University of California
Santa Barbara

Intelligent Software in the Era of Deep Learning

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Yuke Wang

Committee in charge:

Professor Yufei Ding, Chair
Professor Timothy Sherwood
Professor Tevfik Bultan
Professor Yuan Xie
Doctor Ang Li

June 2024

The Dissertation of Yuke Wang is approved.

Professor Timothy Sherwood

Professor Tevfik Bultan

Professor Yuan Xie

Doctor Ang Li

Professor Yufei Ding, Committee Chair

June 2024

Intelligent Software in the Era of Deep Learning

by

Yuke Wang

To my family, advisor, and colleagues

Acknowledgements

This dissertation wouldn't be possible without the help from many people. I want to thank my excellent advisor, Professor Yufei Ding, for her help and encouragement. She provided much guidance and advice over the years on identifying the essence of problems and solving technical challenges. She spent numerous hours teaching me how to write papers and prepare slides. She always encourages me to aim high during hard times.

I would like to thank the rest of my thesis committee members – Professor Tim Sherwood, Professor Tevfik Bultan, Professor Yuan Xie, and Doctor Ang Li. Their feedback and advice are essential for completing this dissertation.

I also want to thank Professor Tevfik Bultan, Tim Sherwood, Yuan Xie, Jonathan Balkind, and Xifeng Yan, who taught me a lot about software engineering, and computer architecture, and shared their valuable experience from both academia and industry.

I want to thank Dr. Ang Li and Dr. Tong Geng for their insightful suggestions and discussions for several of our collaborative projects. I am grateful to have Dr. Yuan Xie for his guidance on my first research internship at Alibaba. I would also thank Dr. Mehrzad Samadi for bringing me to the exciting world of the genomic processing pipeline at NVIDIA Parabrick. I want to thank Dr. Michael Garland for helpful discussions at NVIDIA Research. I want to thank Dr. Saeed Maleki for his support and discussion at Microsoft Research. I learned a lot from each of them and appreciate their help during this exciting research and internship journey.

I want to thank my lab mates and friends: Boyuan Feng, Zheng Wang, Hezi Zhang, Anbang Wu, Guyue Huang, Zhaodong Chen, Gushu Li, and Liu Liu. I enjoy every moment that we have worked and played together.

Finally, I want to thank my parents for their unconditioned support. Their kindness and self-motivation set role models for me.

Curriculum Vitæ

Yuke Wang

2121 Henley Hall,
Santa Barbara, CA 93106
Homepage: https://wang-yuke.com

Phone: (+1) 805-259-9421
Email: yuke_wang@cs.ucsb.edu
[Google Scholar][Github][Linkedin]

2018 – Now	**Ph.D. in Computer Science** University of California, Santa Barbara, USA Advisor: Dr. Yufei Ding
2014 – 2018	**B.E. in Software Engineering** University of Electronic Science and Technology of China, China Advisor: Dr. Yu Tang

Employment

Summer 2023	**Research Intern** Microsoft Research, USA. Supervisor: Saeed Maleki
Summer 2022	**Research Intern** NVIDIA Research, USA. Supervisor: Michael Garland
Summer 2021	**High-Performance Engineering Intern** NVIDIA, USA. Supervisor: Mehrzad Samadi
Summer 2020	**Research Intern** Alibaba DAMO Academy, USA. Supervisor: Yuan Xie

Areas of Research

Yuke's research interests include **Deep-Learning (DL) Systems**, and **GPU-based Parallel and Distributed Computing**. His Ph.D. research spans deep neural networks (**DNNs**), graph neural networks (**GNNs**), and deep reinforcement learning (**DRL**) and their system-level optimization and acceleration on GPUs. The ultimate goal of Yuke's research is to facilitate *efficient*, *scalable*, and *secure* deep learning in the future.

Efficient DL: **GNNAdvisor** [OSDI'21], **QGTC** [PPoPP'22], **TC-GNN** [ATC'23].
Scalable DL: **MGG** [OSDI'23], **El-Rec** [SC'22], **RAP** [ASPLOS'24].
Secure DL: **ZENO** [ASPLOS'24], **Faith** [ATC'22], **UAG** [AAAI'21].

Publications

Selected

OSDI'23	Yuke Wang, Boyuan Feng, Zheng Wang, Tong Geng, Ang Li, Kevin Barker, Yufei Ding, "*MGG: Accelerating Graph Neural Networks with Fine-grained intra-kernel Communication-Computation Pipelining on Multi-GPU Platforms*", the USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2023.
USENIX ATC'23	Yuke Wang, Boyuan Feng, Zheng Wang, Guyue Huang, Yufei Ding, "*TC-GNN: Bridging Sparse GNN Computation and Dense Tensor Cores on GPUs*", the USENIX Annual Technical Conference (ATC), 2023.
PPoPP'22	Yuke Wang, Boyuan Feng, Yufei Ding, "*QGTC: Accelerating Quantized Graph Neural Networks via GPU Tensor Core*", the ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP), 2022.
OSDI'21	Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, Yufei Ding, "*GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs*", the USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2021 .
SC'21	Boyuan Feng*, Yuke Wang*, Tong Geng, Ang Li, Yufei Ding, *"APNN-TC: Accelerating Arbitrary-Precision Neural Networks on Tensor Cores"*, the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC), 2021. (* : **equal contribution**)
CCGrid'21	Yuke Wang, Boyuan Feng, Gushu Li, Georgios Tzimpragos, Lei Deng, Yuan Xie, Yufei Ding, "*TiAcc: Triangle-inequality based Hardware Accelerator for K-means on FPGAs*", the IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), 2021.
IPDPS'21	Yuke Wang, Boyuan Feng, Yufei Ding, "*DSXplore: Optimizing Convolutional Neural Networks via Sliding-Channel Convolutions*", the IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2021.

TCAD'21 Yuke Wang, Boyuan Feng, Gushu Li, Lei Deng, Yuan Xie, Yufei Ding, "*STPAcc: Structural TI-based Pruning for Accelerating Distance-related Algorithms on CPU-FPGA Platforms*", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

CIKM'21 Yuke Wang, Boyuan Feng, Xueqiao Peng, Yufei Ding, "*An Efficient Quantitative Approach for Optimizing Convolutional Neural Network*", The ACM Conference on Information and Knowledge Management (CIKM), 2021 .

ICTAI'20 Boyuan Feng*, Yuke Wang*, Xu Li, Shu Yang, Xueqiao Peng, Yufei Ding, "*SGQuant: Squeezing the Last Bit on Graph Neural Networks with Specialized Quantization*", the IEEE International Conference on Tools with Artificial Intelligence (ICTAI), 2020. (*: **equal contribution**).

Other

ASPLOS'24 Zheng Wang, Yuke Wang, Jiaqi Deng, Da Zheng, Ang Li, Yufei Ding. "*RAP: Resource-aware Automated GPU Sharing for Multi-GPU Recommendation Model Training and Input Preprocessing.*", ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2024.

ASPLOS'24 Boyuan Feng, Zheng Wang, Yuke Wang, Shu Yang, Yufei Ding. "*ZENO: A Type-based Optimization Framework for Zero Knowledge Neural Network Inference*", ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2024.

ISCA'23 Hezi Zhang, Anbang Wu, Yuke Wang, Gushu Li, Hassan Shapourian, Alireza Shabani, Yufei Ding"*A Compilation Framework for Photonic One-Way Quantum Computation.* ", International Symposium on Computer Architecture, 2023.

MLSys'23 Guyue Huang, Yang Bai, Liu Liu, Yuke Wang, Bei Yu, Yufei Ding, Yuan Xie. "*ALCOP: Automatic Load-Compute Pipelining in Deep Learning Compiler for AI-GPUs.*", Sixth Conference on Machine Learning and Systems, 2023.

USENIX ATC'22 Boyuan Feng, Tianqi Tang, Yuke Wang, Zhaodong Chen, Zheng Wang, Shu Yang, Yuan Xie, Yufei Ding. "*Faith: An Efficient Framework for Transformer Verification on GPUs*", the USENIX Annual Technical Conference (ATC), 2021.

SC'22	Zheng Wang, Yuke Wang, Boyuan Feng, Dheevatsa Mudigere, Bharath Muthiah, Yufei Ding. "*EL-Rec: Efficient Large-scale Recommendation Model Training via Tensor-Train Embedding Table*", the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC), 2022.
USENIX ATC'21	Boyuan Feng, Yuke Wang, Gushu Li, Yuan Xie, Yufei Ding, "*Palleon: A Runtime System for Efficient Video Processing toward Dynamic Class Skew*", the USENIX Annual Technical Conference (ATC), 2021.
PPoPP'21	Boyuan Feng, Yuke Wang, Guoyang Chen, Weifeng Zhang, Yuan Xie, Yufei Ding, "*EGEMM-TC: Accelerating Scientific Computing on Tensor Cores with Extended Precision*", the ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP), 2020.
USENIX ATC'21	Boyuan Feng, Yuke Wang, Gushu Li, Yuan Xie, Yufei Ding, "*Palleon: A Runtime System for Efficient Video Processing toward Dynamic Class Skew*", the USENIX Annual Technical Conference (ATC), 2021.
AAAI'21	Boyuan Feng, Yuke Wang, Yufei Ding, "*UAG: Uncertainty-aware Attention Graph Neural Network for Defending Adversarial Attacks*", the Thirty-Fifth AAAI Conference on Artificial Intelligence (AAAI), 2021.
ICASSP'21	Boyuan Feng, Yuke Wang, Yufei Ding, "*SAGA: Sparse Adversarial Attack on EEG-based Brain Computer Interface*", IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2021.
ICML'20	Liu Liu, Lei Deng, Zhaodong Chen, Yuke Wang, Shuangchen Li, Jingwei Zhang, Yihua Yang, Zhenyu Gu, Xing Hu, Yufei Ding, Yuan Xie, "*Boosting Deep Neural Network Efficiency with Dual-Module Inference*", the International Conference on Machine Learning (ICML), 2020.

Professional Service

[03/2023]	Journal of Supercomputing Paper Reviewer
[02/2023]	IEEE Transactions on Neural Networks and Learning Systems Reviewer
[02/2023]	PLDI'23 Artifact Evaluation Committee
[11/2022]	ECOOP'23 Artifact Evaluation Committee

[11/2022]	PPoPP'23 Artifact Evaluation Committee
[10/2022]	CGO'23 Artifact Evaluation Committee
[10/2022]	MLSys'23 External Review Committee
[10/2022]	IEEE Transactions on Computers Reviewer
[09/2022]	USENIX Security'23 Artifact Evaluation Committee
[09/2022]	ASPLOS'23 Artifact Evaluation Committee
[08/2022]	POPL'23 Artifact Evaluation Committee
[08/2022]	ACM Computing Survey Reviewer
[07/2022]	MICRO'22 Artifact Evaluation Committee
[06/2022]	SIGCOMM'22 Artifact Evaluation Committee
[04/2022]	ISSTA'22 Artifact Evaluation Committee
[04/2022]	OSDI'22 Artifact Evaluation Committee
[04/2022]	USENIX ATC'22 Artifact Evaluation Committee
[01/2022]	PLDI'22 Artifact Evaluation Committee
[01/2022]	EuroSys'22 Artifact Evaluation Committee
[11/2021]	ASPLOS'22 Artifact Evaluation Committee
[10/2021]	SOSP'21 Graduate Student Mentor
[10/2021]	Artificial Intelligence Review Paper Reviewer
[10/2021]	Journal of Supercomputing Paper Reviewer
[08/2021]	SOSP'21 Artifact Evaluation Committee
[07/2021]	MICRO'21 Artifact Evaluation Committee
[07/2021]	SC'21 Artifact Evaluation Committee
[10/2020]	AAAI'21 Paper Reviewer Committee

Awards

[05/2023]	Graduate Division Dissertation Fellowship of UCSB
[07/2022]	2022 USENIX Student Travel Grant for OSDI'22/USENIX ATC'22
[06/2022]	2021-2022 Graduate Student External Award in CS Department of UCSB
[11/2021]	2022-2023 NVIDIA Graduate Fellowship (Top 10 out of global applicants)
[10/2021]	2021 ACM PACT Student Research Competition (First Prize Winner)
[09/2021]	2021 SIGIR Student Travel Grant
[06/2021]	2020-2021 Outstanding Publication Award in CS Department of UCSB
[06/2020]	2020 Summer GSR recipient in CS Department of UCSB
[06/2019]	2019 Summer GSR recipient in CS Department of UCSB
[10/2017]	Outstanding Graduates Award of UESTC
[10/2017]	First-class People's Scholarship (2/20 in the Elite Program)
[04/2017]	Interdisciplinary Contest In Modeling (ICM) [Honorable Mention]
[04/2017]	Suzhou Industrial Zone Scholarship (2/20 in the Elite Program)
[10/2016]	International Software Testing Qualifications Board (Certified Tester)
[04/2016]	First-class People's Scholarship (4/116)

Teaching Experience

[09/2019]	Teaching Assistant of CS160 (Translation of Programming Languages)
[07/2019]	Teaching Assistant of CS8 (Python Programming Language)
[01/2019]	Teaching Assistant of CS16 (C++ Programming Language)

Opensource Project

MGG	Accelerating Graph Neural Networks with Fine-grained intra-kernel Communication-Computation Pipelining on Multi-GPU Platforms. `https://github.com/YukeWang96/MGG_OSDI23.git`
TC-GNN	Bridging Sparse GNN Computation and Dense Tensor Cores on GPUs. `https://github.com/YukeWang96/TC-GNN_ATC23.git`
QGTC	Accelerating Quantized GNN via GPU Tensor Core. `https://github.com/YukeWang96/QGTC_PPoPP22.git`
GNNAdvisor	An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs. `https://github.com/YukeWang96/GNNAdvisor_OSDI21.git`
APNN-TC	Arbitrary Precision Neural Networks on Ampere GPU Tensor Cores. `https://github.com/YukeWang96/APNN-TC_SC21.git`
DSXplore	Convolutional Neural Networks via Sliding-Channel Convolutions. `https://github.com/YukeWang96/DSXplore_IPDPS21.git`

Student Mentoring

Xiaoya Zhou	Accelerating the Large Language Model through Systemic Optimizations. (Undergrad at UCSB) [04/2023-Now]
Anshuman Dash	Automating the Optimization Flow of Graph Neural Networks via Dynamic Compilation. (Undergrad at UCSB) [09/2022-12/2022]
Qijun Zhang	Optimizing the Computation Efficiency of the Large-Scale Deep Learning via Holistic System Design. (Now as Ph.D. at HKUST) [06/2022-09/2022]
Xueqiao Peng	Optimizing Convolutional Neural Network with Quantitative Approach. (published at CIKM'21) (Now as Ph.D. at Ohio State University) [06/2020-09/2020]

Talks

[10/2023]	Guest Lecture at the University of Rochester ECE403, hosted by Tong Geng.
[07/2023]	Invited Talk on Graph Learning Acceleration at CUHK and CityUHK, hosted by Hong Xu and Qiang Su.
[11/2022]	Gesture Lecture at NCSU CS591, hosted by Xipeng Shen.
[11/2022]	Technical Talk at AWS AI at Santa Clara, hosted by Yida Wang.
[10/2022]	SAMPLE Talk at the University of Washington, hosted by Zihao Ye.
[04/2022]	NVIDIA GTC'22.

References

Dr. Yufei Ding
Associate Professor
UC at San Diego
yufeiding@ucsd.edu

Dr. Timothy Sherwood
Professor
UC at Santa Barbara
sherwood@cs.ucsb.edu

Dr. Tevfik Bultan
Professor
UC at Santa Barbara
bultan@cs.ucsb.edu

Dr. Michael Garland
Senior Research Director
NVIDIA Research
mgarland@nvidia.com

Dr. Mehrzad Samadi
Senior Engineering Manager
NVIDIA
msamadi@nvidia.com

Dr. Ang Li
Senior Computer Scientist
Pacific Northwest National Laboratory
ang.li@pnnl.gov

Intelligent Software in the Era of Deep Learning

By Yuke Wang

With the end of Moore's Law and the rise of compute- and data-intensive deep learning (DL) applications, the focus on arduous new processor design has shifted towards a more effective and agile approach: ***Intelligent Software*** to maximize the performance gains of DL hardware like GPUs. There are several highlights of such intelligent software design. First, it would maximize the execution *efficiency* of existing and emerging DL algorithms on powerful platforms like GPUs. Second, it would promote the *adaptiveness* of systems to handle a diverse range of inputs. Third, it would maintain sufficient *portability* and *scalability* across a diverse range of platforms, such as mobile devices and high-performance clusters.

In this thesis, I will first highlight the importance of software innovation to bridge the gap between the increasingly diverse DL applications and the existing powerful DL hardware platforms. The second part of my thesis will recap my research work on DL system software innovation, focusing on 1) *Precision Mismatch* between DL applications and high-performance GPU units like Tensor Cores (e.g., QGTC [PPoPP '22] and APNN-TC [SC '21]), to improve the efficiency of quantized deep learning on powerful GPU platforms, and 2) *Computing Pattern Mismatch* between the sparse and irregular DL applications, such as Graph Neural Networks, and the dense and regular tailored GPU computing paradigm (e.g., GNNAdvisor [OSDI '21] and MGG [OSDI '23]), to highlight system adaptability and scalability. Finally, I will conclude this thesis with my vision and future work for building efficient, scalable, and secure DL systems.

Contents

Chapter 1

Introduction

1.1 Motivation

With the prosperity of DL applications, the performance of DL has raised significant interest. As shown in Figure 1.1, there has been a giant leap in terms of training demands from the DL algorithms, as highlighted by those popular GPT models for generative AI. In order to support these intensive AI computations, on the hardware side, CPUs have achieved minor throughput improvements (1.5×) over recent years. The promising DL accelerators like GPUs have achieved an evident 15× throughput improvement. Despite such improvements in hardware like GPUs, there is still a large gap between the DL algorithm and hardware.

To bridge this gap, my research analysis reveals two fundamental insights. The first insight is the ***precision mismatch***. As shown in Figure 1.2, on one side, we have DL and scientific applications that require high data and compute precision, while on the other side, we have hardware units (e.g., GPU Tensor Cores) that have lower data/compute precision but high performance. Therefore, these high-precision applications cannot directly benefit from the high-performance units on GPU hardware. In addition to

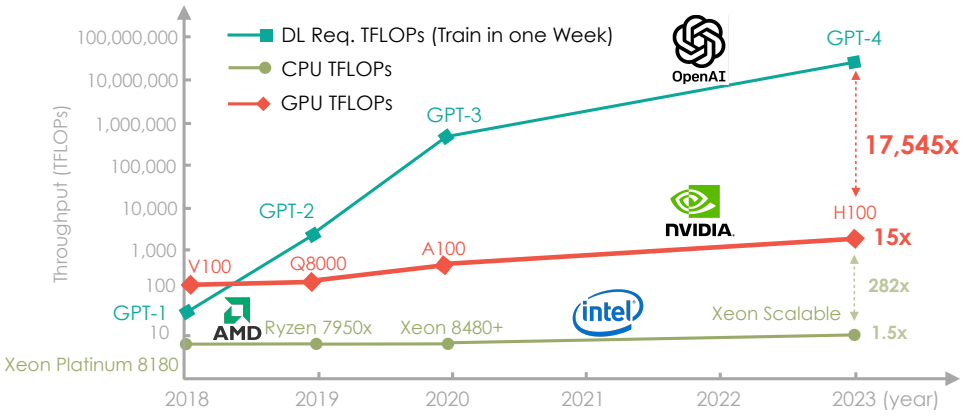


Figure 1.1: The trend of deep learning model and hardware compute capability.

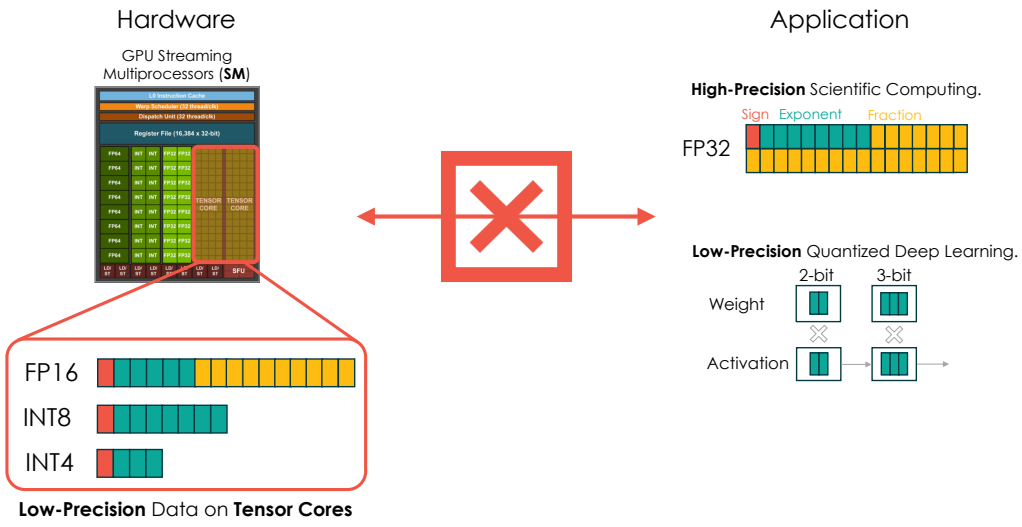


Figure 1.2: The illustration of Precision Mismatch in Deep Learning.

these high-precision scientific applications, there are also diverse quantized low-precision deep learning applications. The precision mismatch could still occur because different DL applications have varying precision requirements (e.g., one layer might require 2-bit precision while another layer needs 3-bit precision for weights and activations), while hardware units can only offer limited precision options (e.g., 4-bit or 8-bit precision).

The *second* insight is the ***compute pattern mismatch***. As shown in Figure 1.3, on one side, existing hardware accelerator devices like GPUs are mostly tailored for

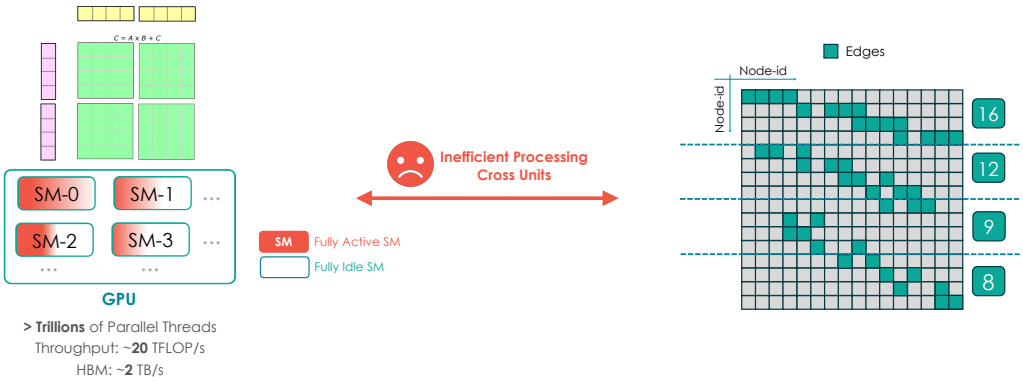


Figure 1.3: The illustration of Compute Pattern Mismatch in Deep Learning.

dense computations because those computations are usually easier to optimize. On the other side, DL applications are now becoming more diverse and could come with Sparse and Irregular Inputs (e.g., Graphs) that cannot well utilize Dense and Regular GPUs. One typical example is the sparse matrix multiplication in GNN computation, where the adjacent matrix of a graph consists of highly scattered edges among different node pairs. There will be a natural imbalance of edges when we distribute them among several GPU SMs. Such irregularity of workloads induces GPU hardware underutilization and sub-optimal performance.

With these two major insights, we would like to highlight my Ph.D. research: ***Promoting ML Hardware with Intelligent Software***. Specifically, my research is composed of three major dimensions, as shown in Figure 1.4. The first dimension defines the precision of data from diverse DL applications. The second dimension defines the sparsity of DL application inputs. With these two dimensions defined, my research work strengthens the individual capability of DL hardware with improved efficiency and adaptiveness. Furthermore, to handle the inputs at scale, we introduce the third dimension for platforms, ranging from single-device to multiple-device collaboration. With the third dimension, we empower the holistic system design with scalability.

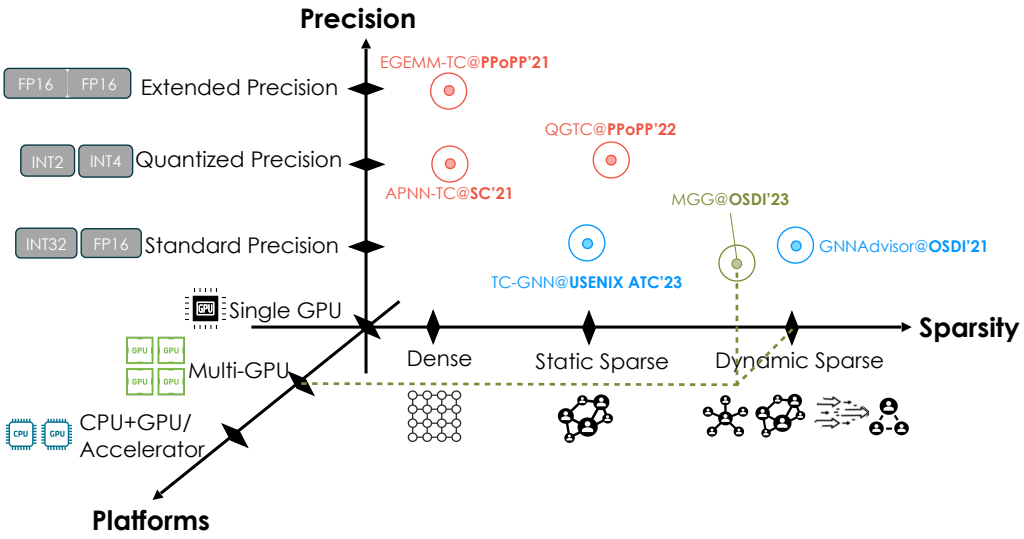


Figure 1.4: The overview of my PhD research.

1.2 Overview of My Dissertation

Prior Research Projects

Today's GPU-based system is the de facto standard to support DNN computation with intensive computing and memory demands. However, due to the increasing complexity and diversity of DL algorithms (e.g., DNNs and GNNs), existing DL computing frameworks may suffer from sub-optimal performance and GPU underutilization, which is due to the fact that tailored system designs and optimizations could hardly handle the unique characteristics of diverse DL workloads. My research [1, 2, 3, 4, 5, 6, 7, 8] tackles this challenge systematically by introducing a *holistic runtime system design with input, algorithm, and hardware awareness*. Instead of resorting to fixed hard-coded computing libraries widely adopted in the conventional DL frameworks (e.g., PyTorch and Tensorflow), we capture essential input features and capitalize on their performance-related benefits by developing customized kernel runtime and optimization heuristics. With such new design paradigms, we enjoy the benefits of both inputs and hardware and

accommodate diverse DL workloads and hardware.

Closing the precision gap with novel algorithm-system co-design. Over the years, GPUs have evolved to incorporate many more new hardware features, and one of the most successful ones is the Tensor Cores (TCs). Meanwhile on the algorithm side, accelerating neural networks with quantization has been widely studied. Unfortunately, prior algorithmic efforts with diverse precision types (e.g., 1-bit weights and 2-bit activations) are usually restricted by limited precision support on GPUs (e.g., 1-bit and 4-bit). To bridge such a gap, my work, **QGTC** [4] (PPoPP '22), introduces the first TC-based computing framework for quantized mini-batch GNNs, to support any-bit-width computation on GPUs. QGTC features a novel quantized low-bit arithmetic design based on low-bit data representation and bit-decomposed computation. QGTC incorporates a novel TC-tailored CUDA kernel design with 3D-stacked bit compression, zero-tile jumping, and non-zero tile reuse to improve the performance systematically. QGTC also integrates an effective bandwidth-optimized subgraph packing strategy to maximize the transferring efficiency between the CPU host and GPU device. Extensive experiments demonstrate that QGTC achieves an average 3.17× speedup compared with the state-of-the-art Deep Graph Library (DGL) framework across diverse settings.

Closing the computing pattern gap with Intelligent Runtime System Design. As the emerging trend of graph-based deep learning, Graph Neural Networks (GNNs) excel in their capability to generate high-quality node feature vectors (embeddings). However, the existing one-size-fits-all GNN implementations are insufficient to catch up with the evolving GNN architectures, the ever-increasing graph sizes, and the diverse node embedding dimensionalities. My research **GNNAdvisor** [3] (OSDI '21, a flagship system venue), introduces an intelligent runtime system for boosting the GNN's performance on a GPU-based platform. GNNAdvisor combines the input-level (e.g., graph and node embedding) and system-level (e.g., GPU kernel design) optimizations to

improve the GNN performance comprehensively. GNNAdvisor largely reduces the execution latency by 2×-3× compared with the state-of-the-art GNN frameworks on GPUs, which can maximize energy efficiency and performance-per-dollar gains when deploying on cloud-based platforms. This work has been opensourced for benefiting future exploration and has inspired many new designs from industry (e.g., BGL from Bytedance, SparseTIR from TVM) and academia (e.g., Marius++ from UW-Madison, Point-X from UMich) in a similar domain since its publication.

To leverage the high-performance GPU Tensor Core units for sparse computation, my work, **TC-GNN** [8] (USENIX ATC '23), bridges the Sparse GNN Computation and Dense TCs on GPUs Specifically, we introduce a sparse graph translation technique, which makes the sparse and irregular GNN input graphs easily fit the dense computing of TCs for acceleration. We build a TC-tailored algorithm and GPU kernel design for CUDA core and TC collaboration on GPUs to handle different sparse GNN computations.

The increasing size of input graphs for graph neural networks (GNNs) highlights the demand for using multi-GPU platforms. My research introduces **MGG** [9] (OSDI '23), a novel system design to accelerate full-graph GNNs on multi-GPU platforms. The core of MGG is its novel fine-grained dynamic software pipeline to facilitate fine-grained computation-communication overlapping within a GPU kernel. Specifically, MGG introduces GNN-tailored pipeline construction and GPU-aware pipeline mapping to facilitate workload balancing and operation overlapping. MGG also incorporates an intelligent runtime design with analytical modeling and optimization heuristics to dynamically improve the GNN execution performance. Extensive evaluation reveals that MGG outperforms state-of-the-art full-graph GNN systems across various settings: on average 4.41×, 4.81×, and 10.83× faster than DGL, MGG-UVM, and ROC frameworks, respectively.

Besides the graph neural networks, sparsity also exists in conventional neural networks. I introduce, **DSXplore** [2] (IPDPS '21), the first optimized design for exploring

deep separable convolutions on CNNs. Specifically, at the algorithm level, DSXplore incorporates a novel factorized kernel – sliding-channel convolution (SCC), featuring input-channel overlapping to balance the accuracy performance and the reduction of computation and memory cost. SCC also offers enormous space for design exploration by introducing adjustable kernel parameters. Further, at the implementation level, we carry out an optimized GPU implementation tailored for SCC by leveraging several key techniques, such as the input-centric backward design and channel-cyclic optimization. Intensive experiments on different datasets across mainstream CNNs show the advantages of DSXplore in balancing accuracy and computation/parameter reduction over the standard convolution and the existing DSCs.

To further reduce model complexity, my work introduces 3D-Receptive Field (**3DRF**) [6] (CIKM '21), an explainable and easy-to-compute metric, to estimate the quality of a CNN architecture and guide the search process of designs. To validate the effectiveness of 3DRF, we build a static optimizer to improve the CNN architectures at both the stage level and the kernel level. Our optimizer not only provides a clear and reproducible procedure but also mitigates unnecessary training efforts in the architecture search process. Extensive experiments and studies show that the models generated by our optimizer achieve up to 5.47% accuracy improvement and up to 65.38% parameters deduction, compared with state-of-the-art CNN model structures like MobileNet and ResNet.

Chapter 2

QGTC: Accelerating Quantized Graph Neural Networks via GPU Tensor Core

Over the most recent years, quantized graph neural network (QGNN) attracts lots of research and industry attention due to its high robustness and low computation and memory overhead. Unfortunately, the performance gains of QGNN have never been realized on modern GPU platforms. To this end, we propose the first Tensor Core (TC) based computing framework, **QGTC** [1], to support any-bitwidth computation for QGNNs on GPUs. We introduce a novel quantized low-bit arithmetic design based on the low-bit data representation and bit-decomposed computation. We craft a novel TC-tailored CUDA kernel design by incorporating 3D-stacked bit compression, zero-tile jumping, and non-zero tile reuse techniques to improve the performance systematically. We incorporate an effective bandwidth-optimized subgraph packing strategy to maximize the transferring efficiency between CPU host and GPU device. We integrate QGTC with Pytorch for better programmability and extensibility. Extensive experiments demonstrate

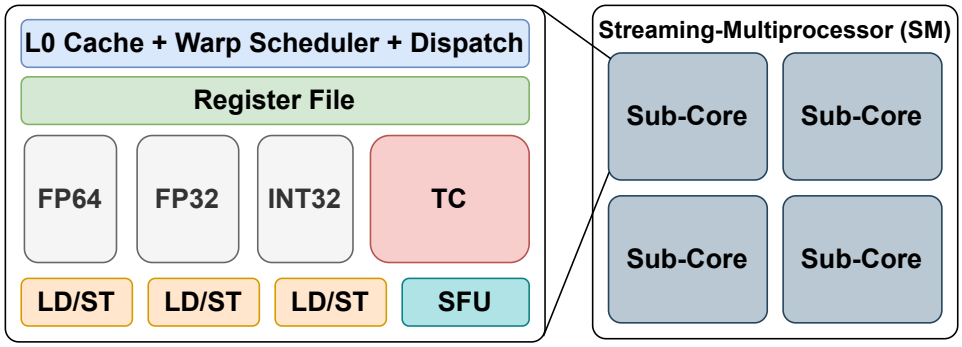


Figure 2.1: GPU Streaming-Multiprocessor (SM) with TC Design. Note that FP64, FP32, INT32, LD/ST, and SFU are double-precision, single-precision, integer, load/store, and special function units, respectively.

that QGTC can achieve evident inference speedup (on average 2.7×) compared with the state-of-the-art DGL framework across diverse settings.

2.1 Introduction

With the popularity surge of the graph neural networks (GNNs) [10, 11, 12], research around the full-precision GNNs has been widely studied in terms of its algorithms [10, 13] and execution performance [14, 15, 16] over traditional graph analytical methods, such as Random Walk [17]. On the other side, quantized GNN [18, 19] (QGNN) recently attracts lots of attention thanks to its negligible accuracy loss, resilience towards malicious attacks, and significantly lower computations and memory overhead. We summarize several key features of GNNs that make them intrinsically suitable for quantization. ***First***, the adjacent matrix of GNNs is naturally well-suited for quantization, since we only need to use 0/1 to indicate the existence of edge connections. Thus, using low bits for such information can save both memory and computation. ***Second***, the quantization of weight and node embedding can also be beneficial. Because the tiny precision loss in quantization can largely be offset by the node information fusion through the iterative

neighbor aggregation process of GNNs. The quantization of floating-point numbers can absorb input perturbations from adversarial attacks.

Despite such great theoretical success of QGNN, the realization of such benefits on high-performance GPUs is still facing tremendous challenges. Existing GPU-based GNN frameworks [16, 15, 3] are designed and tailored for GPU CUDA cores, which are intrinsically bounded by its peak throughput performance and can only handle the byte-based data types (*e.g.*, `int32`). Although quantized computation can be achieved via pure algorithmic emulation, the actual bit-level performance gains could hardly be harvested, since all underlying arithmetic operations still have to rely on those well-defined data types from CUDA/C++ libraries.

To tackle these challenges, we decide to move forward with the recent GPU hardware feature – **Tensor Core** (TC). The modern NVIDIA GPU with TC design is illustrated in Figure 2.1. TC provides the native support of bit-level operations (`XOR`, `AND`), which could be the major ingredient for quantized computation. Besides, TC can easily beat CUDA core with a significantly higher throughput performance (more than 10×) on conventional NN operations (*e.g.*, linear transformation and convolution). This demonstrates the potential of using TC in accelerating QGNNs. However, directly using TC for QGNN computation is encountering several challenges. ***First***, the current TC can only support limited choices of bitwidth (*e.g.*, 1-bit and 4-bit), which may not be able to meet the demands of users for any-bitwidth (*e.g.*, 2-bit) computation. ***Second***, TC initially tailored for GEMM computation may not directly fit the context of sparse GNN computation. A huge amount of computation and memory access efforts would be wasted on those non-existed edges. This is because the hard constraint of TC input matrix tile-size (*e.g.*, 8×128 for 1-bit GEMM) has to be satisfied, which may require excessive zero paddings. ***Third***, the low-bit computation would cause the compatibility issue, since the existing deep-learning frameworks [20, 21] cannot directly operate on the low-bit data.

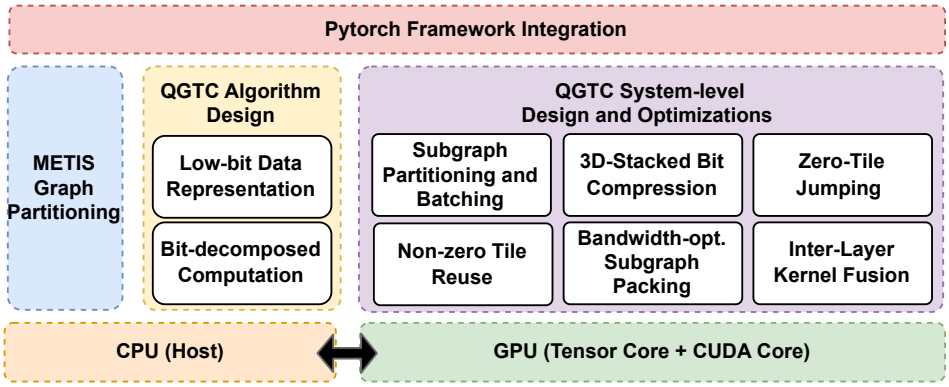


Figure 2.2: QGTC Overview.

Therefore, we remark there are several aspects to be considered in order to use TC for QGNNs: 1) ***Hardware-level Support***. This inspires us to explore the high-performance GPU hardware features that can efficiently support the QGNN computation. Even though it is hard to find such a GPU hardware feature that can directly support any-bitwidth QGNN, some indirect hardware features would potentially be helpful. For example, NVIDIA introduced the 1-bit TC-based GEMM on Turing Architecture, which essentially can be used to composite any-bitwidth GEMM. 2) ***Software-level Optimizations***. This motivates us to optimize the kernel computation according to the characters of QGNN. GNN computation is featured with a highly sparse and irregular scheme. It is intrinsically not favorable for the dense GPU computation flow tailored for the traditional NN operators. Thus, how to handle such input-level irregularity from the computation and memory perspectives is essential to the performance of QGNN. For example, subgraph partitioning [22] based mini-batch GNN computation has been used to increase the computation efficiency without compromising model accuracy performance. 3) ***Framework-level Integration***. This encourages us to bridge the gap between quantized low-bit implementations and deep-learning frameworks built for full-precision computation. Therefore, our whole system-level design can be seamlessly integrated with the state-of-the-art mainstream NN frameworks to benefit the execution

performance and the developing productivity.

To this end, we introduce QGTC[2], the first framework (Figure 2.2) to support any-bitwidth QGNN on GPU TC.

At the input level, we incorporate the METIS [22] graph partitioning to generate a set of dense subgraphs from the highly irregular and sparse input graphs. The insight here is that nodes in real-world graphs are likely to form clusters, and such information can be used to benefit the efficiency of GNN computing and model algorithmic performance.

At the algorithm level, we leverage the insight that any-bitwidth QGNN computation can always be decomposed into the 1-bit computation. Each bit in the output can be generated by different combinations of bits from the input. Thus, we use quantized low-bit data representation and bit-decomposed computation base on the "atomic" 1-bit type.

At the GPU kernel level, we craft a low-bit computation design tailored for QGNN computation on batched dense subgraphs. We address the key performance bottleneck of the low-bit GNN computing from the memory and computing perspectives. Specifically, we use only 1-bit binarized representation for the subgraph adjacent matrix, which is memory efficient for representing the presence/absence of edge connections between nodes. Besides, we use a 3D-stacked bit-compression technique for maintaining quantized low-bit node embedding features and weights. In addition, we fully exploit the intra-subgraph sparsity through zero-tile skipping and non-zero tile reuse, which can further avoid unnecessary computations and improve the data locality.

At the framework level, we integrate QGTC with the state-of-the-art Tensor-based PyTorch framework. We introduce the new notion of bit-Tensor data type and bit-Tensor computation and warp them up as a new set of PyTorch API extensions. End-users can directly interact with the QGTC PyTorch APIs to access all functionalities. This largely

[2]QGTC is open-sourced at `github.com/YukeWang96/PPoPP22_QGTC.git`

improves the programmability and extensibility.

Overall, we summarize our key contributions as:

- We propose a novel 1-bit composition technique for any-bitwidth arithmetic design, which can support QGNN with diverse precision demands.
- We introduce a highly efficient implementation of QGNN built on top of the GPU Tensor Core by applying a series of computation optimizations (*e.g.*, subgraph partitioning and batching, and zero-tile jumping) and memory optimizations. (*e.g.*, 3D-stacked bit-compression and non-zero tile reuse).
- We integrate QGTC with PyTorch by introducing bit-Tensor data type and bit-Tensor computation for better programmability and extensibility.
- Extensive experiments demonstrate the advantages of QGTC in terms of better performance compared with the state-of-the-art DGL framework on mainstream GNN models across various datasets.

2.2 Background and Related Work

In this section, we will introduce the background of GNNs, the quantization of GNNs, and basics of GPU Tensor Core.

2.2.1 Graph Neural Networks

Graph neural network (GNN) is an effective tool for graph-based machine learning. The detailed computing flow of GNNs is illustrated in Figure 3.2. GNNs basically compute the node feature vector (embedding) for node v at layer $k+1$ based on the embedding

information at layer k ($k \geq 0$), as shown in Equation 4.1,

$$\begin{aligned} a_v^{(k+1)} &= \textbf{\textit{Aggregate}}^{(k+1)}(h_u^{(k)} | u \in \mathbf{N}(v) \cup h_v^{(k)}) \\ h_v^{(k+1)} &= \textbf{\textit{Update}}^{(k+1)}(a_v^{(k+1)}) \end{aligned} \tag{2.1}$$

where $h_v^{(k)}$ is the embedding vector for node v at layer k; $a_v^{(k+1)}$ is the aggregation results through collecting neighbors' information (*e.g.*, node embeddings); $\mathbf{N}(v)$ is the neighbor set of node v. The aggregation method and the order of aggregation and update could vary across different GNNs. Some methods [10, 12] just rely on the neighboring nodes while others [11] also leverage the edge properties that are computed by applying vector dot-product between source and destination node embeddings. The update function is generally composed of standard NN operations, such as a single fully connected layer or a multi-layer perceptron (MLP) in the form of $w \cdot a_v^{(k+1)} + b$, where w and b are the weight and bias parameter, respectively. The common choices for node embedding dimensions are 16, 64, and 128, and the embedding dimension may change across different layers.

The most recent advancement of GNN is its batched computation [23], which has also been adopted by many state-of-the-art GNN computing frameworks [15, 16] for large graphs that cannot be easily fit into the GPU/CPU memory for computation directly. Batched GNN computation has been highlighted with good accuracy and runtime performance [23] in comparison with full-graph computation. Batched GNN computation takes several steps. ***First***, it decomposed the input graphs by employing the state-of-the-art graph partitioning toolset, such as METIS [22], which can minimize the graph structural information loss meanwhile maximizing the number of edge connections within each subgraph (*i.e.*, improving the subgraph modularity). ***Second***, it feeds the small subgraphs into the GNN models for computation, which will generate the node feature vector for each subgraph. Third, the generated node embeddings can be used

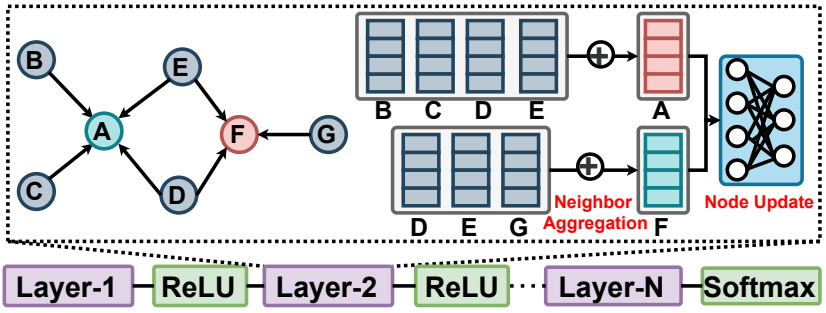


Figure 2.3: GNN General Computation Flow.

in multiple downstream tasks, such as node/graph classification, link prediction, and community detection [24, 25, 17, 26].

2.2.2 Quantization of GNNs

Besides the research efforts on full-precision GNNs, the recent focus also shifted towards the quantized GNNs. For example, Boyuan et al. [18] propose the first framework for running quantized GNNs, and several types of quantization schemes can be applied on GNNs (*e.g.*, the quantization based on the GNN layer, node degrees, and the edge weights). And their experimental results also demonstrate the effectiveness of the GNN quantization in terms of memory saving and model accuracy. Shyam et al. [19] introduce an architecturally agnostic and stable method, Degree Quant, to improve performance over existing quantization-aware training baselines commonly used on other architectures (*e.g.*, CNNs). They achieve up to 4.7× speedups on CPU when using `int8` compared with `float`. Compared with the full-precision GNNs, low-bit GNNs bring the benefit of model robustness towards the adversarial attacks and the low computation and memory overheads. However, work from [18] only showcases the theoretical memory and computation benefits via software-level quantization simulation, where its underlying computation is still carried out in 32-bit full-precision `float`. Work from [19] only demonstrates such gains on CPUs, which has limited applicability in the real-world GNN computation set-

tings. This encourages us to harvest its real performance benefits on the modern widely used GPU platforms.

2.2.3 Tensor Core on GPUs

The recent advancement of GPU hardware technology has pushed computing power to a new level. Among those innovations, the most significant one is the Tensor Core (TC) on NVIDIA GPU. Different from scalar-scalar computation on CUDA Cores, TC provides a matrix-matrix compute primitive, which can deliver more than 10× higher computation throughput. The initial version of TC is designed for handling the GEMM with half-precision input and full-precision output. More variants (*e.g.*, `int8`, `int4`, and `int1` inputs with 32-bit unsigned integer (`uint32`) output) have been introduced since the recent CUDA release (11.0) and newer GPU microarchitectures (*e.g.*, Turing and Ampere).

In particular, TC supports the compute primitive of $\mathbf{D} = \mathbf{A} \times \mathbf{B} + \mathbf{C}$, where matrix tile $\mathbf{A}$ and $\mathbf{B}$ are required to be a certain type of precision (*e.g.*, 1-bit), while matrix tile $\mathbf{C}$ and $\mathbf{D}$ use `uint32`. Depending on the input data precision and the version of GPU microarchitecture, the matrix tile size of $\mathbf{A}(M \times K)$, $\mathbf{B}(K \times N)$, and $\mathbf{C}(M \times N)$ may have different choices. For example, 1-bit TC computing requires $M = N = 8$ and $K = 128$. Different from the CUDA Cores which requires users to define the execution flow of each thread (*i.e.*, work of individual threads). TC requires the collaboration of a warp of threads (32 threads) (*i.e.*, work of individual warps). This can be reflected in two ways. ***First***, before calling TC for computation, all registers of a warp of threads need to collaboratively store the matrix tile into a new memory hierarchy (called *Fragment* [27]), which allows data sharing across registers. This intra-warp sharing provides opportunities for fragment-based memory optimizations. ***Second***, during the computation, these

Listing 2.1: Basic WMMA APIs for TCU in CUDA C.

```
// define the register fragment for matrix A (1-bit).
wmma::fragment<matrix_a, M, N, K, b1, row_major> a_frag;
// load a tile of matrix A to register fragment.
wmma::load_matrix_sync(a_frag, A, M);
// matrix-matrix multiplication (1-bit x 1-bit -> 32-bit)
wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);
// store the C matrix tile from register to matrix C.
wmma::store_matrix_sync(C, c_frag, N, mem_row_major);
```

loaded matrix fragments will be taken as the TC input to generate the output fragment, which also consists of the registers from each thread in a warp. Data movements among these registers are also managed by a warp of threads collaboratively.

