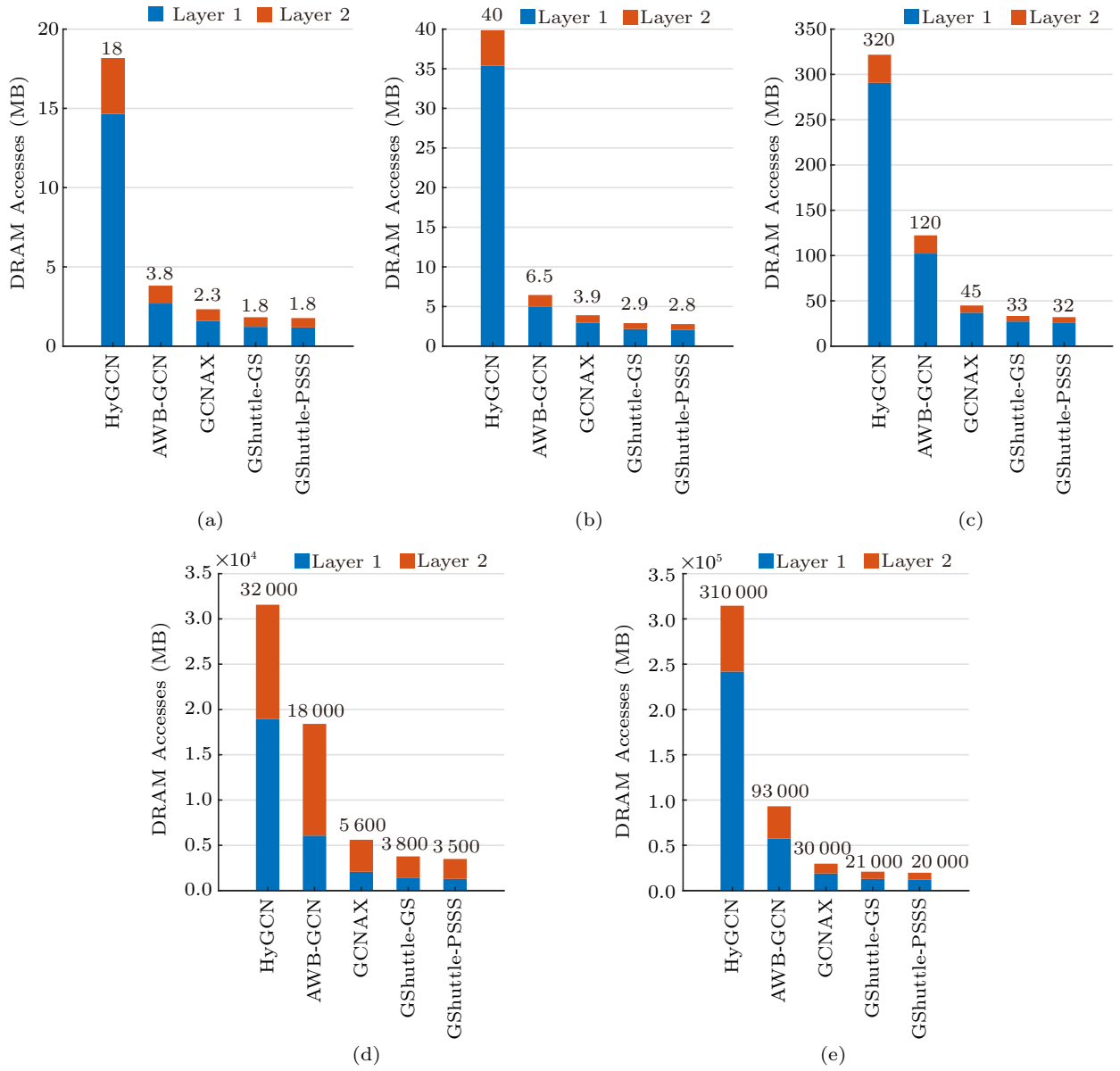

Fig.5. DRAM accesses of GShuttle and the baseline approaches. (a) Cora. (b) Citeseer. (c) Pubmed. (d) Nell. (e) Reddit.