Prior research efforts have been devoted to accelerating high-performance computing workloads with TC. Ahmad et al. [28] process the batched small-size GEMM on TC for acceleration. Ang and Simon [29] leverage 1-bit GEMM capability on Turing GPU TC for accelerating binary neural network inference. Dakkak et al. [30] accelerates the half-precision scan on TC by transforming the scan to a GEMM. Boyuan et al. [31] introduce GEMM-based scientific computing on TC with extended precision. QGTC enlarges the application range of TC by accelerating GNNs for any-bitwidth quantized GNN computation, which is not directly covered by any existing research, any release of cuBLAS [32], or CUTLASS [33] library, and GPU TC hardware.

TC can be used in several ways. The simplest one is to call cuBLAS [32] `cublasSgemmEX` API. However, cuBLAS API only supports computation on the most common fixed bit-width on TC, such as 8-bit, half-precision (16-bit), thus, it cannot support any bitwidth precision directly. The second way is to call the Warp Matrix Multiply-Accumulate (WMMA) (`nvcuda::wmma`) API [34] in CUDA C++ to operate TC directly. There are basically four types of operations (Listing 4.1). In this project, we follow the second way for more low-level implementation customization for batched GNN computation.

Because it can offer more design/implementation flexibility for compositing arbitrary-bit computation and ease the optimization (*e.g.*, data loading and reuse) for batched GNN-specific workloads at the GPU kernel.

2.3 QGTC Algorithm Design

In this section, we first introduce the basics of low-bit computation. Then we will discuss our TC-tailored algorithm design for quantized GNN.

2.3.1 1-bit Composition for Quantized Ops.

Over the last few years, quantized deep neural networks (QDNNs) [18, 19] have been extensively studied, largely due to their memory saving and high computation performance. In GNN, however, similar work is largely lagging behind. Work from [18] demonstrates that GNN is actually insensitive to quantization, even very low-bit quantization would not lead to evident accuracy loss because of the graph-like aggregation operations that can amortize such quantization influence. Another work from [35] also demonstrates that even the binarized GNN would be beneficial in some application scenarios. In this work, we foresee that the support for any-bitwidth precision computation on GNN is vital to satisfy various users' demands (*e.g.*, execution time).

Given a quantization bit q and the 32-bit floating-point value $\alpha \in \mathcal{R}$, we quantize it as a q-bit value by using

$$\alpha^{(q)} = \left\lfloor \frac{\alpha - \alpha_{min}}{scale} \right\rfloor. \tag{2.2}$$

where α_{min} is an empirical lower bound that can be determined by users or application settings; *scale* is the ratio between the range ($|\alpha_{max} - \alpha_{min}|$, where α_{max} is an empirical upper bound) and the q-bit representation range (2^q); $\lfloor \cdot \rfloor$ is the floor function.

For any-bitwidth computation on quantized values, we propose a new type of arithmetics based on the "atomic" 1-bit computation widely used in the binarized NN [36].

Any-bitwidth Scalar-Scalar Multiplication: Assuming we have a 3-bit scalar value (a) and multiply it with a 2-bit scalar value (b). we can first represent these two values as

$$\begin{aligned} a &= at_2 \cdot 2^2 + at_1 \cdot 2^1 + at_0 \cdot 2^0 \\ b &= bt_1 \cdot 2^1 + bt_0 \cdot 2^0 \end{aligned} \tag{2.3}$$

where at_* and bt_* indicate the bit value (0/1) at the certain bit position after bit decomposition. By following the general rule of multiplication, we can get $a \cdot b$ as

$$a \cdot b = (at_2 \cdot 2^2 + at_1 \cdot 2^1 + at_0 \cdot 2^0)(bt_1 \cdot 2^1 + bt_0 \cdot 2^0) \tag{2.4}$$

through simplification we can get that

$$\begin{aligned} a \cdot b = at_2bt_1 \cdot 2^3 + (at_1bt_1 + at_2bt_0) \cdot 2^2 \\ +(at_0bt_1 + at_1bt_0) \cdot 2^1 + at_0bt_0 \cdot 2^0 \end{aligned} \tag{2.5}$$

Any-bitwidth Vector-Vector Multiplication: We extend the any-bitwidth scalar-scalar computation towards any-bitwidth vector-vector computation between a 3-bit vector $\overrightarrow{v_i}$ and 2-bit vector $\overrightarrow{v_j}$, each of which has k elements. Therefore, the above scalar-scalar multiplication formula can be extended to k-dimension vector-vector multiplication

$$\begin{aligned} \overrightarrow{v_i} \cdot \overrightarrow{v_j} = \sum_y^k a^{(y)} \cdot b^{(y)} = \sum_y^k at_2^{(y)}bt_1^{(y)} \cdot 2^3 \\ + \sum_y^k (at_1^{(y)}bt_1^{(y)} + at_2^{(y)}bt_0^{(y)}) \cdot 2^2 \\ + \sum_y^k (at_0^{(y)}bt_1^{(y)} + at_1^{(y)}bt_0^{(y)}) \cdot 2^1 + \sum_y^k at_0^{(y)}bt_0^{(y)} \cdot 2^0 \end{aligned} \tag{2.6}$$

From the above formula, we can see that in order to compute the result of any-bitwidth

vector-vector multiplication, we first do bit decomposition on all elements in each vector then do bit-bit multiplication between elements from each vector, and finally do bit shifting and reduction to get the final result. For example, after bit-decomposition of $\overrightarrow{v_i}$ and $\overrightarrow{v_j}$, we can get $\overrightarrow{v_i}$ at bit position 2 as $at_2^{(y)}$ and $\overrightarrow{v_j}$ at bit position 1 as $bt_1^{(y)}$, where $y \in [0, k)$. From the multiplication and addition, we can get the multiplication result of $\overrightarrow{v_i} \cdot \overrightarrow{v_j}$ at bit position 3. Such a 1-bit vector-vector multiplication can be effectively implemented as

$$ans_{i,j} = popcnt(\overrightarrow{v_i} \& \overrightarrow{v_j}) \tag{2.7}$$

where $popcnt()$ counts the total number of $1s$ of the result in its bit representation (*e.g.*, *popcnt* will return 3 for a binary number 1011). A similar procedure can be applied to generate the result at bit position 0, 1, and 2. After all these individual bits in temporary results are ready, we can do bit shifting and reduction to get the final result. Based on such any-bitwidth vector-vector results, we can easily derive the any-bit matrix-matrix multiplication scheme, where each element in the output matrix can be seen as the results of any-bitwidth vector-vector multiplication.

2.3.2 Quantized Computation in GNNs

Applying any-bitwidth precision computation in GNNs would find two major specialties. First, the adjacent matrix ($\mathbf{A}$) of GNNs only needs to use binary (1-bit) numbers to represent the presence/absence of edges. Second, the node embedding matrix ($\mathbf{X}$) and the weight matrix ($\mathbf{W}$) can be represented with any bitwidth to meet the different precision demands.

As described in Algorithm 1, each layer of any-bitwidth GNN consists of a *neighbor aggregation* and a *node embedding update* phase. Specifically, neighbor aggregation conducts $\mathbf{X\_new} = \mathbf{A} \cdot \mathbf{X}$ through a *1-bit-and-s-bit* matrix multiplication and the node

Algorithm 1 1-layer Quantized GNN Computation.

Require: Full-bit adjacent matrix $\mathbf{A}$ ($N \times N$), node embedding matrix $\mathbf{X}$ ($N \times D$), and weight matrix $\mathbf{W}$ ($N \times H$).
Ensure: Updated full-bit node embedding matrix $\hat{\mathbf{X}}$ ($N \times H$).

```
1  A_bin = bitDecompse(A, 1)[0]
2  X_list = bitDecompse(X, s)
3  W_list = bitDecompse(W, t)
4  X_new_list = list(); C_dict = dict(); X̂ = zeros_as(X)
5  for xIdx in len(X_list) do
6    X_new_list.append(BMM(A_bin, X_list[xIdx]))
7  end for
8  for xIdx in len(X_new_list) do
9    for wIdx in len(W_new_list) do
10     bitIdx = xIdx + wIdx
11     tmp_C = BMM(X_new_list[xIdx], W_list[wIdx])
12     C_dict[bitIdx].append(tmp_C)
13   end for
14 end for
15 for bitIdx in len(C_dict) do
16   for Idx in len(C_dict[bitIdx]) do
17     X̂[Idx] += C_dict[bitIdx][Idx] ≪ bitIdx
18   end for
19 end for
```

update conducts $\hat{\mathbf{X}} = \mathbf{X\_new} \cdot \mathbf{W}$ through a *s-bit-and-t-bit* matrix multiplication. At Line 1 to 3, we do **bitDecompose** for subgraph adjacency matrix $\mathbf{A}$, embedding matrix $\mathbf{X}$, and weight matrix $\mathbf{W}$. For scalar `int32` numbers, our **bitDecompose** will first quantize it to another `int32` number in a n-bit data range $[0, 2^n - 1]$ by using Equation 2.2. Then, it applies bit-shifting to extract each bit (0/1) from the quantized `int32` number. Our 3D stacked bit compression (Section 2.4.2) happens after the above first and second steps are applied to each element of a matrix, and it will pack the extracted bits for the whole matrix together. Here for ease of algorithm description, we maintain different bits of a matrix as the list, *e.g.*, $\mathbf{X}[1]$ stands for the 0's bits for all elements inside the $\mathbf{X}$. At Line 5 to 7, we apply bit-matrix multiplication between each bit matrix from $\mathbf{X}$ and the binary 1-bit matrix $\mathbf{A_{bin}}$, the results of this step will still be a set of bit matrices

and be stored in a list. At Line 8 to 14, we apply the similar bit-matrix multiplication between $\mathbf{X}$ and $\mathbf{W}$, and the results of this step will be stored as bit-matrix as well for the following final-result generation. To avoid any data overflow during the reduction (Line 15 to 19), $\hat{\mathbf{X}}$ should also use a full-bit data type (*e.g.*, `int32`). For large graphs, their adjacent matrices cannot be easily fit into the GPU device memory directly. In this scenario, we employ METIS [22] for graph partitioning and run GNN as batched subgraph computation, which is used by the most popular cluster-GCN [23] design. Considering that the number of subgraphs generated by METIS [22] is usually within the reasonable size (2,000 to 20,000), such a batched GNN computation can be accommodated on a single modern GPU without violating its memory constraints. Note that to reduce the runtime overhead, the bit-decomposition of the matrix $\mathbf{W}$ and $\mathbf{A}$ can be pre-computed and cached before the GNN computation at each layer. The major reason behind this is that across different GNN layers of the same subgraphs, the adjacent matrix $\mathbf{A}$ can be reused. On the other side, across different subgraphs at the same GNN layers, the weight matrix $\mathbf{W}$ can be reused for the later-on computation.

2.4 Implementation

2.4.1 Subgraph Partitioning and Batching

Real-world graphs usually come with a large number of nodes and highly irregular graph structure (edge connections). This brings two levels of difficulties for GNN computing. The first one is the memory consumption, since GPU device memory cannot accommodate all nodes, edges, and node embedding features at the same time. The second one is the inefficient execution since the irregular and sparse edge connections lead to low memory access efficiency and poor computation performance. To this end,

in QGTC, we combine the state-of-the-art graph partitioning technique METIS [22] and subgraph batch processing [23] to handle different sizes of input graphs effectively. Compared with other solutions, such as graph clustering approaches [37, 38] and BFS-based methods [39], METIS achieves a better quality of its captured subgraph partitions (more edges in each subgraph) and the significantly higher runtime performance owing to its partial parallelization. Note that the number of subgraphs/partitions is determined by users and is passed as a runtime parameter to METIS.

After the subgraph partitioning, we will conduct a batching step for GNN computation on GPUs. This step gathers a set of subgraph partitions based on user-defined batch size. Later, during the GNN computing, subgraphs are loaded to GPU memory by batch. Using the partitioning and batching strategy for GNN computing gives users control of workloads at two levels of granularity. ***First***, the workload granularity is defined by the number of subgraphs/partitions. This would manage the size of each subgraph partition and the edge connection density of each subgraph. In general, the more number of the subgraphs/partitions would lead to denser edges connections within each subgraph, which may bring better computation and memory locality. ***Second***, the processing granularity is controlled by the batch size. This would determine the size of graphs that will be fit into the GPU at each round of execution. The selection of batch size would maximize the utilization of the GPU while respecting the GPU computation and memory resource constraints.

2.4.2 3D-Stacked Bit Compression

Existing NN frameworks are developed for full-precision computation, which leads to two major challenges: ***First***, the low-bit quantized data type cannot directly borrow the full-precision data type as the "vehicle" for computation. The major reason is that the full precision data type such as `float` and `int32` cannot bring any benefits to the

memory or computation saving. ***Second***, low-bit quantization would not fit any type of bit alignment, since its bit-level boundary mostly cannot be divisible by the size of a byte (8-bit), making it hard to retrieve its actual value.

To this end, we propose a novel *3D-stacked bit-compression* technique to handle any-bitwidth data type effectively. The major idea is to compress any-bitwidth input with 32-bit alignment for ease of value retrieval and memory alignment. As exemplified in Figure 2.4(a), we have an input matrix with the shape of 3-bit$\times M \times K$. For each bit of the element in the matrix, we store it in a bit matrix (1-bit$\times M \times K$) stacked along the vertical z axis. During the computation of any-bitwidth matrix multiplication $\mathbf{C} = \mathbf{A} \times \mathbf{B}$, two types of 3D-stacked bit-compression are employed. For matrix $\mathbf{A}$, we use the *column-wise compression* with 32-bit alignment, as illustrated in Figure 2.4(b). The main reason for choosing column-wise compression is that the matrix multiplication would benefit from coalesced across-column memory access along each row of the matrix $\mathbf{A}$. 32-bit alignment can benefit the read performance by coalesced loading from the global memory to fragment. After the compression on matrix $\mathbf{A}$ (1-bit$\times M \times K$), we will get a 32-bit compressed 3-bit $\mathbf{A_c}$ with the shape of 3-bit$\times(\mathbf{PAD8}(M) \times \lfloor \mathbf{PAD128}(K)/32 \rfloor)$, where **PAD8** and **PAD128** are for padding rows/columns that cannot be divisible by the basic TC computing size ($M(8) \times N(8) \times K(128)$). For matrix $\mathbf{B}$, we use the *row-wise compression* with 32-bit alignment, as shown in Figure 2.4(c) which can benefit the across-row access along each column of matrix $\mathbf{B}$. After the compression on matrix $\mathbf{B}$ (1-bit$\times M \times$K), we will get a 32-bit compressed 2-bit $\mathbf{B_c}$ with the shape of 2-bit$\times \lfloor \mathbf{PAD128}(K)/32 \rfloor \times \mathbf{PAD8}(N)$ for the output layer. Note that if the $\mathbf{A} \times \mathbf{B}$ is the hidden layer of a GNN model, the padding strategy on matrix $\mathbf{B}$ would be slightly different considering that the result matrix $\mathbf{C}$ will become a new matrix $\mathbf{A}$ in the next layer which demands 128-bit padding. In this case, to avoid additional padding overhead, we will get the 2-bit $\mathbf{B_c}$ with the shape of 2-bit$\times \lfloor \mathbf{PAD128}(K)/32 \rfloor \times \mathbf{PAD128}(N)$.

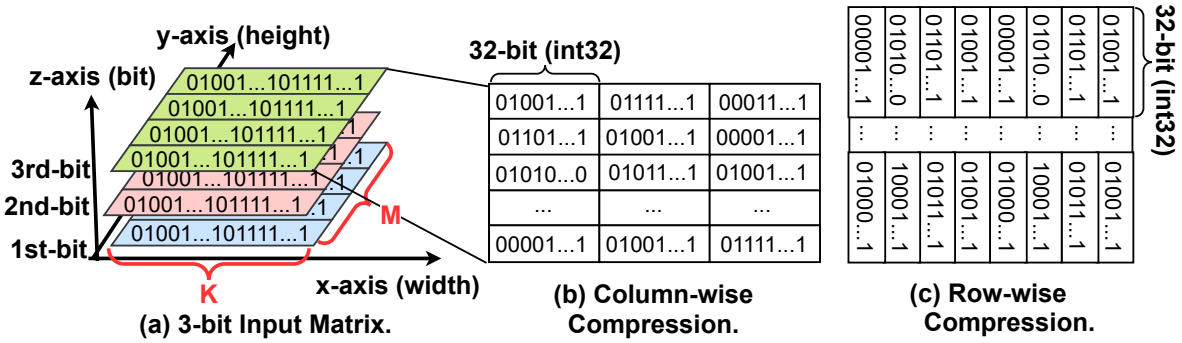


Figure 2.4: 3D-Stacked Bit Compression. Note that every 32 bits are compressed and stored in little-endian.

Compared with the previous work [40] that also leverages bit-level data packing, there are several differences. The ***first*** one is the *padding strategy*. Padding of QGTC on different tensor dimensions could be different, where bit-level padding is ignored in the work from [40]. For example, QGTC may PAD8 or PAD128, depending on the following computation is carried out in low-bit or 32-bit format, thereby, avoiding unnecessary conversion. The ***second*** one is the *packing datatype*. Work from [40] uses `uint4` for packing continuous 128 bits, while QGTC uses a 32-bit format for better interoperability with PyTorch. The ***third*** one is the *bit-level layout*. Work from [40] doesn't consider more bit-level layout optimization. In QGTC, for GEMM operation ($\mathbf{C} = \mathbf{AX}$), we use a column-wise compression for the matrix $\mathbf{A}$ and a row-wise compression for $\mathbf{X}$.

2.4.3 Zero-tile Jumping

Even though the subgraph partitioning such as METIS [22] makes the subgraph denser (more number of edge connections within each subgraph), there are still some TC tiles (*i.e.*, the input matrix tile for a single TC computation) that are filled with all-zero elements. Therefore, directly iterating through these zero tiles would introduce the cost of unnecessary memory (loading data from the global memory to thread-local

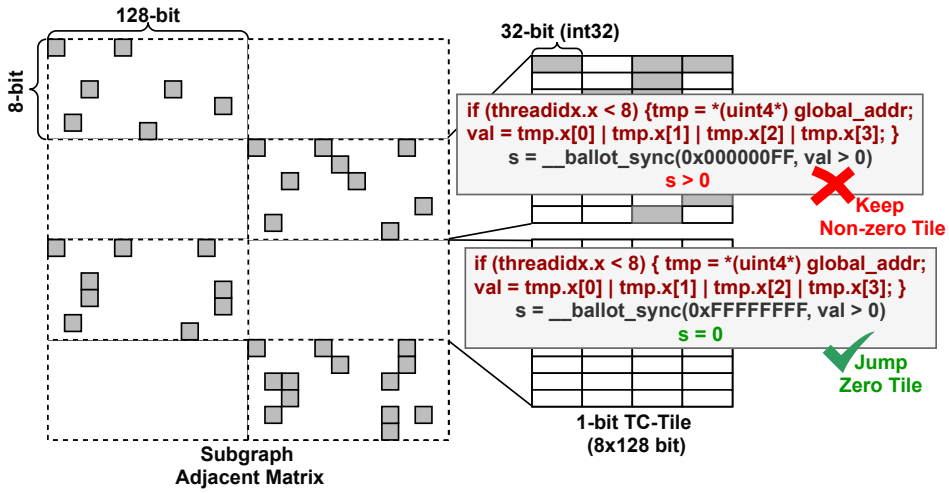


Figure 2.5: Zero-tile Jumping. Note that each small grey square box (on the left side) indicates an edge connection between two nodes within a graph. Each grey rectangular box (on the right side) indicates at least one of its 32 consecutive small square boxes is grey (the presence of an edge).

registers) and computation (processing 1-bit TC-GEMM on input adjacent matrix tile that contains all-zero elements). Based on this observation, we introduce a novel zero-tile jumping technique to reduce unnecessary computations by bitwise `OR` operations and warp-level synchronization primitives.

As illustrated in Figure 2.5, each 1-bit TC-GEMM would work on the tile size of 8×128 register fragment. This can be well partitioned into 8×4 `int32` elements. To check whether the 8×128 tile contains all-zero elements, we first employ only 8 threads from a warp of threads to fetch an `uint4_v` vector data (each `uint4_v` element in CUDA consists of 4 `int32` elements placed in continuous memory address). The reason for using `uint4_v` is to maximize the memory access efficiency by issuing fewer global memory requests. Once all `uint4_v` elements have been loaded. Each thread will apply bitwise `OR` across all 4 `int32` elements, which will check whether each row of a TC-tile is all-zero. The next step is to tell whether the whole tile is all-zeros across different rows, we will use the warp-level primitive to sync the information across these 8 active threads in the

warp. This step will generate an `int32` number. If this number is zero, it will indicate all elements in this input TC-tile are zero. Otherwise, we still need to conduct the 1-bit TC-GEMM on the current tile. We will give a more quantitative analysis of such zero-tile jumping in Section 5.4.3.

2.4.4 Non-Zero Tile Reuse

In addition to jumping over the zero tiles, we further consider reusing the non-zero tiles to improve data locality. In the aggregation step of the GNN computation, we generate the output feature map at different bit-level separately. For example, when we choose 1-bit adjacent subgraph matrix and a 4-bit feature embedding matrix, we will execute the iteration 4 times to generate the output. One straightforward solution, called *cross-bit reduction*, is to generate the complete output matrix tile at each bit level first. This requires loading the matrix tile imperatively, as shown in Figure 2.6(a). However, this would cause one problem that each non-zero tile from the adjacent matrix will be repetitively loaded when computing with each bit matrix from the embedding matrix.

In fact, we can consider reordering the steps in a way that we can maximize the benefit of each non-zero tile of the subgraph adjacency matrix. As shown in Figure 2.6(b), we introduce a *cross-tile reduction* strategy. Specifically, for each loaded non-zero fragment, we will use it to generate an output tile at all bit levels and do a localized reduction (only on the current tile) to generate a partial aggregation result. Once this part has been done, we will move forward to the next non-zero tile and repeat the same process until all non-zero tiles have been processed. The complexity of loading the nonzero tiles can be reduced from $\mathcal{O}(n)$ to $\mathcal{O}(1)$, where n is the number of bits for node embeddings.

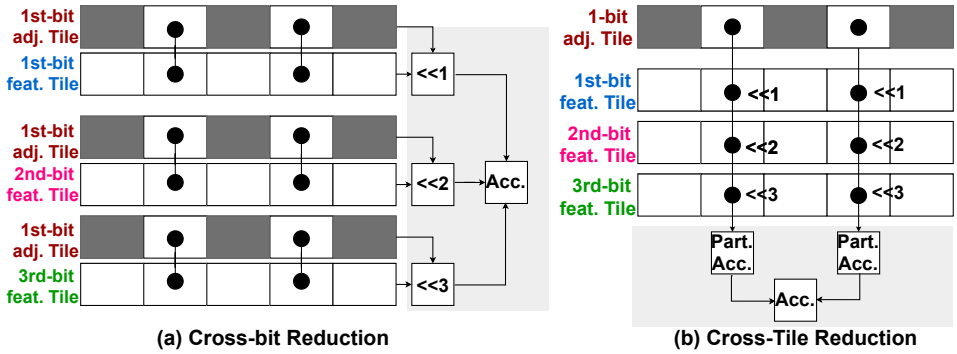


Figure 2.6: Non-zero Tile Reuse. Note that the grey box indicates the zero-tile of the subgraph adjacent matrix, while the white box with a block solid dot inside represents the non-zero tiles of the subgraph adjacent matrix.

2.4.5 Inter-layer Kernel Fusion

Across the GNN layers, we incorporate the low-bit data transferring. Specifically, the output of the one hidden layer will directly be handed over to the next layer as the input. There are several strategies we use. ***First***, we apply data quantization and bit-decomposition at the end of the computation kernel such as the neighbor aggregation and node update. This can help to avoid outputting the result to the slow global memory and apply the data manipulation again. ***Second***, standalone activation function kernels such as `ReLU` and `tanh`, can be effectively fused into our computation kernel as a `device` function, which can directly operate the shared memory results to achieve high performance. For the batch normalization (BN) layers that follow the graph convolution layers, we can also do layer fusion based on

$$\mathbf{BN}(x_{i,j}) = \left(\frac{x_{i,j} - \mathbb{E}[x_{*,j}]}{\sqrt{Var[x_{*,j}] + \epsilon}} \right) \cdot \gamma_j + \beta_j \tag{2.8}$$

where β_j, γ_j, and ϵ are the BN parameters that can be incorporated into the low-bit convolutional kernel to avoid launching a BN kernel. One thing worth noting is that computation at the hidden layer and the output layer is slightly different. For hidden

layers, each kernel has the quantization + bit-decomposition on the output activation, since the next layer relies on the low-bit data as the input for computation. While for the last layer, once the full-precision accumulation is complete, it will directly output the full-precision result for the `softmax` layer (considering the node classification task) to generate logits that demand high precision.

2.4.6 Bandwidth-Optimized Subgraph Packing

During the GNN computation of the subgraphs, data communication between the CPU host and GPU device is also unavoidable. It will swap the subgraph data (such as edge lists and node embedding) in/out of the GPU device. One basic approach is to transfer the dense adjacent matrix in floating point numbers considering that the size of a single subgraph is generally within the range of the modern GPU memory. However, this could easily lead to a huge amount of data traffic between the CPU and GPU host. The transferring performance is heavily bounded by PCIe bandwidth (32 GB/s for PCIe 4.0x16) between the CPU host and the GPU device. For the node embedding matrix, the current practice is to transfer the node embedding matrix by initializing another standalone PCIe transferring, which incurs additional overheads and is unable to maximize the bandwidth usage.

To overcome these issues, we employ a new strategy, called *bandwidth-optimized subgraph packing*. Instead of directly migrating the dense adjacent matrix or sparse adjacent matrix in single-precision floating point, we just transfer the compressed low-bit adjacent matrix and low-bit embedding matrix. This can significantly minimize the data traffic on the high-cost PCIe-based data communication. Besides, we compress the low-bit adjacent matrix and low-bit embedding matrix into a compound memory object (by using `torch.nn.Module` and `register_buffer`) on the host first and then initiate the

transferring of this memory object from the host CPU to GPU device.

2.5 Integration with PyTorch

Besides the highly efficient kernel design and data transferring optimization, for better usability and programmability, we integrate QGTC with the popular PyTorch framework. However, there are two key technical challenges. The first one is how to represent the quantized low-bit number in those Tensor-based frameworks that are built on byte-based data types (*e.g.*, `int32`). The second one is how to apply valid computation between the quantized low-bit number and those well-defined byte-based numbers. For example, how could we get the correct results when we do arithmetic multiplication between a 2-bit number and a 32-bit integer number. To this end, we introduce two new techniques.

Bit-Tensor Data Type: We use the 32-bit `IntTensor` in PyTorch as the "vehicle" for holding any-bitwidth quantized data. And we leverage our 3D-stacked bit compression technique (Section 2.4.2) to package the quantized data. We offer a PyTorch API `Tensor.to_bit(nbits)` for such data type conversion functionality. Note that existing PyTorch APIs, such as `print`, are only defined for those complete data types, such as Int. Therefore, to access the element value of a bit-Tensor, we provide `Tensor.to_val(nbits)` to decode a bit-Tensor as `int32` Tensor (converting each element from a low-bit number to an `int32` number). This can make our design compatible with existing PyTorch functionalities.

Bit-Tensor Computation: We handle two different types of computation: 1) the operations that only involve bit-Tensor and 2) the operations that involve both bit-Tensor and `float` or `int32` Tensor. For the first type of operations, we built two APIs based on whether we want to get the `int32` output or still get the quantized low-bit output as a bit Tensor. For any-bitwidth MM with low-bit output, the API is `bitMM2Bit(C,`

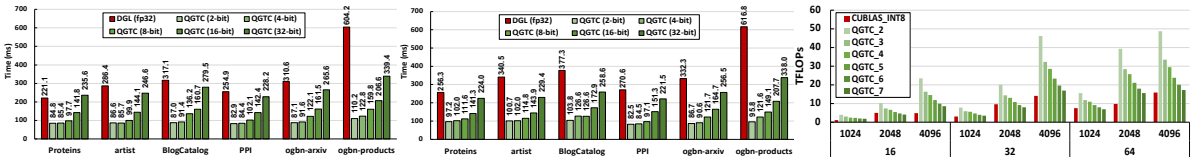
Table 2.1: Datasets for evaluation.

Type	Dataset	#Vertex	#Edge	Dim.	#Class
I	Proteins	43,471	162,088	29	2
	artist	50,515	1,638,396	100	12
II	BlogCatalog	88,784	2,093,195	128	39
	PPI	56,944	818,716	50	121
III	ogbn-arxiv	169,343	1,166,243	128	40
	ogbn-products	2,449,029	61,859,140	100	47

`A, B, bit_A, bit_B, Bit_C)`, where `A` and `B` are bit Tensors, `bit_A/B/C` are bitwidth parameters. For any-bitwidth MM with `int32` output, the API is `bitMM2Int(C, A, B, bit_A, bit_B)`. For the second type of operations, we will first decode a bit-Tensor as a `float`/`int32` Tensor by using `Tensor.to_val(nbits)`. Then we call the official APIs in PyTorch for the regular full-precision computation.

2.6 Evaluation

Benchmarks: We choose two most representative GNN models widely used by previous work [15, 16, 14] on the node classification task to cover different types of aggregation. **1) Cluster GCN** [10] is one of the most popular GNN model architectures. It is also the key backbone network for many other GNNs, such as GraphSAGE [12], and differentiable pooling (Diffpool) [41]. For Cluster GCN evaluation, we use the setting: *3 layers with 16 hidden dimensions per layer.* **2) Batched GIN** [13] differs from cluster GCN in its order of aggregation and node update. Batched GIN aggregates neighbor embedding before the node feature update (via linear transformation). GIN demonstrates its strength by capturing the graph properties that cannot be collected by GCN according to [13]. Therefore, improving the performance of GIN will benefit more advanced GNNs, such as GAT [11]. For batched GIN evaluation, we use the setting: *3 layers with 64 hidden dimensions per layer.* For quantization bitwidth, we cover the bitwidth settings



(a) (b) (c)

Figure 2.7: End-to-end performance comparison with (a) DGL on Cluster GCN and (b) DGL on Batched GIN. (c) Compared with TC-based `cuBLASgemmEX` (`int8`) on GNN aggregation kernel throughput performance (in TFLOPs). Note that "QGTC_3" stands for QGTC with 3-bit data representation for node embedding matrix.

from the existing quantized GNN studies [18, 19] and also conduct a comprehensive experimental analysis on different bitwidth settings.

Baselines: In our experiments, we choose several baselines for comparison. For end-to-end runtime performance comparison, we choose **Deep Graph Library (DGL)** [15], which is the state-of-the-art GNN framework on GPUs. DGL is built with a highly optimized CUDA-based GNN kernel as the backend and uses PyTorch as its front-end. For GNN aggregation kernel performance comparison, we choose the state-of-the-art quantized GEMM implementation on GPU Tensor Core from **cuBLAS** [32] with `int8` precision and **CUTLASS** [33] with `int4` precision.

Datasets: We cover all three types of datasets, which have been used in many previous GNN-related work [15, 16]. Details of these datasets are listed in Table 5.4. Specifically, **Type I** graphs are the popular GNN datasets evaluated by many GNN algorithmic papers [10, 13]. **Type II** graphs [42] are the popular benchmark datasets for graph kernels in many frameworks for GNN algorithmic research. **Type III** graphs [43] are challenging GNN datasets in terms of the large number of nodes and edges. These graphs demonstrate high irregularity in its structures. Note that we do graph partitioning by using METIS [22] and set the number of total subgraphs as 1,500 as prior work [23, 44].

Platforms & Metrics: QGTC backend is implemented with C++ and CUDA C, while QGTC front-end is implemented in Python. Our major evaluation platform is

a Ubuntu server (16.04) with an 8-core 16-thread Intel Xeon Silver 4110 CPU@2.8GHz with 64GB host memory and an NVIDIA Ampere RTX3090 GPU with 24GB device memory. The GPU device kernel is compiled with CUDA (v11.0) and the CPU host code is compiled with GCC 7.5.0 with the compilation option of "-std=c++14 -O3" for integration with the PyTorch framework. To measure the performance speedup, we calculate the averaged latency of 200 rounds of end-to-end results.

2.6.1 Compared with DGL

In this section, we conduct detailed end-to-end comparison with DGL framework under the different choices of bitwidth. As shown in Figure 2.7(a) and Figure 2.7(b), QGTC achieves on average 2.6× and 2.8× end-to-end inference speedup compared to DGL over three types of datasets for cluster GCN and batched GIN, respectively. We also notice that the performance benefit is closely related to the bitwidth we choose, as we can see that from 16-bit to 32-bit the performance shows a large difference compared with the 2-bit to 8-bit setting. We next provide a detailed analysis and give insights for each type of dataset. With a fewer number of bits for both the weights and the node embedding features, QGTC is more likely to reach higher performance. This is because a smaller size of bitwidth would lead to less memory access and fewer computations at the bit level. DGL reaches an inferior performance due to 1) FP32 computation comes with the high computation complexity compared with our QGTC low-bit design; 2) DGL can only rely on CUDA cores for computation which is naturally bounded by the peak computation performance compared with our QGTC on TC with higher throughput performance. Compared with cluster GCN, experimental results on the batched GIN shows higher benefits of QGTC over DGL. This is because batched GIN applies the node update first before the neighbor aggregation, which leads to higher computation-to-communication ratio. QGTC achieves relatively higher performance improvements on

Table 2.2: Model accuracy *w.r.t.* quantization bitwidth.

Settings	FP32	16 bits	8 bits	4 bits	2 bits
ogb-product	0.791	0.791	0.783	0.739	0.620
ogb-arxiv	0.724	0.708	0.707	0.685	0.498

Type III datasets. The major reason is that under the same number of partitions, the size of each partition (subgraph) will increase due to more number of nodes/edges. This also improves the computation intensity that will highlight QGTC's performance advantages of quantized low-bit computation on GPU Tensor Cores.

Accuracy *w.r.t.* Quantization Bits To build the QGNN model, we apply quantization-aware training and evaluate the model testing accuracy *w.r.t.* quantization bits on two large Type III datasets on GCN model for demonstration. As shown in Table 2.2, the GNN model is resilient to the low-bit quantization and can maintain the model accuracy to a large extent. Combining these results with our above performance evaluation result under different quantization bits, we can conclude that making the right tradeoff between the runtime performance and model accuracy is meaningful and can bring benefits to different application settings.

2.6.2 Compared with other baselines

Compared with cuBLAS-`int8` on TC. We further compare our low-bit computation (from 2-bit to 7-bit) with respect to the state-of-the-art `cuBLASgemmEX` for quantized (`int8`) GEMM solution on Tensor Core in terms of their throughput performance. Note that `int8` is the cuBLAS currently supported minimum bits for quantized computation on Tensor Core. In this study, we mainly focus on the computation of **AX** (*i.e.*, $N \times N \times D$, where N is the number of nodes and D is the node embedding dimension) for the neighbor aggregation phase. As shown in Figure 2.7(c), QGTC achieves significant throughput improvement compared with Tensor Core cuBLAS (`int8`) in low-bit settings.

Table 2.3: Compared with CUTLASS-`int4` (TFLOPs).

N	**Dim**	**CUTLASS (`int4`)**	**QGTC (1-bit)**	**QGTC (2-bit)**	**QGTC (3-bit)**	**QGTC (4-bit)**
2048	32	10.36	32.65	19.99	14.40	11.30
4096	32	12.28	81.41	46.23	32.27	24.75
8192	32	12.67	94.58	50.82	35.22	26.31
2048	64	21.40	63.94	39.41	29.83	22.15
4096	64	24.66	89.18	51.21	35.17	25.38
8192	64	24.70	104.66	55.16	40.77	31.07

The major reason is our QGTC design effectively reduces the computation and the data movements at the bit level, thereby, harvesting the real performance gains of the low-bit quantization on GPUs. When the number of bits for quantization is approaching 8-bit in the computation, the performance gains would decrease due to the increase of bit-level computations.

Compared with CUTLASS-`int4` on TC We also compare against the latest CUTLASS [33](v2.7) with the `int4` Tensor Core GEMM in terms of throughput (TFLOPs) for **AX**. The results are summarized at Table 2.3, where we can clearly see the performance advantage in terms of throughput over the CUTLASS implementation. Note that all reported decimal numbers are in TFLOPS; N is the adjacent matrix size and Dim is the node feature embedding dimension. The graph adjacent matrix is stored in 1-bit. QGTC (2-bit) means the 2-bit representation for the embedding matrix. The major reason behind such performance improvement is that our QGTC design can use the 1-bit binary for representing graph adjacency matrix and n-bit (n=1,2,3,4) for node embedding matrix, while CUTLASS `int4` only have the support of 4-bit $\times$ 4-bit. Thus, we have to use a 4-bit presentation for both adjacent matrix and embedding matrix during computation.

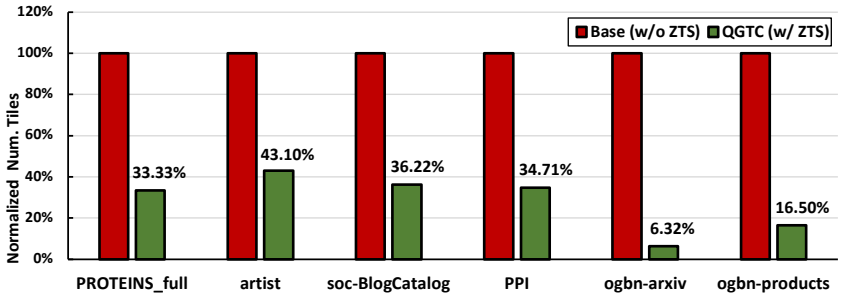


Figure 2.8: Zero-tile jumping efficiency. The percentage (%) on each green bar indicates the ratio of the number of tiles processed w/ jumping versus w/o jumping solution.

2.6.3 Additional Studies

In this section, we will conduct detailed studies to demonstrate the effectiveness of our design and optimizations.

Zero-Tile Jumping. We would compute the ratio of the non-zero TC tiles (8×128) that are actually involved in our computation versus the total number of TC tiles in the adjacent matrix. As shown in Figure 2.8, our zero-tile jumping technology can largely save the efforts for processing all-zero tiles. Based on our observation, the source of such all-zero TC tiles comes from two levels. The first type of all-zero TC tiles is coming from batching subgraphs. Because there is no edge connection among nodes across different subgraphs. This type of all-zero TC tiles dominates the overall collected number of all-zero tiles. The second type of all-zero tiles comes from the missing edge connections between the nodes within each subgraph. While this type of all-zero tiles is minor in its quantity compared with the first type. It potentially reduces memory access and computation.

Adjacency Matrix Size. We will demonstrate the subgraph adjacency matrix size impact on the performance of QGTC. Specifically, adjacency matrix size can be controlled by specifying the *number of subgraphs* (in METIS) and *batching size* (in data loader). The size of the adjacency matrix will impact the performance of aggregation in terms of

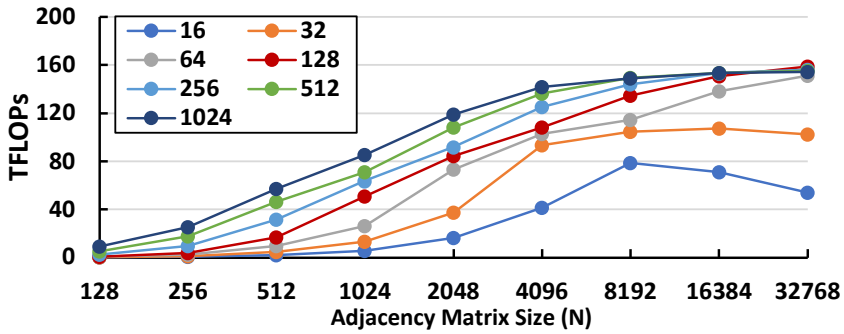


Figure 2.9: Adjacency matrix size impact. Note that we choose the common subgraph size N={128, 256, ..., 32768} and the hidden embedding dimension D={16, 32, ..., 1024}.

computations and data movements, meanwhile, it will also determine whether our GNN computation can fully utilize the available GPU resources.. We use 1-bit for both the adjacency matrix and node embedding matrix in this study. As shown in Figure 2.9, we can observe that under the same size of D, with the increase of the number of nodes (i.e., the value of N), our major 1-bit GEMM computation kernel would scale up its performance well. Note that different colored lines represent different embedding sizes, and we mainly focus on the computation of $\mathbf{AX}$ (*i.e.*, $N \times N \times D$, where N is the number of nodes and D is the node embedding dimension) for neighbor aggregation phase. in the settings of small subgraph size (128 to 512), the increase of the overall computation throughput is not evident, because the computation size is small and most of the available GPU resources such as SMs would achieve low utilization. While in the range of subgraph size (512 to 16,384), we can notice a more significant increase in the TFLOPs performance. Because in these settings, more computations from the bit-level data manipulation would trigger more SMs to participate in the BMM computation, thereby, improving the overall GPU throughput. For those large subgraph sizes ($>$ 16,384) the overall throughput would hardly increase, mainly because all available GPU computation units are almost fully in use. One specialty of those batched GNN computations *w.r.t.* the traditional NN computation is that batch GNN have more skewed-sized matrices in terms of the

ratio between N and D. This, to some degree, limits the achievable peak performance on TC. What is also worth noticing is that among different lines (different choices of D), the larger D usually leads to better utilization of the GPU, since more computation and memory resources of the GPU will become active for higher throughput.

Non-zero Tile Reuse. We will demonstrate the effectiveness of our non-zero tile reuse by a control-variable study. We eliminate the number of non-zero tiles impact on performance by setting all tiles to non-zero tiles (*i.e.*, filling the initial matrix with all ones). Then we choose the neighbor aggregation process ($\hat{\mathbf{X}} = \mathbf{AX}$) for the study and fix the D to 1024. We change N from 1,024 to 8,192. Three bit combinations are used in our evaluation, where $\mathbf{A}$ is consistently using 1-bit while $\mathbf{X}$ is using 4, 8, and 16 bit.

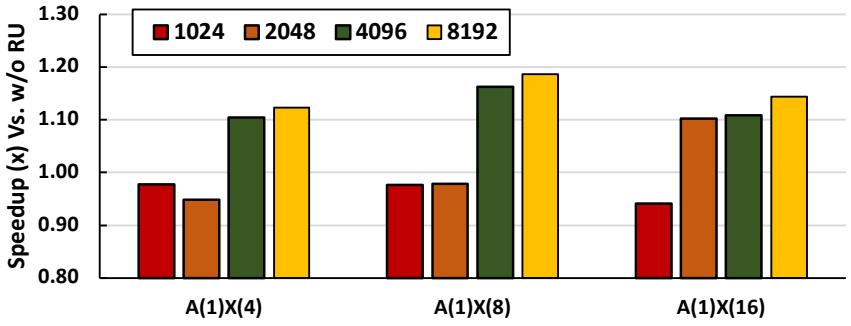


Figure 2.10: Non-zero tile reuse effectiveness. Note that we choose subgraph size N={1024, 2048, 4096, 8192} for this study.

As described in Figure 2.10, our non-zero tile reuse can improve the throughput performance on those large matrix sizes with the higher number of bits. The major reason behind this is that reuse the non-zero tile can largely reduce the global memory access for fetching the same 1-bit adjacency matrix tile repetitively, which is the key performance bottleneck for those large metrics. The setting (w/o nonzero-tile) reuse shows more advantage on the smaller size matrix because the overhead of recurrent loading the same adjacency matrix tile is not pronounced compared with GEMM operations on TC. This study inspires us to come up with a more intelligent strategy or heuristics to determine

under which condition applying the non-zero tile reuse will bring performance benefits and we would leave this for our future work for exploration.

Chapter 3

GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs

As the emerging trend of graph-based deep learning, Graph Neural Networks (GNNs) excel for their capability to generate high-quality node feature vectors (embeddings). However, the existing one-size-fits-all GNN implementations are insufficient to catch up with the evolving GNN architectures, the ever-increasing graph sizes, and the diverse node embedding dimensionalities. To this end, we propose **GNNAdvisor** [1], an adaptive and efficient runtime system to accelerate various GNN workloads on GPU platforms. First, GNNAdvisor explores and identifies several performance-relevant features from both the GNN model and the input graph, and uses them as a new driving force for GNN acceleration. Second, GNNAdvisor implements a novel and highly-efficient 2D workload

[1]Published at USENIX OSDI'21. USENIX permits authors to retain their ownership of the copyrights in their works. Reprinted from *GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs*. USENIX Symposium on Operating Systems Design and Implementation. 07/2021.

management, tailored for GNN computation to improve GPU utilization and performance under different application settings. Third, GNNAdvisor capitalizes on the GPU memory hierarchy for acceleration by gracefully coordinating the execution of GNNs according to the characteristics of the GPU memory structure and GNN workloads. Furthermore, to enable automatic runtime optimization, GNNAdvisor incorporates a lightweight analytical model for an effective design parameter search. Extensive experiments show that GNNAdvisor outperforms the state-of-the-art GNN computing frameworks, such as Deep Graph Library (3.02× faster on average) and NeuGraph (up to 4.10× faster), on mainstream GNN architectures across various datasets.

3.1 Introduction

Graph Neural Networks (GNNs) emerge to stand on the front-line for handling many graph-based deep learning tasks (*e.g.,* node embedding generation for node classification [45, 46, 47] and link prediction [48, 49, 50]). Compared with standard methods for graph analytics, such as random walks [25, 51] and graph Laplacians [52, 53, 54], GNNs highlight themselves with the interleaved two-phase execution of both graph operations (scatter-and-gather [55]) at the *Aggregation* phase, and Neural Network (NN) operations (matrix multiplication) at the *Update* phase, to achieve significantly higher accuracy [10, 13, 11] and better generality [12]. Yet, the state-of-the-art GNN frameworks [15, 16, 14, 56], which follow a one-size-fits-all implementation scheme, often suffer from poor performance when handling more complicated GNN architectures (*i.e.,* more layers and higher hidden dimensionality in each layer) and diverse graph datasets.

Specifically, previous work that supports both GNN training and inference can be classified into two categories. The first type [56, 14] is built on popular graph processing systems and is combined with NN operations. The second type [16, 15], in contrast,

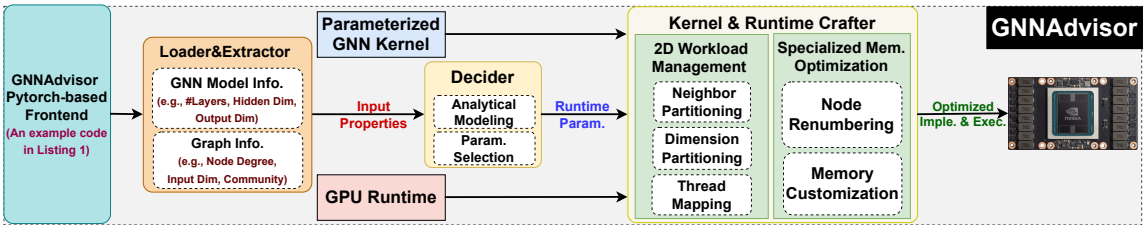


Figure 3.1: Overview of GNNAdvisor.

starts with deep learning frameworks and is extended to support vector-based graph operations. However, these existing solutions are still preliminary and inevitably fall short in the following three major aspects, even on common computing platforms such as GPUs.

Failing to leverage GNN input information. GNN models demonstrate great diversity in terms of layer sequences, types of aggregation methods, and the dimension size of node embeddings. These profoundly impact the effectiveness of various system optimization choices. The diversity of input graphs further complicates the problem. Unfortunately, current GNN frameworks [16, 15, 14] follow a one-size-fits-all optimization scheme and fail to craft an optimization strategy that maximizes efficiency for a particular GNN application's settings. Some classical graph systems [57, 58] have exploited input characteristics to facilitate more efficient optimizations, but they only focus on simple graph algorithms like PageRank [59] while having no support for GNN models.

Optimizations not tailored to GNN. While the update phase in GNNs involves NN operations that are dense in computation and regular in memory access, the aggregation phase is usually sparse in computation and highly irregular in memory access. Without dedicated optimization, it will inevitably become the performance bottleneck. Existing GNN frameworks [16, 15, 14] simply extend the optimization schemes from classical graph systems [60, 56], and do not address the difference between GNN and graph processing. For example, each node is associated with an embedding attribute in GNNs while each node only has a single scalar attribute in traditional graph processing. Such

difference invokes novel design principles for GNNs towards more aggressive parallelism and more efficient memory optimization.

Listing 3.1: Example of a 2-layer GCN in GNNAdvisor.

```
import GNNAdvisor as GNNA
import torch
# import other packages ...

# Create a GCN class.
class GCN(torch.nn.Module):
    def __init__(self, inDim, hiDim, outDim, nLayers):
        self.layers = torch.nn.ModuleList()
        self.layers.append(GNNA.GCNConv(inDim, hiDim))
        for i in range(nLayers - 2):
            layer = GNNA.GCNConv(hiDim, hiDim)
            self.layers.append(layer)
        self.layers.append(GNNA.GCNConv(hiDim, outDim))
        self.softmax = torch.nn.Softmax()

    def forward(self, X, graph, param):
        for i in range(len(self.layers)):
            X = self.layers[i](X, graph, param)
            X = self.ReLU(X)
        X = self.softmax(X)
        return X

# Define a two-layer GCN model.
model = GCN(inDim=100, hiDim=16, outDim=10, nLayers=2)

# Loading graph and extracting input propertities.
graphObj, inputInfo = GNNA.LoaderExtractor(graphFile,
                                           model)
# Set runtime parameters automatically.
X, graph, param = GNNA.Decider(graphObj, inputInfo)

# Run model.
predict_y = model(X, graph, param)

# Compute loss and accuracy.
# Gradient backpropagation for training.
```

Poor runtime support for input adaptability. Prior GNN frameworks [16, 15, 14] rely on a Python-based high-level programming interface for ease of user implementation. These frameworks employ static optimizations through a compiler or manually-optimized libraries. Nevertheless, some critical performance-related information for a GNN is only available at runtime (*e.g.*, node degree and embedding size). Without adaptable designs that can leverage such runtime information, we would easily suffer from an

inferior performance because of the largely under-utilized the GPU computing resources and inefficient irregular memory access. This limitation motivates the need for runtime environments with flexible designs to handle a wide spectrum of inputs effectively.

To this end, we propose, GNNAdvisor[2], an adaptive and efficient runtime system for GNN acceleration on GPUs. GNNAdvisor leverages Pytorch as the front-end to improve programmability and ease user implementation. We show a representative 2-layer Graph Convolutional Network (GCN) [10] in GNNAdvisor at Listing 5.1. At the low level, GNNAdvisor is built with C++/CUDA and integrated with Pytorch framework by using Pytorch Wrapper. It can be viewed as a new type of Pytorch operator with a set of kernel optimizations and runtime support. It can work seamlessly with existing operators from the Pytorch Framework. Data is loaded with the data loader written in Pytorch and passed as a Tensor to GNNAdvisor for computation on GPUs. Once the GNNAdvisor completes its computation at the GPU, it will pass the data Tensor back to the original Pytorch framework for further processing. As detailed in Figure 4.1, GNNAdvisor consists of several key components to facilitate the GNN optimization and execution on GPUs. First, GNNAdvisor introduces an input **`Loader&Extractor`** to exploit the input-level information that can guide our system-level optimizations. Second, GNNAdvisor incorporates a **`Decider`** consisting of analytical modeling for automatic runtime parameter selection to reduce manual effort in design optimization, and a lightweight node renumbering routine to improve graph structural locality. Third, GNNAdvisor integrates a **`Kernel&Runtime Crafter`** to customize our parameterized GNN kernel and CUDA runtime, which consists of an effective 2D workload management (considering both the number of neighbor nodes and the node embedding dimensionality) and a set of GNN-specialized memory optimizations.

Note that in this project, we mainly focus on the setting of single-GPU GNN com-

[2]GNNAdvisor is open-sourced at `https://github.com/YukeWang96/GNNAdvisor_OSDI21.git`

puting, which is today's most popular design adopted as the key component in many state-of-the-art frameworks, such as DGL [15] and PyG [16]. Single-GPU GNN computing is desirable for two reasons: First, many GNN applications with small to medium size graphs (*e.g.*, molecule structure) can easily fit the memory of a single GPU. Second, in the case of large-size graphs that can only be handled by out-of-GPU-core and multi-GPU processing, numerous well-studied graph partition strategies (*e.g.*, METIS [22]) can cut the giant graphs into small-size subgraphs to make them suitable for a single GPU. Therefore, the optimization of both the out-of-GPU-core (*e.g.*, GPU streaming processing) and multi-GPU GNN computation still largely demands performance improvements on a single GPU. Moreover, while our paper focuses on GNNs, our proposed methodology can be applied to optimize various types of irregular workload (*e.g.*, social network analysis) targeting GPUs as well.

Overall, we make the following contributions:

- We are the first to explore GNN input properties (*e.g.*, GNN model architectures and input graphs), and give an in-depth analysis of their importance in guiding system optimizations for GPU-based GNN computing.

- We propose a set of GNN-tailored system optimizations with parameterization, including a novel 2D workload management and specialized memory customization on GPUs. We incorporate the analytical modeling and parameter auto-selection to ease the design space exploration.

- Comprehensive experiments demonstrate the strength of GNNAdvisor over state-of-the-art GNN execution frameworks, such as Deep Graph Library (average 3.02×) and NeuGraph (average 4.36×), on mainstream GNN architectures across various datasets.

3.2 Background and Related Work

In this section, we introduce the basics of GNNs and two major types of GNN computing frameworks: *GPU-based graph systems* and *deep learning frameworks*.

3.2.1 Graph Neural Networks

Figure 3.2 visualizes the computation flow of GNNs in one iteration. GNNs compute the node feature vector (embedding) for node v at layer $k+1$ based on the embedding information at layer k ($k \geq 0$), as shown in Equation 4.1,

$$\begin{aligned} a_v^{(k+1)} &= \mathbf{Aggregate}^{(k+1)}(h_u^{(k)} | u \in \mathbf{N}(v) \cup h_v^{(k)}) \\ h_v^{(k+1)} &= \mathbf{Update}^{(k+1)}(a_v^{(k+1)}) \end{aligned} \tag{3.1}$$

where $h_v^{(k)}$ is the embedding vector for node v at layer k; $h_v^{(0)}$ is computed from the task-specific features of a vertex (e.g., the text associated with the vertex, or some scalar properties of the entity that the vertex represents) via some initial embedding mapping that is used only for this ingest of symbolic values into the embedding space; $a_v^{(k+1)}$ is the aggregation results through collecting neighbors' information (*e.g.*, node embeddings); $\mathbf{N}(v)$ is the neighbor set of node v. The aggregation method and the order of aggregation and update could vary across different GNNs. Some methods [10, 12] just rely on the neighboring nodes while others [11] also leverage edge properties, by combining the dot product of the end-point nodes of each edge, along with any edge features (edge type and other attributes). The update function is generally composed of standard NN operations, such as a single fully connected layer or a multi-layer perceptron (MLP) in the form of $w \cdot a_v^{(k+1)} + b$, where w and b are the learnable weight and bias parameters, respectively. The common choices for node embedding dimensions are 16, 64, and 128, and the embedding dimension may change across different layers.

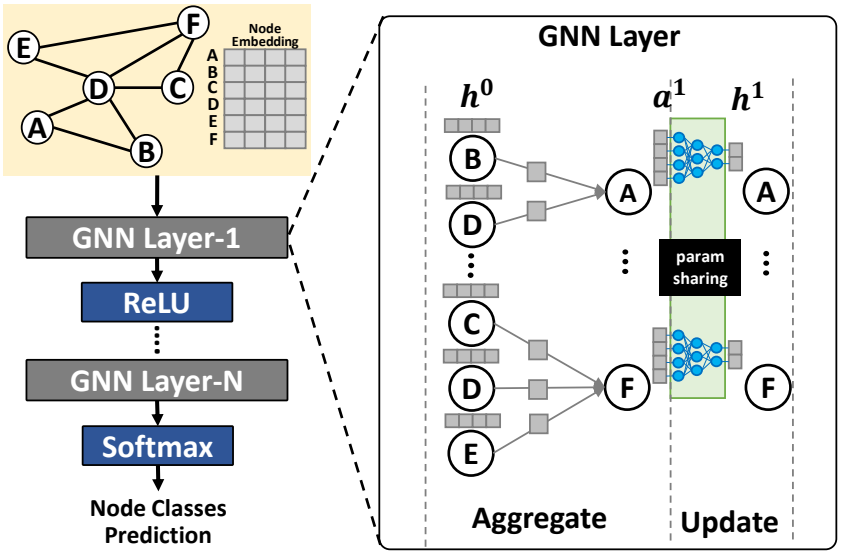


Figure 3.2: GNN General Computation Flow.

After passing through several iterations of aggregation and update (*i.e.*, several GNN layers), we will get the output embedding of each node, which can usually be used for various downstream graph-based deep learning tasks, such as node classification [45, 46, 47] and link prediction [48, 49, 50]. Note that the initial node embedding for GNN's input layer may come with the original graph dataset or can be generated by a set of graph embedding algorithms, such as [25, 61, 62], which is not included in the computation of GNNs models (generating the hidden and output node embeddings).

3.2.2 Graph Processing Systems

Numerous graph processing systems [60, 63, 56, 64, 65] have been proposed to accelerate traditional graph algorithms. The major commonalities of these systems include the vertex/node-centric programming abstraction, edge-centric processing, and system optimizations to reduce the computation irregularity (*e.g.*, workload imbalance) and memory access irregularity (*e.g.*, non-coalesced global memory access). However, extending these graph processing systems to support GNN computing meets with substantial challenges.

First, the common algorithm optimizations in graph processing may not benefit GNNs. For example, graph traversal algorithms, such as Breadth-first Search, rely on iterative computing on node frontiers (active neighbors). Therefore, a set of frontier-based optimizations, such push-pull traversal [64, 65], and frontier filtering [64, 65, 56], have been extensively studied. However, GNNs consistently maintain fixed-sized frontiers (all neighbors) of each node across iterations.

Second, the system optimization techniques for graph processing would benefit GNNs only after careful adaption and calibration. For example, node/edge-centric processing [56, 65] and shard-based graph representation [60] are tailored for processing nodes/edges represented with a single scalar attribute. In GNNs, there's another dimension for data parallelism, namely the embedding dimension, which tends to be large Therefore, previous design trade-offs between the coarse-grained node-level parallelism and node-value locality should be further extended to balance dimension-wise parallelism and node-embedding locality at a finer granularity.

Third, some essential functionalities of GNN computing are missing in graph systems. For example, the node update based on NN computing for both the forward value propagation and the complicated backward gradient propagation is not available in graph systems [60, 63, 56, 64, 65, 66, 67]. In contrast, Pytorch and Tensorflow feature an analytic differentiation function for automatic gradient computations on various deep learning model architectures and functions. Therefore, extending the graph-processing system to support GNN computing requires non-trivial efforts, and thus we develop GNNAdvisor on top of a deep learning framework.

3.2.3 Deep Learning Frameworks

Various NN frameworks have been proposed, such as Tensorflow [68], and Pytorch. These frameworks provide end-to-end training and inference support for traditional deep-

learning models with various NN operators, such as linear and convolutional operators. These operators are highly optimized for Euclidean data (*e.g.*, image) but lack support for non-Euclidean data (*e.g.*, graph) in GNNs. Extending NN frameworks to support GNN that takes the highly irregular graphs as the input is facing several challenges.

First, NN-extended GNN computing platforms [16, 15] focus on programmability and generality for different GNN models but lack efficient backend support to achieve high performance. For example, Pytorch-Geometric (PyG) [16] uses the torch-scatter [69] library implemented with CUDA as its major building block of graph aggregation operations. The torch-scatter implementation scales poorly when encountering large sparse graphs with high-dimensional node embedding because its kernel design essentially borrows the design principles of graph-processing systems by using excessive high-overhead atomic operations to support node embedding propagation. A similar scalability problem is also observed in Deep Graph Library (DGL) [15], which incorporates an off-the-shelf Sparse-Matrix Multiplication (SpMM) (*e.g.*, *csrmm2* in cuSparse [70]) for simple sum-reduced aggregation [10, 12] and leverages its own CUDA kernel for more complex aggregation scheme with edge attributes [13, 11].

Second, major computation kernels [16, 15] are hard-coded without design flexibility, which is essential to handle diverse application settings with different input graph sizes and node embedding dimensionality. From the high-level interface, users are only allowed to define the way of composing these kernels externally. Users are not allowed to customize kernels internally based on the known characteristics of GNN model architectures, GPU hardware, and graph properties.

3.3 Input Analysis of GNN Applications

In this section, we argue that the GNN input information can guide the system optimization, based on our key observation that different GNN application settings would favor different optimization choices. We introduce two types of GNN input information and discuss their potential performance benefits and extraction methods.

3.3.1 GNN Model Information

While the GNN update phase follows a relatively fixed computing pattern, the GNN aggregation phase shows high diversity. The mainstream aggregation methods of GNNs can be categorized into two types:

The first type is aggregation (*e.g.*, *sum*, and *min*) with only the embeddings of neighbor nodes, as in Graph Convolutional Network (GCN) [10]. For GNNs with this type of aggregation, the common design practice is to reduce the node embedding dimensionality during the update phase (*i.e.*, multiplying the node embedding matrix with the weight matrix) [10, 16, 15] before the aggregation (gather information from neighbor node embedding) at each GNN layer, thereby, largely reducing the data movements during the aggregation. In this case, improving memory locality would be more beneficial, in that more node embeddings can be cached in fast memory (*e.g.*, L1 cache of GPUs) to exploit performance benefits.

The second type is aggregation with special edge features (*e.g.*, weights, and edge vectors that are computed by combining source and target nodes) applied to each neighbor node, as in Graph Isomorphism Network (GIN) [13]. This type of GNN must work on large full-dimensional node embeddings to compute the special edge features at the node aggregation. In this case, the fast memory (*e.g.*, shared memory of GPU Stream-Multiprocessors) is not large enough to exploit memory locality. However, improving

computation parallelization (*e.g.*, workload partitioning along the embedding dimension) would be more helpful, considering that workloads can be shared among more concurrent threads for improving overall throughput.

We illustrate this aggregation-type difference with the mathematical equations for GCN and GIN. With GCN, the output embedding $\mathbf{X}$ is computed as follows:

$$\mathbf{X}' = \hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\mathbf{X}\mathbf{W}, \tag{3.2}$$

where $\hat{\mathbf{D}}$ is the diagonal node degree matrix; $\mathbf{W}$ is the weight matrix; $\hat{\mathbf{A}}$ is the graph adjacency matrix. For GIN, the output embedding $\mathbf{X}$ for each layer is computed as follows:

$$\mathbf{x}_i' = h\left((1+\epsilon)\cdot\mathbf{x}_i + \sum_{j\in\mathcal{N}(i)}\mathbf{x}_j\right) \tag{3.3}$$

where h denotes a neural network, *e.g.*, an MLP, which maps node features x with input embedding dimension and output embedding dimension; ϵ is a configurable/trainable parameter depending on the users' demands or application settings; $\mathcal{N}(i)$ denotes the neighbor IDs of the node i.

Assume we have GCN and GIN with hidden dimension 16, and the input dataset has a node embedding dimension of 128. In the case of GCN, we will first do node update (GEMM[3]-based linear transformation) of the node embedding, thus, at the aggregation, we only need to do aggregation on nodes with hidden dimension 16. In the GIN case, we have to do neighbor aggregation on nodes with 128 dimensions then do node update to linearly transform node embedding from 128 to 16 dimensions. Such an aggregation difference would also lead to different optimization strategies, where GCN would prefer more memory optimization on the small node embeddings while GIN would prefer more computing parallelism on the large node embeddings.

To conclude, the type of aggregation in GNNs should be considered for system-level

[3]General Matrix-Matrix Multiplication.

optimization and it can be obtained by GNNAdvisor's built-in parser of GNN model proprieties.

3.3.2 Graph Information

Node Degree & Embedding Dimensionality: Real-world graphs generally follow the power-law distribution [71] of node degrees. Such distribution already causes workload imbalance in traditional graph processing systems [64, 72, 73]. In GNN aggregation, such workload imbalance would be exacerbated due to the higher dimensionality of the node embeddings if we perform node-centric workload partitioning. Moreover, node embedding would invalidate some cache-based optimizations that are originally applied to graph processing systems, since caches are usually small in size and insufficient to hold enough nodes with their embeddings. For example, in the graph processing scenarios with a scalar attribute for each node, we can improve performance by putting 16×10^3 nodes on the 64KB L1 cache of each GPU thread block. However, in typical GNNs with a 64-dimension embedding for each node, we can only fit 256 nodes on each GPU block's cache.

With node degree and embedding dimensionality information, new optimization opportunities for GNNs may appear because we can estimate the node's workload and its concrete composition based on such input information. If the workload size is dominated by the number of node neighbors (*e.g.*, large node degree), we may customize the design that could concurrently process more neighbors to increase the computing parallelism among neighbors. On the other hand, if the workload size is dominated by node embedding size (*e.g.*, high-dimensional node embedding), we may consider boosting the computing parallelism along the node embedding dimension. Note that the node degree and embedding dimension information can be extracted based on the loaded graph struc-

ture and node embedding vectors. GNNAdvisor manages the GNN workload based on such information (Section 3.4).

Graph Community: Graph community [74, 75, 76] is one key feature of real-world graphs, which describes that a small group of nodes tend to hold "strong" intra-group connections (many edges) while maintaining "weak" connections (fewer edges) with the remaining part of the graph. A motivating example of GNN optimization with graph community structure is shown in Figure 3.3a. Existing node-centric aggregation employed by many graph processing systems [56, 60] is shown in Figure 3.3b, where each node will first load its neighbors and then do aggregation independently. This strategy can achieve great computation parallelism when each neighbor has a lightweight scalar attribute. In this case, the benefit of loading parallelization would offset the downside of duplicate loading of some shared neighbors. However, in GNN computing where node embedding size is large, this node-centric loading would trigger significant unnecessary memory access since the cost of duplicate neighbor loading is now dominant and not offset by per-node parallelism For example, aggregation of node a, b, c, d, and e would load the embeddings of 15 nodes in total and most of these loads are repeated (both node a and b load the same node d during the aggregation). Such loading redundancy is exacerbated with the increase of embedding dimensionality. On the other side, by considering the community structure of real-world graphs, unnecessary data loading for these "common" neighbors can be well reduced (Figure 3.3c), where aggregation only requires loads of 5 distinct nodes.

This idea sounds promising, but the effort to realize its benefits on GPUs is non-trivial. Existing approaches [77, 76] of exploiting the graph communities mainly target CPU platforms with a limited number of parallelized threads and MB-level cache sizes for each thread. Their major goal is to exploit the data locality for every single thread. GPUs, on the other side, are equipped with a massive number of parallel threads and

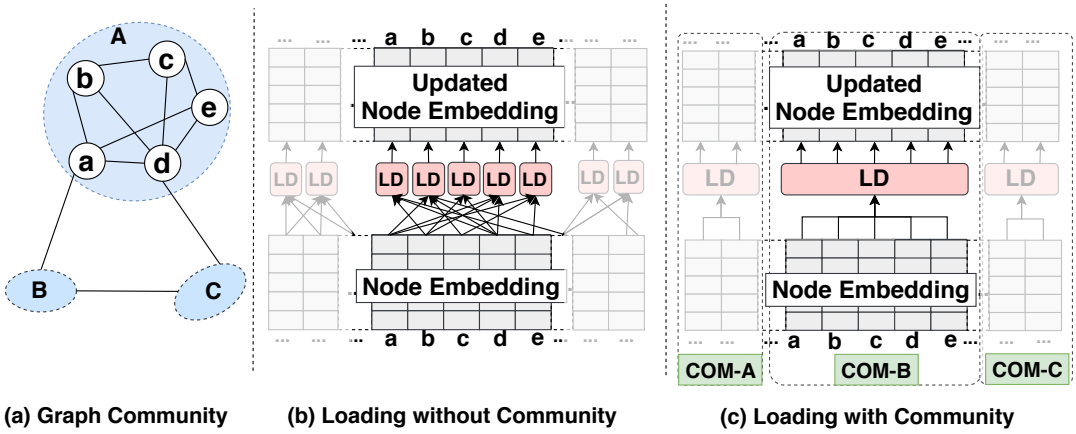


Figure 3.3: Graph community and its potential benefits. Note that "LD": loading operation. "COM": community.

KB-level cache sizes per thread. Therefore, the key to exploiting graph community on GPUs is to effectively exploit the data locality among threads by leveraging the L1 cache. Specifically, we need first capture the communities of a graph and then map such locality from input level (node-ID adjacency) to underlying GPU kernels (thread/warp/block-ID adjacency). The major hardware-level insight is that threads close in their IDs are more likely to share memory and computing resources, thus, improving the data spatial and temporal locality. GNNAdvisor handles all these details through community-aware node renumbering and GNN-specialized memory optimizations (Section 3.5).

3.4 2D Workload Management

GNNs employ a unique space in graph computations, due to the representation of each node by a high-dimensional feature vector (the embedding). GNN workloads grow in two major dimensions: *the number of neighbors* and *the size of the embedding dimension*. GNNAdvisor incorporates an input-driven parameterized 2D workload management tailored for GNNs, including three techniques: *coarse-grained neighbor partitioning*, *fine-grained dimension partitioning*, and *warp-based thread alignment*.

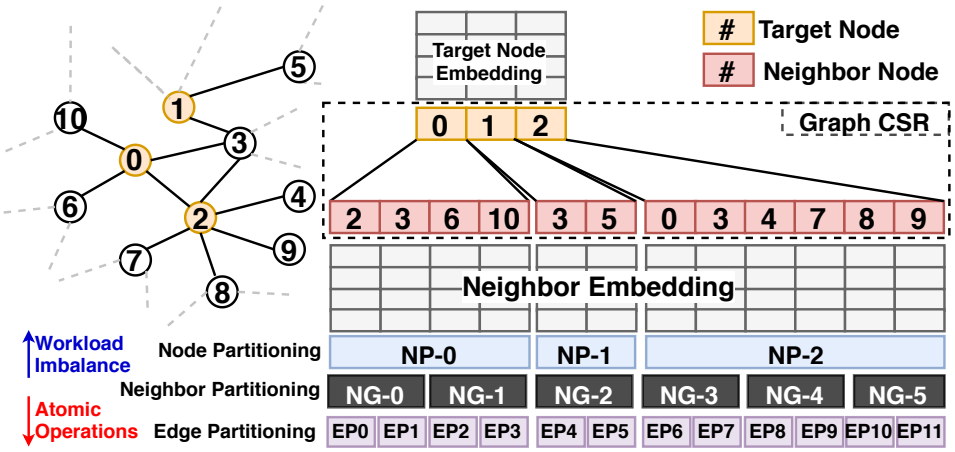


Figure 3.4: Neighbor Partitioning. Note that "NP": Node Partitioning; "EP": Edge Partitioning; "NG": Neighbor Group.

3.4.1 Coarse-grained Neighbor Partitioning

Coarse-grained neighbor partitioning is a novel workload balance technique tailored to GNN computing on GPUs. It aims to tackle the challenge of *inter-node workload imbalance* and *redundant atomic operations*.

Specifically, based on the loaded graph compressed-sparse row (CSR) representation, our coarse-grained neighbor partitioning will first break down the neighbors of a node into a set of equal-sized neighbor groups, and treat the aggregation workload of each neighbor group (NG) as the basic workload unit for scheduling. Figure 3.4 exemplifies an undirected graph and its corresponding neighbor partitioning result. The neighbors of Node-0 are divided into two neighbor groups (NG-0 and NG-1) with a pre-determined group size of 2. Neighbors (Node-3 and Node-5) of Node-1 are covered by NG-2, while the neighbors of Node-2 are spread among NG-{3,4,5}. To support the neighbor group, we introduce two components, the neighbor-partitioning module and the neighbor-partitioning graph store. The former is a lightweight module built on top of the graph loader by partitioning the graph CSR into equal-size groups. Note that each neighbor group only covers the neighbors of one target node for ease of scheduling and synchronization. The neighbor-

partitioning graph store maintains the tuple-based meta-data of each neighbor group, including its IDs, starting and ending position of its neighbor nodes in the CSR representation, and the source node. For example, the meta-data of NG-2 will be stored as (2, 1, (4, 6)), where 2 is the neighbor-group ID, 1 is the target node ID, (4, 6) is the index range of the neighbor nodes in CSR.

The benefits of applying the aggregation based on partitioning neighbors are threefold: 1) compared with the more coarse-grained aggregation based on node/vertex-centric partitioning [60], neighbor partitioning can largely mitigate the size irregularity of the workload units, which would improve GPU occupancy and throughput performance; 2) compared with the more fine-grained edge-centric partitioning (used by existing GNN frameworks, such as PyG [16], for batching and tensorization, and graph processing systems [65, 56] for massive computing parallelization), the neighbor-partitioning solution can avoid the overheads of managing many tiny workload units that might hurt the performance in many ways, such as scheduling overheads and the excessive amount of synchronizations; 3) it introduces a performance-related parameter, **neighbor-group** size (ngs), which is used for design parameterization and performance tuning. Neighbor partitioning works at a coarse granularity of individual neighbor nodes. It can largely mitigate the workload imbalance problem for low-dimension settings. For high-dimensional node embeddings, we employ a fine-grained dimension partitioning discussed in the next subsection to further distribute workloads of each neighbor group to threads. Note that when the number of neighbors is not divisible by the neighbor group size, it will raise neighbor-group imbalance. Such irregularity can be amortized by setting the neighbor-group size to a small number (*e.g.*, 3).

3.4.2 Fine-grained Dimension Partitioning

GNN distinguishes itself from traditional graph algorithms in its computation on the node embedding. To explore the potential acceleration parallelism along this dimension, we leverage a fine-grained dimension partitioning to further distribute the workloads of a neighbor group along the embedding dimension to improve aggregation performance. As shown in Figure 3.5, the original neighbor-group workloads are evenly distributed to 11 consecutive threads, where each thread manages the aggregation along one dimension independently (*i.e.*, accumulation of all neighbor node embeddings towards the target node embedding). If the dimension size is larger than the number of working threads, more iterations would be required to finish the aggregation.

There are two major reasons for using dimension partitioning. First, it can accommodate a more diverse range of embedding dimension sizes. We can either increase the number of concurrent dimension workers or enable more iterations to handle the dimension variation flexibly. This is essential for modern GNNs with increasingly complicated model structures and different sizes of embedding dimension. Second, it introduces another performance-related parameter – the number of working threads (**dimension-worker** (*dw*)) for design customization. The value of this parameter can help to balance the thread-level parallelism and the single thread efficiency (*i.e.*, computation workload per thread).

3.4.3 Warp-based Thread Alignment

While the above two techniques answer how we balance GNN workloads logically, how to map these workloads to underlying GPU hardware for efficient execution is still unresolved. One straightforward solution is to assign consecutive threads to concurrently process workloads from different neighbor groups (Figure 3.6a). However, different behav-

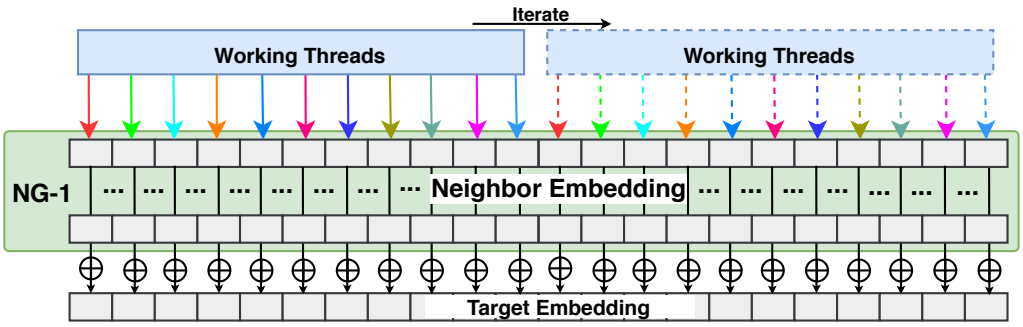


Figure 3.5: Dimension Partitioning. $\bigoplus$: Accumulated add.

iors (*e.g.*, data manipulation and memory access operations) among these threads would result in thread divergence and GPU underutilization. Threads from the same warp proceed in a single-instruction-multiple-thread (SIMT) fashion and the warp scheduler can only serve one type of instruction per cycle. Therefore, different threads have to wait for their turn for execution until the Stream-Multiprocessor (SM) warp scheduler issues their corresponding instructions.

To tackle this challenge, we introduce a warp-aligned thread mapping in coordination with our neighbor and dimension partitioning to systematically capitalize on the performance benefits of balanced workloads. As shown in Figure 3.6b, each warp will independently manage the aggregation workload from one neighbor group. Therefore, the execution of different neighbor groups (*e.g.*, NG-0 to NG-5) can be well parallelized without inducing warp divergence. There are several benefits in employing warp-based thread alignment. First, inter-thread synchronization (*e.g.*, atomic operations) can be minimized. Threads of the same warp are working on different dimensions of the same neighbor group, thus no conflicts occur for either global or shared memory accesses by threads from the same warp.

Second, the workload of a single warp is reduced and different warps will process more balanced workloads. Therefore, more small warps can be managed flexibly by SM warp schedulers to improve overall parallelism. Considering the unavoidable global memory

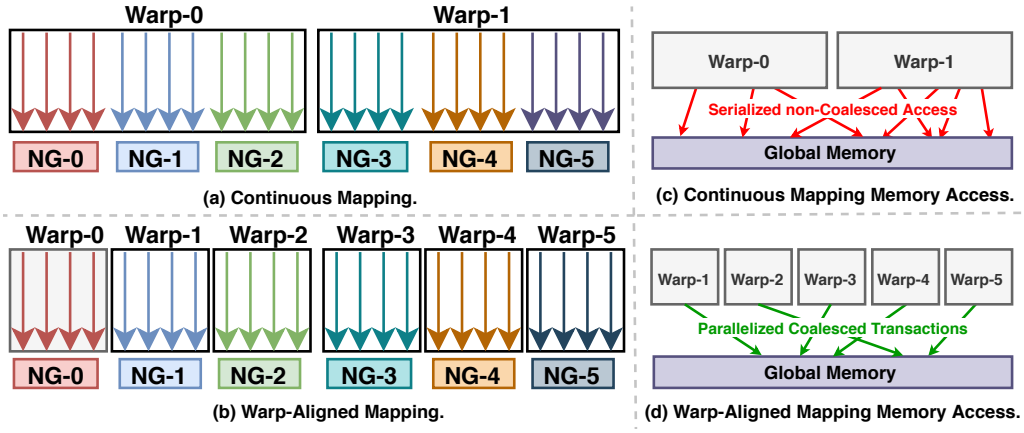


Figure 3.6: Warp-based Thread Alignment.

access of each warp during aggregation, increasing the number of warps can improve SM occupancy to hide latency. Third, memory access can be coalesced. Threads with consecutive IDs from the same warp will access continuous memory addresses in global memory for node embeddings. Therefore, compared with continuous thread mapping (Figure 3.6c), warp-aligned thread mapping can merge memory requests from the same warp into one global memory transaction (Figure 3.6d).

3.5 Specialized Memory Optimization

To further exploit the benefits of 2D workload, we introduce GNN-specialized memory optimizations, *community-aware node renumbering* and *warp-aware memory customization*.

3.5.1 Community-aware Node Renumbering

To explore the performance benefits of graph community (Section 3.3.2), we incorporate lightweight node renumbering by reordering node IDs to improve the temporal/spatial locality during GNN aggregation without compromising output correctness. The

key idea is that the proximity of node IDs would project to the adjacency of computing units on GPU where they get processed. In GNNAdvisor, our 2D workload management assigns neighbor groups of a node to consecutive warps based on their node ID. If two nodes are assigned with consecutive IDs, their corresponding neighbor groups (warps) would be close to each other in their warp IDs as well. Thus, they are more likely to be scheduled closely on the same GPU SM with a shared L1 cache to improve the data locality on loaded common neighbors. To apply node renumbering effectively, two key questions must be addressed.

When to apply: While graph reordering provides potential benefits for performance, we still need to figure out *what kind of graph would benefit from such reordering optimization*. Our key insight is that for graphs already in a shape approximating block-diagonal pattern in their adjacency matrix (Figure 3.7a), reordering could not bring more locality benefits, since nodes within each community are already close to each other in terms of their node-IDs. For graphs with a more irregular shape (Figure 3.7b), where edge connections are distributed among nodes with an irregular pattern, the reordering could bring notable performance improvement (up to 2× speedup, later discussed in Section 5.4.3). To this end, we propose a new metric – *Averaged Edge Span* (AES), to determine whether it is beneficial to conduct a graph reordering.

$$\textbf{AES} = \frac{1}{\#E} \sum_{(src_{id}, trg_{id}) \in E} |src_{id} - trg_{id}| \tag{3.4}$$

where E is the edge set of the graph; $\#E$ is the number of total edges; src_{id} and trg_{id} are the source and target node IDs of each edge. Computing AES is lightweight and can be done on-the-fly during the initial graph loading. Our profiling of a large corpus of graphs also shows that when $\sqrt{AES} > \lfloor \frac{\sqrt{\#N}}{100} \rfloor$ node numbering is more likely to improve runtime performance.

How to apply: We leverage Rabbit Reordering [58], which is a fully parallelized and

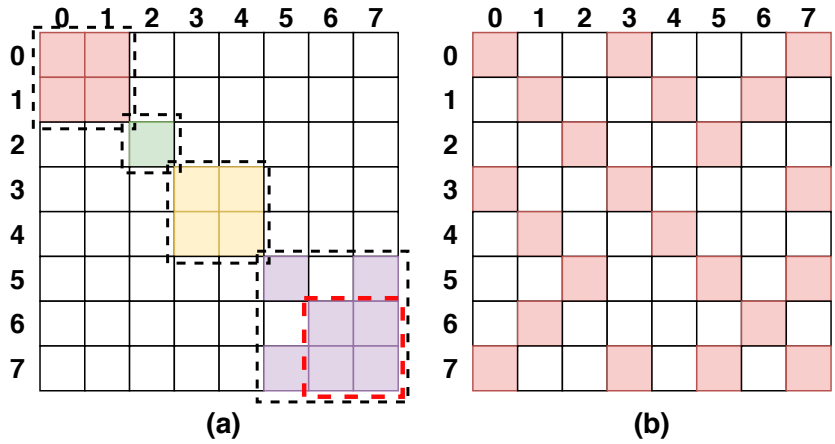


Figure 3.7: Graph Edge Connection Patterns. Note that each colored square represents the edge between two nodes. Different colors in (a) represent edges from different communities. The red dot-line box indicates the sub-community.

low-cost graph reordering technique. Specifically, it first maximizes the graph modularity by hierarchically merging edges and clustering nodes. And it then generates node order within each cluster through DFS traversal. Rabbit Reordering has also been proved to outperform other graph clustering approaches [22, 78, 37, 38, 39], including Community-based methods, such as METIS [22], and BFS-based methods, such as Reverse Cuthill-McKee (RCM) [39]) in terms of better quality (data locality) of the captured graph communities, the ease of parallelization, and performance. More importantly, Rabbit Reordering can capture the graph communities hierarchically (*i.e.*, a set of smaller sub-communities are included in a larger community, as exemplified in Figure 3.7a). Such communities at different levels of granularities would be a good match for the GPU cache hierarchy, where smaller sub-communities (occupying one SM) can enjoy the data locality benefit from the L1 cache, while larger communities (occupying multiple SMs) can enjoy the data locality from the larger L2 cache. We quantitatively discuss such a locality benefit in Section 4.7.2.

3.5.2 Warp-aware Memory Customization

Existing works [16, 56] utilize a large number of global memory accesses for reading and writing the embedding and a large number of atomic operations for aggregation (a reduction operation). However, this approach leads to heavy overhead and fails to exploit the potential benefits from shared memory. In particular, when aggregating on a target node with k neighbor groups (each has ngs neighbors with Dim-Dimensional embeddings) into a Dim-dimensional embedding, it involves $O(k \cdot ngs \cdot Dim)$ atomic operations and $O(k \cdot ngs \cdot Dim)$ global memory accesses.

By contrast, we propose a warp-centric shared memory optimization technique. Our key insight is that by customizing shared memory layout according to the block-level warp organization pattern (Figure 3.7), we can significantly reduce the number of atomic operations and global memory access. First of all, we reserve a shared memory space ($4 \times Dim$ bytes for floating-point embeddings) for the target node of each neighbor group (warp), such that the threads from a warp can cache the intermediate results of reduction in shared memory. Later on, within a thread block, we designate only one warp (called *leader*) for copying the intermediate results of each target node to global memory considering that neighbors of each node can be spread across different warps. The detailed customization procedure is described in Algorithm 2. Specifically, each warp (maintained in *warpPtr*) has three properties: *nodeSharedAddr* (a shared memory address for the aggregation result of a neighbor-group), *nodeID* (the ID of the target node), and *leader* (a boolean flag indicating whether the current warp is a leader warp for flushing out the result from the shared memory to the global memory). The major customization routine (Line 4 to Line 22) handles different warps based on their index position relative to thread blocks. Note that such a shared memory customization is low-cost and is done only once on-the-fly with the regular graph initialization process before the GPU kernel

Algorithm 2 Warp-aware Memory Customization.

```
   ▷ Compute #neighbor-groups (#warps).
1: warpNum = neighborGroups = computeGroups(ngs);
   ▷ Compute the number of warps per thread block.
2: warpPerBlock = floor(threadPerBlock/threadPerWarp)
   ▷ Initialize tracking variables.
3: cnt = 0; local_cnt = 0; last = 0;
4: while cnt < warpNum do
      ▷ Warp in the front of a thread block.
5:    if cnt % warpPerBlock == 0 then
6:       warpPtr[cnt].nodeSharedAddr = local_cnt × Dim;
7:       last = warpPtr[cnt].nodeID;
8:       warpPtr[cnt].leader = true;
      ▷ Warp in the middle of a thread block.
9:    else
         ▷ Warp with the same target node as
           its predecessor warp.
10:      if warpPtr[cnt].nodeID == last) then
11:         warpPtr[cnt].nodeSharedAddr = local_cnt × Dim;
         ▷ Warp with the different target node as
           its predecessor warp.
12:      else
13:         local_cnt++;
14:         warpPtr[cnt].nodeSharedAddr = local_cnt × Dim;
15:         last = warpPtr[cnt].nodeID;
16:         warpPtr[cnt].leader = true;
17:      end if
18:   end if
      ▷ Next warp belongs to a new thread block.
19:   if (++cnt)%warpPerBlock == 0 then
20:      local_cnt = 0;
21:   end if
22: end while
```

execution.

In our design, when a target node with k neighbor groups (each has ngs neighbors with Dim-dimensional embeddings), it involves $O(Dim)$ atomic operations and $O(Dim)$ global memory accesses. To this end, we can save the atomic operations and global memory access by $(k \cdot ngs)\times$, thus significantly accelerating the aggregation operations. Here, we treat ngs as a hyper-parameter to balance memory access efficiency and computation parallelism, and we further discuss its value selection in Section 3.6.

3.6 Design Optimization

The parameters in our GPU kernel configurations can be tuned to accommodate various GNN models with graph data sets. But it is not yet known how to automatically select the parameters which can deliver the optimal performance. In this section, we introduce the analytical model and the auto parameter selection in the **`Decider`** of GNNAdvisor.

Analytical Modeling: The performance/resource analytical model of GNNAdvisor has two variables, workload per thread (WPT), and shared memory usage per block ($SMEM$).

$$\textbf{WPT} = ngs \times \frac{Dim}{dw}, \quad \textbf{SMEM} = \frac{tpb}{tpw} \times Dim \times FloatS \tag{3.5}$$

where ngs and dw is the neighbor-group and dimension-worker size (Section 3.4.2), respectively; Dim is the node embedding dimension; $IntS$ and $FloatS$ are both 4-byte on GPUs; tpb is the thread-per-block and tpw is the thread-per-warp; tpw is 32 for GPUs, while tpb is selected by users.

Parameter Auto Selection: To determine the value of the ngs and dw, we follow two steps. First, we determine the value of dw based on tpw (hardware constraint) and Dim (input property), as shown in Equation 3.6. Note that we develop this equation by profiling different datasets and GNN models.

$$dw = \begin{cases} tpw & Dim \geq tpw \\ \frac{tpw}{2} & Dim < tpw \end{cases} \tag{3.6}$$

Second, we determine the value of ngs based on the selected dw and the user-specified tpb. The constraints include making $WPT \approx 1024$ and $SMEM \leq SMEMperBlock$. Note that $SMEMperBlock$ is 48KB to 96KB on modern GPUs [79, 80]. Across different GPUs, even though the number of CUDA cores and global memory bandwidth would

be different, the single-thread workload capacity (measured by WPT) remains similar. tpb is usually chosen as a power of 2 but less than or equal 1024. Our insight based on micro-benchmarking and previous literature [81] shows that smaller blocks (1 to 4 warps, *i.e.*, $32 \leq tpb \leq 128$) can improve SM warp scheduling flexibility and avoid tail effects, thus leading to higher GPU occupancy and throughput. We further demonstrate the effectiveness of our analytical model in Section 5.4.3.

3.7 Evaluation

In this section, we comprehensively evaluate GNNAdvisor in terms of the performance and adaptability on various GNN models, graph datasets, and GPUs.

3.7.1 Experiment Setup

Benchmarks: We choose the two most representative GNN models widely used by previous work [15, 16, 14] on node classification tasks to cover different types of aggregation. 1) Graph Convolutional Network (GCN) [10] is one of the most popular GNN model architectures. It is also the key backbone network for many other GNNs, such as GraphSAGE [12], and differentiable pooling (Diffpool) [41]. Therefore, improving the performance of GCN will also benefit a broad range of GNNs. For GCN evaluation, we use the setting: *2 layers with 16 hidden dimensions*, which is also the setting from the original paper [10]. 2) Graph Isomorphism Network (GIN) [13]. GIN differs from GCN in its aggregation function, which weighs the node embedding values from the node itself. In addition, GIN is also the reference architecture for many other advanced GNNs with more edge properties, such as Graph Attention Network (GAT) [11]. For GIN evaluation, we use the setting: *5 layers with 64 hidden dimensions*, which is the setting used in the original paper [13].

Baselines: we choose several baseline implementations for comparison. *1) Deep Graph Library (DGL)* [15] is the state-of-the-art GNN framework on GPUs, which is built upon the famous tensor-oriented platform – Pytorch. DGL significantly outperforms the other existing GNN frameworks [16] over various datasets on many mainstream GNN architectures. Therefore, we make an in-depth comparison with DGL in our evaluation; *2) Pytorch-Geometric (PyG)* [16] is another GNN framework in which users can define their edge convolutions when building customized GNN aggregation layers; *3) NeuGraph* [14] is a dataflow-centered GNN system on GPUs built on Tensorflow [68]; *4) Gunrock* [56] is the GPU-based graph processing framework with state-of-the-art performance on traditional graph algorithms (*e.g.*, PageRank).

Datasets: We cover all three types of datasets, which have been used in previous GNN-related work [15, 16, 14]. Type I graphs are the typical datasets used by previous GNN algorithm papers [10, 13, 12]. They are usually small in the number of nodes and edges, but rich in node embedding information with high dimensionality. Type II graphs [42] are the popular benchmark datasets for graph kernels and are selected as the built-in datasets for PyG [16]. Each dataset consists of a set of small graphs, which only have intra-graph edge connections without inter-graph edge connections. Type III graphs [82, 10] are large in terms of the number of nodes and edges. These graphs demonstrate high irregularity in structure, which is challenging for most of the existing GNN frameworks. Details of these datasets are listed in Table 5.4.

Platforms & Metrics: We implement GNNAdvisor's backend with C++ and CUDA C and its front-end with Python. Our major evaluation platform is a server with an 8-core 16-thread Intel Xeon Silver 4110 CPU [83] and a Quadro P6000 [80] GPU. Besides, we use Tesla V100 [79] GPU on the DGX-1 system [84] to demonstrate the generality of GNNAdvisor. Runtime parameters of different input settings are optimized by GNNAdvisor **`Decider`**. To measure the performance speedup, we calculate the averaged

Table 3.1: Datasets for Evaluation.

Type	Dataset	#Vertex	#Edge	Dim.	#Class
I	Citeseer	3,327	9,464	3,703	6
	Cora	2,708	10,858	1,433	7
	Pubmed	19,717	88,676	500	3
	PPI	56,944	818,716	50	121
II	PROTEINS_full	43,471	162,088	29	2
	OVCAR-8H	1,890,931	3,946,402	66	2
	Yeast	1,714,644	3,636,546	74	2
	DD	334,925	1,686,092	89	2
	TWITTER-Partial	580,768	1,435,116	1,323	2
	SW-620H	1,889,971	3,944,206	66	2
III	amazon0505	410,236	4,878,875	96	22
	artist	50,515	1,638,396	100	12
	com-amazon	334,863	1,851,744	96	22
	soc-BlogCatalog	88,784	2,093,195	128	39
	amazon0601	403,394	3,387,388	96	22

latency of 200 end-to-end inference (forward propagation) or training (forward+backward propagation).

3.7.2 Compared with DGL

In this section, we first conduct a detailed experimental analysis and comparison with DGL on GNN inference, then extend our comparison for GNN training. As shown in Figure 3.8, GNNAdvisor achieves 4.03× and 2.02× speedup on average compared to DGL [15] over three types of datasets for GCN and GIN on inference, respectively. We next provide detailed analysis and give insights for each type of datasets.

Type I Graphs: The performance improvement against DGL is significantly higher for GCN (on average 6.45×) than GIN (on average 1.17×). The major reason is their different GNN computation patterns. For GCN, node dimension reduction (DGEMM) is always placed before aggregation. This largely reduce data movement and thread synchronization overheads during the aggregation phase, which could gain more benefits from GNNAdvisor's 2D workload management and specialized memory optimization for data locality improvements. GIN, on the other side, has aggregation phase that must

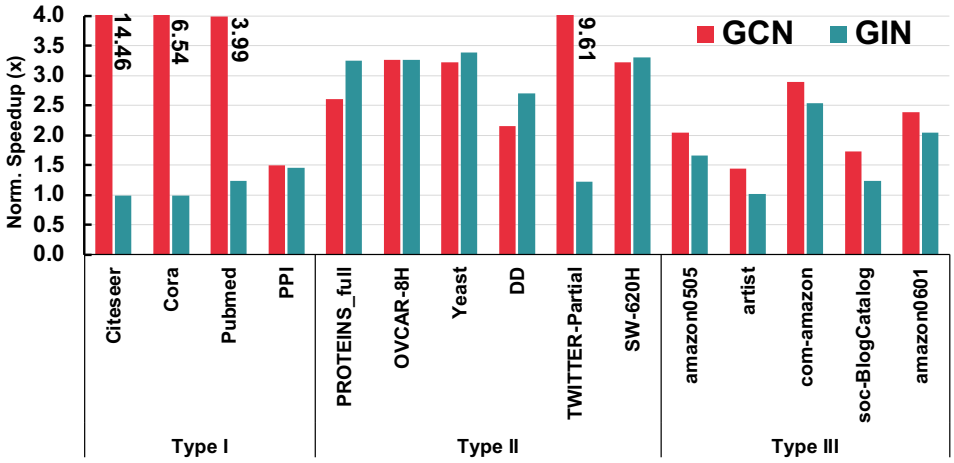


Figure 3.8: Inference speedup (×) over DGL on GCN and GIN.

be finished before the node dimension reduction. Thus, it cannot avoid high-volume memory access and data movements during the aggregation phase. Therefore, it gets lower benefits from the data locality and the shared memory on GPUs for fast and low-overhead memory access. However, our fine-grained dimension partitioning can still handle these high-dimensional cases effectively.