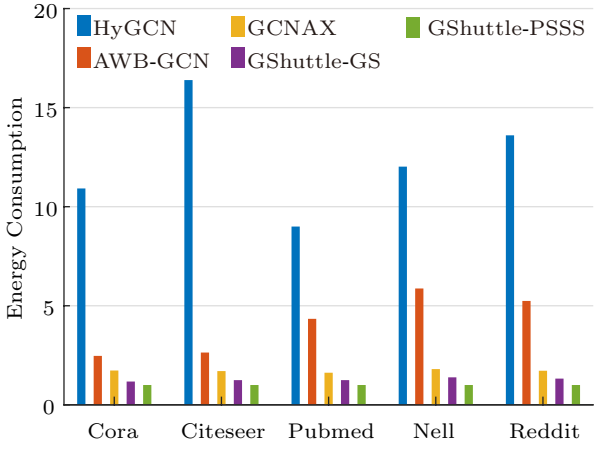

Fig.6. Energy savings of GShuttle over the baselines.

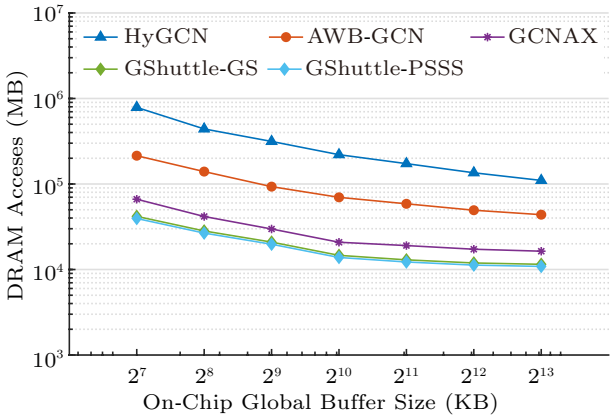

Fig.7. Number of DRAM accesses w.r.t. the variation on-chip buffer size (tested on the Reddit dataset).

ing patterns shared by typical GNNs. GRIP[26] is designed for low-latency inference of GNNs, which splits GNN inference into a fixed set of edge- and vertex-centric execution phases that can be implemented in hardware, and then specializes each unit for the unique computational structure found in each phase. GraphACT[27] is dedicated for the acceleration of training GCNs on CPU-FPGA heterogeneous systems, which incorporates multiple algorithm-architecture co-optimizations. VersaGNN[28] is systolic-array-based versatile GNN accelerator that unifies dense and sparse matrix multiplication in GNNs.

Since the loop optimization techniques proposed in this paper can significantly improve memory access efficiency, which is of paramount importance for GCN acceleration, we believe that our approach can be easily applied to the other accelerators.

6 Conclusions

Motivated by the observation that most of the energy is consumed by memory accesses for state-of-the-art GCN accelerators, this paper presented GShuttle to optimize memory access efficiency for such accelerators. GShuttle employs two algorithms to solve the memory access optimization problem raised in GCN accelerators, both of which can find the optimal design variables of GCN dataflow under certain design constraints. Our results showed that GShuttle could significantly reduce the number of DRAM and SRAM accesses for GCN accelerators. Since the optimizations on DRAM accesses are orthogonal to those on computation, we expect that GShuttle can be applied to many existing GCN accelerators such as HyGCN and AWB-GCN.

Acknowledgment The authors appreciate the constructive comments provided by the anonymous reviewers.

References

[1] Jiang W W, Luo J Y. Graph neural network for traffic forecasting: A survey. arXiv: 2101.11174, 2021. https://arxiv.org/abs/2101.11174, Dec. 2022.

[2] Shi W J, Rajkumar R. Point-GNN: Graph neural network for 3D object detection in a point cloud. In *Proc. the 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, Jun. 2020, pp.1711–1719. DOI: 10.1109/CVPR42600.2020.00178.

[3] Wee C Y, Liu C Q, Lee A, Poh J S, Ji H, Qiu A Q, The Alzheimers Disease Neuroimage Initiative. Cortical graph neural network for AD and MCI diagnosis and transfer learning across populations. *NeuroImage*: *Clinical*, 2019, 23: 101929. DOI: 10.1016/j.nicl.2019.101929.

[4] Zhang Z W, Cui P, Zhu W W. Deep learning on graphs: A survey. *IEEE Trans. Knowledge and Data Engineering*, 2022, 34(1): 249–270. DOI: 10.1109/TKDE.2020.2981333.

[5] Yang H X. AliGraph: A comprehensive graph neural network platform. In *Proc. the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, Jul. 2019, pp.3165–3166. DOI: 10.1145/3292500.3340404.