Type II Graphs: Performance shows less difference between GCN (4.02×) and GIN (2.86×) on the same datasets except for *TWITTER-Partial*, which has the highest node embedding dimension (1323) in Type II graphs. It is worth noticing that the speedup for GIN is consistently better compared with Type I. There are two major reasons: 1) node feature dimension is much lower (average 66.5, excluding *TWITTER-Partial*) versus Type I (average 1421), which can gain more performance benefits from data spatial and temporal locality of our specialized memory optimizations; 2) Type II graphs intrinsically have good locality in their graph structure. The reason is that Type II datasets consist of small graphs with very dense intra-graph connections but no inter-graph edges, plus nodes within each small graph are assigned with consecutive IDs. Therefore, the performance gains of such graph-structure locality can be scaled up when combined with GNNAdvisor's efficient workload and memory optimizations.

Type III Graphs: The speedup is also evident (average 2.10× for GCN and average 1.70× for GIN) on graphs with a large number of nodes and edges, such as *amazon0505*. The reason is the high overhead inter-thread synchronization and global memory access can be well reduced through our 2D workload management and specialized memory optimization. Besides, our community-aware node renumbering further facilitates an efficient workload sharing among adjacent threads (working on a group of nodes) through improving the data spatial/temporal locality. On the dataset *artist*, which has the smallest number of nodes and edges within Type III, we notice a lower performance speedup for GIN. And we find that the *artist* dataset has the highest standard deviation of graph community sizes within Type III graphs, which makes it challenging to 1) use the group community information to capture the node temporal and spatial locality in the GNN aggregation phase, and 2) capitalize on the performance benefits of using such a community structure for guiding system-level optimizations (*e.g.*, warp-aligned thread mapping and shared memory customization) on GPUs, which have a fixed number of computation and memory units within each block/SM.

Kernel Metrics: For detailed kernel metrics analysis, we utilize NVProf [85] to measure two performance-critical (computation and memory) CUDA kernel metrics: *Stream Processor (SM) efficiency* and *Cache (L1 + L2 + Texture) Hit Rate*. GNNAdvisor achieves on average 24.47% and 12.02% higher SM efficiency compared with DGL for GCN and GIN, respectively, which indicates that our 2D workload management can strike a good balance between the single-thread efficiency and the multi-thread parallelism that are crucial to the overall performance improvement. GNNAdvisor achieves on average 75.55% and 126.20% better cache hit rate compared with DGL for GCN and GIN, correspondingly, which demonstrates the benefit of specialized memory optimizations.

Training Support: We also evaluate the training performance of GNNAdvisor on

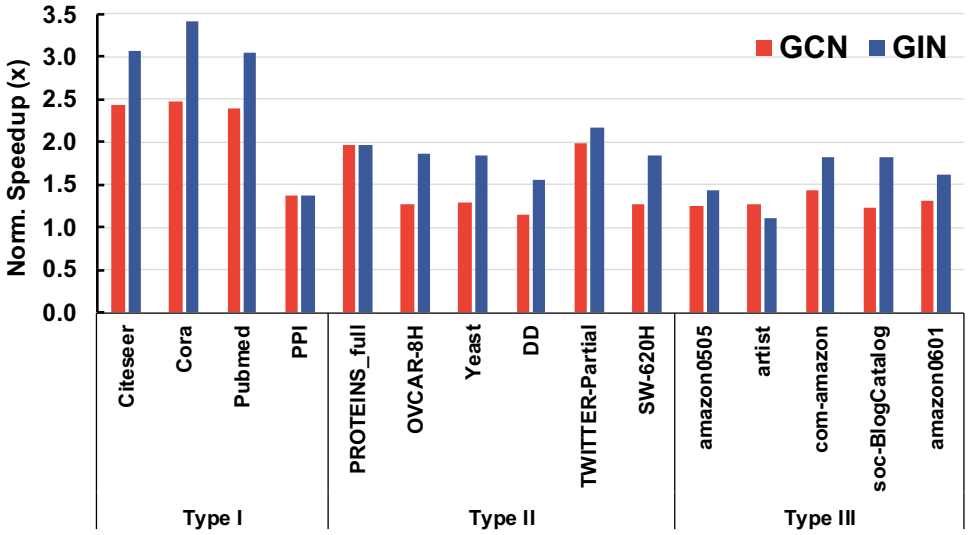


Figure 3.9: Training speedup (×) over DGL on GCN and GIN.

all three types of datasets compared with the DGL on both GCN and GIN. Compared with inference, training is more challenging, since it involves more intensive computation with the forward value propagation and the backward gradient propagation, both of which heavily rely on the underlying graph aggregation kernel for computation. As shown in Figure 3.9, GNNAdvisor consistently outperforms the DGL framework with average 1.61× and average 2.00× speedup on GCN and GIN, respectively, which shows the strength of our input-driven optimizations. The key difference between training and inference of GNNs is two-fold: First, backpropagation is needed in training. This step benefits from our improvements, as the backpropagation step is similar to the forward computation during the inference, and all the proposed methods are still beneficial; Second, training incurs extra memory and data movement overheads for storing/accessing the activations of the forward pass until gradients can be propagated back.

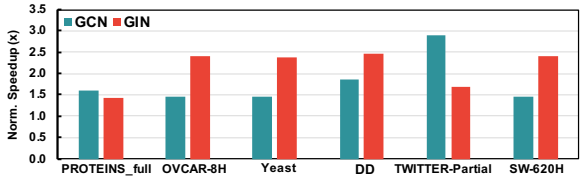


Figure 3.10: Training speedup (×) over PyG on GCN and GIN.

Table 3.2: Latency (ms) comparison with NeuGraph (NeuG).

Dataset	NeuG (ms)	Ours (ms)	Speedup
reddit-full	2460	599.69	4.10×
enwiki	1770	443.00	3.99×
amazon	1180	474.57	2.48×

3.7.3 Compared with other Frameworks

We compare with DGL on all input settings, since DGL is the overall best-performance GNN framework. In this section, we further compare GNNAdvisor with three other representative GNN computing frameworks on their best settings.

Compared with PyG: As shown in Figure 3.10, GNNAdvisor can outperform PyG with 1.78× and 2.13× speedup on average for GCN and GIN, respectively. For GCN, GNNAdvisor achieves significant speedup on datasets with high-dimensional node embedding, such as *TWITTER-Partial*, through 1) node dimension reduction before aggregation and 2) workload sharing among neighbor partitions and dimension partitions. For GIN, GNNAdvisor reaches 2.45× speedup on datasets with a higher average degree, such as *DD*, since GNNAdvisor can effectively distribute the workload of each node along their embedding dimension to working threads while balancing the single-thread efficiency and inter-thread parallelism. PyG, however, achieves inferior performance because 1) it has poor thread management in balancing workload and controlling synchronization overhead; 2) it heavily relies on the scatter-and-gather kernel, which lacks flexibility.

Compared with NeuGraph: For a fair end-to-end training comparison with NeuGraph that has not open-sourced its implementation and datasets, we 1) use the GPU

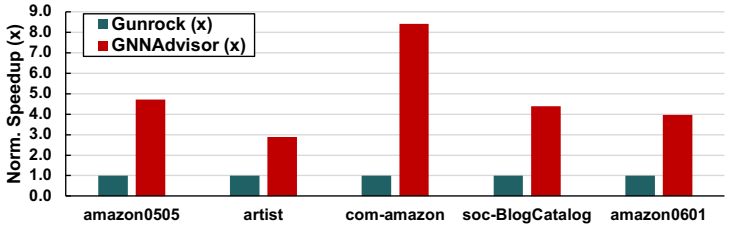
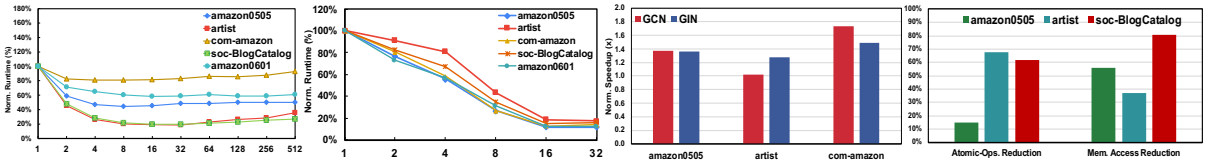


Figure 3.11: Speedup (×) comparison with Gunrock.

(Quadro P6000 [80]) that is comparable with the GPU of NeuGraph (Tesla P100 [86]) in performance-critical factors, such as GPU architecture (both have the Pascal architecture) and the number of CUDA cores; 2) use the same set of inputs as NeuGraph on the same GNN architecture [14]; 3) use the datasets that are presented in their paper and are also publicly available. As shown in Table 3.2, GNNAdvisor outperforms NeuGraph with a significant amount of margin (1.3× to 7.2× speedup) in terms of computation and memory performance. NeuGraph relies on general GPU kernel optimizations and largely ignores the input information. Moreover, the optimizations in NeuGraph are built-in and fixed inside the framework without performance tuning flexibility. In contrast, GNNAdvisor leverages GNN-featured GPU optimizations and demonstrates the key contribution of input insights for system optimizations.

Compared with Gunrock: We make a performance comparison between GNNAdvisor and Gunrock [56] on a single neighbor aggregation kernel of GNNs (*i.e.*, the Sparse-Matrix Dense-Matrix Multiplication (SpMM)) over the Type III graphs. As shown in Figure 3.11, GNNAdvisor outperforms Gunrock with 2.89× to 8.41× speedup. There are two major reasons behind such a evident performance improvement on the sparse GNN computation: 1) Gunrock focuses on graph-algorithm operators (*e.g.*, frontier processing) but lacks efficient support for handling high-dimensional node embedding; 2) Gunrock leverages generic optimizations without considering the input differences, thus, losing the adaptability for handling different GNN inputs efficiently.



(a) (b) (c) (d)

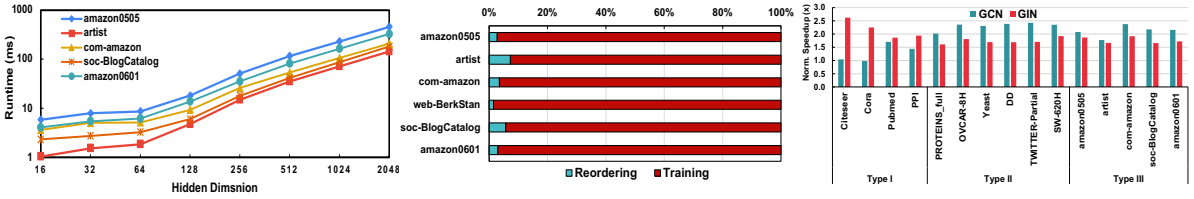
Figure 3.12: Optimization Analysis. (a) Normalized latency as the neighbor group size (ngs) grows (latency at $ngs = 1$ is set as 100%); (b) Normalized latency as the number of dimension workers grows (latency at $dw = 1$ is set as 100%); (c) Normalized speedup when using node renumbering compared to without renumbering; (d) Normalized GPU kernel metrics when using block-level optimizations compared to without block-level optimizations.

3.7.4 Optimization Analysis

In this section, we explore and analyze the optimizations used in Sections 3.4 and 3.5 in detail.

Neighbor partitioning: From Figure 4.10a, we can see that with the increase of the neighbor-group size, the running time of GNNAdvisor will first decrease. The increase of the neighbor-group size saturates the computation capability of each thread meanwhile improving the data locality and reducing the number of atomic operations (*i.e.*, inter-thread synchronization overhead). However, when the neighbor-group size becomes larger than a certain threshold (*e.g.*, 32 for the *artist* dataset), each thread reaches its computation capacity upper bound, and further increasing the neighbor-group size offers no more performance benefit instead increases the overall latency.

Dimension partitioning: As shown in Figure 4.10b, the dimension worker impact is more evident in performance compared with the neighbor-group size at the range from 1 to 16. When the number of dimension worker increases from 16 to 32, the runtime performance shows very minor difference due to the already balanced single-worker efficiency and multi-worker parallelism. Therefore, further increase the number of dimension workers brings no more benefits.



(a) (b) (c)

Figure 3.13: Additional Studies. (a) Latency (ms) analysis as the hidden dimension grows on GCN; (b) Overhead (%) analysis for node renumbering; (c) Speedup ($\times$) on Tesla V100 over Quadro P6000 (set as $1\times$).

Node renumbering: We demonstrate the benefit of node renumbering by profiling Type III datasets for GCN and GIN. As shown in Figure 4.10c, renumbering nodes within a graph can bring up to 1.74× and 1.49× speedup for GCN and GIN, respectively. The major reason is that our community-aware node renumbering can increase the data spatial and temporal locality during GNN aggregation.

To quantify such locality benefits, we extract the detailed GPU kernel metric – memory access in terms of read and write bytes from DRAM for illustration. Our CUDA kernel metric profiling results show that node renumbering can effectively reduce the memory access overhead (on average 40.62% for GCN and 42.33% for GIN) during the runtime since more loaded node embeddings are likely to be shared among the nodes with consecutive IDs. We also notice one input case that benefits less from our optimization – *artist*, since 1) the community size inside *artist* displays a large variation (high standard deviation), making it challenging to capture the neighboring adjacency and locality; 2) such a variation hurdles system-level (computation and memory) optimizations to effectively capitalize on the locality benefits of renumbering.

Block-level optimization: We show the optimization benefits of our block-level optimization (including warp-aligned thread mapping, and warp-aware shared memory customization). We analyze two kernel metrics (*atomic operations reduction* and *DRAM access reduction*) on three large graphs for illustration. As shown in Figure 4.10d, GN-

NAdvisor can effectively reduce the atomic operations and DRAM memory access by an average 47.85% and 57.93%. This result demonstrates 1) warp-aligned thread mapping based on neighbor partitioning can effectively reduce a large portion of atomic operations; 2) warp-aware shared memory customization can avoid a significant amount of global memory access.

3.7.5 Additional Studies

Hidden dimensions of GNN: In this experiment, we analyze the impact of the GNN architecture in terms of the size of the hidden dimension for GCN and GIN. As shown in Figure 3.13a, we observe that with the increase of hidden dimension of GCN, the running time of GNNAdvisor is also increased due to more computation (*e.g.*, additions) and memory operations (*e.g.*, data movements) during the aggregation phase and a larger size of the node embedding matrix during the node update phase. Meanwhile, we also notice that GIN shows a larger latency increase versus GCN, mainly because of the number of layers (2-layer GCN *vs.* 5-layer GIN) that make such a difference more pronounced.

Overhead analysis: Community-aware node renumbering is the major source of overhead for leveraging GNN input information, and other parts are negligible. Here as a case study, we evaluate its overhead on the training phase of GCN on Type III graphs, given the optimization decision from our GNNAdvisor **`Decider`** (as discussed in Section 3.5). Here we use training for illustration; inference in a real GNN application setting would also use the same graph structure many times [12, 10, 10] with different node embeddings inputs. As shown in Figure 3.13b, node-renumbering overhead is consistently small (average 4.00%) compared with overall training time. We thus conclude that such one-time overhead can be amortized over GNN running time, which demonstrates its applicability in real-world GNN applications.

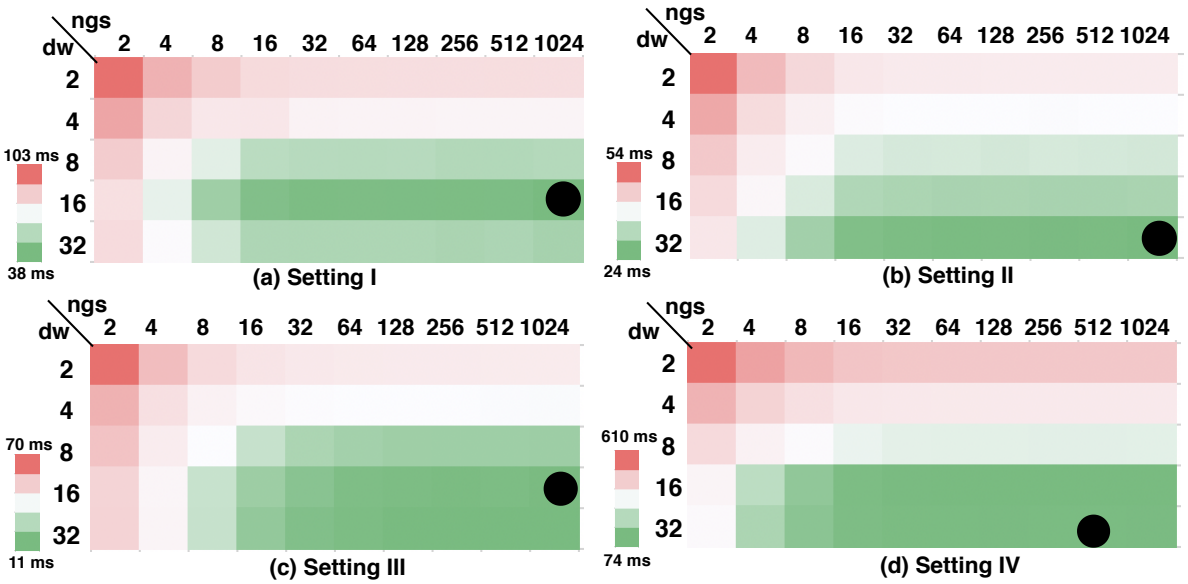


Figure 3.14: Parameter Selection for Four Settings. Note that the solid-black dot indicates the parameter (*dw* and *ngs*) selected by GNNAdvisor **`Decider`** based on analytical modeling.

Performance on Tesla V100: To demonstrate the potential of GNNAdvisor in the modern data-center environment, we showcase the performance of GNNAdvisor on an enterprise-level GPU – Tesla V100 [79]. As shown in Figure 3.13c, GNNAdvisor can scale well towards such a high-end device, which can achieve 1.97× and 1.86× speedup compared with P6000 for GCN and GIN due to more computation resources (*e.g.*, 2.6× SMs, and 1.33× CUDA cores, and 1.13× throughput performance) and higher memory bandwidth (*e.g.*, 2.08× peak memory bandwidth). This comparison shows that GNNAdvisor well adapts towards more advanced GPU hardware for seeking better performance. We also foresee that our current work of GNNAdvisor can be extended to the multi-GPU or distributed data center, benefiting overall performance by improving single GPU efficiency.

Parameter selection: To show the effectiveness of our analytical modeling in kernel parameter selection, we consider four different settings: I: *amazon0505* on GCN at

P6000 GPU as our base setting; II: *amazon0505* GCN on V100 to demonstrate device adaptation; III: *amazon0505* and *soc-BlogCatalog* on P6000 to demonstrate adaptation to different datasets; IV: *amazon0505* on GIN at P6000 to demonstrate adaptation to a different GNN model architectures. As shown in Figure 4.11, our parameter selection strategy can pinpoint the optimal low-latency design for the above four settings. This demonstrates the effectiveness of our analytical modeling in assisting parameter selection to optimize the performance of GNN computation.

Chapter 4

MGG: Accelerating Graph Neural Networks with Fine-Grained Intra-Kernel Communication-Computation Pipelining on Multi-GPU Platforms

The increasing size of input graphs for graph neural networks (GNNs) highlights the demand for using multi-GPU platforms. However, existing multi-GPU GNN systems optimize the computation and communication individually based on the conventional practice of scaling dense DNNs. For irregularly sparse and fine-grained GNN workloads, such solutions miss the opportunity to jointly schedule/optimize the computation and communication operations for high-performance delivery.

To this end, we propose **MGG** [1] , a novel system design to accelerate full-graph

[1]Published at USENIX OSDI'23. USENIX permits authors to retain their ownership of the copyrights

GNNs on multi-GPU platforms. The core of MGG is its novel dynamic software pipeline to facilitate fine-grained computation-communication overlapping within a GPU kernel. Specifically, MGG introduces GNN-tailored pipeline construction and GPU-aware pipeline mapping to facilitate workload balancing and operation overlapping. MGG also incorporates an intelligent runtime design with analytical modeling and optimization heuristics to dynamically improve the execution performance. Extensive evaluation reveals that MGG outperforms state-of-the-art full-graph GNN systems across various settings: on average 4.41×, 4.81×, and 10.83× faster than DGL, MGG-UVM, and ROC, respectively.

4.1 Introduction

Over the recent years, graph-based deep learning has attracted lots of attention from the research and industry communities. Among various graph-learning methods, graph neural network (GNN) [10, 13, 11] gets highlighted most due to its success in many deep learning tasks (*e.g.*, node feature vector (embedding) generation for node classification [45, 46, 47] and link prediction [48, 49, 50]). GNNs consist of several layers, where layer $k+1$ computes the embedding for a node v based on the embeddings at the previous layer k ($k \geq 0$) by applying

$$a_v^{(k+1)} = \mathbf{Aggregate}^{(k+1)}(h_u^{(k)} | u \in \mathbf{N}(v) \cup h_v^{(k)})$$
$$h_v^{(k+1)} = \mathbf{Update}^{(k+1)}(a_v^{(k+1)})$$

where $h_v^{(k)}$ is the embedding of node v at layer k. The *Aggregate* function accumulates neighbors'($\mathbf{N}(v)$) embeddings of node v. The *Update* function consists of a fully-

in their works. Reprinted from *MGG: Accelerating Graph Neural Networks with Fine-grained intra-kernel Communication-Computation Pipelining on Multi-GPU Platforms.* USENIX Symposium on Operating Systems Design and Implementation. 07/2023.

connected NN layer. The neighbor aggregation (*Aggregate*) is the key bottleneck that dominates the overall computation due to its high computation sparsity and irregularity [3, 87]. Compared with conventional graph analytics (e.g., random walk [25, 51]), GNN features higher accuracy [10, 13] and better generality [12, 88] on various applications.

GNN computation on large input graphs (millions/billions of nodes and edges) usually counts on powerful multi-GPU platforms (e.g., NVIDIA DGX [89]) for scaling up the performance. The multi-GPU system (that can potentially store all data required for the computation in the aggregate memory of all GPUs on a single machine) can benefit from aggregated memory capacity and bandwidth (HBM and NVLinks) with more GPUs. There is also a popular trend for state-of-the-art hyper-scale systems employing GPU-centric building blocks. For example, the recent NVIDIA DGX SuperPod [90] consists of 32×DGX-H100 servers (each with 8×H100). Unfortunately, the runtime performance of GNNs does not scale proportionally with the aggregated compute capability and memory capacity of the platform. This is mainly because the irregular and sparse local memory access of neighbor aggregation in the single-GPU settings now "scales" to more expensive inter-GPU communication (i.e., remote memory access). Such intensive inter-GPU communication becomes the new critical path of multi-GPU GNN execution and offsets the performance gains from multi-GPU computation parallelism.

Based on this observation, we highlight a more promising way of formalizing GNN computation on multi-GPU systems. Our key insight is that GNN execution can be more precisely abstracted as a fine-grained dynamic software pipeline to encourage communication and computation overlapping, which will largely hide the communication cost. The opportunities for building such fine-grained pipelines widely exist at different granularities in GNNs. For instance, on a single graph node, the remote neighbor access can be overlapped with the local neighbor computation. Among different graph nodes,

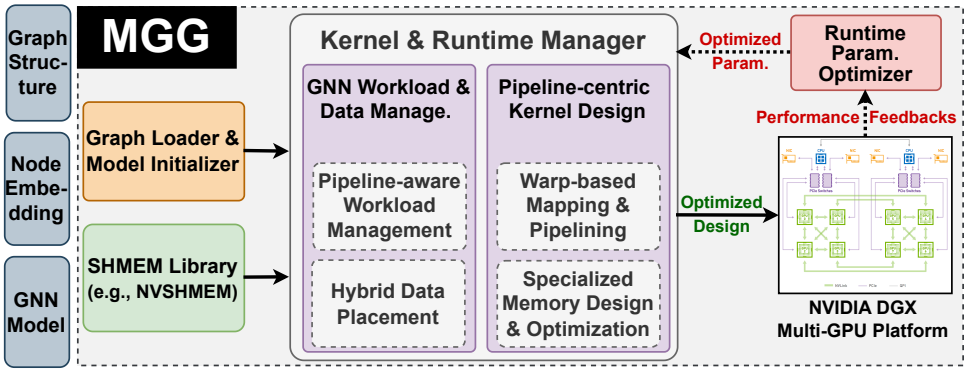


Figure 4.1: Overview of MGG.

the remote neighbor access for certain nodes would potentially be overlapped with the local neighbor computation of some other nodes. However, prior research could hardly exploit such benefits since they rely on hardware and software infrastructures tailored for coarse-grained [91, 92] and regular communication patterns [14, 93]. To capitalize on the fine-grained pipelining benefits, there are three major challenges.

The first challenge is *how to craft the pipeline structure*. A work-efficient pipeline for GNNs demands comprehensively considering multiple factors (e.g., the operations and the number/granularity of each pipeline stage) to best fit the GNN algorithm and multi-GPU computation/communication. The second challenge is *how to map the pipeline to the GPU processing units*. Given the GPU's architectural complexity (e.g., multi-granular processing units and multi-layer memory hierarchy), different mapping and primitive choices would bring performance and design flexibility tradeoffs. The third challenge is *how to find and adapt toward the "optimal" pipeline configuration swiftly*. Given the diversity of GNN inputs (*e.g.*, graph structures) and hardware (*e.g.*, different types/numbers of GPUs), pinpointing the best-off design configuration with high-performance delivery relies on combined insights from the properties of the software pipeline, GNN inputs, and GPU programming and execution paradigms.

To this end, we introduce a set of principles for multi-GPU GNN acceleration via

a fine-grained dynamic software pipeline. *To construct fine-grained pipelines*, the original coarse-grained irregular GNN computation should be breakdown into fine-grained operations. The joint optimization of the GNN workload granularity and data layout should be carried out to facilitate operation overlapping. *To map pipelines to GPUs*, the proper GPU logical processing units (e.g., thread, warp, and block) should be selected for promoting GPU kernel efficiency and design flexibility. In addition, the right choice of communication primitives (e.g., NVSHMEM [94]) should be determined to provide fine-grained inter-GPU communication support. *To adapt pipelines dynamically*, customized kernel templates with tunning knobs should be devised. This will help to maintain pipelining effectiveness across a diverse range of GNN inputs and hardware platform settings.

We crystallize the above principles into MGG[2], a holistic system design and implementation for multi-GPU GNNs (Figure 4.1). Given the GNN models and inputs, MGG will automatically generate pipeline-centric GPU kernels for multi-GPU platforms and dynamically improve the kernel performance based on runtime feedback. The core of MGG is its **Kernel & Runtime Manager**, which constructs GNN-tailored pipelines and maps such pipelines to proper communication primitives and GPU logical processing units. It can also dynamically orchestrate GPU kernels based on new configurations. MGG also incorporates a **Runtime Parameter Optimizer**, which will monitor the performance (e.g., latency) from the actual execution and generate new configurations for the next iteration based on the analytical performance model and optimization heuristics. To the best of our knowledge, we are the first to explore the potential of GPU kernel operation pipelining for accelerating irregular GNN workloads. Moreover, MGG can be generalized to other applications (e.g., deep-learning recommendation model (DLRM) [95]) that are sharing similar irregular communication demands (§4.7.3).

[2]MGG is open-sourced at `https://github.com/YukeWang96/MGG_OSDI23`

Overall, we make the following contributions in this paper:

- We propose a GNN-tailored pipeline construction technique (§4.4) with pipeline-aware workload management and hybrid data placement, for efficient communication-computation pipelining in a GPU kernel.
- We introduce a GPU-aware pipeline mapping strategy (§4.5), encompassing warp-based mapping and pipelining, and specialized memory designs and optimizations to comprehensively promote kernel performance.
- We devise an intelligent runtime with lightweight analytical modeling and optimization heuristics to dynamically improve the performance of GNN training (§4.6).
- Comprehensive experiments demonstrate that MGG can outperform state-of-the-art multi-GPU GNN systems across various GNN benchmarks. Additionally, MGG can be generalized to other DL applications, like DLRM.

4.2 Related Work

Recent deep-learning applications expand their scope from handling structured dense inputs (e.g., images) to unstructured sparse inputs (e.g., graphs). Along with such algorithmic/application expansion is the exploration of new system designs and optimizations for more efficient deep learning. One of the most important topics is the ability to handle large-scale inputs, which are usually out of the computation and memory capacity of one GPU. For scaling regular deep-learning applications, like dense DNNs, various abstractions (e.g., data and model parallel) and high-performance communication libraries (e.g., NCCL [96]) have been developed. While the scaling approach for irregular GNN applications is still initial and suffers from unsatisfactory performance.

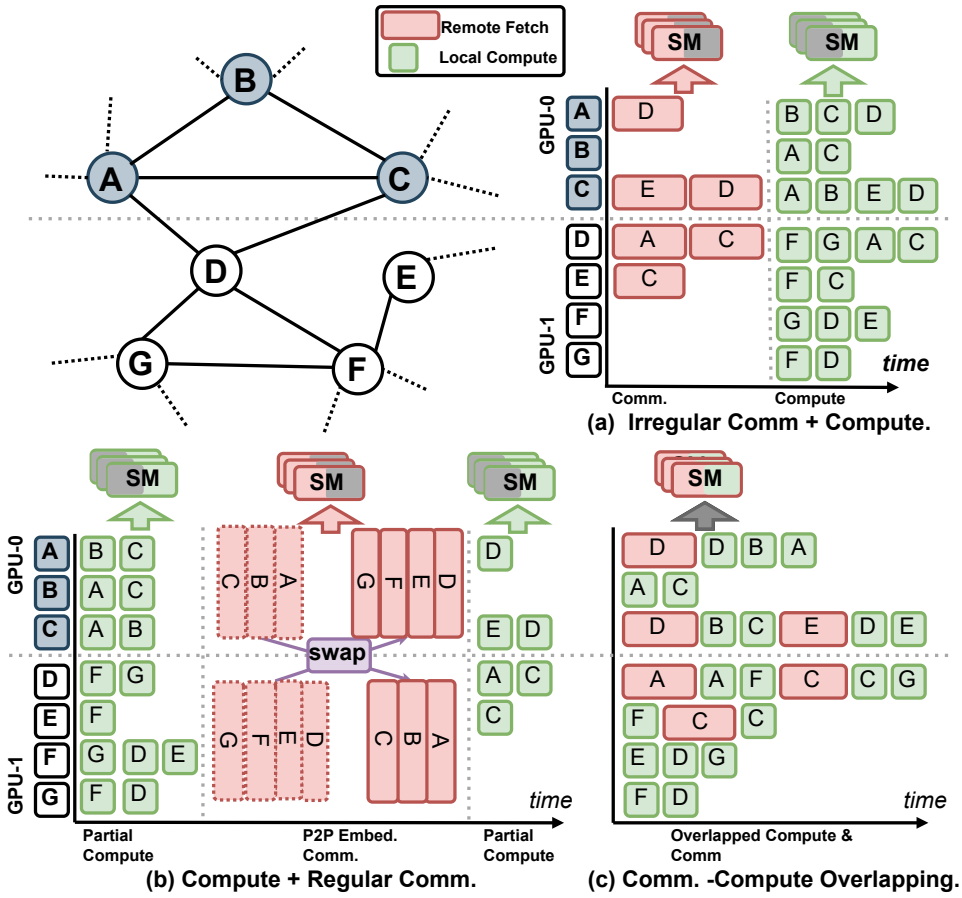


Figure 4.2: Different Multi-GPU GNN strategies for computation and communication. Note that red and green boxes indicate aggregation workload on remote and local neighbors. "SM" boxes with grey areas indicate potential idleness.

Compared to scaling dense DNNs, scaling sparse GNNs is significantly more challenging. The irregular fine-grained sparse GNNs workload cannot fit the regular coarse-grained workload abstraction for dense DNNs. The cost of irregular communication in GNNs cannot be easily amortized by simply batching more requests as dense DNNs due to their randomness and sparseness. Scaling strategies largely vary among different GNN inputs while tiling/schedule strategies would be reused across different inputs of dense DNNs. Therefore, an array of dedicated designs have been introduced to scale the sparse GNNs, focusing on three major directions.

Operator Specialization for Sparse Communication: This is the mainstream solution that treats the communication as a standalone operator for irregularly sparse GNN communication (Figure 4.2(a)). DGL [15] is the state-of-the-art GNN framework and its most recent update incorporates PyTorch-Direct [92] (a GNN-tailored communication design based on zero-copy memory [97]) for large-scale GNN training across GPUs. Work from [98] introduces a communication planning algorithm for distributed GNNs by considering links, communication, contention, and load balancing. However, these efforts optimize the communication standalone and thus miss the opportunities to jointly optimize computation and communication operations/schedules which can potentially reduce the overall latency and improve GPU utilization.

Algorithm Modification for no Communication: The second typical type is to eliminate irregular communication by altering algorithms [99, 100, 15, 101]. They harness various algorithmic adaption solutions, such as neighbor sampling and mini-batch to prefetch the remote neighbors to local devices, and then train the GNN model in a data-parallel fashion as the traditional dense DNN. However, existing research [91, 102] shows that such an algorithmic modification would compromise the accuracy of GNN models compared to the original GNNs. It would also destabilize the algorithmic performance (*e.g.*, the lower convergence speed and final accuracy) under different inputs and sampling configurations.

Schedule Transformation for Dense Communication: The third type is to transform irregular communication to regularized communication (e.g., AlltoAll, P2P), which has been optimized by existing communication kernels (Figure 4.2(b)). ROC [91] delegates communication to its underlying NVIDIA Legion runtime [103], which manages irregular remote neighbor access via a DMA engine. It batches fine-grained embeddings into large embedding tiles on CPUs to facilitate coarse-grained data movement between the host and GPUs. NeuGraph [14] tiles the large node embedding matrices

by rows (as embedding chunks) and then forwards each chunk to GPUs sequentially via coarse-grained P2P communication. P3 [93] spots the potential of transforming irregular embedding communication to regular all-to-all communication for embedding column tiles. However, this type of effort would introduce many unnecessary data movements and non-trivial overhead to transform original algorithms and data inputs.

To sum up, existing designs explore solutions in a limited scope and have yet to extend their solution search to a broader context by exploring the synergy between the multi-GPU GNN workloads, GPU execution paradigms, and communication patterns. Therefore, these designs could hardly enjoy the full potential of multi-GPU platforms.

4.3 Motivation

Different from prior solutions, we propose a new view for multi-GPU GNN workload. We spot that by removing the explicit barrier between the computation and communication stage in multi-GPU GNNs, we can co-schedule the operations from both stages in a holistic way that can reduce the GPU resource idleness and promote performance (Figure 4.2(c)). For example, when GPUs initiate remote access requests and are waiting for the arrival of remote data, the idle cycles of GPUs can be fulfilled by other local computing workloads. Such insight enables us to abstract the multi-GPU GNN workload as a fine-grained dynamic software pipeline for communication and communication overlapping. Specifically, "Fine-grained" means that the operations at each pipeline stage are tiny (e.g., the aggregation of one neighbor's embeddings) versus DNN layers."Dynamic" means that the division of computation into pipeline stages would vary among different inputs in contrast to DNNs with a relatively fixed pipeline. Such a new design is motivated by our three major observations.

GNN Workload Speciality: The first observation reveals the specialty of GNN

Listing 4.1: NVSHMEM APIs in CUDA C.

```
// Initialize an NVSHMEM context on CPUs.
nvshmem_init();
// Get the current GPU device ID on CPUs.
int gpu_id = nvshmem_team_my_pe(NVSHMEMX_TEAM_NODE);
// Set the GPU based on its device ID on CPUs.
cudaSetDevice(gpu_id);
// Define NVSHMEM memory visible for all GPUs on CPUs.
d_shared_mem = (void*) nvshmem_malloc (num_bytes);
// Define global memory visible only for the current GPU.
cudaMalloc((void**) &d_mem, num_bytes);
// Remote access API called by a thread/warp/block.
__device__ nvshmem_float_get_{warp/block}(void *dst, const void *src, size_t nelems, int
    src_gpu_id);
// Sync all GPUs within an NVSHMEM context on CPUs.
nvshmem_barrier_all();
// Release NVSHMEM objects on CPUs.
nvshmem_free(d_shared_mem);
// Terminate the current NVSHMEM context on CPUs.
nvshmem_finalize();
```

workloads, which feature two major types of partial dependency that facilitate pipelining [104]. The first type is the fine-grained neighbor aggregation dependency, where the neighbor embeddings of individual graph nodes are aggregated either sequentially or in parallel with proper synchronization. The second type is the dynamic execution dependency on limited processing units, where different operations would compete for limited GPU resources (e.g., SMs) during the runtime. Such two types of dependencies expose new opportunities for us to amortize communication costs by overlapping neighbor aggregation from different nodes.

GPU Execution Characteristics: The second observation highlights the characteristics of the GPU execution paradigm. One key design principle of GPUs is their massive computation/communication parallelism to amortize the unit cost of individual computation/communication operations [105]. The underlying mechanism of GPU hardware design to facilitate this is to simultaneously schedule multiple logical processing units (e.g., threads/warps/blocks) to share the hardware processing units (i.e., GPU

SMs). Such a design provides the essential ingredient for pipelining, which is that computation and communication operations can co-run on the same units at the same time to fulfill the idle GPU cycles and maximize the utilization of the GPU hardware processing units. Moreover, with the precise control of GPU kernel launching parameters (e.g., the size of the block and shared memory), the effectiveness of co-running heterogeneous operations can be adjusted so that we can flexibly accommodate different inputs while maintaining high-performance delivery.

Multi-GPU Programming Support: The third observation features the recent advancement of the GPU communication technique and its programming support. The one highlighted most is the NVSHMEM [94], which provides GPU intra-kernel APIs for fine-grained (several to tens of bytes) inter-GPU communication (Listing 4.1). NVSHMEM is the main communication backend for MGG. Other existing techniques such as Zero-copy memory can also serve as an alternative to NVSHMEM for fine-grained communication. The performance will be similar while NVSHMEM offers better programmability. Some other traditional strategies for inter-GPU communication, would either offer too coarse-grained communication solutions (e.g., unified virtual memory [106] uses KB-level communication granularity) or resort to the default communication strategies of existing multi-GPU-based runtime system (e.g., NVIDIA Legion [103]) without GNN-tailored communication optimization.

These observations and insights motivate MGG, a holistic multi-GPU GNN system with a novel view of GNN workloads as an operation pipeline. MGG automates the pipeline construction, detailed pipeline mapping, and dynamic input-driven pipeline adaption, to improve the GNN scaling.

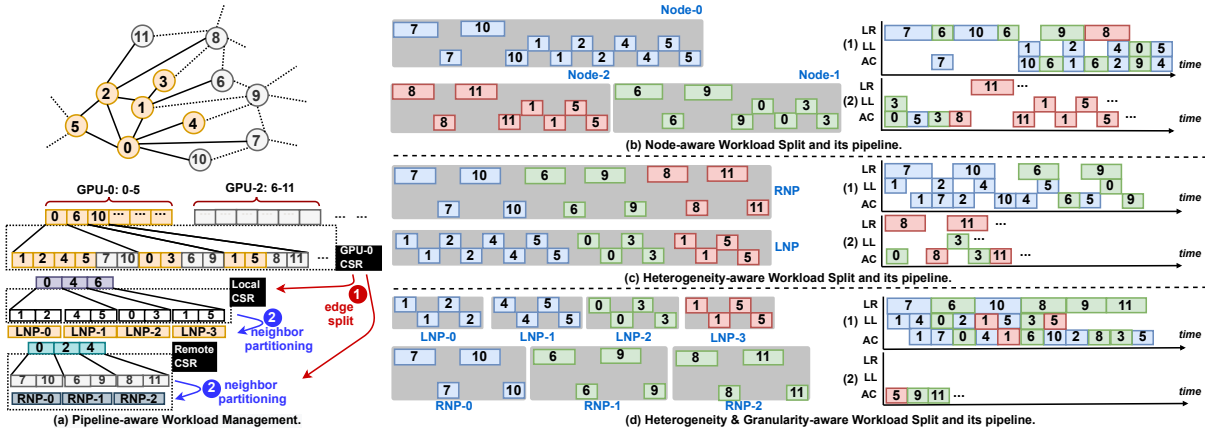


Figure 4.3: (a) Pipeline-aware workload management. "LNP"/"RNP" indicate local/remote workload partitions. (b)(c)(d) Different strategies of workload decomposition and pipelining. Each box indicates a certain (local/remote) aggregation workload and its length indicates its relative latency. "LR": loading remote neighbors, "LL": loading local neighbors, "AC": aggregation computation. Each grey rectangular shadow indicates a workload partition to be processed by one GPU processing unit. (1) and (2) indicate that the same pipeline is chunked into two parts along its time axis due to space limitations.

4.4 GNN-tailored Pipeline Construction

Constructing a GNN-tailored pipeline are facing two major challenges: *1) How to effectively partition and schedule multi-GPU GNN workloads* so that pipeline efficiency can be maximized; *2) How to properly layout input* so that the hierarchy of GNN inputs and the memory/storage of multi-GPU systems can be carefully matched to facilitate pipeline execution. MGG addresses these challenges with *Pipeline-aware Workload Management* and *Hybrid GNN Data Placement*.

4.4.1 Pipeline-aware Workload Management

Managing irregularly sparse GNN workloads for pipelining is challenging and could hardly benefit from the prior practice and exploration of the DNN pipeline [107, 108].

Difference from DNN pipeline *First*, balancing the GNN workloads among GPUs

has to jointly optimize the computation capacity and the computation/communication irregularity. While the DNN pipeline only needs to balance the computation/memory capacity, since its pipeline stages are well-structured and their inputs are regularly dense. Distributed DNNs require dense regular communication (e.g., Allreduce) that is naturally fit for existing GPU interconnects optimized for throughput and has been optimized by many libraries (e.g., NCCL). In contrast, distributed full-graph GNN (with the entire graph cached on GPUs) is much more challenging since it requires sparse irregular communication that is naturally at odds with the existing hardware interconnects, and fewer efforts have optimized its performance. *Second*, the GNN pipeline workload is more irregular and non-structural and can easily cause pipeline stalls/bubbles. For example, remote neighbor aggregation would have different stages (remote access + aggregation) compared with local neighbor aggregation (local access + aggregation), making it challenging to mix those two heterogeneous workloads. While in the DNN pipeline, all inputs should consistently pass through the same pipeline stages. *Third*, GNN pipeline stages are more fine-grained (e.g., fetching individual embeddings) compared with coarse-grained layers (e.g., GEMMs and Convolutions) in the DNN pipeline. Such small workload granularity enables different pipeline stages to overlap with each other on GPU processing units, like Streaming Multiprocessors (SMs). In contrast, DNN pipelines can only overlap layer-wise computation and communication operations among different GPUs.

With the above insights, we propose a three-stage dynamic software pipeline design. The three stages include *loading remote neighbors* (**LR**), *loading local neighbors* (**LL**), and *aggregation computation* (**AC**). Aggregation of a certain neighbor will only take two stages. The remote neighbor aggregation will take the stage LR and AC while local neighbor aggregation will take the stage LL and AC. The stage-wise pipelining is achieved with two steps: 1) assigning aggregation workload to different GPU logical processing units (LPUs), like warps and blocks, and 2) scheduling different LPUs on

the same GPU SM to overlap their execution. Three-phase pipeline can generalize to different GNN models, which essentially consist of the different numbers of basic remote and local operations. For example, GCN has a lower local-vs-remote operation ratio while GAT features a higher local-versus-remote operation ratio. Three-phase pipeline can also capture differences among inputs. For instance, a more sparse graph will have a higher remote-to-local operation ratio.

However, the direct construction and execution of such three-stage pipelines would be inefficient, because of its ignorance of GNN workload heterogeneity and irregularity on multi-GPU platforms. To address these challenges, MGG highlights a GNN-tailored pipeline construction strategy to build and optimize the software pipeline in three steps.

Step-1: Workload-aware inter-GPU pipeline workload balancing. This step aims to construct the "raw" pipeline and balance workloads among pipelines on different GPUs. Our insight is that GPUs with massive processing units (e.g., SMs) will serve many pipelines concurrently, and the key to maximizing GPU performance and utilization is to ensure that each pipeline will get a similar amount of workload, thereby avoiding execution critical path on certain "long" pipelines. We, therefore, develop a *range-constrained binary search algorithm* (Algorithm 3) based on prior graph partitioning exploration [109]. Our solution features a lower runtime cost to split the GNN input graph into chunks (one chunk per GPU) while balancing the number of edges within each chunk. Then the workload from the same chunk is grouped by nodes as *workload partitions* mixed local and remote neighbors (Figure 4.3(b)). From its potential execution pipeline, we can see many idle cycles (indicated by blank spaces in different pipeline stages) which would result in low pipeline efficiency and GPU resource occupancy. Note that in the software pipeline, workloads from different partitions can be overlapped as they will be processed by different LPUs. While the workloads from the same partition are sequentially processed by one LPU and their relative order should be maintained

even after being mixed with other partitions.

Step-2: Heterogeneity-aware pipeline bubble reduction. The pipeline constructed from the previous step is still inefficient due to its scattered workloads among stages, namely pipeline bubbles. The optimization in this step is to minimize such pipeline bubbles for better pipeline efficiency. The key is to reduce the heterogeneity of workload partitions that hinders effective overlapping. To achieve this, we categorize the sparse multi-GPU GNN computation into two types. The first type has local neighbor access only, which has shorter execution latency. The second type has remote neighbor access, which features high latency overhead. We delicately handle different types of workloads via grouping (Figure 4.3(a)-❶), where two separate CSRs for *local* and *remote* subgraphs will be built. The aggregation will be conducted on local and remote subgraphs separately and followed by a result synchronization at the end. Such a remote-local split is also backed by the fact that on platforms with all-to-all GPU interconnections (e.g, DGX-A100/H100), accessing different GPUs under the same data granularity has approximately equal communication cost [110]. Such heterogeneity awareness in workload partitioning (Figure 4.3(c)) enables a more densely overlapped workload between the stage LR and LL/AC.

Step-3: Granularity-aware intra-GPU pipeline enhancement. While the second optimization improves pipeline efficiency by reducing the workload heterogeneity, there is still plenty of room for further enhancement. The optimization in this step is to facilitate a more balanced workload distribution among pipeline stages. This key is to find the proper workload granularity for local and remote subgraphs so that those originally sequentially processed workload partitions can be overlapped. Our key observation is that nodes in the local/remote subgraphs would have a diverse number of neighbors. Such a specialty makes it challenging for massively parallel GPUs to harvest the real performance gains due to the imbalance workload and diverged execution flow.

Algorithm 3 Range-constrained Binary Search.

Require: Graph node pointer array ($nPtr$), edge list array ($eList$), and the number of GPUs ($numGPUs$).
Ensure: List of graph edge split points ($numGPUs - 1$).

```
outList = {}; lastPos = 0
▷ Compute approximated #edges per GPU.
ePerGPU = (len(eList) + numGPUs - 1) / numGPUs
for sId in [0, 1, ..., numGPUs - 1] do
    nid = binSearch(nPtr, ePerGPU, lastPos, numNodes)
    lastPos = nid
    outList[sId] = nid
end for
return outList
Function (binSearch)nPtr, ePerGPU, lastPos, numNodes
i = lastPos; j = numNodes
target = min(nPtr[i] + ePerGPU, nPtr[numNodes])
while i < j do
    mid = (nPtr[i] + nPtr[j]) / 2
    if mid > target then
        j = (i + j) / 2
    else
        i = (i + j) / 2
    end if
end while
return i
```

Therefore, we approximate such coarse-grained irregular workloads with fine-grained fixed-sized partitions so that the workload imbalance across nodes can be amortized. For example, with 2 neighbors per partition (Figure 4.3(a)-❷), we can get a more balanced workload among nodes in their local and remote neighbor aggregation. With such granularity awareness, the individual pipeline can be further condensed along its time axis with more overlapping of the LL and AC stage. (Figure 4.3(d)). Meanwhile, the irregular workload can be more evenly distributed to GPU SMs for higher GPU utilization. On the other side, partition granularity should also be balanced with synchronization overhead, since more fine-grained partitioning can bring more parallelism at the cost of more synchronization overhead. This is because workloads from different partitions for the same target node need to be reduced via synchronization, like inter-thread shuffling

and atomics.