[6] Lerer A, Wu L, Shen J, Lacroix T, Wehrstedt L, Bose A, Peysakhovich A. PyTorch-BigGraph: A large-scale graph embedding system. arXiv: 1903.12287, 2019. https://arxiv.org/abs/1903.12287, Dec. 2022.

[7] Yan M Y, Chen Z D, Deng L, Ye X C, Zhang Z M, Fan D R, Xie Y. Characterizing and understanding GCNs on GPU. *IEEE Computer Architecture Letters*, 2020, 19(1): 22–25. DOI: 10.1109/LCA.2020.2970395.

[8] Zhang Z H, Leng J W, Ma L X, Miao Y S, Li C, Guo M Y. Architectural implications of graph neural networks. *IEEE Computer Architecture Letters*, 2020, 19(1): 59–62. DOI: 10.1109/LCA.2020.2988991.

[9] Geng T, Li A, Shi R B, Wu C S, Wang T Q, Li Y F, Haghi P, Tumeo A, Che S, Reinhardt S, Herbordt M C. AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing. In *Proc. the 53rd Annual IEEE/ACM International Symposium on Microarchitecture* (*MICRO*), Oct. 2020, pp.922–936. DOI: 10.1109/MICRO50266.2020.00079.

[10] Ma Y F, Cao Y, Vrudhula S, Seo J S. Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks. In *Proc. the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Feb. 2017, pp.45–54. DOI: 10.1145/3020078.3021736.

[11] Yan M Y, Deng L, Hu X, Liang L, Feng, Y J, Ye X C, Zhang Z M, Fan D R, Xie Y. HyGCN: A GCN accelerator with hybrid architecture. In *Proc. the 2020 IEEE International Symposium on High Performance Computer Architecture* (*HPCA*), Feb. 2020, pp.15–29. DOI: 10.1109/HPCA47549.2020.00012.

[12] Li J J, Louri A, Karanth A, Bunescu R. GCNAX: A flexible and energy-efficient accelerator for graph convolutional neural networks. In *Proc. the 2021 IEEE International Symposium on High-Performance Computer Architecture* (*HPCA*), Mar. 2021, pp.775–788. DOI: 10.1109/HPCA51647.2021.00070.

[13] Galal S, Horowitz M. Energy-efficient floating-point unit design. *IEEE Transactions on Computers*, 2011, 60(7): 913-922.

[14] Kipf T N, Welling M. Semi-supervised classification with graph convolutional networks. arXiv: 1609.02907, 2016. https://arxiv.org/abs/1609.02907, Dec. 2022.

[15] Hamilton W L, Ying R, Leskovec J. Inductive representation learning on large graphs. In *Proc. the 31st Interna-*

tional Conference on Neural Information Processing Systems, Dec. 2017, pp.1024–1034.

[16] Xu K, Hu W H, Leskovec J, Jegelka S. How powerful are graph neural networks? arXiv: 1810.00826, 2018. https://arxiv.org/abs/1810.00826, Dec. 2022.

[17] Allen J R, Kennedy K. Automatic loop interchange. In *Proc. the 1984 SIGPLAN Symposium on Compiler Construction*, Jun. 1984, pp.233–246. DOI: 10.1145/502874.502897.

[18] Zhang C, Li P, Sun G Y *et al*. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proc. the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Feb. 2015, pp.161–170. DOI: 10.1145/2684746.2689060.

[19] Pugh W. Uniform techniques for loop optimization. In *Proc. the 5th International Conference on Supercomputing*, Jun. 1991, pp.341–352. DOI: 10.1145/109025.109108.

[20] Pal S, Beaumont J, Park D H, Amarnath A, Feng S Y, Chakrabarti C, Kim H S, Blaauw D, Mudge T, Dreslinski R. OuterSPACE: An outer product based sparse matrix multiplication accelerator. In *Proc. the 2018 IEEE International Symposium on High Performance Computer Architecture* (*HPCA*), Feb. 2018, pp.724–736. DOI: 10.1109/HPCA.2018.00067.

[21] Nie J. Memory-driven data-flow optimization for neural processing accelerators [Ph.D. Thesis]. Princeton University, 2020. https://www.proquest.com/openview/41fe23f43fd65cafaa8c2e051aed4059/1?pq-origsite=gscholar&cbl=18750&diss=y, Jan. 2023.

[22] Sen P, Namata G, Bilgic M *et al*. Collective classification in network data. *AI Magazine*, 2008, 29(3): 93-106. DOI: 10.1609/aimag.v29i3.2157.