MGG design can also be generalized to multiple machines with a minor adaptation. For example, in Figure 4.3(d), when there are inter-node (over Inifite-Band) remote neighbors (longer latency due to lower inter-node communication speed), the size of remote neighbor partitioning (RNP) should be adjusted to a smaller size (e.g., from 2 to 1 remote neighbor) to facilitate better overlapping with local computation.

4.4.2 Hybrid GNN Data Placement

In collaboration with our multi-step pipeline construction, we introduce a *hybrid GNN data placement* strategy to exploit the benefits of different types of memory in SHMEM-enabled multi-GPU systems. The major impact of such hybrid placement on pipelining is two-fold. First, placing GNN data in different memory spaces will lead to different ratios of local and remote workloads, thus, affecting workload balance among pipelines. Second, different memory spaces will offer different access performances (e.g., latency), thereby, affecting the execution efficiency of the individual pipelines, such as the number of pipeline bubbles.

Our strategy focuses on two major aspects. Firstly, for workload balance among pipelines, we leverage NVSHMEM "shared" global memory to store the node embeddings (**NEs**) of the whole graph (Figure 4.4 *left*). Our major consideration here is that such shared global memory space can be accessed by all GPUs with the approximated equal access speed, which is vital to facilitate a more even distribution of remote workloads to GPUs in terms of their size and unit access costs. In addition, NEs are generally large in terms of size (due to high dimensionality), which are beyond the device memory limit of a single GPU. Therefore, NEs are ideal to be placed in shared global memory space with sufficient space (with aggregated memory of different GPUs), which also provides

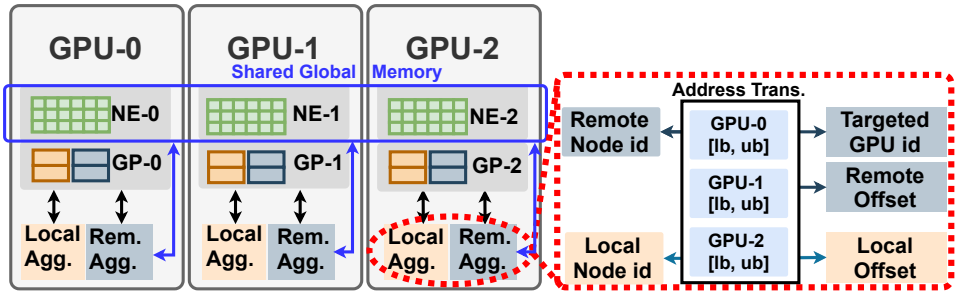


Figure 4.4: MGG Storage Layout and Communication Pattern. Note that "NE-i" is the node embedding partition stored on the i-th GPU. "GP-i" is the neighbor partition processed by the i-th GPU. "GPU-i [lb, ub]" is the node-id range [lowerbound, upperbound] of the node embeddings on the i-th GPU.

direct remote access support across GPUs. Specifically, we will partition the NEs of input graphs into n equal-sized partitions (where n is the number of GPUs) and place each of them in one GPU's shared global memory space.

Secondly, for the efficiency of individual pipelines, we allocate the "private" global memory space for storing partitioned graph structure (**GP**) data, which is only visible to kernels on the current GPU. Our key insight is that GP (e.g., edge lists), is all scalar values and usually small in size, and will only be accessed by the local GPU. Therefore, GP is ideal to be placed in individual GPUs' DRAM. Such a placement is also important to reduce unnecessary and inefficient remote access on those tiny scalars for fewer pipeline bubbles. In our design, GP data (e.g., edges) from private GPU global memory will be processed by a *address translation* unit for fetching correct NEs on local/remote GPU since the NE indices are rebased to zero on each GPU (Figure 4.4 *right*).

4.5 GPU-aware Pipeline Mapping

Efficient pipelining also demands effective mapping of well-constructed pipeline workload and their schedules to the low-level GPU logical processing units (*e.g.*, GPU threads/warps/blocks) to overlap computation and communication. To achieve this, we propose

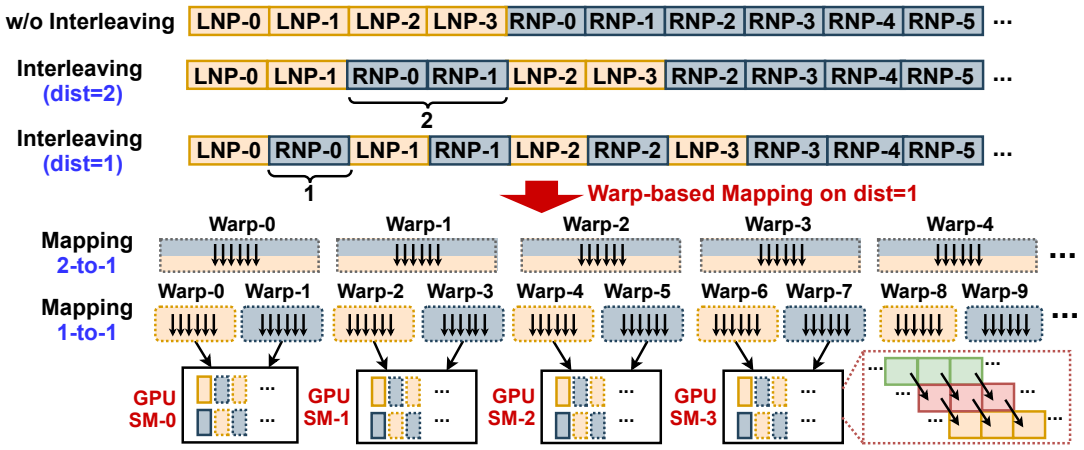


Figure 4.5: Warp-based Mapping and Pipelining. Note that "LNP" refers to the local neighbor partitions; "RNP" refers to the remote neighbor partitions. Workload and Warps are matched based on colors. Tiny boxes in GPU SM indicate decomposed workload operations for overlapped execution.

Warp-based Mapping & Pipelining and *Specialized Memory Design & Optimization* to jointly optimize the pipeline execution efficiency, GPU utilization, and end-to-end design flexibility.

4.5.1 Warp-based Mapping & Pipelining

An effective pipeline mapping demands comprehensive consideration of two major aspects. 1) *Which type of GPU logical processing units (e.g., warps, blocks) should be used for pipeline workload partitions?* We choose GPU *warp* as the basic working unit to handle the workload of each partition. This is because threads in a warp can collaboratively work on different dimensions of a node embedding simultaneously. Whereas using a single or several threads (less than the size of a warp, 32 threads) would hardly explore the computation parallelism and would cause warp-level divergence. Besides, NVSHMEM remote access initiated by a warp of threads would merge the requests into one remote memory transaction to amortize the overhead. 2) *Which pattern of mapping should be used for benefiting pipeline execution efficiency?* The most straightforward way

is to continuously map the neighbor partitions from the local and remote workload list to GPU warps with continuous IDs (Figure 4.5). However, this strategy would easily suffer from workload imbalance among GPU SMs. This is because warps with continuous IDs are more likely to be placed into the same thread block, which is assigned to one SM for processing. Therefore, SMs assigned with warps for handling remote neighbor partitions would lead to much longer latency than SMs assigned with warps for processing local neighbor partitions. Such a workload imbalance would lead to poor GPU utilization and runtime execution performance.

To this end, we introduce our novel *workload interleaving* strategy to balance the workload among SMs on GPUs. Each warp of threads running on GPU would handle one or more pairs of local/remote workload partitions. To more precisely calibrate the warp-to-SM mapping for different pipeline stages to achieve efficient pipelining, we introduce a new metric – *interleaving distance*. We give examples with the interleaving distance equals 1 and 2 for illustration (Figure 4.5). By mixing different types (both local and remote) of workload together, better GPU utilization can be achieved since when one warp is blocked for high-cost remote access, other warps that are working on local computation can still be served by the SMs warp scheduler for filling up these idle GPU cycles. Moreover, such a design would improve design flexibility. For instance, given an input graph with a selected neighbor partition size, we can adjust the size of interleaving distance and the workload per warp so that waiting cycles of the remote access can be hidden by the computation cycles of the neighbor aggregation. Thus, each warp can be fully utilized while the design can achieve sufficient parallelism.

MGG currently processes the neighbors of adjacent nodes (based on node-ids) to the same thread block where the same block will be scheduled on the same SM. If there are common remote neighbors for those adjacent nodes, their remote requests will be merged. Improving such locality requires reordering the graph nodes to maximize their common

neighbors. Such an exploration is orthogonal to our current contribution. In future GPUs, there is a trend to explore the locality among independent processing units. For instance, in Hopper, several thread blocks can be grouped together as thread-block groups. We can explore the tradeoff between the locality benefits and group synchronization overhead.

4.5.2 Specialized Memory Design & Optim.

Efficient software pipelining also demands careful management of high-bandwidth shared memory for promoting data access efficiency and asynchronized primitives for exploiting intra-warp operation pipelining.

GPU SM Shared Memory Layout: Based on our MGG's warp-based workload design, we propose a *block-level shared memory orchestration* to maximize the performance gains. We have several key insights for such a dedicated memory layout design within each thread block. *First*, our neighbor-partition-based workload will generate the intermediate results that can be cached at the high-speed shared memory for reducing the frequent low-speed global memory access. *Second*, NVSHMEM-based remote data access demands a local scratch-pad memory (e.g., registers, shared and global memory) to hold the remote data for local operations.

For the *local* neighbor aggregation, we reserve a shared memory space with D (D is the embedding dimension) floating-point numbers for embeddings of the target node in each neighbor partition so that threads from a warp can cache the intermediate results of partial reduction in shared memory. For the *remote* neighbor aggregation, the shared memory space is doubled $2 \times wpb \times D$ (wpb is the warps per block). The reason is that we need the first half $wpb \times D$ for caching the partial aggregation results of each warp and the remaining for the remotely accessed neighbor embeddings. For each MGG kernel design, we will first identify the warp-level information, like warp IDs. Then within each

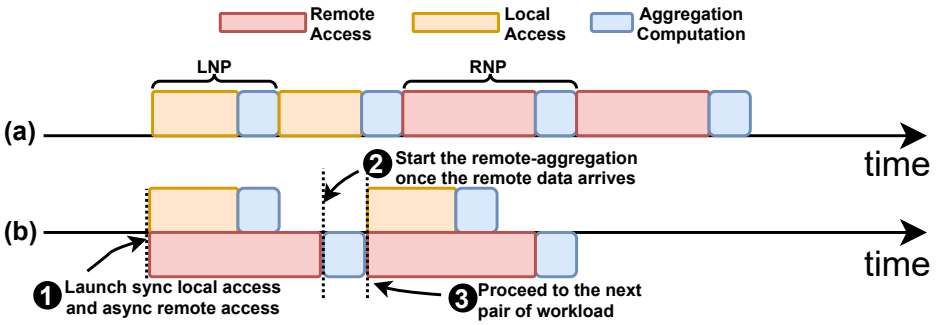


Figure 4.6: Illustration of (a) w/o and (b) w/ asynchronized primitives for overlapping computation and communication of an individual warp. Note that the length of each rectangular box indicates the estimated latency cost of each operation.

thread block, we define the customized shared memory layout by splitting the contiguous shared memory address into three different parts for neighbor ids, partial aggregation results, and the remotely-fetched node embeddings. We use the dynamic shared memory for design flexibility since those parameters (*e.g.*, *wpb* and D) can only be determined at runtime. During execution, we will first calculate the total shared memory size per block and then pass it as a kernel launching parameter.

Pipelined Memory Operation: §4.5.1 have discussed assigning local (LNP) and remote (RNP) neighbor aggregation workloads to warps so that different warps can overlap their computation and communication to fully saturate the active cycles of the GPU SM scheduler. However, only exploiting the inter-warp communication-computation overlap is not enough to maximize the utilization of GPU resources. We further explore the overlapping of the computation and communication at the intra-warp level by carefully scheduling the memory operations. Figure 4.6(a) shows the case with two LNPs and two RNPs by using the synchronized remote access, we can just sequentially process the two LNPs and the two RNPs. The long-latency remote access can happen only after the completion of its preceding LNP. This could lead to a longer GPU stall for memory operations and low GPU SM utilization. Our profiling also shows that without overlapping,

the remote access usually dominates the overall execution (around 60% of overall latency) compared to the time for local data access plus the time for aggregation computation (around 40% of overall latency). Such observation justifies our design to mainly hide the latency from remote access.

To amortize the cost of remote access for each warp, we introduce *asynchronized remote memory operations* (Figure 4.6(b)). This improved design consists of two major steps. First, we can simultaneously launch the local memory access while initializing the remote memory access for fetching the node embedding (❶), therefore, the time for remote access can be amortized by the processing of LNP. Second, once the remote access is completed, the current warp will start aggregation on the remotely-fetched node embedding data (❷). The next step will start the new iteration of the previous two steps, which will process a new pair of LNP and RNP.

4.6 Intelligent Runtime Design

In this section, we will discuss our intelligent runtime design with performance/resource analytical modeling and heuristic-based cross-iteration optimization strategy.

Performance-Resource Analytical Modeling: The performance/resource model of MGG has two variables: *workload per warp* (WPW) and *shared memory usage per block* ($SMEM$), which can be measured by

$$\begin{aligned} WPW &= 2 \cdot ps \cdot D \cdot dist, \\ SMEM &= ps \cdot wpb \cdot IntS + 2 \cdot wpb \cdot D \cdot FloatS \end{aligned} \tag{4.1}$$

where ps, wpb, and D are the sizes of neighbor partition, warp per block, and node embedding dimension, respectively; $dist$ is the interleaved distance of local/remote workloads (§4.5.1); $IntS$ and $FloatS$ are both 4 bytes on GPUs. To determine the value of the ps, wpb, and $dist$ of a given input graph, we will first compute the total number of

warps by using

$$numWarps = \frac{\max\{local, remote\}}{dist} \tag{4.2}$$

where *local* and *remote* are the number of local and remote partitions, respectively. Then we compute the total number of blocks and the estimated block per SMs by using

$$\begin{aligned} numBlocks &= \frac{numWarps}{wpb}, \\ blocksPerSM &= \frac{numBlocks}{numSMs} \end{aligned} \tag{4.3}$$

Later, based on our micro-benchmarking results on diverse datasets, we define our parameter search space and constraints: 1) $ps \in [1 \dots 32]$ to balance the computation parallelism and synchronization overhead; 2) $dist \in [1 \dots 16]$ to effectively overlap the computation and remote memory access; 3) $wpb \in [1 \dots 16]$ to maintain SM warp scheduling flexibility for better occupancy and throughput; 4) $numSMs \leq c_1$, $SMEM \leq c_2$, where c_1 and c_2 are hardware constraints [111], *e.g.*, NVIDIA A100 has 108 SMs and 164KB shared memory per SM.

Heuristic-based Cross Iteration Optimization To optimize the design of MGG, the parameter *ps*, *dist*, and *wpb* are initialized as the value 1 at the beginning. Then we optimize one parameter in each of the following iterations. *First*, we increase the *ps* to maximize the warp utilization. When further increasing the *ps* would also increase the latency, we would stop the search on *ps* and switch to *dist*. *Second*, we apply a similar strategy to locate the value of *dist* that can maximize the overlap of local computation and remote access. *Third*, we increase *wbp* to maximize the utilization of the entire SM. If any increase of *wpb* would increase the latency, we know that there may be too large thread blocks or too heavy workloads on individual warps that lower SM warp scheduling efficiency or computation parallelism. We would "retreat" (*i.e.*, decrease) *ps* to its second-highest value if necessary and restart the increase of *wpb*. This optimization algorithm

Table 4.1: Datasets for Evaluation.

Dataset	#Vertex	#Edge	#Dim	#Class
reddit(**RDD**) [15]	232,965	114,615,892	602	41
enwiki-2013(**ENWIKI**) [82]	4,203,323	202,623,226	300	12
it-2004 (**IT04**) [112]	41,291,594	1,150,725,437	256	64
ogbn-paper100M(**PAPER**) [93]	111,059,956	1,615,685,872	128	64
ogbn-products(**PROD**) [113]	2,449,029	61,859,140	100	47
ogbn-proteins(**PROT**) [113]	132,534	39,561,252	8	112
com-orkut(**ORKT**) [82]	3,072,441	117,185,083	128	32

will stop when any decrease of *ps* and increase of *wpb* would lead to higher latency than the top-3 lowest latency. The latency of each iteration during the optimization will be recorded by a configuration lookup table. Finally, the configuration with the lowest latency will be applied.

This particular optimization order of parameters (*ps*, *dist*, and *wpb*) is based on two major aspects: (i) *Spatially speaking*, the granularity is from coarse-grained algorithm-level partitioning through *ps*, to medium-grained pipeline construction through *dist* (according to the partition plan), to fine-grained pipeline-to-warp fine-tuning through *wpb* (according to the pipeline design). (ii) *Temporally speaking*, the three optimizations are applied at loading-time (*ps* to decide layout), kernel initialization (*dist* to decide pipeline), and runtime (*wpb* to decide pipeline mapping), respectively.

The above parameter adaption for dynamic pipelining is vital for design/optimization generality. This is because the characteristics of graphs (#nodes/edges and embedding sizes) would lead to different efficiency of kernel pipelines. Our later experimental studies (as shown in Figure 4.11) demonstrate its benefits with up to 70% of performance improvements.

4.7 Evaluation

Benchmarks & Datasets Despite the diversity of GNN models, the fundamental computation and communication paradigm (vector-based scatter-gather operation) in

multi-GPU GNNs remains the same. We evaluate two distinctive and representative GNN models on *node classification* tasks:

The first type of GNN model uses a *non-discriminated* neighbor aggregation strategy, where all neighbors contribute equally when doing the aggregation. We choose **Graph Convolutional Network (GCN)** [10], which is the most popular GNN model and is also the key backbone network for many other GNNs, such as GraphSAGE [12] and Differentiable Pooling [41]. We use *2 layers with 16 hidden dimensions* for GCN, which is also the setting from the original paper [10]. The computation of a 2-layer GCN can be expressed as

$$Z = Softmax(\hat{A}\ ReLU(\hat{A}XW^1)W^2). \tag{4.4}$$

where $\hat{A}$ is the adjacent matrix of the input graph with self-loop edges, and X is the input node embedding matrix, where $X \in R^{N \times D}$; N is the number of nodes in a graph; D is the size of node embedding dimensions. W^1 and W^2 are trainable weight matrices in layer-1 and layer-2, respectively.

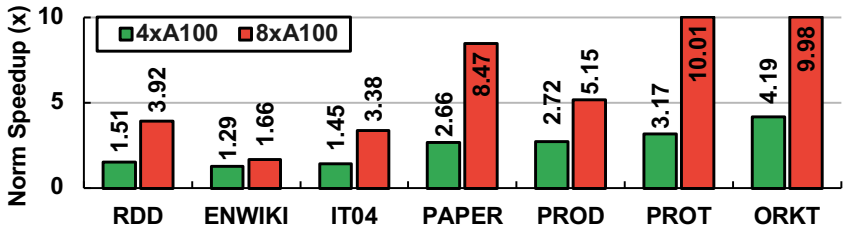
The second type uses a *discriminated* neighbor aggregation strategy, where neighbors would contribute differently depending on their calculated edge-specific features. We choose **Graph Isomorphism Network (GIN)** [13], which aims to distinguish the graph structure that cannot be identified by GCN. Each layer of GIN can be expressed as

$$h_v^{l+1} = MLP^l((1+\epsilon^l)\dot{h}^l + \sum_{u \in N_{(v)}} h_u^l). \tag{4.5}$$

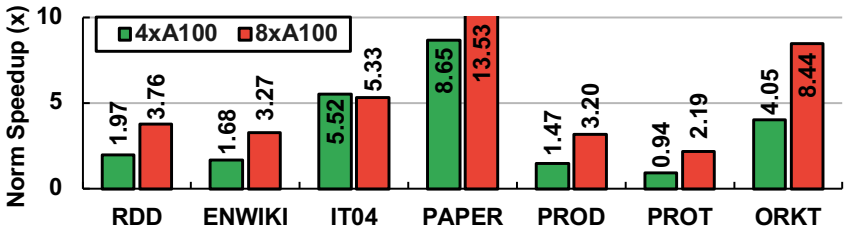
where l is the layer ID and $l \in \{0, 1\}$, MLP is a fully-connected neural network, h_v is the node embedding for node v, and $N_{(v)}$ stands for the neighbors of node v. GIN mainly differs from GCN in its aggregation function, which introduces a weight parameter as the ratio of contribution from its neighbors and the node itself. In addition, GIN is the reference architecture for many other advanced GNNs with more edge properties,

such as Graph Attention Network [11]. For GIN evaluation, we use *5 layers with 64 hidden dimensions*, which is also the setting used in the original paper [13]. Graphs (Table 5.4) used in our evaluation are large in their number of nodes and edges that demand multi-GPU capability for effective GNN computation. *#Class* is the output dimension (*#labels*) for the node classification task. *#Dim* is the embedding dimension of the input graph.

Baselines In this evaluation, we compared MGG with several existing systems that support large full-graph GNN (i.e., caching the entire graph on GPUs) on multi-GPU platforms. **1) Deep Graph Library (DGL)** [15] is the state-of-the-art framework for large-scale GNNs across GPUs. It leverages PyTorch-Direct [114] as the communication backend for GPU-initiated zero-copy memory access [97] to fetch neighbors embedding from the CPU host. **2) MGG-UVM** [115] is a GNN design by adapting MGG to leverage unified virtual memory (UVM). UVM has been highlighted in handling irregular graph computations (such as PageRank) on large graphs [115]. However, [115] is not open-sourced, we thus generalize the pipeline kernel designs and optimizations (§4.4 and §4.5) of MGG to build such a UVM baseline and incorporate optimizations from [115]. Note that UVM and zero-copy memory are different communication backends [1]. Thus, MGG-UVM does not implement zero-copy data transfer. We remark UVM is the key communication protocol before the new hardware support for fine-grained direct GPU-GPU communication (e.g., NVSHMEM). UVM is more coarse-grained and will require the engagement of CPUs (e.g., host memory management) for communication. The reason to use MGG-UVM is to show that if there is no advanced hardware support (e.g., NVSHMEM) for fine-grained direct GPU-GPU communication, the benefits of our elaborated pipeline can be offset by UVM communication overhead. **3) ROC** [91] is a popular distributed multi-GPU system for full-graph computation. ROC highlights its learning-based partitioning and leverages NVIDIA Legion [103] runtime for communication and



(a) GCN Model.



(b) GIN Model.

Figure 4.7: Performance comparison with DGL. Note that full-graph PAPER on DGL requires A100-80GB.

task scheduling.

Other multi-GPU GNN designs, like NeuGraph [14] and *P*3 [93], are not publicly available. Initially, we plan to evaluate MGG on AMD ROC_SHMEM [116]. However, as indicated in its document, the existing ROC_SHMEM is an experimental prototype and is not officially ready to be applied in practice due to very strict software limitations (*e.g.*, only supports ROCm v4.3) and hardware (*e.g.*, only supports AMD GFX9 GPUs), which are quite challenging to find and deploy and not supported by any existing GNNs frameworks [98, 92, 91] for comparison. We believe that once ROC_SHMEM becomes ready and generally applicable, MGG can be easily migrated to AMD multi-GPU platforms.

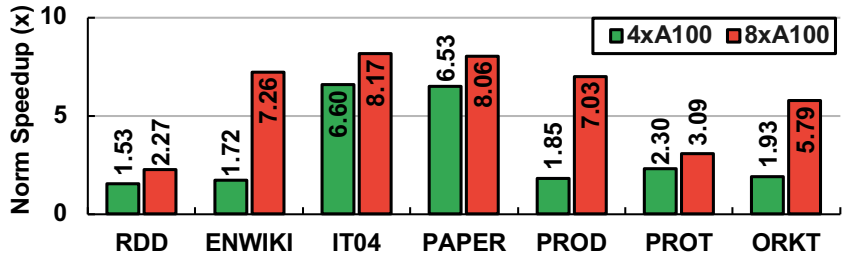
There is no existing design that can leverage GPU-to-GPU communication only for distributed full-graph GNN computation. We try our best to measure the best-possible baseline performance. DGL and ROC have longer latency in the earlier iteration due to cache warmup for node embedding on GPU memory. We thus perform warm up iterations until their per-iteration latency becomes stable, and then measure their performance with

minimized CPU-GPU data movements.

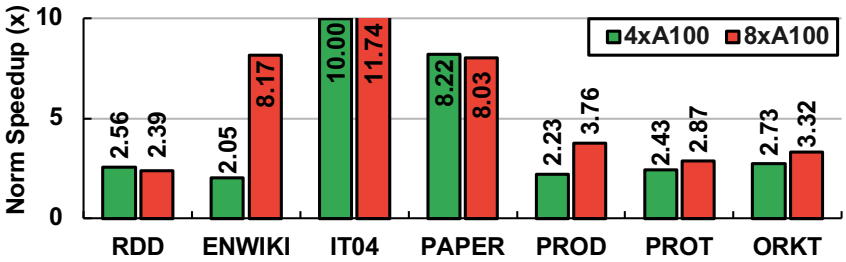
Platforms & Tools The implementation of MGG consists of ∼9K LoC. We compile and link MGG with CUDA (v11.2), OpenMPI (v4.1.1), NVSHMEM (v2.0.3), and cuDNN (v8.2) library. Our major platform is an NVIDIA DGX-A100 with dual AMD Rome 7742 processors (each with 64 cores, 2.25 GHz), 1TB host memory, and 8×A100 GPUs (40 GB) connected via NVSwitch, which offers 600 GB/s GPU-to-GPU bi-directional bandwidth. For the modeling study, we also leverage DGX-1 with 4×V100 GPUs connected via NVLinks. We use NVIDIA NSight Compute to get the kernel-level profiling metrics. Speedup is averaged over 100 runs.

4.7.1 End-to-End Performance

Compared with DGL In this section, we will compare with the state-of-the-art DGL framework, which leverages PyTorch-Direct for cross-GPU communication. We evaluate different datasets and platform settings (with 4 and 8 A100 GPUs). As shown in Figure 4.7, MGG outperforms DGL with averaged 4.25× and 4.57× speedups on GCN and GIN models, respectively. We also notice a trend that MGG demonstrates a more pronounced speedup with more GPUs. With the increasing number of GPUs, DGL suffers from heavy memory access contention, since multiple GPUs are initiating massive requests to access the neighbor embeddings on the CPU host memory. Another observation is that on GIN ($D = 64$) with higher hidden dimensionality for smaller datasets (e.g., PROD and PROT), the performance gap between DGL and MGG is smaller compared to GCN ($D = 16$) since as indicated in [92], zero-copy memory would be beneficial from more coarse-grained data movement (with larger embedding vector) that can saturate the PCIe cache line (128 Bytes). While such an advantage of DGL diminishes for those larger datasets (e.g., IT04 and PAPER) on GIN due to significantly



(a) GCN Model.



(b) GIN Model.

Figure 4.8: Performance comparison with MGG-UVM.

Table 4.2: Additional performance comparison of MGG and DGL on GraphSAGE and GAT.

Model	RDD	ENWIKI	IT04	PAPER	PROD	PROT	ORKT
SAGE	4.97×	1.76×	1.99×	3.53×	7.05×	3.39×	3.53×
GAT	2.65×	1.62×	2.06×	3.04×	2.06×	3.39×	3.04×

increased sparsity and irregularity. In addition, compared with MGG, DGL assumes the one-size-fits-all communication strategy would work well for all input datasets. Therefore, it ignores the importance of the inputs and hardware properties, which would bring non-trivial (more than 30%) benefits (§4.7.2).

MGG can also be extended to cover other GNN models. The following results show the speedups of MGG over DGL on GraphSAGE with layerwise node neighbor sampling and GAT with dot-product edge attention. Table 4.2 shows that the performance results of GAT and SAGE also agree with our prior observations on the GCN and GIN, demonstrating the generality and effectiveness of our proposed design and optimizations to handle more complex dataflow (e.g., edge attention and softmax) in multi-GPU GNN computation.

Despite that MGG (NVSHMEM) and DGL (with CPU-GPU zero-copy memory [97]) both rely on GPU-initiated communication and overlap communication with computation, their underlying mechanism is different, and MGG shows more performance advantages. MGG can leverage inter-GPU communication while DGL can only rely on CPU-GPU communication with limited bandwidth. This makes the communication costs pronounced in DGL and offsets the performance gains from massive thread-level parallelism. This experiment also shows that MGG can serve as a drop-in replacement for the existing communication backend of DGL to improve large-scale full-graph GNN computation.

Compared with MGG-UVM In this experiment, we compare MGG with its UVM-based counterpart, MGG-UVM, which uses UVM in place of NVSHMEM for remote communication. Figure 4.8 shows that MGG achieves 4.58× speedup and 5.04× speedup on average compared to MGG-UVM on GCN and GIN, respectively. The MGG-UVM leverages the page-faulting-based remote data access that is more coarse-grained (around 4 KB) in comparison with a single node embedding size (less than 0.4KB), which leads to higher overhead and lower effective bandwidth usage per embedding transfer. Such an overhead would exacerbate with more GPUs and also make MGG-UVM challenging for GPU SM schedulers to effectively dispatch instructions for the next available warps. This is mainly because most of the warps wait for the long-cycle page-faulting and migration.

We notice that with the increase of the dimension size (i.e., data movement granularity), the speedup over MGG-UVM becomes higher. We later found out that the increase of data-movement granularity actually increases the overall page-fault counts. This is because embedding vectors are generally stored continuously for memory efficiency instead of aligning with the size of memory pages. Therefore, increasing the size of individual embedding also increases the likelihood of triggering multiple pagefaults per embedding transfer.

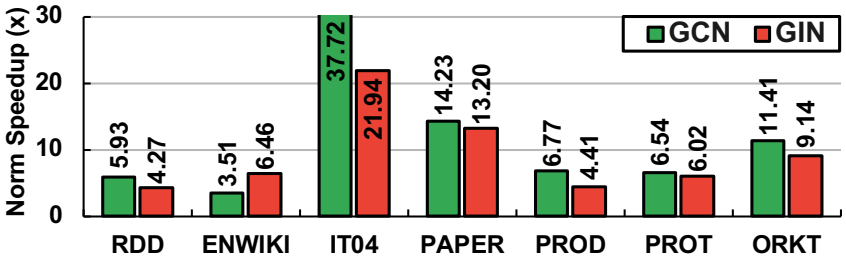


Figure 4.9: Performance comparison with ROC with 8×A100.

Comparing among datasets, for graphs (e.g., PAPER) with more nodes/edges and lower average node degree, MGG would demonstrate more speedups since these graphs exhibit more irregular and sparse access that can not well fit into regular fix-sized pages. This also indicates the importance of amortizing communication overhead. Thanks to pipeline-centric workload management, we can effectively amortize such costs with careful operation scheduling.

We further measure two performance-critical GPU kernel metrics that are the key indicators of our pipeline efficiency (§4.4.1): *Achieved Occupancy* (the ratio of the average active warps per active cycle to the maximum number of warps supported in an SM) and *SM utilization* (the utilization of all available SMs on a single GPU). MGG improves SM utilization (by 21.15% on average) and occupancy (by 39.20% on average) compared to MGG-UVM. This indicates that MGG can effectively 1) distribute irregular workloads to SMs to balance workloads among pipelines and improve the overall GPU utilization, and 2) overlap the remote access and local aggregation computation from different warps to reduce pipeline bubbles and maximize SM occupancy.

Compared with ROC In this experiment, we compare MGG with ROC [91] on their officially released GCN model implementation. We originally plan to evaluate both 4 and 8 GPU settings. However, ROC reports many out-of-memory (OOM) errors for those large graphs on GCN/GIN model and medium graphs on the GIN model due to its aggressive caching of those intermediate tensors on GPUs. Therefore, we keep our

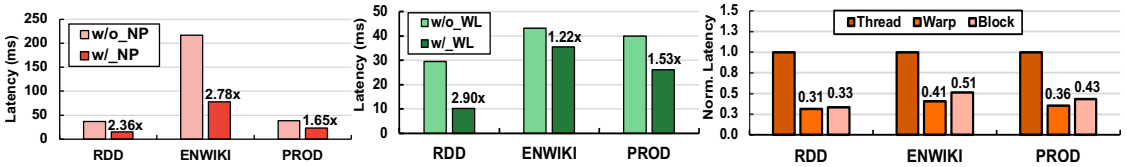


Figure 4.10: Optimization Analysis: (a) Neighbor Partitioning; (b) Workload Interleaving; (c) Choice of Communication Primitives.

comparison to 8 GPUs. Performance-critical ROC runtime configurations (e.g., #CPU cores, GPU/host memory size) are optimized to fully utilize the DGX-A100.

Figure 4.9 shows that MGG achieves averaged 12.30× and 9.35× speedups over ROC on GCN and GIN, respectively. MGG demonstrates a more pronounced speedup over ROC on the larger graph (e.g., IT04 and PAPER), which has more irregular neighbor embedding access. The Legion runtime of ROC relies on the DMA engine for bulky data (batched embeddings) transfer between host and GPU memory, leading to higher throughput but inferior latency performance. Besides, ROC relies on a separate communication-computation design, where computation happens after the full completion of communication. Such a design eliminates the opportunity to fill idle GPU cycles with computation during communication. In addition, the learning-based partitioning (to reduce communication) of ROC shows benefits on relatively smaller datasets (e.g., RDD and PROT) but hard to find optimal partition plans for large graphs due to the input structure complexity.

4.7.2 Optimization Analysis

Neighbor Partitioning (NP) We compare MGG with a baseline design without applying the neighbor partitioning technique (*i.e.*, each aggregation workload consists of all local/remote neighbors) on 4×A100. We apply the workload interleaving for both implementations and fix the warp-per-block size to 2 to eliminate the impact from other

performance-related factors. Figure 4.10(a) shows higher latency (averaged 2.26×) for designs without applying neighbor partitioning, since the workload imbalance becomes more severe across different warps without neighbor partitioning, especially for those graphs with many remote access demands, leading to limited computing parallelism and GPU underutilization.

Workload Interleaving (WL) We compare MGG with a baseline design without workload interleaving (*i.e.*, remote neighbor aggregation and local neighbor aggregation are mapped separately to the GPU warps. We fix the neighbor partition size to 16 and the warp-per-block size to 2. Figure 4.10(b) shows that MGG consistently outperforms the non-interleaved baseline with an average of 1.89× speedup. Without interleaving the local/remote workload, the workload distribution would be highly skewed, where the heavy and intensive remote aggregation would be gathered on certain warps close to each other while the lightweight local aggregation would be gathered on some other warps close to each other. This leads to inefficient warp scheduling and higher latency.

Communication Primitives We adopt MGG with different NVSHMEM primitives at the *thread*, *warp*, and *block* levels. We fix the number of GPUs to 2, the hidden dimension to 16, the neighbor partition size to 2, and the distance of workload interleaving to 2. Figure 4.10(c) shows that warp-level NVSHMEM primitives (*e.g.*, `nvshmemx_float_warp_get`) for remote accessing can bring the lowest latency. For thread-level NVSHMEM primitives (*e.g.*, `nvshmem_float_get`), it would not coalesce the remote memory access to reduce unnecessary transactions. For the block-level NVSHMEM primitives (*e.g.*, `nvshmemx_float_block_get`), the higher overhead comes from collaborating a block of threads for remote access, since thread blocks (usually consisting of multiple warps) is larger than a single warp, thus, leading to higher synchronization and scheduling cost. This study also shows that our choice of warp-level primitives strikes a good balance between memory access efficiency and scheduling flexibility.

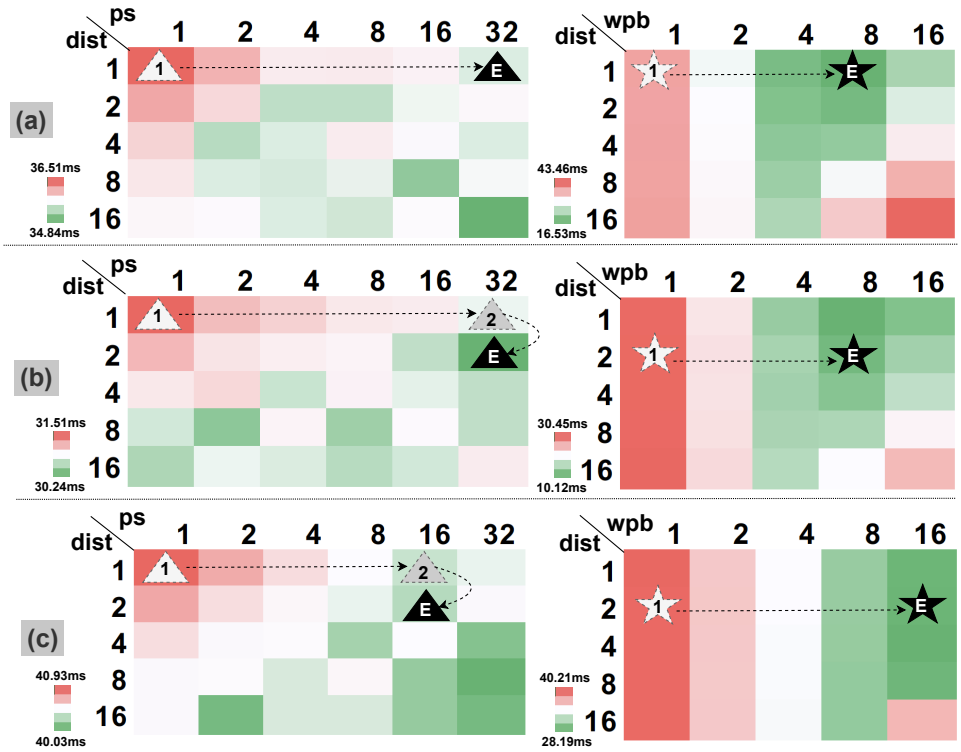


Figure 4.11: Parameter selection for three different settings. (a), (b), and (c) are for setting I, II, and III, respectively. Note that the left-side figures show the runtime latency for different combinations of *ps* and *dist*, while the right-side figures show the latency for different combinations of *wpb* and *dist*. The solid black triangle with "E" is the searched "optimal" combination for *ps* and *dist*, while the black solid star with "E" is the searched "optimal" *wpb* given *dist* and *ps*.

Modeling and Optimization We further analyze the effectiveness of our lightweight analytical model for design space search. Specifically, three key parameters are studied, the size of neighbor partitioning (*ps*), the interleaving distance (*dist*), and the warps per block (*wpb*). We consider three different settings on a 2-layer GCN model: I: RDD on 4×A100 as the basic setting. II: RDD on 8×A100 to demonstrate the adaptability toward the different numbers of GPUs. III: RDD on 4×V100 [79] to demonstrate the adaptability toward the different types of GPUs. We decompose searching results into two parts corresponding to the output of the ***second*** and ***third*** steps of the optimization discussed in §4.6.

Figure 4.11 shows that our performance modeling and parameter selection strategy can pinpoint the low-latency design for the above three settings. The overall searching process only requires about 10 iterations to reach the final "optimal" settings. Note that here we show latency results for all possible settings for comparison. While in practice, we only need to traverse a small part of the whole design space (as indicated by the boxes touched by the dot lines). By comparing the final optimal runtime configuration setting and the initial configuration, we can see that modeling and cross-iteration optimization can decrease the execution time by up to 68%. In the end-to-end GNN training (usually more than 100 iterations), such a latency saving would also be significant.

4.7.3 Additional Study

Accuracy-latency Tradeoff This study will analyze the accuracy-latency tradeoff between GNNs with sampling and full-graph (w/o sampling) on 8×A100. Table 4.3 shows an evident node classification accuracy increase (2% to 5%) of GNN w/o sampling over GNN w/ sampling. The accuracy of sampling-based GNN would be affected by many factors (*e.g.*, sampling rate at each GNN layer and graph structure). It is thus highly tricky to choose the "optimal" value for those factors. Here we follow the conventional way for GNN sampling [15]. The accuracy difference agrees with previous GNN algorithmic work [12]. In many real-world applications (e.g, e-commerce), such an accuracy advantage of full-graph GNNs are be more preferred by users. Because even 1% accuracy would make significant profit gains when deploying services at scale while the latency penalty is relatively minor.

Generality to other applications The design of MGG can be generalized to other similar applications. We demonstrate the typical and popular deep-learning recommendation model (DLRM) [95, 117, 118] that has been widely used in the industry. In

Table 4.3: Accuracy-Latency of GNNs w/ and w/o sampling.

Dataset	**Accuracy w/ sampling**	**Accuracy w/o sampling**	**Latency (w/o *vs.* w/ sampling)**
RDD	0.937	0.957	1.07×
PROT	0.776	0.825	1.25×

Table 4.4: DLRM [95] with MGG in Embedding Lookup.

Implementation	DLRM [95]	DLRM (MGG)
Time (ms)	315.27	119.66

multi-GPU DLRM, the large embedding tables are partitioned by rows and stored in different GPUs. The DLRM inputs (embedding access queries) will request embeddings from tables on different GPUs and then apply operations (e.g., elementwise addition or dot product) on those fetched embeddings. Such embedding lookup is highly sparse and irregular and dominates ($> 80\%$ latency [117, 119]) the overall DLRM computation. We improve the mainstream DLRM system [95] with the design and optimizations of MGG to accelerate embedding lookup and element-wise addition and compare with the original system (which relies on NCCL) [95] under 4-GPU settings on the popular Criteo Kaggle [120] dataset. Table 4.4 shows that DLRM with MGG effectively reduces the lookup time (2.64×). The fine-grained remote access of MGG can reduce redundant inter-GPU traffic by using NCCL and offset the cost by massively parallel GPU-initiated communication.

Chapter 5

TC-GNN: Bridging Sparse GNN Computation and Dense Tensor Cores on GPUs.

Recently, graph neural networks (GNNs), as the backbone of graph-based machine learning, have demonstrated great success in various domains (*e.g.*, e-commerce). However, the performance of GNNs is usually unsatisfactory due to the highly sparse and irregular graph-based operations. To this end, we propose **TC-GNN** [1] , the first GNN acceleration framework based on GPU Tensor Core Units (TCUs). The core idea is to reconcile the "Sparse" GNN computation with the high-performance "Dense" TCUs. Specifically, we conduct an in-depth analysis of the sparse operations in mainstream GNN computing frameworks. We introduce a novel sparse graph translation technique to facilitate TCU processing of the sparse GNN workload. We implement an effective CUDA core and TCU collaboration design to fully utilize GPU resources. We integrate TC-GNN with

[1]Published at USENIX ATC'23. USENIX permits authors to retain their ownership of the copyrights in their works. Reprinted from *TC-GNN: Bridging Sparse GNN Computation and Dense Tensor Cores on GPUs*. USENIX USENIX Annual Technical Conference. 07/2023.

the PyTorch framework for high programmability. Rigorous experiments show an average of 1.70× speedup over the state-of-the-art DGL framework across various models and datasets.

5.1 Introduction

Over the recent years, with the increasing popularity of graph-based learning, graph neural networks (GNNs) [10, 13, 121] become dominant in the computing of essential tasks across a wide range of domains, like e-commerce, financial services, and etc. Compared with standard methods for graph analytics, such as random walk [25, 51, 17] and graph laplacians [52, 53, 54], GNNs highlight themselves with significantly higher accuracy [10, 13, 11] and better generality [12]. From the computation perspective, GNNs feature an interleaved execution phase of both graph operations (scatter-and-gather [55]) at the ***Aggregation*** phase and Neural Network (NN) operations (matrix multiplication) at the ***Update*** phase. Our experimental studies further show that the aggregation phase which involves highly sparse computation on irregular input graphs generally takes more than 80% of the running time for both GNN training and inference. Existing GNN frameworks, *e.g.*, Deep Graph Library [15] and PyTorch Geometric [16], are mostly built upon the popular NN frameworks that are originally optimized for dense operations, such as general matrix-matrix multiplication (GEMM). To support sparse computations in GNNs, their common strategy is to incorporate sparse primitives (such as cuSPARSE [70]) for their backend implementations. However, cuSPARSE leverages the sparse linear algebra (LA) algorithm which involves lots of high-cost indirect memory accesses on non-zero elements of a sparse matrix. Therefore, cuSPARSE cannot enjoy the same level of optimizations (e.g., data reuse) as its dense counterpart, such as cuBLAS [32]. Moreover, cuSPARSE is designed to only utilize CUDA cores. Therefore,

It cannot benefit from advancements in GPU hardware features, like Tensor Core Units (TCUs) on the recent NVIDIA Ampere and Hopper GPUs. Such a design is also the trend of many other AI-tailored accelerators/units (e.g., Google TPU [122] and Matrix Core [123] on AMD GPUs) and can significantly boost the performance of dense LA algorithms (e.g., GEMM and Convolution) in most conventional deep-learning applications (e.g., CV [124] and NLP [125]).

This work focuses on exploring the potential of TCUs for accelerating such GNN-based graph learning and our design/optimization principles will also benefit other similar AI hardware [122, 123] for sparse deep-learning workloads. We remark that making TCUs effective for general GNN computing is a non-trivial task. Our initial study shows that naively applying the TCU to sparse GNN computation would even result in inferior performance compared with the existing sparse implementations on CUDA cores. There are several challenges. ***First***, directly resolving the sparse GNN computing problem with the pure dense GEMM solution is impractical due to the extremely large memory cost ($\mathcal{O}(N^2)$, where N is the number of nodes). Besides, traversing the matrix tiles already known to be filled with all-zero elements would cause excessive unnecessary computations and memory access. ***Second***, simply employing TCUs to process non-zero matrix tiles of the sparse graph adjacency matrix would still waste most of the TCU computation and memory access efforts. This is because TCU input matrix tiles are defined with fixed dimension settings (*e.g.*, $height(16) \times width(8)$), whereas the non-zero elements of a sparse graph adjacency matrix are distributed irregularly. Thus, it requires intensive zero-value padding to satisfy such a rigid input constraint. ***Third***, although the recent CUDA release update enables TCUs to exploit the benefit of certain types of sparsity [126], it only supports blocked SpMM, where non-zero elements must first fit into well-shaped blocks and the number of blocks must be the same across different rows. Such an input restriction makes it hard to handle highly irregular sparse graphs in real-world GNN

applications.

To this end, we introduce, **TC-GNN**[2], the first TCU-based GNN acceleration design on GPUs. Our key insight is to *let the sparse input graph fit the dense computation of TCUs*. ***At the input level***, instead of exhaustively traversing all sparse matrix tiles and determining whether to process each tile, we develop a new *sparse graph translation* (SGT) technique that can effectively identify those non-zero tiles and condense non-zero elements from these tiles into fewer number of "dense" tiles. Our major observation is that neighbor sharing is very common among nodes in real-world graphs. Therefore, applying SGT can effectively merge the unnecessary data loading of the shared neighbors among different nodes to avoid high-cost memory access. SGT is generic to any kind of sparse pattern of input graphs and can always yield the correct results as the original sparse algorithm. ***At the GPU kernel level***, for efficiently processing GNN sparse workloads, TC-GNN exploits the benefits of CUDA core and TCU collaboration. The major design idea is that the CUDA core, which is more powerful at fine-grained thread-level execution, would be a good candidate for managing memory-intensive data access. While TCU, which is more powerful in handling simple arithmetic operations (*e.g.*, multiplication and addition), would be well-suited for compute-intensive GEMM on dense tiles generated from SGT. ***At the framework level***, we integrate TC-GNN with the popular PyTorch framework. Thereby, users only need to interact with their familiar PyTorch programming environment by using TC-GNN APIs. This can significantly reduce extra learning efforts, and improve user productivity and code portability.

To sum up, we summarize our contributions as follows:

- We conduct a detailed analysis (§5.2) of existing solutions (*e.g.*, SpMM on CUDA cores) and identify the potentials of TCUs for accelerating sparse GNN workloads.

[2]TC-GNN is open-sourced at `https://github.com/YukeWang96/TC-GNN_ATC23.git`

- We introduce a sparse graph translation technique (§5.3.1). It can make the sparse and irregular GNN input graphs easily fit the dense computing of TCUs for acceleration.
- We build a TCU-tailored algorithm (§5.3.2) and GPU kernel design (§5.3.3) for CUDA core and TCU collaboration on GPUs to handle different sparse GNN computation.
- Extensive experiments show TC-GNN achieves 1.70× speedup on average over the state-of-the-art GNN computing framework, Deep Graph Library, across various mainstream GNN models and dataset settings.