[23] Carlson A, Betteridge J, Kisiel B *et al*. Toward an architecture for never-ending language learning. In *Proc. the 34th AAAI Conference on Artificial Intelligence*, July 2010, pp.1306–1313.

[24] Auten A, Tomei M, Kumar R. Hardware acceleration of graph neural networks. In *Proc. the 57th ACM/IEEE Design Automation Conference* (*DAC*), Jul. 2020. DOI: 10.1109/DAC18072.2020.9218751.

[25] Liang S W, Wang Y, Liu C *et al*. EnGN: A high-throughput and energy-efficient accelerator for large graph neural networks. *IEEE Trans. Computers*, 2021, 70(9): 1511–1525. DOI: 10.1109/TC.2020.3014632.

[26] Kiningham K, Re C, Levis P. GRIP: A graph neural network accelerator architecture. arXiv: 2007.13828, 2020. https://arxiv.org/abs/2007.13828v1, Dec. 2022.

[27] Zeng H Q, Prasanna V. GraphACT: Accelerating GCN training on CPU-FPGA heterogeneous platforms. In *Proc. the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Feb. 2020, pp.255–265. DOI: 10.1145/3373087.3375312.

[28] Shi F, Jin A Y, Zhu S C. VersaGNN: A versatile accelerator for graph neural networks. arXiv: 2105.01280, 2021. https://arxiv.org/abs/2105.01280, Dec. 2022.

Jia-Jun Li received his B.E. degree from the Department of Automation, Tsinghua University, Beijing, in 2013. He received his Ph.D. degree from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, in 2019. From 2019 to 2021, he was a postdoc researcher with the Department of Electrical and Computer Engineering, George Washington University, Washington. He is currently an associate professor with the School of Astronautics, Beihang University, Beijing. His current research interests include machine learning and heterogeneous computer architecture.

Ke Wang received his Ph.D. degree in computer engineering from the George Washington University, Washington, in 2022. He received his M.S. degree in electrical engineering from Worcester Polytechnic Institute, Worcester, in 2015, and his B.S. degree in electrical engineering from Peking University, Beijing, in 2013. He is currently an assistant professor of the Department of Electrical and Computer Engineering at the University of North Carolina at Charlotte. His research work focuses on parallel computing, computer architecture, interconnection networks, and machine learning.

Hao Zheng received his Ph.D. degree in computer engineering from George Washington University, Washington, in 2021. He is an assistant professor in the Department of Electrical and Computer Engineering at the University of Central Florida, Orlando. His research interests are in the broad area of computer architecture and parallel computing, with emphasis on interconnection networks, AI chips for emerging applications, and energy-efficient manycore architecture designs.

Ahmed Louri is the David and Marilyn Karlgaard Endowed Chair Professor of the Department of Electrical and Computer Engineering at the George Washington University, Washington, which he joined in August 2015. He is also the director of the High Performance Computing Architectures and Technologies Laboratory. Dr. Louri received his Ph.D. degree in computer engineering from the University of Southern California, Los Angeles in 1988. From 1988 to 2015, he was a professor of Electrical and Computer Engineering at the University of Arizona, Tucson, and during that time, he served six years (2000 to 2006) as the chair of the Computer Engineering Program. From 2010 to 2013, Dr. Louri served as a program director in the National Science Foundation's (NSF) Directorate for Computer and Information Science and Engineering. He directed the core computer architecture program and was on the management team of several cross-cutting programs. Dr. Louri conducts research in the broad area of computer architecture and parallel computing, with emphasis on interconnection networks, optical interconnects for scalable parallel computing systems, reconfigurable computing systems, and power-efficient and reliable Network-on-Chips (NoCs) for multicore architectures. Recently he has been concentrating on: energy-efficient, reliable, and high-performance many-core architectures; accelerator-rich reconfigurable heterogeneous architectures; machine learning techniques for efficient computing, memory, and interconnect systems; emerging interconnect technologies (photonic, wireless, RF, hybrid) for NoCs; future parallel computing models and architectures (including convolutional neural networks, deep neural networks, and approximate computing); and cloud-computing and data centers. He is the recipient of the 2020 IEEE Computer Society Edward J. McCluskey Technical Achievement Award, for pioneering contributions to the solution of on-chip and off-chip communication problems for parallel computing and manycore architectures. Dr. Louri is a fellow of IEEE, and he is currently the Editor-in-Chief of the IEEE Transactions on Computers. More information can be found at https://hpcat.seas.gwu.edu/Director.html.