5.2 Motivation

In this section, we will discuss the major technical thrust for us to leverage TCUs for accelerating sparse GNN computation. We use the optimization of SpMM as the major example in this discussion, and the acceleration of SDDMM would also benefit from similar optimization principles.

5.2.1 SpMM on CUDA cores

As the major component of sparse linear algebra operation, SpMM has been incorporated in many off-the-shelf libraries [70, 127, 128, 129, 130]. The close-sourced cuSPARSE [70] library developed by NVIDIA is the most popular solution and it can deliver state-of-the-art performance for most GPU-based SpMM computation. cuSPARSE has also been widely adopted by many GNN frameworks, such as Deep Graph Library (DGL) [15], as the backend for sparse operations. To understand its characters, we profile DGL on one layer of a GCN [10] model (*neighbor aggregation* + *node update*) on NVIDIA

Table 5.1: Profiling of GCN Sparse Operations.

Dataset	Aggr. (%)	Update (%)	Cache(%)	Occ.(%)
Cora	88.56	11.44	37.22	15.06
Citeseer	86.52	13.47	38.18	15.19
Pubmed	94.39	5.55	37.22	16.24

RTX3090. We report two key kernel matrices for only neighbor aggregation kernel, including L1/texture cache hit rate (*Cache*) and the achieved Streaming-Multiprocessor (SM) occupancy (*Occ.*). We select three representative GNN datasets: Cora with 3,327 nodes, 9,464 edges, and 3,703 node embedding dimensions; Citeseer with 2,708 nodes, 10,858 edges, and 1,433 dimensions; Pubmed with 19,717 nodes, 88,676 edges, and 500 dimensions.

From Table 5.1, we have several observations: ***First***, the aggregation phase usually dominates the overall execution of the GNN execution. From these three commonly used GNN datasets, we can see that the aggregation phase usually takes more than 80% of the overall execution time, which demonstrates the key performance bottleneck of the GNNs is to improve the performance of the sparse neighbor aggregation. ***Second***, sparse operations in GNNs show very low memory performance. The column *Cache* of Table 5.1 shows GNN sparse operations could not well benefit from the GPU cache system, thus, showing a low cache-hit ratio (around 37%) and frequent global memory access. ***Third***, sparse operations of GNNs show very inefficient computation. As described in the column *Occupancy* of Table 5.1, the sparse operation of GNNs could hardly keep the GPU busy because 1) its low computation intensity (the number of non-zero elements in the sparse matrix is generally small); 2) its highly irregular memory access for fetching rows of the dense matrix during the computation, resulting in memory-bound computation; 3) it currently can only leverage CUDA cores for computation, which naturally has limited throughput performance. On the other side, this study also points out several potential directions for improving the SpMM performance on GPUs, such as improving the com-

Table 5.2: Medium-size Graphs in GNNs.

Dataset	# Nodes	# Edges	Memory	Eff.Comp
OVCR-8H	1,890,931	3,946,402	14302.48 GB	0.36%
Yeast	1,714,644	3,636,546	11760.02 GB	0.32%
DD	334,925	1,686,092	448.70 GB	0.03%

putation intensity (*e.g.*, assigning more workload to each thread/warp/block), boosting memory access efficiency (*e.g.*, crafting specialized memory layout for coalesced memory access), and breaking the computation performance ceiling (*e.g.*, using TCUs).

5.2.2 Dense GEMM on CUDA Cores/TCUs

While the dense GEMM is mainly utilized for dense NN computation (*e.g.*, linear transformation and convolution), it can also be leveraged for GNN aggregation under some circumstances. For example, when an input graph has a very limited number of nodes, we can directly use the dense adjacency matrix of the graph and accelerate the intrinsically sparse neighbor aggregation computation on CUDA cores/TCUs by calling cuBLAS [32]. However, such an assumption may not hold even for medium-size graphs in real-world GNNs.

As shown in Table 5.2, for these selected datasets, the memory consumption of their dense graph adjacent matrix (as a 2D `float` array) would easily exceed the device memory constraint of today's GPU (less than 100GB). Even if we assume the dense adjacent matrix can fit into the GPU memory, the extremely low effective computation (the last column of Table 5.2) would also be a major obstacle for us to achieve high performance. We measure the effective computation as $\frac{nnz}{N \times N}$, where nnz is the number of the non-zero elements (indicating edges) in the graph adjacent matrix and N is the number of nodes in the graph. The number of nnz is tiny in comparison with the $N \times N$. Therefore, computation and memory access on zero elements are wasted.

5.2.3 Hybrid Sparse-Dense Solution

Another type of work [131, 126] takes the path of mixing the *sparse control* (tile-based iteration) with *Dense GEMM computation*. They first apply a convolution-like (2D sliding window) operation on the adjacent matrix and traverse all possible dense tiles that contain non-zero elements. Then, for all identified non-zero tiles, they invoke GEMM on CUDA cores/TCUs for computation. However, this strategy has two shortcomings. ***First***, the sparse control itself would cause a high overhead. Based on our empirical study, the non-zero elements are highly scattered on the adjacent matrix of a sparse graph. Therefore, traversing all blocks in a super large adjacent matrix would be time-consuming. ***Second***, the identified sparse tiles would still waste lots of computation. The irregular edge connections of the real-world graphs could hardly fit into these fixed-shape tile frames. Therefore, most of the dense tiles would still have very few non-zero elements.

Inspired by the above studies, we make several design choices in order to achieve high-performance sparse GNN operations. ***First***, we choose the hybrid sparse-dense solution as the starting point. This can give us more flexibility for optimizations at the sparse control (*e.g.*, traversing fewer tiles) and dense computation (*e.g.*, increasing the effective computation/memory access when processing each tile). ***Second***, we employ shared memory as the key space for GPU kernel-level data management. It can help us to re-organize the irregular GNN input data in a more "regularized" way such that both the memory access efficiency and computing performance can be well improved. ***Third***, we choose TCUs as our major computing unit since they can bring significantly higher computing throughput performance in comparison with CUDA cores. This also indicates the great potential of using TCUs for harvesting more performance gains.

Finally, we crystallize all of our ideas and insights into TC-GNN that effectively coor-

Table 5.3: Comparison among Sparse GEMM, Dense GEMM, Hybrid Sparse-Dense, and TC-GNN. Note that **MC**: Memory Consumption, **EM**: Effective Memory Access, **CI**: Computation Intensity, **EC**: Effective Computation.

Solution	**MC**	**EM**	**CI**	**EC**
Sparse GEMM (§5.2.1)	Low	Low	Low	High
Dense GEMM (§5.2.2)	High	High	High	Low
Hybrid Sparse-Dense (§5.2.3)	High	Low	Low	High
TC-GNN (This work)	**Low**	**High**	**High**	**High**

dinates the execution of GNN sparse operations on dense TCU. We show a brief qualitative comparison among TC-GNN and the above three solutions in Table 5.3. Note that *Memory Consumption* is the size of memory used by the sparse/dense graph adjacency matrix; The *Effective Memory Access* is the ratio between the size of the accessed data that is actually involved in the later computation and the total data being accessed; The *Computation Intensity* is the ratio of computing operations versus the data being accessed; The *Effective Computation* is the operations for generating the final result versus the total operations.

5.3 TC-GNN Design

In this section, we will first give an overview of TC-GNN through its high-level programming interface and then detail the TCU-aware GNN algorithm design. As detailed in Listing 5.1, TC-GNN consists of several key components to facilitate the programming of GNN models on GPU TCUs. TC-GNN introduces a set of pre-built popular GNN layers (e.g., `TCGNN.GCNConv`) that can be easily connected with some other existing neural network layers (e.g., `ReLU` and `softmax`), to help users define their own GNN model quickly. For those non-conventional GNN layers, users can directly use our low-level APIs (e.g., `TCGNN.spmm` and `TCGNN.sddmm`) to express the GNN computation easily. TC-GNN introduces an input **`Loader`** to load the GNN input graph as a *rawGraph* and capture the key

Listing 5.1: Example of a 2-layer GCN in TC-GNN.

```
import TCGNN, torch
# include other packages ...
class GCN(torch.nn.Module):
    def __init__(self, inDim, hiDim, outDim):
        self.layer1 = TCGNN.GCNConv(inDim, hiDim)
        self.layer2 = TCGNN.GCNConv(hiDim, outDim)
        self.softmax = torch.nn.Softmax()

    def forward(self, tiledGraph, param):
        tiled_adj, X = tiledGraph.adj, tiledGraph.X
        X = self.layer1(X, tiledAdj, param)
        X = self.ReLU(X)
        X = self.layer2(X, tiledAdj, param)
        X = self.softmax(X)
        return X
# Define a two-layer GCN model in TC-GNN.
model = GCN(inDim=100, hiDim=16, outDim=10)
# Load graph and extract input information.
rawGraph, info = TCGNN.Loader(graphFilePath)
# Generate TCU tile and runtime configuration.
tiledGraph, config = TCGNN.Preprocessor(rawGraph, info)
# Run model through forward computation.
predict_y = model(tiledGraph, config)
# Compute loss and accuracy.
# Gradient backpropagation for training.
```

input information for system-level optimizations. TC-GNN incorporates a **`Preprocessor`** to build tiles from *rawGraph* and generate TCU-aware *tiledGraph* (§5.3.1), and optimize runtime configuration (e.g., warps per block) for our TCU-tailored GPU kernel (§5.3.2 and §5.3.3) based on input. Finally, we train the initialized GNN model defined in TC-GNN as the regular GNN models defined in other frameworks through forward and backward computation.

5.3.1 TCU-aware Sparse Graph Translation

As the major component of TC-GNN, we introduce a novel *Sparse Graph Translation* (SGT) technique to facilitate the TCU acceleration of GNNs. Our core idea is that *the pattern of the graph sparsity can be well-tuned for TCU computation through effective graph structural manipulation meanwhile guaranteeing output correctness*. Our key observation is that neighbor sharing is common in real-world graphs and has been exploited

Algorithm 4 TCU-aware Sparse Graph Translation.

Require: Graph adjacency matrix **A** (*nodePointer*, *edgeList*).
Ensure: Result of *winPartition* and *edgeToCol*.

▷ Compute the total number of row windows.
$numRowWin = \lceil \frac{numNodes}{winSize} \rceil$
for *winId* **in** 1 to *numRowWin* **do**
  ▷ EdgeIndex range of the current rowWindow.
    $winStart = nodePointer[winId \cdot winSize]$
    $winEnd = nodePointer[(winId + 1) \cdot winSize]$
  ▷ Sort the edges of the current rowWindow.
    $eArray = \textbf{Sort}(winStart, winEnd, edgeList)$
  ▷ Deduplicate edges of the current rowWindow.
    $eArrClean = \textbf{Deduplication}(eArray)$
  ▷ #TC blocks in the current rowWindow.
    $winPartition[winId] = \lceil \frac{eArrClean.size}{TC\_BLK\_w} \rceil$
  ▷ Edges-to-columnID mapping in TC Blocks.
  for *eIndex* **in** $[winStart, winEnd]$ **do**
    $eid = edgeList[eIndex]$
    $edgeToCol[eIndex] = eArrClean[eid]$
  end for
end for

for various tasks like link prediction [132]. Our evaluated datasets (Section 5.4) have 18% to 47% (averaged 29%) neighbor similarity. Specifically, we condense (remap) the highly-scattered neighbor ids into highly-condensed new neighbor ids that can facilitate the dense TCU computation paradigm. Also, such condensing should not compromise any original information (*e.g.*, edge connections) and can generate the exact output as the conventional design.

As exemplified in Figure 5.1a and Figure 5.1b, we take the regular graph in CSR format as the input and condense the columns of each row window (in the red-colored rectangular box) to build TCU blocks (TC_block) (*a.k.a.*, the input operand shape of a single MMA instruction), in the orange-colored rectangular box. *nodePointer* is the row pointer array *edgeList* is the edges of each node stored continuously. In this paper, we demonstrate the use of standard MMA shape for TF-32 of TCU on Ampere GPU architecture, and other MMA shapes [133] can also be used under different precision

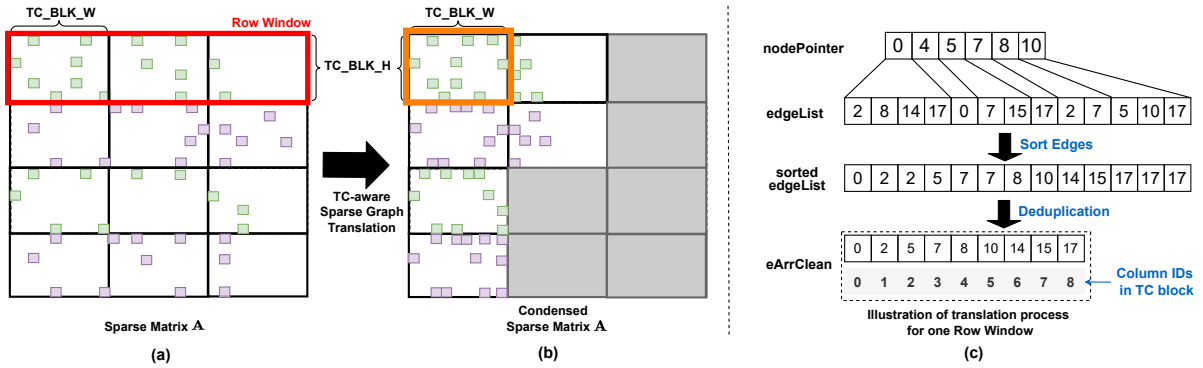


Figure 5.1: Illustration of Sparse Graph Translation. Note that the grey area indicates the TCU blocks that will be directly skipped.

(*e.g.*, `half` and `int8`) and GPU architecture (*e.g.*, Turing).

SGT takes several steps for processing each row window, as detailed in Algorithm 4 and visualized in Figure 5.1c. *winPartition* is an array for maintaining the number of TC blocks in each row window. *edgeToCol* is an array for maintaining the mapping between the edges and their corresponding position in the graph after SGT. Note that *edgeToCol* has the same length as *edgeList* but with *column-id* from *eArrClean*. *colToRow* maps *column-id* of adjacency matrices to the *row-id* of embedding matrices. We choose the size of the row window ($winSize$=TC_BLK_H) and column width (TC_BLK_W) according to TCU MMA specification (*e.g.*, TC_BLK_H=16, TC_BLK_W=8 in TF-32). After condensing the graph within each row window, the time complexity of sliding the TC_block can be reduced from $\mathcal{O}(\frac{N}{TC\_BLK\_W})$ to only $\mathcal{O}(\frac{nnz_{unique}}{TC\_BLK\_W})$, where N is the total number of nodes in the graph and nnz_{unique} is the size of the unique neighbor within the current row window, which equals $eArrClean.size$ in Algorithm 4. The density (computation intensity) of each identified TCU block can be largely improved. Considering the case in Figure 5.1, after the sparse graph translation, we can achieve 2× higher density on individual TCU blocks (Figure 5.1b) compared with the original one (Figure 5.1a).

Compared to existing sparse matrix formats (e.g., Blocked-Ellpack [126]) which use

Algorithm 5 TC-GNN Neighbor Aggregation.

Require: Condensed graph structure ($nodePointer$, $edgeList$, $edgeToCol$, $winPartition$) and node embedding matrix ($\mathbf{X}$).
Ensure: Updated node embedding matrix ($\hat{\mathbf{X}}$).

for $winId$ **in** 1 to $numRowWindows$ **do**
  ▷ #TC blocks of the row window.
  $numTCblocks = winPartition[winId]$
  ▷ edge range of TC blocks of the row window.
    $edgeRan =$ **GetEdgeRange**$(nodePointer, winId)$
  for $TCblkId$ **in** 1 to $numTCblocks$ **do**
    ▷ The edgeList chunk in current TC block.
      $edgeChunk =$ **GetChunk**$(edgeList, edgeRan, TCblkId)$
    ▷ Neighbor node Ids in current TC block.
      $colToNId =$ **GetNeighbors**$(edgeChunk, edgeToCol)$
    ▷ Initiate a dense tile ($ATile$).
      $ATile =$ **InitSparse**$(edgeChunk, winId)$
    ▷ Initiate a dense tile ($XTile$).
      $XTile, colId =$ **FetchDense**$(colToNId, X)$
    ▷ Compute $XnewTile$ via TCU GEMM.
      $XnewTile =$ **TCcompute**$(ATile, XTile)$
    ▷ Store $XnewTile$ of $\hat{X}$.
      $\hat{X} =$ **StoreDense**$(XNewTile, winId, colId)$
  end for
end for

the regular matrix tiles to cover the irregularly scattered non-zero elements, SGT reduces the irregularity of non-zero-elements layout to fit them into fewer number TCU blocks, thus, reducing the unnecessary computation and memory overhead. SGT is applicable for both the SpMM and SDDMM in GNN sparse operations and can be easily parallelized because the processing of individual row windows is independent. In most cases, SGT only needs to execute once and its result can be reused across many epochs/rounds of GNN training/inference.

Additionally, SGT can be generally used with other accelerators (e.g., AMD-GPUs with matrixCore and TPUs) that offer similar dense MM primitives. CPUs have no direct alternative to TensorCore-like MM primitives. However, with AVX-vectorized instructions, CPUs can benefit from SGT by setting BLK_H=1 and BLK_W=(#elements-per-AVX-instruction). TC-GNN currently targets GNN training. SGT is conducted once before training. SGT cost can be offset by training iterations (averaged 2% for 200

iterations as DGL).

5.3.2 TCU-tailored GNN Computation

Besides the effective way to condense the sparse tiles, the next major challenge is *how to tailor the computation schedule of GNN algorithms* so that we can capitalize on the performance of condensed sparse graphs and the powerful TCUs. We focus on two major types of computation in GNNs.

Neighbor Aggregation The major part of GNN sparse computing is neighbor aggregation, which can generally be formalized as SpMM operations by many state-of-the-art frameworks [15]. And they employ the cuSPARSE [70] on CUDA cores as a black-box technique for supporting sparse GNN computation. In contrast, our TC-GNN design targets at TCU for the major neighbor aggregation computation which demands a specialized algorithmic design. TC-GNN focuses on maximizing the net performance gains by gracefully batching the originally highly irregular SpMM as dense GEMM computation and solving it on TCU effectively. As illustrated in Algorithm 5, the node aggregation processes all TC blocks from each row window. *nodePointer* and *edgeList* are directly from graph CSR, while *edgeToCol* and *winPartition* are generated from SGT discussed in the previous section. Note that **InitSparse** is to initialize a sparse tile in dense format according to the translated graph structure of the current TC block. Meanwhile, **FetchDense** returns a dense node embedding matrix tile $XTile$ for TCU computation, and the corresponding column range $colId$ (embedding dimension range) of matrix $\mathbf{X}$. This is to handle the case that the width of one $XTile$ could not cover the full-width (all dimensions) of $\mathbf{X}$. Therefore, the $colId$ will be used to put the current TCU computation output to the correct location in the updated embedding matrix $\hat{\mathbf{X}}$.

Edge Feature Computing Previous research [121, 11] has demonstrated the great

Algorithm 6 TC-GNN Edge Feature Computation.

Require: Condensed graph data (*nodePointer*, *edgeList*, *edgeToCol*, *winPartition*) and node embedding matrix (**X**).
Ensure: Edge Feature List (*edgeValList*).

for *winId* **in** 1 to *numRowWin* **do**
  ▷ #TC blocks in the row window.
  $numTCblocks = winPartition[winId]$
  ▷ Edge range of TC blocks of the row window.
    $edgeRan =$ **GetEdgeRange**$(nodePointer, winId)$
  for *TCblkId* **in** 1 to *numTCblocks* **do**
    ▷ EdgeList chunk in current TC block.
      $edgeChunk =$ **GetChunk**$(edgeList, edgeRan, TCblkId)$
    ▷ Neighbor node Ids in current TC block.
      $colToNId =$ **GetNeighbors**$(edgeChunk, edgeToCol)$
    ▷ Fetch a dense tile ($XTile_A$).
      $XTile_A =$ **FetchDenseRow**$(winId, TCblkId, X)$
    ▷ Fetch a dense tile ($XTile_B$).
      $XTile_B =$ **FetchDenseCol**$(colToNId, edgeToCol, X)$
    ▷ Compute *edgeValTile* via TCU GEMM.
      $edgeValTile =$ **TCcompute**$(XTile_A, XTile_B)$
    ▷ Store *edgeValTile* to *edgeValList*.
      StoreSparse$(edgeValList, edgeValTile, edgeList, edgeToCol)$
  end for
end for

importance of incorporating the edge feature for a better GNN model algorithmic performance (*e.g.*, accuracy, and F1-score). The underlying building block to generate edge features is the Sampled Dense-Dense Matrix Multiplication (SDDMM)-like operation. In TC-GNN, we support SDDMM with the collaboration of the above sparse graph translation and TCU-tailored algorithm design, as described in Algorithm 6. The overall algorithm structure and inputs are similar to the above neighbor aggregation. The major difference is the output. In the case of neighbor aggregation, our output is the updated dense node embedding matrix ($\hat{\mathbf{X}}$), where edge feature computing will generate a sparse output with the same shape as the graph edge lists. Note that fetching the $XTile_A$ only needs to consecutively access the node embedding matrix **A** by rows while fetching the $XTile_B$ requires first computing the TCU block column-id to node-id (*colToNId*) to fetch the corresponding neighbor node embeddings from the same node embedding matrix **X**.

Despite the dataflow similarity with dense-GEMM computation (e.g., CUTLASS [33]),

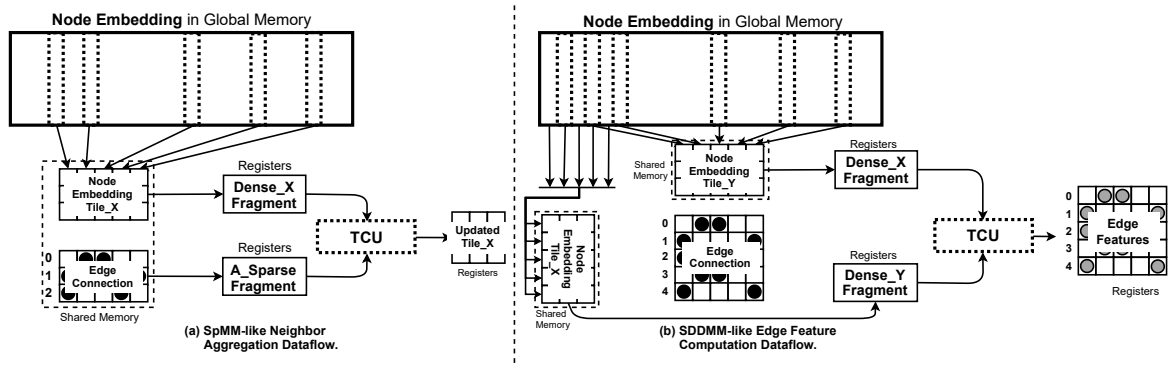


Figure 5.2: TCU-optimized Dataflow Design for (a) Neighbor Aggregation and (b) Edge Feature Computing in GNNs.

TC-GNN has to overcome the limited parallelism (imbalance workload) and sparse/irregular access with novel algorithmic and kernel designs. While these challenges are absent in dense-GEMM computation with naturally high parallelism and data-access locality.

5.3.3 TCU-centric Workload Mapping

In collaborating with our TCU-tailored algorithm design, an effective mapping of our algorithmic design to low-level GPU primitives is indispensable for high-performance delivery. We discuss two key techniques: *GPU-aware Workload Decomposition* and *TCU-optimized dataflow design*.

GPU-aware Workload Decomposition

Different from previous work [15, 16] focusing on CUDA cores only, TC-GNN highlights itself with CUDA core and TCU collaboration through effective two-level workload mapping. The idea is based on the fact that CUDA cores work in SIMT fashion and are operated by individual threads, while TCU designated for GEMM computation requires collaboration from a warp of threads (32 threads). Our key design principle is to *mix these two types of computing units as a single GPU kernel*, which can efficiently coordinate the kernel execution at different levels of execution granularity.

In TC-GNN, we operate CUDA cores by thread blocks and manage TCU by thread warps. Specifically, threads running CUDA cores from the same thread block will load data (*e.g.*, edges) from the global memory to shared memory. Note that in our design we assign each row window (discussed in §5.3.1) to one thread block. The number of threads in each block should be divisible by the number of threads in each warp (32) for better performance. Once threads running on CUDA cores (CUDA-core threads) finish the data loading, threads from each warp (TCU threads) will operate TCU for GEMM computation (including loading the data from the shared memory to thread-local registers (fragments), applying GEMM computation on data in registers, accumulating results on registers, and storing the final results back to global memory). Note that there would be a large overlap of the CUDA-core threads and TCU threads, both of which are threads from the same blocks but running at a different time frames. In general, we use more CUDA-core threads than TCU threads considering that global memory access demands more parallelization.

There are two major benefits of such two-level workload decomposition. First, threads from the same block can work together to improve the memory access parallelization to better utilize memory bandwidth. Second, warps from the same block can reuse the loaded data, including the information (*e.g.*, column index mapping) of the translated graph and the tiles from the dense node embedding matrix. Therefore, we can avoid redundant high-cost global memory operations.

TCU-optimized Dataflow Design

As the major technique to improve the GPU performance, shared memory is customized for our TCU-based sparse kernel design for re-organizing data layout for dense TCU computation and reducing the redundant global memory traffic. Our design takes the TCU specialty into careful consideration from two aspects, 1) the input matrix tile

size of the TCU, which is M(16)×N(16)×K(8) in the case of TF-32, and 2) the tile fragment layout for fast computation. The common practice of the loaded tile A and B are stored in row-major and column-major for better performance. Next, we will detail our TCU-optimized dataflow design for both neighbor aggregation and edge feature computation.

Neighbor Aggregation In Figure 5.2a, shared memory is mainly used for caching several most frequently used information, including the tile of sparse matrix A (`sparse_A`), the column-id of the sparse matrix **A** to row-id of the node embedding matrix of a graph **X** (`sparse_AToX_index`), and the dense tile of **X** (`dense_X`). When handling each TCU block, we assign all threads from the same block of threads for loading the sparse tile while allowing several warps to concurrently load the dense row tile from the matrix X. The reasons for enforcing such caching are two-fold. First, it can bridge the gap between the sparse graph data and the dense GEMM computing that requires continuous data layout. For example, the adjacent matrix **A** is input as CSR format that **cannot** be directed feed to TCU GEMM computation, therefore, we use a shared memory `sparse_A` to initialize its equivalent dense tile. Similarly, we cache rows of **X** according to the columns of **A** to the row of **X** mapping after our sparse graph translation, where originally scattered columns of **A** (the rows of **X**) are condensed. Second, it can enable data reuse on `sparse_AToX_index` and `sparse_A`. This is because in general, the `BLK_H` (16) cannot cover all dimensions of a node embedding (*e.g.*, 64), multiple warps will be initiated of the same block to operate TCU in parallel to work on non-overlapped dense tiles while sharing the same sparse adjacency matrix tile.

Edge Feature Computation Similar to the shared memory design in neighbor aggregation, for edge feature computing, as visualized in Figure 5.2b, the shared memory is utilized for `sparse_A`, `sparse_AToX_index`, and `dense_X`. We assign all threads from the same block of threads for loading the sparse tile while allowing several warps to

concurrently load the dense row tile from the matrix $\mathbf{X}$. Compared with dataflow design in neighbor aggregation, edge feature computing demonstrates several differences.

First, the sizes of `sparse_A` are different. In the neighbor aggregation computation, the sparse matrix $\mathbf{A}$ is used as one operand in the SpMM-like computation, therefore, the minimal processing granularity is 16×8, while in edge feature computing by following SDDMM-like operation, the sparse matrix $\mathbf{A}$ serves as the output matrix, thus, maintaining the minimum processing granularity is 16×16. To reuse the same translated sparse graph as SpMM, we need to recalculate the total number of TC blocks. ***Second***, iterations along the embedding dimension would be different. Compared with neighbor aggregation, edge feature computing requires the result accumulation along the embedding dimension. The result will only be output until all iterations have finished. In neighbor aggregation, the node embedding vector is divided among several warps, each of which will output their aggregation result to non-overlapped embedding dimension ranges in parallel. ***Third***, the output format has changed. Compared with SpMM-like neighbor aggregation which directly output computing results as an updated dense matrix $\hat{\mathbf{X}}$, SDDMM-like edge feature computing requires a sparse format (the same shape as `edgeList`) output for compatibility with neighbor aggregation and memory space. Therefore, one more step of dense-to-sparse translation is employed.

5.4 Evaluation

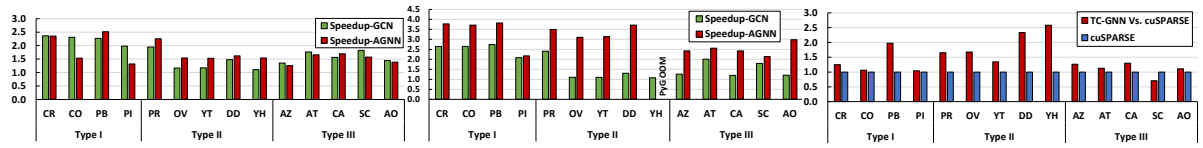
Benchmarks: We choose two representative GNN models widely used by previous work [15, 16, 14] on *node classification* tasks. Specifically, 1) Graph Convolutional Network (**GCN**) [10] is one of the most popular GNN model architectures. It is also the key backbone for many other GNNs (e.g., GraphSAGE [12] and differentiable pooling (Diffpool) [41]). Therefore, improving the performance of GCN will also benefit a broad range

Table 5.4: Datasets for evaluation.

Type	Dataset	Abbr.	#Vertex	#Edge	Dim.	#Class
I	Citeseer	CR	3,327	9,464	3703	6
	Cora	CO	2,708	10,858	1433	7
	Pubmed	PB	19,717	88,676	500	3
	PPI	PI	56,944	818,716	50	121
II	PROTEINS_full	PR	43,471	162,088	29	2
	OVCAR-8H	OV	1,890,931	3,946,402	66	2
	Yeast	YT	1,714,644	3,636,546	74	2
	DD	DD	334,925	1,686,092	89	2
	YeastH	YH	3,139,988	6,487,230	75	2
III	amazon0505	AZ	410,236	4,878,875	96	22
	artist	AT	50,515	1,638,396	100	12
	com-amazon	CA	334,863	1,851,744	96	22
	soc-BlogCatalog	SC	88,784	2,093,195	128	39
	amazon0601	AO	403,394	3,387,388	96	22

of GNNs. For GCN evaluation, we use the setting: *2 layers with 16 hidden dimensions per layer*, which is also the setting from the original paper [10]. 2) Attention-based Graph Neural Network (**AGNN**) [121]. AGNN differs from GCN in its aggregation function, which computes edge features (via embedding vector dot-product between source and destination vertices) before the node aggregation. AGNN is also the reference architecture for many other recent GNNs for better model algorithmic performance. For AGNN, we use: *4 layers with 32 hidden dimensions per layer*.

Baselines: 1) Deep Graph Library (**DGL**) [15] is the state-of-the-art GNN framework on GPUs, which is built with the high-performance CUDA-core-based cuSPARSE [70] library as the backend and uses PyTorch as its front-end programming interface. DGL significantly outperforms other existing GNN frameworks [16] over various datasets on many mainstream GNN model architectures. Therefore, we make an in-depth comparison with DGL. 2) PyTorch Geometric (**PyG**) [16] is another GNN framework. PyG leverages torch-scatter [69] library (highly-engineered CUDA-core kernel) as the backend support, which highlights its performance on batched small graph settings; 3) Blocked-SpMM [126] (**bSpMM**) accelerates SpMM on TCU. It is included in the recent update on the cuSPARSE library. bSpMM requires the sparse matrix with Blocked-Ellpack format for



(a) (b) (c)

Figure 5.3: Speedup over (a) DGL and (b) PyG on GCN and AGNN; (c) Speedup over cuSPARSE bSpMM on TCUs.

computation. Its computation on non-zero blocks can be seen as the hybrid sparse-dense solution (§5.2.3). Note that the bSpMM has not been incorporated into any existing GNN frameworks. We also compare TC-GNN with **tSparse** [134] and **Triton** [135] for non-vendor-developed highly optimized kernels on TCUs.

Datasets, Platforms, and Metrics: We cover three types of datasets (Table 5.4), which have been used in previous GNN-related work [15, 16, 14]. Specifically, **Type I** graphs are the typical datasets used by previous GNN algorithm papers [10, 13, 12]. They are usually small in the number of nodes and edges, but rich in node embedding information with high dimensionality. **Type II** graphs [42] are the popular benchmark datasets for graph kernels and are selected as the built-in datasets for PyG [16]. Each dataset consists of a set of small graphs, which only have intra-graph edge connections without inter-graph edge connections. **Type III** graphs [82, 10] are large in terms of the number of nodes and edges. These graphs demonstrate high irregularity in its structures, which are challenging for most of the existing GNN frameworks. The core design of TC-GNN consists of around 2.5K lines of code. TC-GNN backend is implemented with C++ and CUDA C, and its front end is implemented in Python. Our major evaluation platform is a server with an 8-core 16-thread Intel Xeon Silver 4110 CPU and an NVIDIA RTX3090 GPU. To measure the performance speedup, we calculate the average latency of 200 end-to-end runs.

5.4.1 Compared with DGL

Figure 5.3a shows that TC-GNN achieves 1.70× speedup on average compared to DGL over three types of datasets across GCN and AGNN models on end-to-end training. Our kernel profiling via Nsight Compute shows that TC-GNN achieves high SM occupancy (averaged 85.28%), which is on average 21.05% higher compared to DGL across all datasets.

Type I Graphs: The performance improvements against DGL are significantly higher for GCN (on average 2.23×) compared to AGNN (on average 1.93×). The major reason is their different GNN computation patterns. GCN only consists of a neighbor aggregation phase (SpMM-like operation) and a node update phase (GEMM operation). Whereas in the AGNN, the aggregation phase would also require an additional edge attention value (feature) computation based on SDDMM-like operations. Compared with SpMM-like operations, edge attention computation in SDDMM is more sensitive to the irregular sparse graph structure because of much more intensive computations and memory access. Thus, the performance improvement is relatively lower.

Type II Graphs: TC-GNN achieves averaged 1.38× speedup on GCN and 1.70× speedup on AGNN for the Type II graphs. Speedup on Type II graphs is relatively lower compared with Type I, since Type II datasets consist of a set of small graphs with very dense intra-graph connections but no inter-graph edges. This leads to a lower benefit from the sparse graph translation that would show more effectiveness on highly irregular and sparse graphs. Such a clustered graph structure would also benefit cuSPARSE due to more efficient memory access, *i.e.*, less irregular data fetching from the sparse matrix. In addition, for AGNN, TC-GNN can still demonstrate evident performance benefits over the DGL (CUDA core only) that can mainly contribute to TCU-based SDDMM-like designs that can fully exploit the power of GPU through an effective TCU and CUDA

Table 5.5: Compare TC-GNN with tSparse and Triton.

Dataset	tSparse (ms)	Triton (ms)	TC-GNN (ms)
AZ	18.60	31.64	4.09
AT	9.15	12.86	3.06
CA	13.84	15.50	3.26
SC	9.74	14.38	3.59
AO	11.93	21.78	3.41

core collaboration.

Type III Graphs: The speedup is also evident (on average 1.59× for GCN and average 1.51× for AGNN) on graphs with a large number of nodes and edges and irregular graph structures. The reason is the high overhead global memory access can be well reduced through our spare graph translation. Besides, our dimension-split strategy further facilitates efficient workload sharing among warps by improving the data spatial/temporal locality. On the dataset AT and SC, which have a higher average degree within Type III datasets, we notice a better speedup performance for both GCN and AGNN. This is because 1) more neighbors per node can lead to a higher density of non-zero elements within each tile/fragment. Thus, it can fully exploit the computation benefits of each TCU GEMM operation; 2) it can also facilitate more efficient memory access. For example, in AGNN, fetching one dense embedding x from the dense matrix X can be reused more times by applying a dot-product between x and many columns of the dense matrix X^T (neighbors embeddings).

Additionally, our performance breakdown analysis shows that for graphs with highly scattered and irregular edge distribution, such as Type I and III graphs, SGT would contribute more (averaged 64%) to the overall performance improvements since it helps significantly reduce the unnecessary workload. For graphs with highly dense and more regular edge connections, such as Type II datasets, SGT contributes relatively minor (averaged 23%) to the overall performance since it could not squeeze out more redundant computations from already condensed edge tiles.

5.4.2 Compared with other baselines

Compared with PyG Figure 5.3b shows TC-GNN can outperform PyG with an average of 1.76× speedup on GCN and an average of 2.82× speedup on AGNN. For GCN, TC-GNN achieves significant speedup on datasets with high-dimensional node embedding, such as *Yeast (YT)*, through effective TCU acceleration through a TCU-aware sparse graph translation while reducing the synchronization overhead by employing our highly parallelized TCU-tailored algorithm design. PyG, however, achieves inferior performance because its underlying GPU kernel can only leverage CUDA cores, thus, intrinsically bounded by CUDA core performance.

Compared with cuSPARSE bSpMM Figure 5.3c shows that TC-GNN outperforms bSpMM with on average 1.76× speedup on neighbor aggregation and improves effective computation by 75.8% on average. Our SGT technique can maximize the non-zero density of each non-zero tile and significantly reduce the number of non-zero tiles to be processed. However, bSpMM in cuSPARSE has to comply with the strict input sparse pattern (indicated in official API documentation [136]). For example, all rows in the arrays must have the same number of non-zero blocks. Thus, more redundant computations (on padding non-structural non-zero blocks) in bSpMM lead to inferior performance. We also notice that for SC datasets with a high average node degree and clustered node distribution, bSpMM would benefit more due to its usage of a larger block size of 32×32 (fewer TCU invocations) compared to 16×8 in TC-GNN (more TCU invocations).

Compared with tSparse and Triton From Table 5.5, TC-GNN can outperform tSparse with on average 3.60× speedup on SpMM. The major reason behind this is that TC-GNN can well reduce the graph structural-level irregularity through our novel SGT strategy to benefit the dense TCU-based computation. In contrast, tSparse only considers partitioning the input sparse matrix into dense/sparse tiles based on their

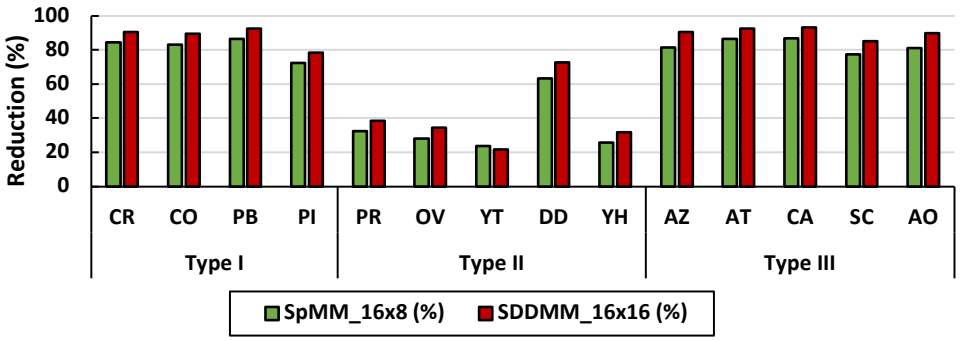


Figure 5.4: SGT Effectiveness on SpMM and SDDMM.

non-zero elements but ignores the potential of compressing non-zero elements into fewer tiles to reduce the workload. TC-GNN also outperforms Triton with on average 5.42× speedup on SpMM. Triton's block-sparse GEMM for TCU acceleration is designed for dense neural networks (focusing on feature maps' sparsity), which is quite different from GNNs (focusing on the graph adjacency matrix's sparsity) with significantly larger sparse matrix size and more irregular pattern.

5.4.3 Additional Studies

SGT Effectiveness & Overhead We conduct a quantitive analysis of SGT in terms of the total number of TCU blocks between graphs w/o SGT and the graphs w/ SGT applied. Note that in the SpMM-based aggregation, the size of TCU blocks is 16×8 since it serves as one of the operands in TCU GEMM. While in SDDMM-based edge feature computation, the size of TCU blocks is 16×16 since it serves as the resulting matrix of TCU GEMM. Figure 5.4 shows that across all types of datasets, our SGT technique can significantly reduce the number of traversed TCU blocks (on average 67.47%). The major reason is that SGT can largely improve the density of non-zero elements within each TCU block. In contrast, the graphs w/o SGT would demonstrate a large number of highly sparse TCU blocks. What is also worth noticing is that on Type II graphs, such a reduction benefit is lower. The reason is that Type II graphs consist of a set

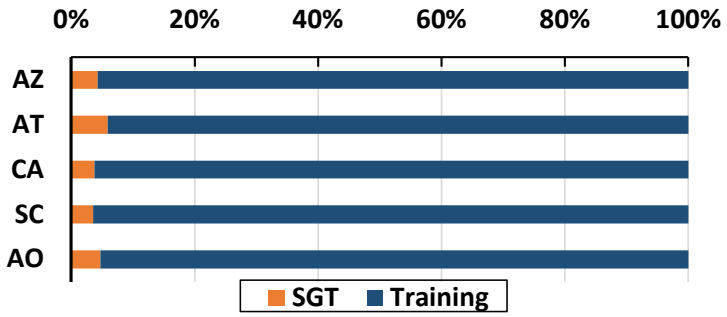


Figure 5.5: The overhead analysis of SGT.

Table 5.6: Sparsity Analysis. Numbers for bSpMM/TC-GNN are in GFLOPs. "DB/W": dense blocks per row window.

DB/W	Sparsity (%)	bSpMM	TC-GNN
1	99.61	773.86	12,686.02
2	99.22	1,597.83	11,010.75
4	98.44	3,348.75	18,164.08
8	96.88	6,528.10	25,883.10
16	93.75	12,955.40	23,865.99
32	87.50	26,061.70	16,629.28

of small subgraphs that only maintain the intra-subgraph connections, which already maintain dense columns. We evaluate the overhead of SGT (Figure 5.5), we find that its overhead is consistently low (on average 4.43%) compared with the overall training time (200 epochs as DGL [15]).

Sparsity Analysis We compare with bSpMM on synthetic matrix data with different sparsity (zero-element ratio). Note that we change the sparsity by varying the number of dense non-zero blocks (16×16) within each row window, the input adjacent matrix size is fixed to 4096×4096 while the dense embedding matrix dimension is fixed to 16. Table 5.6 shows that when sparsity increases from 93.75% to 99.61%, TC-GNN design demonstrates more throughput performance strength (averaged 6.9×) and this is also the common sparsity range (more than 95%) for most input graphs of GNNs. When the sparsity drops to around 87.50% the sparse would demonstrate more advantage due to more dense blocks for computation.

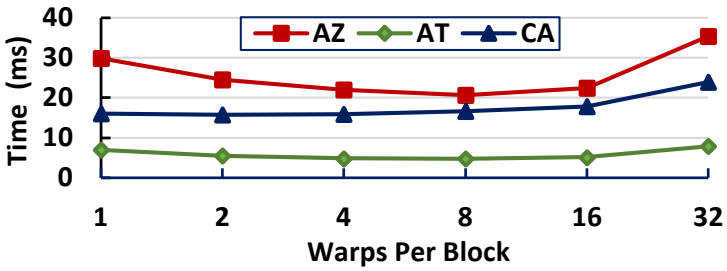


Figure 5.6: Performance Impact of Warps per Block.

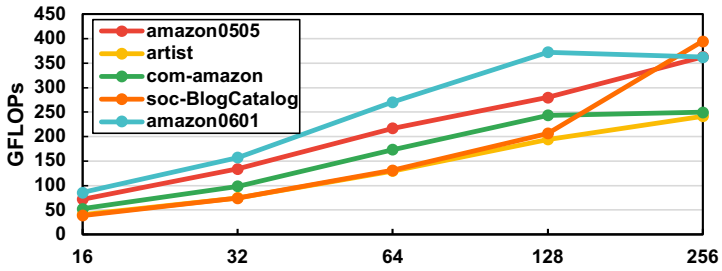


Figure 5.7: Analysis of TC-GNN kernel throughput when increasing the node embedding dimension from 16 to 256.

Warps per Block: Figure 5.6 shows that with the increase of the number of warps, the overall performance for training per epoch would first decrease due to the better parallelism for loading the graph data. However, the number of warps per block would decrease the overall performance under certain circumstances (e.g., 32). All three settings suffer from evident performance degradation. Because the global memory access contention will become severe, leading to lower execution performance. Different datasets would have different "optimal" choices of the warp-per-block parameter. For example, on the CA dataset, 2 warps per block can deliver the best performance, while AZ requires 8 warps per block. Based on our profiling and empirical study, the selection of this parameter should consider the average *#edges* per row window (*avg.edges*), which can be easily get during the preprocessing. Our **`preprocessor`** will generate $warpPerBlock = \lfloor \frac{avg.edge}{32} \rfloor$ to approach the "optimal" performance. For instance, the average edges per row window are 88 for CA, it reaches the best performance at 2 warps per block.

Throughput Analysis: For sparse matrix computations in GNNs, we measure the

throughput performance of SpMM in TC-GNN when the dimension of node embedding increases for a roofline analysis. Sparse matrix computation is largely limited by its memory access performance, which is quite different from the dense GEMM computation which is largely bounded by the computing performance. Figure 5.7 shows that the throughput of TC-GNN can scale proportionally with the growing number of node embedding dimensions. This also indicates that TC-GNN can effectively handle graphs with high-dimensional node embeddings and well utilize GPU resources.

5.5 Related Work and Discussion

Other GPUs TC-GNN can easily generalize to other GPUs (e.g., A6000, H100, and RTX4090) with TCUs via recompilation (`python setup.py install`). TC-GNN also supports different TCU configurations (e.g., precision) by modifying (BLK_H, BLK_W in `TCGNN_conv/config.h`) and four parameters (`M, N, K, dataType`) in `wmma::fragment`, then recompile. For future GPUs with more TCUs, our TC-GNN can also be adapted to accommodate such changes and maintain its performance advantage. There are two future GPU designs that we anticipate. The first direction is to place more TCUs per SM while keeping the total number of SMs unchanged. There will be more active warps per thread block (This is mainly because TCUs are operated by warps) and each warp will process fewer neighbors. The cost of decomposition and mapping can be offset by parallelism among more warps. The second direction is to place more SMs on GPUs while keeping TCUs per GPU unchanged. In this scenario, there will be more thread blocks and each thread block will process neighbors from fewer nodes. The cost can be offset by parallelism among more thread blocks.

Other GNN Frameworks Besides DGL and PyG, other single-GPU GNN frameworks like GNNAdvisor [3], GE-SpMM [130], and fuseGNN [137], tailor their own GNN

layers manually with low-level GPU kernel optimizations. Unfortunately, these designs limit their kernel optimizations to CUDA cores, thus, missing the golden opportunities to exploit the full potential of widely deployed AI-tailored GPUs with TCUs.

Graph Partitioning/Reordering ROC [138] introduces a learning-based graph partitioning to reduce the data movement between CPU and GPU when processing large graphs. Rabbit Order [139] and Reverse Cuthill Mckee Algorithm [140] are focusing on *row reordering/clustering* to improve node/row-wise computation locality. Our sparse-graph translation (SGT) technique is orthogonal and complementary to these graph partitioning and reordering techniques since our SGT focuses on *column (neighbor) re-indexing* to improve neighbor-wise locality for TCU computation.

Distributed GNN Computation There are two major ways of scaling-up GNN computing: 1) *Distributed sampled graphs* [15, 100, 16, 114] (where graph nodes and their embeddings are on the same GPU): TC-GNN can be incorporated directly since all sampled graphs along with their node embeddings are presented at the same GPU. 2) *Distributed full-graph* [141, 14, 138, 142] (where graph nodes and their embeddings may be on different GPUs): TC-GNN needs to be modified slightly by incorporating inter-GPU communication techniques (e.g., Unified Virtual Memory [106] and NVSHMEM [94]) to support the remote neighbor embedding access. We leave such exploration for our future work.

Chapter 6

DSXplore: Optimizing Convolutional Neural Networks via Sliding-Channel Convolutions

As the key advancement of the convolutional neural networks (CNNs), depthwise separable convolutions (DSCs) are becoming one of the most popular techniques to reduce the computations and parameters size of CNNs meanwhile maintaining the model accuracy. It also brings profound impact to improve the applicability of the compute- and memory-intensive CNNs to a broad range of applications, such as mobile devices, which are generally short of computation power and memory. However, previous research in DSCs are largely focusing on compositing the limited existing DSC designs, thus, missing the opportunities to explore more potential designs that can achieve better accuracy and higher computation/parameter reduction. Besides, the off-the-shelf convolution implementations offer limited computing schemes, therefore, lacking support for DSCs with different convolution patterns.

To this end, we introduce, DSXplore [1], the first optimized design for exploring DSCs on CNNs. Specifically, at the algorithm level, DSXplore incorporates a novel factorized kernel – sliding-channel convolution (SCC), featuring input-channel overlapping to balance the accuracy performance and the reduction of computation and memory cost. SCC also offers enormous space for design exploration by introducing adjustable kernel parameters. Further, at the implementation level, we carry out an optimized GPU-implementation tailored for SCC by leveraging several key techniques, such as the input-centric backward design and the channel-cyclic optimization. Intensive experiments on different datasets across mainstream CNNs show the advantages of DSXplore in balancing accuracy and computation/parameter reduction over the standard convolution and the existing DSCs.

6.1 Introduction

With the increasing popularity of AI-driven edge computing and Internet of Things (IoTs), convolutional neural networks (CNNs) have entered a new era with the plethora of tiny devices, which have limited resources, such as the power, and memory budgets. This makes the CNNs that are small in parameter size with low computation costs (FLOPs) highly in demand. Among numerous research and industry efforts, depthwise separable convolutions (DSCs) attract a lot of attention, largely because of their stunning success in reducing FLOPs and parameters.

Existing works around DSCs have been widely studied mainly from an algorithmic perspective. MobileNet [143] has been proposed by replacing the standard convolution with the depthwise separable convolution (depthwise (DW) + pointwise (PW) convolu-

[1] © 2021 IEEE. Reprinted, with permission, from Yuke Wang, *DSXplore: Optimizing Convolutional Neural Networks via Sliding-Channel Convolutions*, International Parallel and Distributed Processing Symposium, 05/2021.

tion) to reduce the model parameters and computation cost significantly. Inspired by the success of MobileNet, Xception [144] architecture has been built to largely simplify more complicated CNNs, such as Inception [145] (with a large number of layers and residual connections) by leveraging DSCs. The major assumption of these works is that the cross-channel correlations and the spatial correlations can be effectively decoupled and there is no need to map them at the same time. Therefore, the high-cost standard convolution can be effectively divided into the lightweight DW convolution to capture the spatial information and PW convolution to capture the channel-wise information.

However, these existing efforts on DSCs are still initial, since they overlook some key points that could be potentially leveraged for further parameter and computation reduction. And we believe there are several reasons behind.

First, there are limited DSC designs to balance accuracy performance and the size of computation/parameters. Existing work on DSCs is mostly derived from the DW+PW design for standard convolution replacement. The more effective DSC schemes that can potentially deliver better accuracy and model size trade-offs still remain uncovered. For example, we can further reduce the computation cost and parameter size by combining the group convolution (GC) (dividing the input and output channel into the same number of groups and only applying standard convolution within each group) with PW.

Second, there is a lack of efficient implementation support for new factorized kernels. Existing works on DSCs heavily rely on the deep-learning infrastructure with standard/group convolutions for their factorized kernel implementation. For example, the DW convolution can be expressed as the extreme case of the GC with the number of groups equal the input channels, while the PW convolution can be expressed as another special case of standard convolution with the 1×1 kernel spatial dimension. Therefore, the better factorized kernel that may bring better accuracy and lower com-

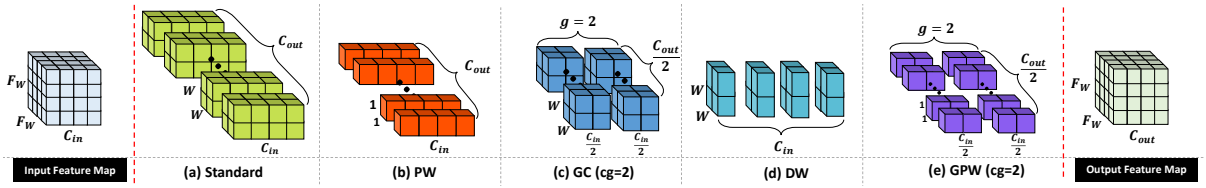


Figure 6.1: Comparison of input/output feature map before/after convolution and the filter dimension of existing CNN convolutions.

putation and memory costs but not in the above categories cannot leverage the existing convolutional primitives for an effective implementation.

To this end, we propose, DSXplore, the first optimized design for exploring the long-existing "buried" DSC potentials. The highlight of DSXplore is our novel factorized convolution kernel – sliding-channel convolution (SCC), which can effectively reduce the computation and parameter size meanwhile maintaining model accuracy to a great extent. In contrast to the previous fixed DW+PW in most existing DSC designs, DSXplore provides an enormous space for exploring new DSC designs by introducing a set of adjustable parameters to SCC – the number of channel groups (cg) and input-channel overlapping (co). Furthermore, we present an optimized implementation of SCC on GPUs by capturing the specialty (*e.g.*, cyclic-channel pattern) of SCC and tailoring parallel computation (*e.g.*, improving thread-level parallelism and reducing atomic operations).

Overall, we make the following contributions

- We propose the first optimized design to uncover the potentials of DSC. We harness our novel sliding-channel convolution (SCC) to balance the accuracy performance and the reduction of computation and memory cost. Moreover, SCC offers an enormous design exploration space with parameterized design strategy.

- We carry out an optimized GPU-implementation tailored for SCC design by incorporating several key techniques, including the output-centric forward and input-centric backward optimization, and the optimization based on the convolutional

specialty (cyclic channel) of its filters.

- We integrated our SCC design with the original Pytorch framework as the drop-in replacement of the existing DSCs to facilitate the training and inference of DSXplore[2] in an end-to-end fashion.

- Intensive experiments show DSXplore achieves better accuracy and lower computation/memory cost compared with the existing DSC. Our proposed optimized GPU implementation can overcome the implementation challenges of SCC by providing notable speedup compared with the implementations by compositing the highly-optimized Pytorch operators.

6.2 Background and Related Work

6.2.1 Standard Convolution

The standard convolution is the most widely used deep-learning operation in many CNNs [146, 147, 148], which targets on images (*a.k.a.*, feature map). In general, we annotate the dimension of 3D feature map as $F_w \times F_w \times C_{in}$, where F_W is the 2D spatial dimension of the feature map while C_{in} is the number of input channels.

The standard convolution relies on a set of (C_{out}) standard convolutional filters, each of which has the size of $W \times W \times C_{in}$ parameters to generate output feature maps, as shown in Figure 7.1(a), where the W is the spatial dimension of the filters (In general, filters usually have a square shape of $W \times W$), C_{in} is the number of channels for the input feature map, and C_{out} is the number filters (equals the channels for the output feature map, since each filter only generates the feature map in one output channel). After applying the standard convolution on the input with the shape of $F_w \times F_w \times C_{in}$,

[2]DSXplore is open-sourced at `https://github.com/YukeWang96/DSXplore.git`

we will get the output feature map O, which has the shape of $F_w \times F_w \times C_{out}$. Note that the mainstream CNNs [146, 147, 143] generally maintain the same feature map spatial dimension at different convolutional layers while only changing the number of the channels across different layers.

Formally, for standard convolution, we have

$$O_{m,n,c} = \sum_{i,j,a} K_{i,j,a,c} * F_{m+i-1,n+j-1,a} \tag{6.1}$$

where $O_{m,n,c}$ is one pixel point in the output feature map; m and n are the spatial indexes in the output feature map ($0 \leq m < F_w$ and $0 \leq n < F_w$); a is the channel index in the input feature map ($0 \leq a < C_{in}$); c is the channel index in the output feature map ($0 \leq c < C_{out}$); i, j, and a are the index used to accumulated the elementwise multiplication values between input feature map and one filter, which is essentially a cube with the shape of $K \times K \times C_{in}$.

The standard convolution not only captures the spatial information by iteratively "gathering" a $W \times W$ square in a 2D sliding-window fashion in each channel but also effectively fuses the information across different channels, as indicated in the Figure 7.1(a), where each kernel filter will "fuse" the information from all C_{in} channels of input feature maps.

6.2.2 Factorized Convolution

While the standard convolution gains success in CNNs across different applications settings (*e.g.*, computer vision), its high computation and memory complexity significantly make it challenging to get widely adopted on devices with limited capability of computation and memory resources.

To break this hurdle, a set of factorized kernels and their combinations are introduced

as a drop-in replacement for the standard convolution to reduce the computation and memory cost meanwhile maintaining the model prediction power. Existing factorized kernels can be divided into four major categories: 1) Pointwise Convolution (PW) [145], which is a standard convolution with 1×1 spatial size, as shown in Figure 7.1(b); 2) Group Convolution (GC) [149] that divides input channels into several groups and performs standard convolution within each group, as exemplified in Figure 7.1(c) with two groups; 3) Depthwise Convolution (DW) [150] which calculates spatial convolution per channel or can be regarded as an extreme case of GC when the group number equals the number of the input channels, as shown in Figure 7.1(d); 4) Group Pointwise Convolution (GPW) [151] that further splits PW into groups, as exemplified in Figure 7.1(e) with two groups.

The most successful example adopted in many CNNs (*e.g.*, Xception [144] and MobileNet [143]) is the depthwise separable convolution (**DSC**). It breaks the original standard convolution into two parts: **depthwise** (DW) convolution and **pointwise** (PW) convolution. The first step (DW) applies C_{in} different $W \times W \times 1$ filters to each of the C_{in} input channels independently, which can be formalized as Equation 7.2

$$\hat{O}_{m,n,a} = \sum_{i,j} K_{i,j,a} * F_{m+i-1,n+j-1,a} \tag{6.2}$$

The second step (PW) applies a filter with 1×1 spatial dimension. As shown in Equation 7.3.

$$O_{m,n,c} = \sum_{a} K_{a,c} * F_{m-1,n-1,a} \tag{6.3}$$

Compared with the standard convolution, DSC brings two folds of benefits. First, it largely reduces the size of weight parameters. In the standard convolution, we have $W \times W \times C_{in} \times C_{out}$ parameters, while in DSC, we only have $W \times W \times C_{in} + C_{in} \times C_{out}$

parameters, which is only $\frac{1}{C_{out}} + \frac{1}{W^2}$ of the parameters in the standard convolution. Second, it can largely reduce the total number of computations (FLOPs) compared with the standard convolution kernel. Specifically, the standard convolution requires $F_w \times F_w \times C_{out} \times W \times W \times C_{in}$ operations, whereas DSC only requires $C_{in} \times F_w \times F_w \times W \times W + C_{out} \times F_w \times F_w \times C_{in}$, which is $\frac{1}{C_{out}} + \frac{1}{W^2}$ of the number of operations in the standard convolution. Further study, such as Shift Convolution [152], adapts the depthwise convolution by using a shift matrix for a more specialized spatial-wise information fusion at individual input channels.

However, the underlying reason why these designs can be successful is not clear. Our work, in contrast, introduces a brand-new DSC convolution kernel – sliding-channel convolution (SCC), focusing on the channel-wise information fusion. It offers explainable parameters (the number of channel groups and the input-channel overlapping ratio) to effectively capture cross-channel information with more flexibility.

6.2.3 Sparse Convolution

As an another way of reducing computation and compressing model size, sparse convolution [153, 154, 155, 156, 157] has been proposed and widely studied. Depending on the granularity of the pruning, existing pruning strategies can also be categorized into two major types, non-structured and structured pruning. The non-structured pruning aims to maximize the ratio of reducing the parameters and saving FLOPs. However, it fails to consider the implementation complexity on modern hardware due to its computation irregularity after pruning. The structured pruning overcomes the weakness of non-structured pruning through applying coarse-grained pruning strategy to largely maintain the computation regularity. However, it may compromise model accuracy to some extent due to such an "imprecise" nature of the structural pruning.

Furthermore, the modern deep-learning frameworks, such as Pytorch and Tensorflow [158], still lack of efficient programming library support for these pruning operations (*a.k.a.*, sparse kernels) on CPUs and GPUs. And most of these sparse kernels are only evaluated in their accuracy performance by masking the feature maps or weight parameter with zero values. At the same time, some of the pruning method is highly input-sensitive, which may lead to non-deterministic model accuracy when encountering the "unseen" real-world datasets. In contrast, our work focusing on factorized kernel design is essentially orthogonal to these pruning approaches but share the commonality of saving computation and parameters of CNNs. And the difference is that our work is to redesign the overall kernel filters and their operation patterns (*e.g.*, the input-to-output channel mapping of kernel filters), while these pruning works target at modifying individual filters or feature maps, such as masking out specific tensors based on their values. We also believe that factorize kernel + pruning is a potential research direction, but it is not our current focus in this paper.

6.3 Sliding-Channel Convolution

Motivated by previous research on DSCs, we believe some "hidden" dimensions are yet to be explored in this direction. And there are three major questions to be answered:

1) Is it possible to further save the parameters and the computation costs by crafting a new DSC convolution scheme?

2) How could we maintain the model accuracy performance by applying such a design?

3) How could we implement such a new design to facilitate end-to-end training on modern hardware, like GPUs?

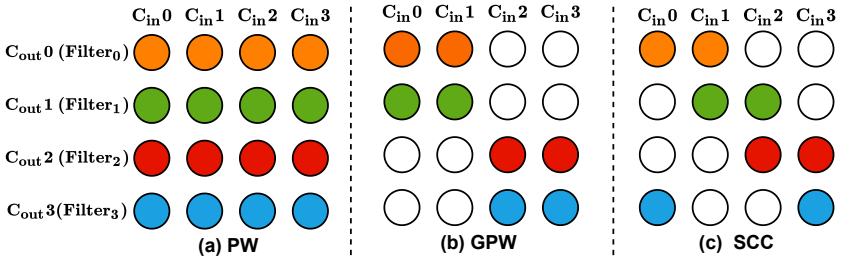


Figure 6.2: Comparison of our proposed SCC to PW and GPW on input-to-output channel mapping of filters. Note that circles in the same color represent input channels covered by one filter.

6.3.1 Sliding-Channel Convolution

To this end, we propose a brand new convolution scheme, named sliding-channel convolution (SCC), in place of the PW convolution in the DW+PW design to capture cross-channel information more effectively. Specifically, it combines the GC to reduce the parameter and computations in the original PW convolution. Most importantly, it overlaps the input-channels of each filter to "recover" the information that is ignored by applying GC. Such specialty makes our SCC different from previous designs (PW and GPW) in different perspectives.

Table 6.1: Comparison among SCC, PW, and GPW.

Convolution	FLOPs	Params.	Acc.
PW†	High	High	High
GPW*	Low	Low	Low
SCC	**Low**	**Low**	**High**

†: PW can be seen as SCC with 1 group with 100% channel overlapping of adjacent filters.

*: GPW can be seen as SCC with m groups with 0% channel overlapping of adjacent filters. m is a parameter that can be determined by users.

Compared with PW: As the second step in DW+PW design, the major role of PW is to cross-channel information by a standard convolution with 1×1 filter. While SCC also shares a similar goal as PW to unify channel-wise information, each kernel filter of SCC only needs to look through a part of the input channels, whereas each kernel filter of PW has to look through all input channels. As illustrated in Figure 6.2(a), the $C_{out}0$

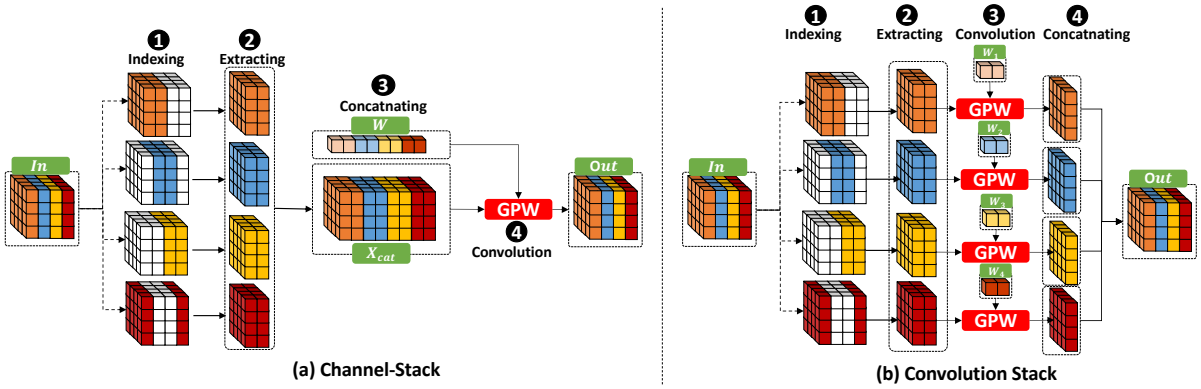


Figure 6.3: Two proposed Implementations for SCC via Pytorch-operator composition: *channel-stack* and *convolution-stack*.

in SCC only need to "gather" information from the input channel $C_{in}0$ and $C_{in}1$, whereas $C_{out}0$ in PW requires information from channel $C_{in}0$, $C_{in}1$, $C_{in}2$ and $C_{in}3$. Note that we use the circle to represent the feature map ($F_w \times F_w$) on each channel. The benefits of our SCC design compared with PW can be highlighted by comparing line 3 and line 1 in Table 6.1, where SCC reduces the FLOPs and parameters meanwhile maintaining accuracy performance.

Compared with GPW: As the variant of the PW convolution, GPW has been studied to amortize the computation and parameter costs of PW. While sharing the commonality of reducing the model complexity as GPW, each filter of our SCC would cover different "windows" of input channels that are partially overlapped with their adjacent filters. The input-channel overlapping in SCC serves as an essential "bridge" to communicate the different input channel information that is originally segregated by channel grouping in GPW. Filters in the conventional GPW, on the other hand, would either fully overlap with their adjacent filters or no overlap with their adjacent filters in terms of input channels they cover. This may lead to 1) redundant information from the same input-channel window, and 2) missing information spanning across different input-channel windows. As illustrated in Figure 6.2(b), the first half of the output channels only uses the first half of the input channels from the input feature map, while the second

half of the output channels only uses the second half of the input channels. In contrast, in our SCC design (Figure 6.2(c), the adjacent two filters would never occupy the same part of input-channel window. For example, filter 2 overlaps with the filter 1, thus, extracting information from both $C_{in}1$ and $C_{in}2$, which cannot be captured in GPW. Besides, we introduce the channel-circulation scheme that further strengthens the capability to extract information, *i.e.*, the last input channel is logically connected with the first input channel to form a cycled input channels. Therefore, as shown in Figure 6.2(c), the input-channel window of filter 3 will include the $C_{in}3$ and $C_{in}0$. The benefits of our SCC design compared with GPW can be demonstrated by comparing line 3 and line 2 in Table 6.1, where SCC wins in terms of accuracy by better utilizing cross-channel information.

Moreover, another highlight of our SCC design is its flexibility, and there are two parameters are introduced: channel group number *cg* and input-channel overlap ratio *co*. For instance, we use SCC-cgX-coY% to denote each filter in the convolution kernel takes $\frac{1}{X}$ number of input channels, while adjacent filters in SCC have y% overlap in their consumed input channels. As exemplified in Figure 6.2(c) with the setting of SCC-cg2-co50%, the input and output channels of the feature map are divided in to two groups, each adjacent filter is overlapped with 50% input channels.

6.3.2 Implementation Challenges

While the SCC allows users to explore more algorithmic benefits from DSCs through combining group convolution and overlapping channels, the implementation of this new kernel is challenging in several aspects. First, existing off-the-shelf high-level CNN building blocks (*e.g.*, cuDNN [159], Conv2D in Pytorch) crafted for standard convolution could not be easily adapted for the implementation of SCC due to the quite different ways of conducting convolution operations. Because each filter of SCC relies on different input

channels to generate feature map on one output channel, as indicated in Figure 6.2. Thus, extra computations are required to pinpoint the corresponding input-channel window at the input feature map before the convolution.

Second, the implementation of SCC could not well benefit by leveraging the highly-optimized low-level NN primitives, such as General Matrix-Matrix operations (GEMM). The major reason is that SCC requires more fine-grained GEMM operations on the matrix that are essentially skewed in its shape. Because each kernel filter does not cover the identical "window" of input channel as its adjacent filter, thus, lacking the reuse of input-channel window. For example, considering a setting with feature map (56×56) in each input channel, the number of input channel is 64 ($C_{in} = 64$) and the number of output channel is 128 ($C_{out} = 128$). When $cg = 2$ (*i.e.*, both input and output channel are divided in to 2 groups), SCC requires 128 times fine-grained GEMM operations between the matrix with shape ((56×56) $\times$ 32) and matrix with shape (32×1). Even though stacking these small matrices in to a larger matrix is an alternative solution, the computation and memory costs are also non-trivial. In contrast, existing convolution kernels, such as GPW, can be well benefited from cuBLAS [32] for implementation. Assuming the same setting ($cg = 2$) as the above example, GC only requires two GEMM operations between a matrix with shape ($(56\times56)\times32$) and a matrix with shape (32×64), which is clearly efficient compared with that in SCC.

To tackle these challenges, we first propose two implementations by compositing existing Pytorch operators. Further, we orchestrate our highly-efficient implementation with a set of kernel-level designs and optimizations tailored for SCC computation. We discuss those details in Section 6.4.

6.4 Implementation

6.4.1 Pytorch Operator Composition

Channel-Stack Implementation As the most straight-forward solution, we implement SCC by compositing the standard Pytorch operators, such as tensor slicing, concatenation, and standard group convolution. Specifically, there are four major steps, as shown in Figure 6.3(a). The first step (❶) is to identify the input channels of kernel filters (*i.e.*, the calculation of the index range of each kernel filter, including its starting and ending location). The second step (❷) is to extract the input feature maps based on the calculated input channel windows from the previous step. Then the third step (❸) concatenates them into a large feature map long their channel dimension. The fourth step (❹) is to apply the standard group convolution (such as `conv2D` in Pytorch) with the number of groups specified as the number of output channels (kernel filters).

However, we could easily find the drawbacks of such an implementation in the follow aspects. First, massively dividing and concatenating tensors (feature maps) in Pytorch is inefficient. It generally requires random indexing and data duplication on tensors, leading to a large amount of data movement that will hurt the performance. Second, Pytorch framework lacks support for concurrently executing the above operations, thus, missing the opportunity for parallelization. Third, the overlapped input channels of each kernel filter incur excessive data redundancy of repeatedly storing the same feature maps, leading to a prohibitively large size of the concatenated tensor that hurdles its applicability on modern GPUs with limited size of memory.

Convolution-Stack Implementation The second implementation circumvents the "huge" concatenated tensor in the above channel-stack implementation by applying convolution operation before concatenating. One major key insight is that the computation

on the large concatenated tensor can be decomposed into the more effective computation on a set of small tensors. As illustrated in Figure 6.3(b), instead of simply combining all the extracted features maps, we can pre-build a set of lightweight convolutions, such as GPW1 - GPW4, each of which will generate the feature map for only one kernel filter. Finally, we concatenate these output feature map together. While this solution can largely overcome the third problem of the above channel-stack implementation, it is still hindered by the excessive inefficient Pytorch operations and lack of computation parallelization.

6.4.2 DSXplore Implementation

To handle the aforementioned challenges, our DSXplore introduces a set of new designs at forward and backward pass tailored for SCC computation. Moreover, DSXplore identifies the channel-cyclic pattern in SCC that can be effectively applied towards different implementations.

Output-Centric Forward Pass We propose an output-centric forward pass to efficiently handle the SCC forward propagation. Adjacent kernel filters are overlapped in terms of their corresponding input channels. Note that the last input channel is logically adjacent to the first input channel to form a channel circle. In the forward pass, we assign $N \times C_{out} \times F_w \times F_w$ GPU threads in total to generate each pixel point in the output feature map, and the workload of each thread is just a simple vector-vector multiplication between the weight and the pixels across different input channels but at the same position of the 2D kernel dimension (*i.e.*, the same (x, y) in $F_w \times F_w$ space). For example, output feature map on channel C_1 with the shape of $F_w \times F_w$ will be generated by the elementwise multiplication between input feature maps ($(F_w \times F_w) \times 2$) and weight parameters (2×1) by leveraging $F_w \times F_w$ threads.

Our proposed forward pass scheme has two major benefits. First, no data duplication for any part of tensors (feature maps) during the whole computation, since each thread only need to fetch its corresponding pixel point across different input channels from the original input tensor directly. Second, better data locality in the input feature map and weight parameters, since adjacent threads within the same GPU block are scheduled to operate on the same output feature map. Third, no inter-thread contention, since the computation of each pixel point is independent from each other and is handled by one thread.

Input-Centric Backward Pass For training our SCC kernel, an efficient backward phase is required for gradient backward propagation to update the model parameters. Compared with the forward pass, backward propagation is generally several times higher in computations due to the gradient backpropagation for both the trainable parameters (e.g., weight, and bias) and input feature maps. The backpropagation of the standard-/group convolution can be easily formalized as the high-performance GEMM operation due to the fact that their kernel filters are either fully overlapped (for filters from the same group) or non-overlapped (for filters from different groups) in terms of their input channels. However, in our SCC kernel, filters are partially overlapped, which challenges the implementation of gradient backpropagation due to lacking support of any existing optimized operation.

While there is a straightforward option to simply reverse the forward computation flow of data propagation to the backward flow for gradient propagation. As illustrated in Figure 6.4(a), we assign threads based on the output gradient map as we previous do in the forward pass. However, the overlapping of input channel for different filters also leads to the excessive amount of thread-level synchronization during the backward pass. The major reason is that the overlapped input-channel feature maps would simultaneously

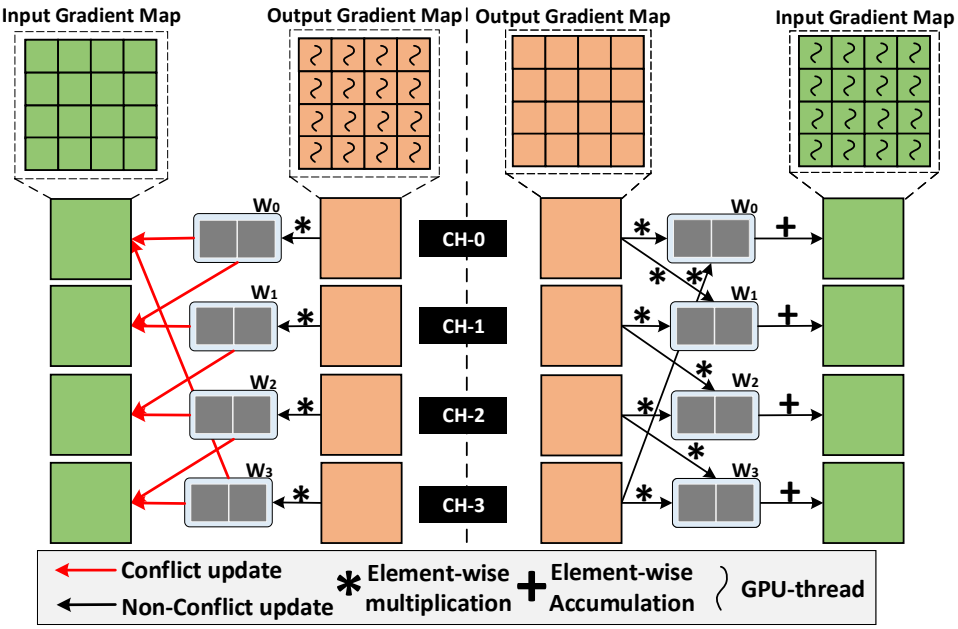


Figure 6.4: SCC backward computation flow: *output-centric* and *input-centric* design. Note that square boxes stand for points on feature maps, while each curved line inside a box stands for a GPU thread for generating value at that point. "CH" is short for "channel".

receive the gradients being "pushed" from different kernel filters that are operated by different groups of threads. Therefore, the computation correctness has to be guaranteed by employing lots of atomic primitives when accumulating gradients for input feature maps. For example, the propagation of output gradient map on CH-2 and CH-3 would incur lots of conflicts during the updating of input gradient map on CH-3.

To overcome these issues, we introduce an input-centric backward scheme that assigns each thread to generate the gradient of each pixel point on the input feature map. As shown in Figure 6.4(b), we assign threads based on the input gradient map instead of the output gradient map, while the direction of the gradient flow remains unchanged. For example, the input gradient map on CH-3 will "pull" gradients from the output gradient maps on CH-2 and CH-3. By harnessing such an input-centric backward scheme, we eliminate the need to use the excessive amount of thread-level synchronization (atomic

operations) for the input gradient map update, meanwhile maintaining the correctness of the backward computation.

Note that for both forward and backward computation, we decide not to move forward with GEMM-based (*e.g.*, cuBLAS [32]) solution according to our experimental study. First, it would lead to launching an excessive amount of small GEMM kernels, each of which can not fully utilize the GPU resources; Second, cuBLAS [32] no longer supports any function call from the CUDA kernel (decorated with `__global__`) since CUDA 10.0. Thus, only the host code running on CPU is allowed to invoke `cublasSgemm`, which largely limits the computation parallelism (even with OpenMP).

Channel-Cyclic Optimization While SCC brings challenges of implementing forward and backward pass, it also comes with a special pattern – *channel cyclic*, which can be leveraged for further optimization. The adjacent kernel filters would "gather" information from the adjacent "windows" of input channels that are overlapped with each other. After several kernel filters, it will encounter the same input-channel window that has been traversed before. For example, as shown in Figure 6.5(a) for the $cg = 2$ and $co = 50\%$ case, every four kernel filters will share the same set of input-channel window. Similarly, for the $cg = 2$ and $co = 33\%$ case, as shown in Figure 6.5(b), every three kernel filters will share the same set of input-channel window. To identify such cyclic channel pattern and compute the input-channel indexes of filters within one cycle, we follow the Algorithm 7, under the given C_{in}, cg, and co.

Cyclic-channel patterns in SCC can be leveraged to optimize our proposed implementations in the above sections. First, it can be applied to the implementations of Pytorch operator composition to reduce the repetitive tensor manipulations and save memory. Specifically, in the channel-stack implementation, we only need to identify and concatenate the first cycle of input feature maps. Then for the settings that are composed of multiple cycles, we just need to duplicate the concatenated feature map of the first cycle

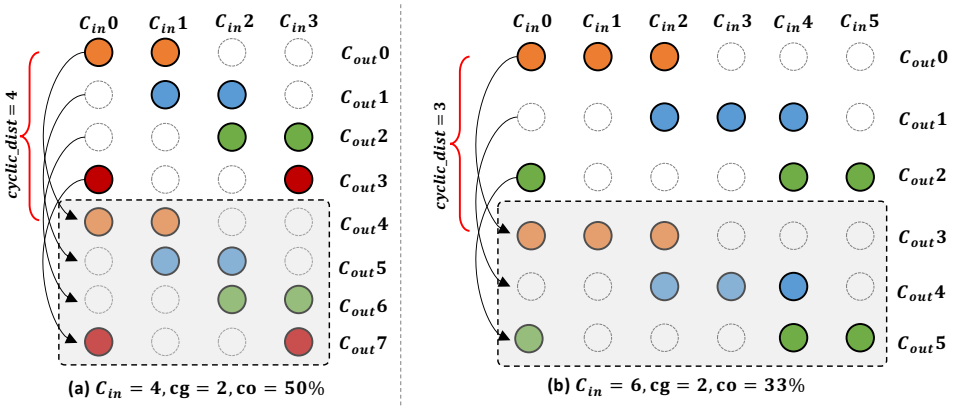


Figure 6.5: Illustration of the channel-cyclic pattern in SCC.

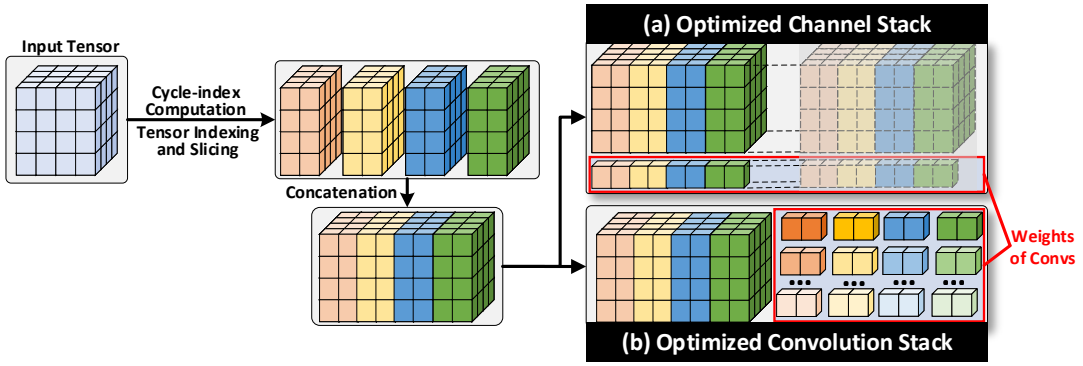


Figure 6.6: Case study of cyclic-channel optimization on Pytorch-based implementations for $C_{in} = 4$, $cg = 2$, $co = 50\%$.

several times and merge them all together along the channel dimension, as exemplified in Figure 6.6(a). This can avoid the high-cost tensor indexing and slicing operations on the original feature map that have been carried out before. In the convolution-stack implementation, we can largely save the memory by keeping the concatenated tensor of the input feature maps in the first cycle, since all the remaining cycles would maintain the same input information, as illustrated in Figure 6.6(b). And we just need to pile enough GPW convolutions (the number of which equals the output channels) that will be trained to extract different information from the same input.

Second, it can be leveraged to optimize our DSXplore by reusing the indexes of the

Algorithm 7 Compute channel indexes of one cycle.

```
channel_map = {}
group_width = input_channel//num_groups
start, end = 0, group_width
start_v, end_v = start, end
cyclic_dist = 0
for oid = 0; oid < output_channel; oid++ do
    item = (start, end)
    if item not in channel_map then
        channel_map.add(item)
        cyclic_dist++
    else
        break
    end if
    start_v = end_v − int(overlap ∗ group_width)
    end_v = start_v + group_width
    start = start_v % input_channel
    end = end_v % input_channel
end for
```

Algorithm 8 DSXplore Channel-cyclic Optimization

```
1: thread_id = blockIdx.x × blockDim.x + threadIdx.x
2: opt_channel_id = get_output_channel(thread_id)
3: idx = output_channel_id  mod cyclic_dist
4: start, end = channel_map[idx]
```

input channel during our SCC-tailored forward pass. Therefore, the recurrent index range only need to be calculated once and will be stored into a map/dictionary like object for the following filters, as illustrated in Algorithm 8. *Line* 1 gets the corresponding kernel filter index (*opt_channel_id*) based on the GPU thread ID. Then based on the *opt_channel_id* and the *cyclic_dist* from the Algorithm 7, we can get the *idx* of feature map to fetch the corresponding starting and ending position of input channels for the filter.

6.5 Evaluation

In this section, we conduct a set of intensive experiments on DSXplore in terms of its accuracy and the runtime performance.

6.5.1 Experimental Setup

Datasets We use CIFAR-10 [160] and ImageNet [161] dataset for evaluation. CIFAR-10 consists of 60,000 32×32 color images in 10 classes, with 6,000 images per class. ImageNet is a large dataset of over 14 million images with up to 1,000 output classes, and it has been widely used for many computer vision research.

CNN Models We run comprehensive experiments on the state-of-the-art CNN models (VGG16 [147] and VGG19 [147], MobileNet [143], ResNet18 [124] and ResNet50 [124]. The major reasons of choosing these CNN models are 1) VGG16 and VGG19 are two most classic CNNs with linearly stacked layers; 2) MobileNet is the typical lightweight model with depth-separable convolution (DW+PW) blocks; 3) ResNet18 and ResNet50 is the representative model with the non-linearly stacked layers (residual connections).

Implementations We include three implementations for comparison: 1) **Pytorch-Base**: the baseline Pytorch implementation using channel-stack design (**CHS**) without channel-cyclic (**CC**) optimization; 2) **Pytorch-Opt**: the optimized Pytorch implementation using convolution-stack (**COS**) design with CC optimization; 3) **DSXplore**: DSXplore implementation comes with our efficient forward/backward pass design and CC optimization. Note that we omit the Pytorch implementation with CHS design and CC optimization, since CHS design still requires replicating the identified recurrent feature maps to build a large tensor before passing through the group convolution, which is identical to the Pytorch-Base implementation in terms of computation and memory complexity.

Platforms & Tools Our major evaluation platform is a server with an Intel Xeon Platinum 8168 (2.7 GHz, 24 cores), 188GB host memory and a Tesla V100 GPU (5,120 CUDA cores, Memory: 32GB, Peak Single-Precision: 15.7 TFLOPs). Note that to measure the end-to-end CNNs training performance, we use the Python `time` library and calculate the averaged running time of 100 measurements under the same setting. For kernel detailed metric analysis, such as GPU memory consumption, we use NVProf profiling tool from Nvidia.

6.5.2 Algorithmic Performance

In this experiment, we will first show the overall accuracy performance of different models optimized by DSXplore on CIFAR-10 in terms of computation (FLOPs), parameter saving and accuracy. We then apply our SCC kernel on ResNet50 and evaluate it on ImageNet dataset to show its applicability towards more complicated model on the large dataset. Moreover, we demonstrate the benefits of our SCC kernel design by using MobileNet on CIFAR-10 for a detailed study with the different number of groups (cg) and the channel overlapping ratios (co). Note that the value of cg should respect 1) the smallest channel number (64 for our selected models) of the convolution layers (excluding the input layer, which is usually 3 for RGB image), 2) balancing the parameter/computation reduction and the model accuracy. Our empirical study shows that $cg > 8$ would lead to significant accuracy degradation.

Overall Accuracy As shown in Table 6.2, DSXplore-optimized CNNs can strike a good balance between the model accuracy and the computation/parameter size. Overall, DSXplore can save 50.04% FLOPs in computation and 75.60% in parameter size on average, meanwhile maintaining accuracy to a great extent. This is because DSXplore leverages the SCC design that can maximize the value from the input information across different

Table 6.2: Accuracy comparison of CNNs on CIFAR-10.

Model	Implementation	MFPLOS	Param. (M)	Acc. (%)
VGG16	Origin	314.67	14.73M	92.64
	DSXplore	94.39	0.85M	92.60
VGG19	Origin	399.75	20.02M	93.88
	DSXplore	114.22	1.16M	92.71
MobileNet	Origin	67.31	3.19M	92.05
	DSXplore	45.29	1.63M	92.02
ResNet18	Origin	581.63	11.17M	95.75
	DSXplore	298.63	0.54M	94.81
ResNet50	Origin	2036.01	23.52M	95.82
	DSXplore	1469.64	12.81M	95.67

Table 6.3: Accuracy comparison (**ImageNet**) for ResNet50.

Network	MFLOPs	Param.	Acc.(%)
Origin	4130	23.67M	76.56
DSXplore	**2550**	**14.34M**	**75.91**

channels. We also notice that on VGG16 and VGG19, DSXplore can save more than 70% FLOPs and more than 90% parameters. The major reason is that the original VGG models rely on standard convolutions that would carry lots of redundant computations and parameters that contribute minor to the final prediction. In contrast, our SCC design in DSXplore can effectively captures the key factors (spatial and cross-channel information) that contribute most to the model prediction capability, thus, reducing the computation and parameters without compromising accuracy. Besides, for the complicated ResNet50 model on the challenging ImageNet dataset (Table 6.3), our SCC design can still reduce the FLOPs and parameters with up to 38.25% and 39.41% compared with the original model while maintaining accuracy.

Detailed Analysis A detailed studies on MobileNet (Table 6.4) further demonstrates the advantage of our SCC design in DSXplore in terms of a better trade-off between the model efficiency and the prediction accuracy. We try three different group numbers $cg \in \{2, 4, 8\}$), as well as two overlapping ratios $co \in \{25\%, 50\%\}$. Our model with DW+SCC-cg2-co25% achieves comparable accuracy performance in comparison with

Table 6.4: Comparison of different settings on MobileNet.

Network	MFLOPs	Param.	Acc.(%)
Baseline (DW+PW)	67.31	3.19M	92.05
DW+GPW-cg2	**45.29**	**1.63M**	**90.11**
DW+GPW-cg4	**34.28**	**0.84M**	**88.88**
DW+GPW-cg8	**28.78**	**0.45M**	**82.69**
DW+SCC-cg2-co25%	**45.29**	**1.63M**	**92.02**
DW+SCC-cg2-co50%	**45.29**	**1.63M**	**91.36**
DW+SCC-cg4-co25%	**34.28**	**0.84M**	**90.63**
DW+SCC-cg4-co50%	**34.28**	**0.84M**	**90.60**
DW+SCC-cg8-co25%	**28.78**	**0.45M**	**88.92**
DW+SCC-cg8-co50%	**28.78**	**0.45M**	**89.23**

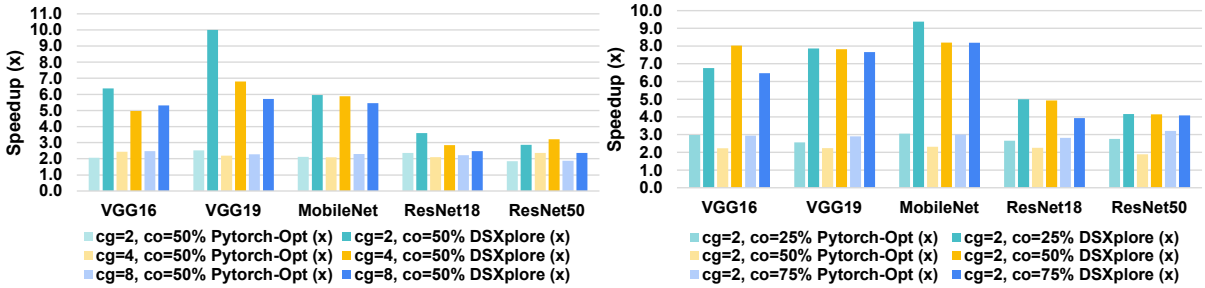


Figure 6.7: Runtime performance comparison on CIFAR10. Note that speedup is normalized *w.r.t.* **Pytorch-Base** Implementation.

baseline DW+PW model while saving about 32.77% FLOPs and 48.90% parameters. The reason is that under the setting of the small group number (*e.g.*, 2), our SCC-based channel sliding help filters learn different information by watching different channels. Note that in the larger group number (*e.g.*, 8), such benefits would be largely offset by the significantly reduced parameters. With an increase in the group number, we observe a significant reduction in both computational cost and parameter usage, along with a slight degradation in prediction accuracy. This aligns well with our expectation that the channel-group number *cg* determines the number of input channels that GPW or SCC would take, and thus also decides the number of computations and parameters of the model.

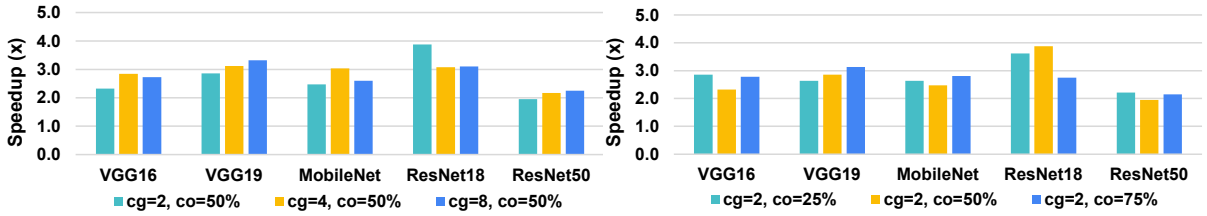


Figure 6.8: Runtime performance comparison on ImageNet. Note that speedup is normalized *w.r.t.* **Pytorch-Opt** Implementation.

6.5.3 Runtime Performance

In this section, we compare DSXplore with Pytorch-Base and Pytorch-Opt on CIFAR-10 and ImageNet for training across different CNNs, including VGG16, VGG19, MobileNet, ResNet18, and ResNet50. We use two set of settings for better coverage, where the first type of settings is $cg \in \{2, 4, 8\}$ and $co = 50\%$ while the second type of settings is $cg = 2$ and $co \in \{25\%, 50\%, 75\%\}$.

As shown in Figure 6.7, across all two types of settings, DSXplore consistently outperforms Pytorch-Base and Pytorch-Opt with average 5.68× and 2.34× speedup, respectively. We also notice that Pytorch-Opt is faster than Pytorch-Base, since our effective convolution-stack design and channel-cyclic optimization can be leveraged to reduce the memory overhead and excessive data movement (tensor slicing and concatenation), such an optimization can deliver 1.86× to 3.20× speedup compared with the Pytorch-Base implementation. DSXplore further boosts the performance on top of that by introducing channel-wise parallelism, which delivers an additional 1.11× to 3.97× speedup on average compared with Pytorch-Opt. Another key observation is that on VGG16 and VGG19, the performance benefits is more significant compared with ResNet18 and ResNet50. The major reason is that VGG models mainly rely on standard convolutions (carrying high computation complexity and large number of parameters) as the major building blocks, while the ResNet models would use either "Basic Blocks" or "Bottleneck Blocks" as the major building blocks. These "Blocks" include additional convolutions that are already

lightweight (such as the dual PW convolutions in Bottleneck Block) and no need to be replaced, thus, replacing the standard convolution of these blocks would have relatively lower impact. Moreover, under the same overlapping ratio, the increase of the *cg* would give more advantage to the Pytorch-based implementations, since the computation of each group convolution is reduced due to the smaller number of channels in each group. On the other hand, however, as notated in Section 6.5.2, the larger *cg* would lower the model accuracy. Therefore, we prefer lower value of *cg* in most settings.

In the comparison on ImageNet for the same set of CNNs, Pytorch-Base cannot even to run due to the excessive amount of the memory consumption. Therefore, we choose the Pytorch-Opt as the baseline for speedup normalization. As shown in Figure 6.8, DSXplore outperforms Pytorch-Opt with 1.95× to 3.88× speedup. This also demonstrate the scalability of DSXplore by effectively exploring the computation parallelism.

6.5.4 Additional Studies

Input-centric Backward Design To demonstrate the benefits of our input-centric backward optimization, we leverage four implementations (**Pytorch-Base**, **Pytorch-Opt**, a DSXplore variant (**DSXplore-Var**) with output-centric backward, which simply reverses the forward propagation flow for backward gradient propagation), and **DSXplore** (with input-centric backward). We evaluate the time of backward gradient propagation only. As shown in Figure 6.9, our input-centric backward achieves an average 15.03×, 4.55×, 1.55× speedup compared with Pytorch-Base, Pytorch-Opt, and DSXplore-Var, respectively. The advantage of DSXplore are two-folds: first, compared with Pytorch implementations, DSXplore with output-centric backward propagation (DSXplore-Var) and input-centric backward propagation (DSXplore) can explore the computation parallelism, meanwhile reducing lots of unnecessary data manipulation;

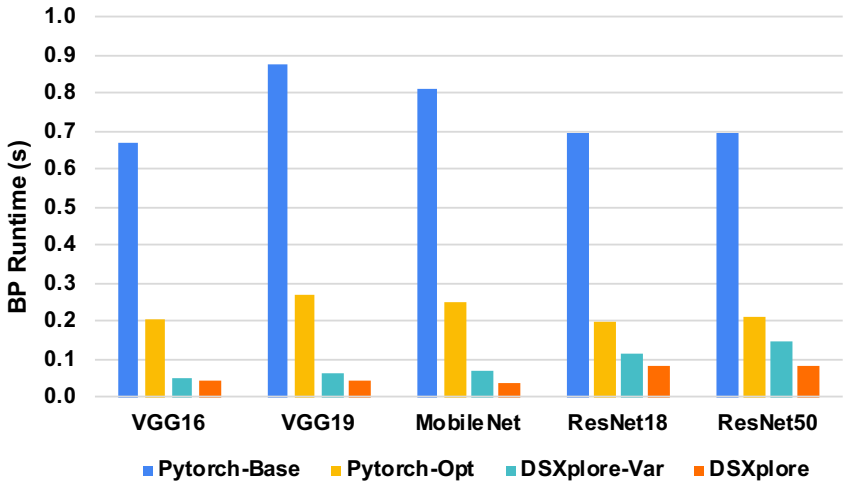


Figure 6.9: Back-propagation optimization.

second, compared with the DSXplore-Var, DSXplore can significantly reduce the atomic operations (more than 90% on average) on updating the gradient of the input tensor based on our kernel profiling via NVProf.

Cyclic-Channel Optimization As another key technical contribution, our CC optimization can effectively reduce the memory overhead from 72.88% to 83.33%, as illustrated in Figure 6.10. This is because the pattern of repeated channels will occur

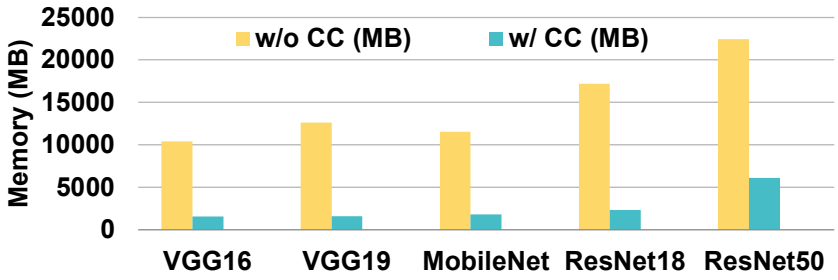


Figure 6.10: Channel-cyclic optimization.

periodically, as described in Section 6.4.2. And we only need to store them once instead of extracting and concatenating them all. To this end, we can avoid most of the data manipulation (*e.g.*, tensor slicing) and movement (*e.g.*, tensor concatenation). What is also worth noticing is that the impact of such optimization would also depend on the value of *co* we choose. Therefore, the cyclic distance would be largely different. For example,

when $co = 50\%$ with $C_{in} = 4$, we will have the cyclic distance of 4. When $co = 25\%$ with $C_{in} = 6$, we will have the cyclic distance as 3. This would also determine the size of concatenated feature map to be stored, thus, affecting the overall memory consumption.

Number of Groups As one of the features that is shared with the existing group convolution, our SCC divides the input and output channels into different groups based on the user-provided group numbers. Experiments from the detailed analysis of previous Section 6.5.2 already gives us the idea about its impact on model accuracy, and this study will help us to analyze its influence on the overall runtime performance in the end-to-end training. As shown in Figure 6.11, the increase in the number of groups will lead to the

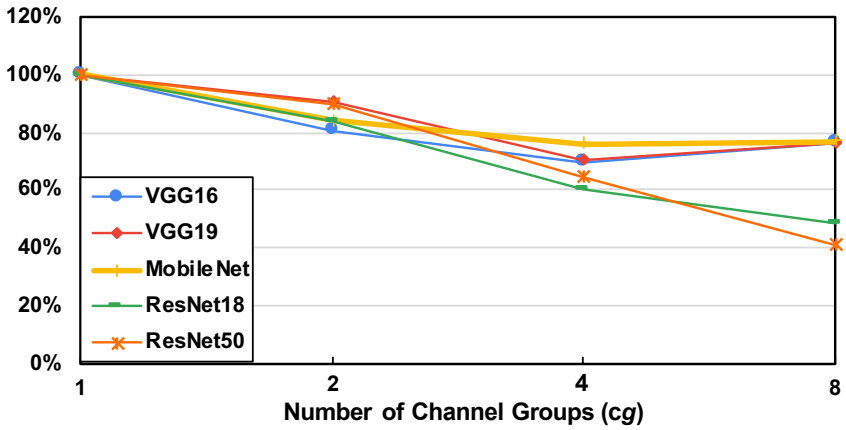


Figure 6.11: The performance impact of the number of groups (cg). Note that we set $co = 50\%$ and the runtime is normalized $w.r.t$ the performance at $cg = 1$.

reduction of the runtime, since with more channel groups, the corresponding group size (the number of input channels) required for each output channel will decrease. Therefore, the overall running time will decrease. Meanwhile, as discussed in the above section, the more number of groups will also leads to a slightly decrease of accuracy and reduction in parameter/computation costs. Therefore, in practice, we should balance the runtime performance and the model accuracy performance when choosing the value of cg.

Input-Channel Overlapping As the key feature that distinguishes SCC from the

existing factorized convolution kernels, the overlapping (co) of the filters' input channel facilitates the information fusion across different channels. Experiments from the detailed analysis of previous Section 6.5.2 already give us an idea about its impact on model accuracy, while this experimental study, on the other side, aims to help us understand how different choices of co would impact our SCC kernel performance. As shown in

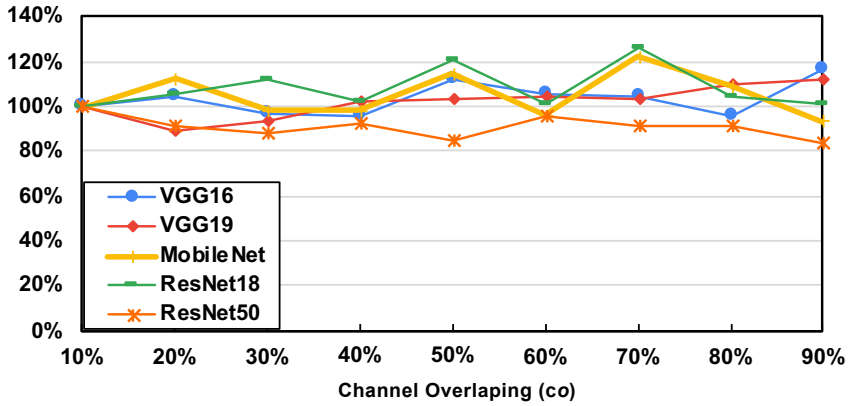


Figure 6.12: The performance impact of the input-channel overlapping ratio (co). Note that we set $cg = 2$ and the runtime is normalized $w.r.t$ the performance at $co = 10\%$.

Figure 6.12, the change of co for the adjacent sliding channel units does not show an evident impact on the runtime, since the overlapping ratio will not change the workloads assigned to different threads during the forward and backward pass. Even though there are some fluctuations in the running time, and it is mostly caused by the data reuse when choosing different co, which is also a very minor impact compared with the change of cg.

Training Batch Size As one of the most important factor of training CNNs, the batch size would impose a profound impact on the training, including the convergence rate, model accuracy, and training speed. In general, the larger batch size would lead to shorter training time. However, it may also degrade the model accuracy performance. In this experiment, we consider different batch sizes ranging from 16 to 1024 with the increase step by the power of 2. We select three CNNs: VGG16, MobileNet, and ResNet18,

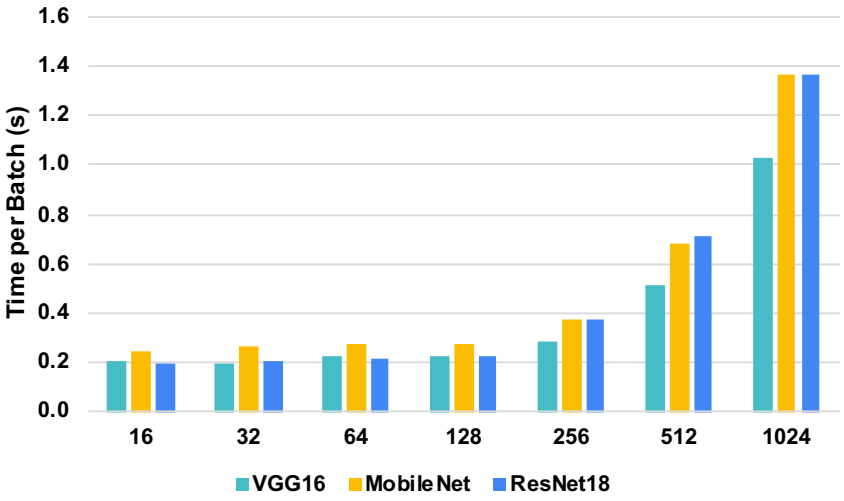


Figure 6.13: Impact of batch size on training performance.

which represent different types of CNN architectures for studies under the most common settings of $cg = 2$ and $co = 50\%$. As shown in Figure 6.13, the overall trend of the running time will increase with respect to the increase of the batch size. We also observe that within a certain range of batch size (less than 128), the increase of batch size does not lead to an evident increase of the running time. This is because of not enough active threads to saturate the available GPU Streaming Processors (SMs) to support the fully parallelized forward and backward computation. However, when the batch size becomes even larger, the active threads will increase correspondingly, which are more likely to compete with each other given the number of SMs for execution.

Chapter 7

An Efficient Quantitative Approach for Optimizing Convolutional Neural Networks.

With the increasing popularity of deep learning, Convolutional Neural Networks (CNNs) have been widely applied in various domains, such as image classification and object detection, and have achieved stunning success in terms of their high accuracy over the traditional statistical methods. To exploit potential of CNN models, a huge amount of research and industry efforts have been devoted to optimizing CNNs. Among these endeavors, CNN architecture design has attracted tremendous attention because of its great potential to improve model accuracy or reduce model complexity. However, existing work either introduces repeated training overhead in the search process or lacks an interpretable metric to guide the design. To clear these hurdles, we propose *3D-Receptive Field (3DRF)* [1], an explainable and easy-to-compute metric, to estimate the quality of a

[1] Reprinted from *An Efficient Quantitative Approach for Optimizing Convolutional Neural Networks.*. ACM International Conference on Information and Knowledge Management. 11/2021.

CNN architecture and guide the search process of designs. To validate the effectiveness of 3DRF, we build a static optimizer to improve the CNN architectures at both the stage level and the kernel level. Our optimizer not only provides a clear and reproducible procedure but also mitigates unnecessary training efforts in the architecture search process. Extensive experiments and studies show that the models generated by our optimizer achieve up to 5.47% accuracy improvement and up to 65.38% parameters deduction, compared with state-of-the-art CNN model structures like MobileNet and ResNet.

7.1 Introduction

Deep convolutional neural networks (CNNs) have achieved significant successes in a broad collection of fields, including object-detection [162], video classification [163], object tracking [164], image segmentation [165] and human pose estimation [166]. Such unparalleled successes attract many interests in CNN architecture design to *improve accuracy* or *reduce complexity*. Examples include an array of efficient models that have been crafted manually (*e.g.,* VGG [147], MobileNet [143], ShuffleNet [167]) and those generated automatically by the neural architecture search (NAS) tools [168, 169, 170, 171, 172]. Yet, two challenges of CNN architecture design remain far from well resolved: 1) missing an interpretable metric, and 2) huge training efforts. The former indicates that some direct and easy-to-interpret metric is still missing to guide the design, while the latter means that the repeated training cost is huge for evaluating different architectures in the search process.

To address these challenges, we propose *3D-Receptive Field (3DRF)*, an interpretable metric, for efficient CNN architecture designs. Particularly, we focus on two levels: the *stage level*[2] and the *kernel level*. At the stage level, we decide the number of convolution

[2]Following many works [124, 151, 173], we define a stage in a CNN as a collection of consecutive

kernels in different stages, while at the kernel level we choose the type of the convolution kernel to use (*i.e.,* standard convolution kernels or efficient factorized kernels [150, 151]). We build up *3DRF* to uniformly conduct the optimization at both levels. The key insight is that the portion of the input tensor that can flow into each output neuron, which we name as *3DRF*, often determines the learning potential of that given stage or kernel. A stage or kernel with larger *3DRF* will have more input elements passing through, leading to a higher potential for extracting useful features and improving the classification accuracy. Therefore, we use *3DRF* to estimate the quality of architecture design in the search process, rather than repeated training.

To validate and showcase the effectiveness of *3DRF*, we propose an *architecture optimizer* to examine CNN architecture designs at stage and kernel level. At stage level, we provide an *organizer* to improve the accuracy of a CNN model while using the same or fewer convolution kernels. The organizer, in effect, removes the convolution kernels that cannot contribute to *3DRF* enough or move the kernels from the positions with marginal contributions to *3DRF* in one stage to another stage with larger contributions. The optimization is based on two key observations: 1) the contributions from the latter kernels in a stage are diminishing since the newly observed input elements are on the marginal positions, which have less impact compared with the central input already observed; 2) when the spatial size of the input tensor to a stage is small, piling more layers can barely learn more features. On the other side, moving some layers to another stage with larger input tensor would promote *3DRF* and better learning capacity.

At the kernel level, we propose a *decomposer* to reduce model complexity without substantively affecting accuracy. The decomposer, in effect, replaces standard convolution kernels [3] with convolution blocks composed of efficient factorized kernels (*e.g.,* Depth-

convolution layers with *input tensors* of the same spatial dimensions (*i.e.,* pooling or convolution kernel with stride ≥ 2 will generate a new stage).

[3]In this paper, we refer to the standard convolution kernel as the one with 3 * 3 * C filters, where C

wise Convolution [150], and Pointwise Convolution [145]). The key guidance behind such replacement is to maintain the same *3DRF* (*i.e.,* the efficient convolution block should observe the same amount of *3DRF* as standard convolutions in order to maintain accuracy). We name this rule as *Rule for Kernel Replacement*. This rule not only allows us to unify all existing convolution blocks used in MobileNet, ShuffleNet, clcNet [173], and Xception [144], but also inspires the discovery of one new basic factorized convolution kernel, as we named *Rolling Pointwise Convolution (RPW)*, and a new convolution block (Depthwise (DW) + RPW). This new *convolution block* turns out to be more efficient than existing factorized kernel designs, like that in MobileNet model.

To facilitate the end-to-end CNN model design, we introduce our design prototype. As shown in the Listing 7.1, we start with importing our 3DRF-based optimization libraries, including a stage optimizer (`stage_opt`) and a kernel optimizer (`kernel_opt`). We will then build a CNN models as we normally do in the regular Pytorch. Here, convolutional layers in the CNN models can be grouped into different stages, where each stages consists of convolutions linearly stacked together. Different stages are sequentially connected. At the end of those stages, we put the linear (fully-connected) layer and a softmax layer layer to generate logits for classification.

In summary, the major contributions of our work are:

- We propose a brand-new interpretable metric *3D-Receptive Field (3DRF)* for guiding CNN architecture designs efficiently. Whereas previous CNN model architecture exploration techniques (*e.g.*, NAS) require huge training and searching efforts.

- We build an end-to-end CNN stage-level organizer for improving the accuracy performance of CNN models at the model architectural level. This can largely ease the manual efforts in arduous CNN model optimization process.

is the number of input channels.

Listing 7.1: Illustration of 3DRF-based Optimizer Prototype.

```
from 3DRF_optimizer import stage_opt, kernel_opt
# import other libraries, such as Pytorch...

# Create an stage of CNN model.
def make_stage(stage_depth):
    layers = nn.sequential()
    for i in range(stage_depth):
        layers.append(nn.conv2D(inChannel, outChannel))
    return layers

# Create a CNN model.
class CNN(nn.module):
   def __init__(self, stageDepth=[2,2,2,2], outClass=10):
        self.stages = torch.nn.moduleList()
        for depth in stageDepth:
            self.stages.append(make_stage(depth))
        self.classifier = nn.Linear(flatDim, outClass)
        self.softmax = nn.softmax()

    def forward(self, X):
        out = X
        for stg in self.stages:
            out = stg(out)
        out = self.classifier(out)
        out = self.softmax(out)
        return out
# Define a simple CNN Model.
model = CNN([2,2,2,2], 10)
# Compute the delta 3DRF for a input model.
info_3DRF = stage_opt.comp_Delta3DRF(model)
# Optimize the model structure with delta 3DRF.
model_opt = stage_opt.optimze_arch(model, info_3DRF)
# Optimize the kernel.
model_final = kernel_opt(model_opt)
# Do regular model training and inference.
```

- We introduce an new type of convolution kernel – Rolling-Pointwise Convolution to reduce the model parameters and the computation FLOPs.

Rigorous evaluations on real-world image datasets (*e.g.*, CIFAR-10/100 [160], and ImageNet [161]), demonstrate the strength of our architecture optimizer in terms of model accuracy, FLOPs and parameters. At the stage level, the organizer improves the accuracy (up to 5.47%) of the manually crafted CNN structures (*e.g.*, MobileNet) by maximizing the contribution to *3DRF*. For instance, the optimized MobileNet achieves 3.7% higher accuracy with 74% fewer parameters and 16% fewer FLOPs compared with

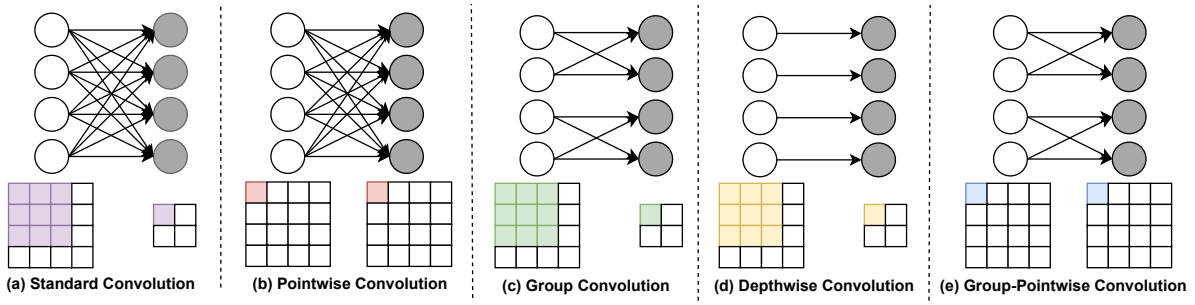


Figure 7.1: Channel mapping (top) and Spatial mapping (bottom) of the standard convolution and factorized convolution kernel.

the original structure. At the kernel level, the newly discovered convolution block achieves higher accuracy (up to 0.58%) with much fewer computations (up to 40.0% reduction) and parameters (up to 90.4% reduction) compared with the existing design. For example, one kernel designed by us has 2.54% higher accuracy and 29.04% fewer FLOPs in comparison with the MobileNet.

7.2 Related Work

7.2.1 Neural Architecture Search (NAS)

NAS methods have been widely studied to automatically construct efficient CNN architectures. NAS frameworks generally come with three major components, 1) *Search space:* The NAS search space is composed of several types of operations (*e.g.*, convolution, fully-connected, and pooling) and the inter-connection among these operators. The design of search space demands domain expertise from both the deep learning and the specific application settings; 2) *Search algorithm:* A NAS search algorithm samples a population of network architecture candidates. It receives the model performance evaluation result (*e.g.,* result) as rewards and optimizes to generate high-performance architecture candidates. 3) *Evaluation strategy:* This step will measure the performance

of candidate models in order to improve the search algorithm.

The most significant part of NAS research has been devoted to the neural architecture search algorithm. And a array of techniques and strategies have been proposed, such as evolutional algorithms [168, 174], hill climbing [175]; multi-objective search [176, 177], and reinforcement learning (RL) [168, 169]. To accelerate the NAS search, ENAS [178] represents the search space using a directed acyclic graph (DAG) and targeting at optimizing the subgraph structure within the large supergraph. Meanwhile, it also introduces a training strategy of parameter sharing among subgraphs to significantly boost the searching efficiency. Work from [179, 180] also follow the similar idea of hierarchical computation graph optimization. Work from [181] further share the parameters of different paths within a block using super-kernel representation. [182] proposes a fine-grained search space comprised of atomic blocks that is much smaller than the ones used in recent NAS algorithms.

Although NAS methods can build high-quality CNN architecture, they have two major drawbacks. First, they require prohibitively expensive computing power and add significant overhead to the design time. For instance, the RL-based method in [172] requires 500 NVIDIA P100 GPUs for more than 4 days to evaluate 20000 candidate neural networks, even after adopting many proxy tasks techniques including early stopping with few epochs, running on a small dataset, and limiting the kernel numbers. Second, the NAS method can identify the design, but it does not explain the general rule behind to obtain such a design, which limits its applicability. Once the task changes, one has to run NAS again. In contrast, our static architecture optimizer gives an alternative solution, offering a clear and reproducible design procedure without training in the architecture search process. Other works still requires non-trivial overhead of CNN runtime profiling for optimization.

7.2.2 Standard Convolution

The widely applied deep learning application demands effective ways to capture the characters of the inputs (*e.g.*, images). Among those techniques, the standard convolution is most widely used in many CNNs [146, 147, 148]. In general, we annotate the input image (I), output feature map (O), and filter (F). The dimension of an image is $[I_w, I_w, C_{in}]$, where I_W is the size of an image while C_{in} is the number of input channels (*e.g.*, the RGB image has 3 input channel). The standard convolution (Figure 7.1a) leverages C_{out} standard convolutional filters with the shape of $[K, K, C_{in}]$, where the K is the filter size, C_{in} is the number of input channels, and C_{out} is the number filters. After applying the standard convolution on the input (with the shape of $[I_w, I_w, C_{in}]$), we will get the output feature map O, which has the shape of $[O_w, O_w, C_{out}]$., where the O_w is size of the output feature map. Note that the mainstream CNNs [146, 147, 143] generally maintain the same feature map spatial dimension at different convolutional layers while only changing the number of the channels across different layers.

Formally, for standard convolution, we have

$$O_{m,n,c} = \sum_{i,j,a}^{K,K,C_{in}} F_{i,j,a,c} * I_{m+i-1,n+j-1,a} \tag{7.1}$$

where $O_{m,n,c}$ is one pixel point in the output feature map; m and n are the spatial indexes in the output feature map ($m \in Z : m \in [0, O_w)$ and $n \in Z : n \in [0, O_w)$); a is the channel index in the input feature map ($a \in [0, C_{in})$); c is the channel index in the output feature map ($c \in Z : c \in [0, C_{out})$); i, j, and a are the index used to accumulated the elementwise multiplication values between input feature map and one filter. The standard convolution will not only extract the spatial information by traversing a $K \times K$ 2D sliding window within each channel but also effectively fuses the information across

different channels (Figure 7.1a), where each kernel filter will gather the information from all input channels.

7.2.3 Kernel Factorization

Besides the standard convolution kernel, recent deep-learning research introduces several factorized kernels [145, 149, 150, 151] and combine them into a convolution block. This can offer another way to improve the computation efficiency of CNN architecture designs while maintaining the prediction power. Existing factorized kernels can be divided into four categories. Specifically, the first type is the Pointwise Convolution (PW) [145]) (Figure 7.1b), which is a standard convolution with 1×1 spatial size. The second type is Group Convolution (GC) [149] (Figure 7.1c) that divides input channels into several groups and performs standard convolution within each group. The third type is Depthwise Convolution (DW) [150] (Figure 7.1d) which calculates spatial convolution per channel or can be regarded as an extreme case of GC when the group number equals the number of the input channels. The last one is Group Pointwise Convolution (GPW) [151] (Figure 7.1e), that further splits PW into groups. Previously, researchers combine some of the factorized kernels into convolution blocks.

Xception [144] and MobileNet [143] demonstrate the successful application of convolutional kernel factorization in the popular CNN models. It breaks the original standard convolution into two parts: **depthwise** (DW) convolution and **pointwise** (PW) convolution. The first step (DW) applies C_{in} different $[W, W, 1]$ filters to each of the C_{in} input channels independently, which can be formalized as Equation 7.2

$$\hat{O}_{m,n,a} = \sum_{i,j}^{K,K} F_{i,j,a}^{(dw)} * I_{m+i-1,n+j-1,a} \tag{7.2}$$

The second step (PW) applies a filter with 1×1 spatial dimension. As shown in Equa-

tion 7.3.

$$O_{m,n,c} = \sum_{a}^{C_{in}} F_{a,c}^{(pw)} * I_{m-1,n-1,a} \tag{7.3}$$

In this paper, we use the idea of *3DRF* to unify these previous convolution blocks. In addition, we create a new type of factorized convolution kernel, named *Rolling Pointwise Convolution (RPW)*, and a new convolution block (DW+RPW) that can outperform the previous designs.

7.3 3D-Receptive Field

In this section, we present *3D-Receptive Field* (*3DRF*) for measuring the representation ability of each neuron in a convolution layer. Then, we derive the *3D-Receptive Field Gain* (*3DRF* Gain) for quantifying the representation ability change when an additional convolution layer is inserted. This *3DRF* Gain is sensitive to the location, type, and combination of the inserted convolution layer, thus guiding the CNN design. We demonstrate the effectiveness of *3DRF* Gain in quantifying representation ability, in terms of its impact on accuracy.

Our 3D-Receptive field is inspired by an existing metric, *receptive field* [183], which quantifies the spatial area of neurons for evaluating a single neuron in the next convolution layer. This receptive field serves well for quantifying the local representation ability in a single traditional convolution layer, where a larger receptive field leads to higher accuracy. However, the receptive field fails to quantify the global representation ability across layers, when a large number of convolution layers with diverse receptive fields stacked in a CNN stage. Moreover, the receptive field fails to consider the channel number, which becomes critical in modern convolution layers (*e.g.*, Depthwise convolution and Channel-wise convolution). By contrast, our *3DRF* provides the first global metric

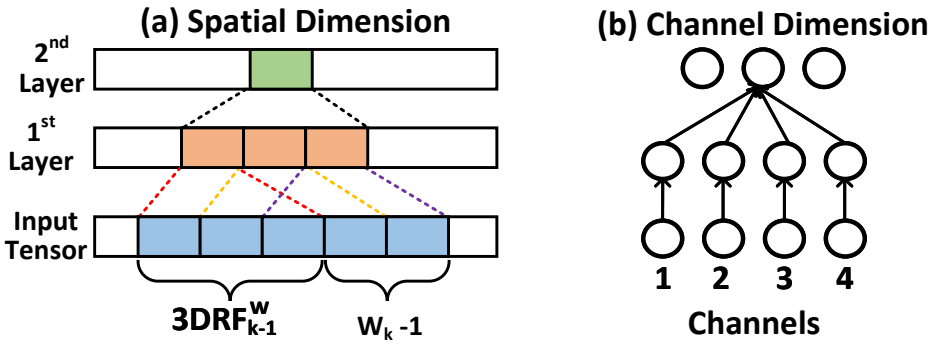


Figure 7.2: Illustration of 3D-Receptive Field (*3DRF*) for convolutions of a single stage.

for quantifying the global representation ability across layers, considering extensively the location, type, and combination of convolution layers. By quantifying the global representation ability, *3DRF* serves as an effective and efficient tool for guiding the CNN design without tediously enumerating and training NN architectures.

7.3.1 Definition of 3D-Receptive Field

For a CNN stage with a sequence of layers, we define the 3D-Receptive Field (*3DRF*) for the k^{th} convolution layer in the current stage as $3DRF_k$. This $3DRF_k$ captures the number of neurons in the initial input tensor to the CNN stage that contributes to computing individual neurons in this layer k. This initial input tensor is the $w_0 \times w_0 \times 3$ input tensor (*e.g.*, input image) in the first stage of a CNN, and a $w_0 \times w_0 \times c_0$ input tensor in later stages. Here, w_0 is the spatial width of the input tensor and c_0 is the channel number of the input tensor. To cater convolution layers with diverse kernel sizes and types, *3DRF* considers two factors of the spatial width $3DRF_k^w$ for the kernel size and the channel number $3DRF_k^c$ for the convolution type:

$$3DRF_k = (3DRF_k^w)^d * 3DRF_k^c \tag{7.4}$$

where $d = 1$ for 1D convolution (Figure 7.2) and $d = 2$ for 2D convolution. We recursively compute the spatial width $3DRF_k^w$ in layer k based on the spatial width $3DRF_{k-1}^w$ in the preceding layer $k-1$ and the kernel width w_k in the current layer k:

$$3DRF_k^w = \min(3DRF_{k-1}^w + w_k - 1, w_0) \tag{7.5}$$

A $min()$ is applied for ensuring that the spatial width $3DRF_k^w$ does not exceed the spatial width w_0 of the input tensor.

We compute recursively the channel number $3DRF_k^c$ in layer k with a property function $g(\cdot,\cdot)$, that captures the channel number $3DRF_{k-1}^c$ in the preceding layer $k-1$ and the convolution type T_k in the current layer k:

$$3DRF_k^c = \min(g(3DRF_{k-1}^c, T_k), c_0) \tag{7.6}$$

A $min()$ is applied for ensuring that the channel number $3DRF_k^c$ does not exceed the channel number c_0 of the input tensor. The property function $g(\cdot,\cdot)$ captures the information flow from the perspective of channel numbers and is designed for individual convolution types. For example, as illustrated in Figure 7.2, we set the property function $g(3DRF_{k-1}^c, PW) = c_0$ for Pointwise (PW) Convolution, since the output neuron of PW observes all input channels. Similarly, we set $g(3DRF_{k-1}^c, DW) = 3DRF_{k-1}^c$ for Depth-wise Convolution (DW), since only one channel from the preceding layer $k-1$ contributes to the neuron in the current layer k. This property function $g(\cdot,\cdot)$ is designed only once for a small set of convolution types. While modern CNNs may have hundreds of convolution layers, these layers often use the same convolution type repeatedly. Thus, the property function can be written once and applied repeatedly for a large number of convolution layers.

7.3.2 Definition of *3DRF* Gain

We derive the *3DRF* Gain ($\Delta 3DRF$) to measure the impact of a convolution layer k over the model representation ability, in terms of the impact over the 3DRF. While *3DRF* quantifies the information flow in a convolution unit as a whole, *3DRF Gain*—denoted by $\Delta 3DRF$— targets at measuring the contribution of a single convolution kernel k in the unit. The goal of introducing $\Delta 3DRF$ is to create a direct indicator that could match the learning power (*i.e.,* prediction accuracy) of a CNN model in the granularity of a single convolution, laying a foundation for static architecture optimization. Specifically, we define $\Delta 3DRF_k$ as the difference in the receptive field with and without the layer k, adjusted with an exponential decay term:

$$\Delta 3DRF_k = \frac{3DRF_k - 3DRF_{k-1}}{3DRF_{k-1}} * e^{-\alpha * \frac{3DRF_{k-1}}{V_0}} \tag{7.7}$$

where $V_0 = w_0 \times w_0 \times c_0$ is the volume of the input tensor. The exponential decay term rescales the impact of the k^{th} layer with regards to the information already observed by 1^{th} to $(k-1)^{th}$ layers. which composes of two major terms: the former calculates the relative increase in *3DRF* incurred by kernel k; the latter introduces an exponentially decay term to rescale the impact of the k^{th} layer with regards to the information already observed by 1^{th} to $(k-1)^{th}$ layers. This decay term is inspired by the observation that the elements in the central region of the input tensor usually have a larger impact than the newly observed elements on the margin: the central input elements have more paths to propagate their values into the output in the forward pass and larger gradient in the backward pass. Note that α is a hyperparameter that should be set larger than 0. In our empirical study, we tried multiple choices and observed no substantial difference in architecture optimization, and we set it to 3 for the rest of this paper.

Table 7.1: Illustration of computing *3DRF* Gain on Variant-3.

k	Layer Type	$3DRF_k^w$	$3DRF_k^c$	$3DRF_k$	$\Delta 3DRF_k$
1	conv3-256	3	128	1152	-
2	conv3-256	5	128	3200	1.17
3	conv3-256	7	128	6272	0.29

Table 7.2: Impact of *3DRF* Gain ($\Delta 3DRF$) over Accuracy.

Network	$\Delta 3DRF$	Accuracy (%)	ΔAccuracy (%)
VGG-11	0	92.68	0
Variant-1	1.73	93.56	0.88
Variant-2	1.60	93.46	0.78
Variant-3	0.29	92.75	0.07
Variant-4	0.0	92.58	-0.10
Variant-5	0.0	92.41	-0.27

7.3.3 Case Study: Accuracy Impact of *3DRF* Gain

We demonstrate the impact of diverse ($\Delta 3DRF$) over the accuracy. Here we generate diverse ($\Delta 3DRF$) by sticking to the same base model and inserting an additional convolution layer at diverse location. More study on the ($\Delta 3DRF$) from varying the type and combination of convolution layers will be conducted later in the evaluation section. As shown in Figure 7.2, we take VGG11 [147] as the baseline structure and run it on CIFAR-10 dataset [160]. Specifically we generate five VGG-variants by inserting a single standard convolution before each max pooling. The inserted convolution layer has the same kernel width and channel number as its preceding layer. For example, we insert a conv3-64 before the first max pooling as the Variant-1, and a conv3-512 before the fifth max pooling as the Variant-5. Specifically, we train these models on the CIFAR-10 training dataset and report the accuracy on the CIFAR-10 testing dataset. We repeat this procedure for ten times and present the average accuracy here. We also present the $\Delta 3DRF$ of each variants for demonstrating the impact of $\Delta 3DRF$ over accuracy.

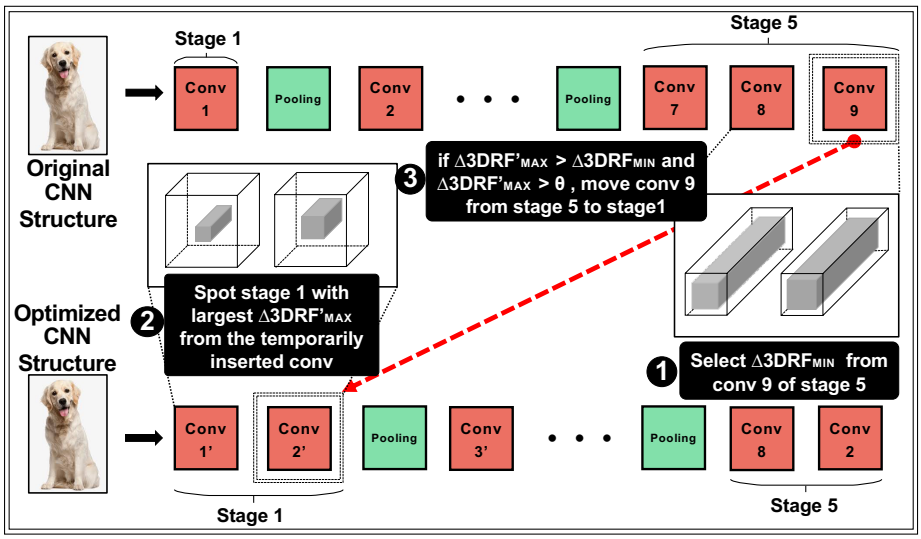


Figure 7.3: Illustration of the Stage-level Organizer.

$\Delta 3DRF$ is calculated by leveraging our proposed Equation 7.7 for the newly inserted layer.

As shown in Table 7.1, the procedure of computing $\Delta 3DRF$ on Variant-3, which inserts an additional layer to the third stage in VGG-11. Originally, the third stage in VGG-11 contains two convolution layers (*i.e.*, the 1^{st} layer and the 2^{nd} layer in Table 7.1). We insert the 3^{rd} convolution layer with the same kernel width and channel number as the first two layers. The input tensor to this third stage is of shape $8 \times 8 \times 128$, leading to a V_0 of 8192. Following Equation 7.4 - 7.6, we can compute $3DRF_k^w$, $3DRF_k^c$, and $3DRF_k$ recursively. The derived $3DRF_k$ can be exploited for computing $\Delta 3DRF$ following Equation 7.7. This procedure can be applied for other VGG-11 model variants, leading to the $\Delta 3DRF$ in Table 7.2.

As shown in Table 7.2, we can clearly figure out the the impact of $\Delta 3DRF$ on CNN model accuracy. Large $\Delta 3DRF$ of the newly inserted layer agrees with notable accuracy gain, as is the case for *Variant-1* and *Variant-2*. For the *Variant-3*, small $\Delta 3DRF$ indicates close-to-saturation information coverage, yielding negligible accuracy improvement from the original model. Variant-4 and Variant-5 has a low $\Delta 3DRF$ of

0, indicates that inserting convolution layers does not improve its *3DRF*. The insight is that, for an input tensor with a small spatial width w_0 of 2 (after 4 times of max pooling from an input image of shape $32 \times 32 \times 3$), a single convolution layer of kernel width 3 is sufficient for capturing all neurons. In fact, Variant-4 and Variant-5 show an accuracy degradation of -0.10% and -0.27% respectively. This degradation shows that a $\Delta 3DRF$ of 0 signals overfitting since all input elements have already been observed by other kernels at such stage. Comparing across variants, Variant-1 has a larger $\Delta 3DRF$ of 1.73 and a larger $\Delta Accuracy$ of 0.88%, compared with Variant-5 with $\Delta 3DRF$ of 0.0 and $\Delta Accuracy$ of -0.27%. This trend demonstrates a strong correlation between the $\Delta 3DRF$ and the $\Delta Accuracy$, thus guiding the NN design in terms of the insertion location.

To sum up, $\Delta 3DRF$ effectively probes the potential of accuracy improvement, and we leverage such an easy-to-compute metric to build our architecture optimizer in Section 7.4.

7.4 Architecture Optimizer via 3DRF

We build a static *Architecture Optimizer* based on *3DRF* and $\Delta 3DRF$. It examines the structure inefficiency in a given CNN architecture and optimizes it at the *stage level* and *kernel level*.

7.4.1 Stage-Level Organizer

Stage-level organizer (Figure 7.3) manages to improve the prediction accuracy of a CNN design by iteratively removing a convolution kernel from a saturated stage or moving it to another stage with more room to absorb new information (*i.e.,* learn from more marginal elements introduced by the kernel).

Three sub-steps are conducted in each iteration. The first step is to find the convolution kernel with minimum $\Delta 3DRF$, which has the lowest contribution to the 3DRF. In consideration of the decaying property of $\Delta 3DRF$ within a stage, this step can be simplified to compute the $\Delta 3DRF$ of the last convolution kernel in each stage. Comparing across stages, we select the convolution layer with the minimum *3DRF* Gain, denoted as $\Delta 3DRF_{MIN}$ in Figure 7.3, and identify the corresponding stage as the *source stage*. This identified convolution layer will be either deleted or moved from the source stage to another stage, in the following steps.

The second step is to spot the stage with the largest room for improving 3DRF. This step follows the insight from our case study that a larger $\Delta 3DRF$ often leads to higher accuracy. We tentatively append the convolution kernel identified in the first step to each stage and compute the corresponding $\Delta 3DRF$. When appending the convolution layer, the input and output channel number will be adjusted for catering to the preceding layers in the source stage and the following layer in the next stage if available. Comparing across stages, we can find the one, called *target stage*, with maximum $\Delta 3DRF$ for the appended layer ($\Delta 3DRF'_{MAX}$ in Figure 7.3). This step follows the insights obtained from our case study that a strong correlation exists between $\Delta 3DRF$ and $\Delta Accuracy$. to conduct architecture optimization.

The third step decides whether moving the last convolution layer from the source stage to the target stage or simply removing this layer. When moving the convolution layer, we adjust the input channel number and the output channel number with the same strategy in the second step. This step follows the insights obtained from our case study to conduct architecture optimization. There are three key choices: 1) If $\Delta 3DRF'_{MAX} > \Delta 3DRF_{MIN}$ *and* $\Delta 3DRF'_{MAX} > \theta$, we move the last kernel from the *source stage* and append it to the *target stage*; 2) If $\Delta 3DRF'_{MAX} < \theta$ *and* $\Delta 3DRF_{MIN} < \theta$, we just remove the last kernel from the *source stage* (no appending); 3) If $\Delta 3DRF_{MIN} > \Delta 3DRF'_{MAX}$

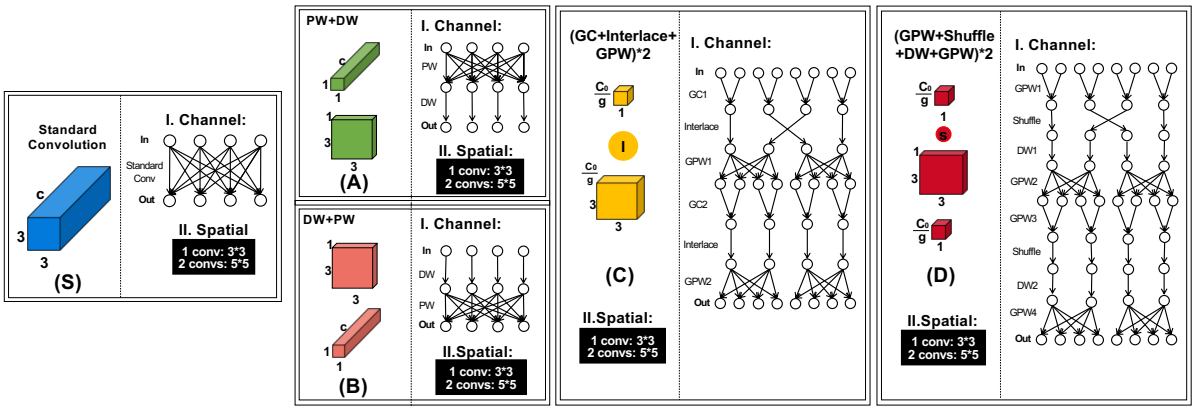


Figure 7.4: Illustration of the *3DRF*, both in the channel (I) and spatial (II) dimension, for the standard kernels (S) and previous convolution blocks (A-D). g is the number of groups for GC and GPW. The arrow denotes the flow from inputs to outputs in the channel dimension, and the number of input channels that could flow into an output neuron would be the channel dimension of *3DRF* for that block. We omit the process of computing the spatial size of *3DRF*, while only giving the computed result based on Equation 7.4 in the figure.

and $\Delta 3DRF_{MIN} > \theta$, we keep the original structure and terminate our optimization procedure. Here the hyperparameter θ is the border we draw empirically to distinguish underfitting from overfitting. For example, θ is set to 0 for VGG. Following this iterative optimization procedure, our organizer manages to mitigate the structure-level inefficiency in a CNN design via static architecture optimization. The experimental results of the organizer can be found in our evaluation.

7.4.2 Kernel-Level Decomposer

At the kernel level, our *decomposer* reduces the computational cost of a CNN architecture design, by substituting its *standard convolution kernels* with less computational expensive *convolution blocks*. The key challenge here is to construct such an efficient and effective *convolution block* with multiple factorized kernels. Previous manual efforts by domain experts have made some progress [148, 173, 151], but the underlying design principle remains unclear. In this paper, we provide the first easy-to-follow design principle,

Rule of Kernel Replacement, to guide the design of efficient convolution blocks.

Rule of Kernel Replacement To avoid significant accuracy degradation and achieve computation efficiency, a convolution block N can replace the standard convolution kernels S only if two conditions are satisfied: 1) *Quality Condition*: $3DRF(N) = 3DRF(S)$ for the same input tensor; 2) *Compact Condition*: $3DRF(N - x) < 3DRF(S)$ if we remove a factorized kernel x from N. The former ensures the effectiveness of N with regards to its learning capacity, while the latter guarantees its optimality in terms of computation efficiency. The rule helps us unify the previous construction of the convolution block, as well as inspires us to build a new convolution blocks and one efficient factorized kernel.

Unifying Existing Convolution Blocks This section shows that the previous four convolution blocks follow the *Rule of Kernel Replacement*: they have the same *3DRF* as the standard convolutions and they are already in the compact form that cannot be further simplified. Figure 7.4 depicts the *3DRF* for a standard convolution block (S) and four previously explored convolution blocks (A-D), in their spatial and channel dimensions. As shown in Figure 7.4 (S), the *3DRF* spatial size $3DRF_1^w$ for S is 3 for one standard convolution and $3DRF_2^w$ is 5 when two standard convolutions are packed together in the block. The *3DRF* channel dimension $3DRF_k^c$ for S equals the number of the input channels to the block.

Convolution block A (adopted by Xception [144]) and B (applied in MobileNet [143]) follow a similar structure. Both A and B successfully maintain the same *3DRF* with that of S with one standard kernel. Specifically, the spatial coverage is managed by DW [4] and channel coverage is taken care of by PW, which communicates the information among all input channels. Convolution block C (used in clcNet [173]) and D (utilized by ShuffleNet [151]), on the other hand, achieve the same *3DRF* with that of S with two

[4]Definitions of factorized kernels like DW can be found in the Related Work Section.

standard kernels. Take block C (shown in Figure 7.4 (C)) as an example, one combination of GC, Interlace, and GPW, can perceive the same spatial region but only half of the entire input channels, compared to a standard convolution kernel. But with one extra GC+Interlace+GPW, the channel dimension gets full coverage. Thus, the *3DRF* is the same for the block with (GC+Interlace+GPW) * 2 and two standard convolutions. The proof of the compactness for four convolution blocks is omitted, but it is clear from the plot that if we remove any of the factorized kernels, the *3DRF* cannot be maintained.

New Kernel Design Inspired by the *Rule of Kernel Replacement*, we discover an unexplored convolution block and a new type of factorized kernel, shown in Figure 7.5. The first block includes a DW, a channel shuffle, and a GWC. The *key insight* of the design is choosing a DW to capture information in the spatial dimension and using a GPW with a shuffle operation to observe full channel information. Since the PW contributes to the majority of the computations in the previous factorized design (more than 95% FLOPs in MobileNet [143]), the usage of GPW to replace PW can largely reduce the computation cost, compared to blocks like (A) and (B).

The convolution block we come up with composes of a DW and a Rolling-Pointwise Convolution (RPW), as shown in the left side of Figure 7.5 (model F). The comparison between RPW and GPW is presented in the right side of Figure 7.5. Different from GPW, RPW is the new factorized convolution kernel we invented, where adjacent convolution filters partially overlap in the channel dimension. The overlapped part serves as a bridge to communicate the different channel information and allows the later kernel to observe different channels without channel shuffle. Specifically, there are two parameters that come with RPW: group number g and overlap ratio o. For instance, RPW-gX-oY% denotes each filter in the convolution kernel takes $\frac{1}{X}$ number of input channels, while adjacent filters in RPW have y% overlap in their consumed channels. The newly designed block outperforms previous designs in accuracy, memory and computation efficiency,

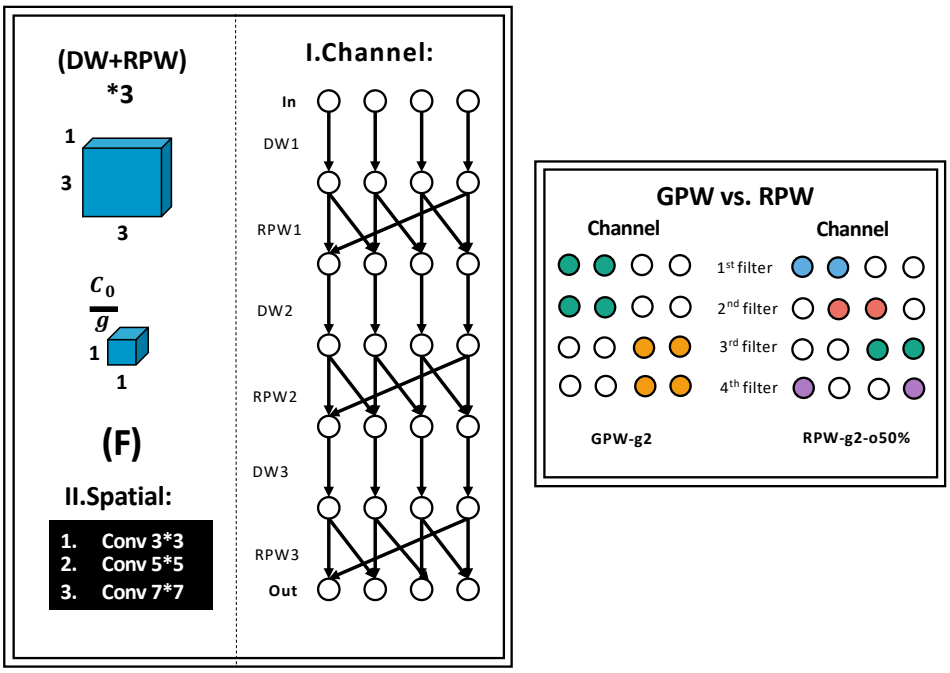


Figure 7.5: **Left**: DW+RPW convolution block design. **Right**: Comparison of RPW kernel with GPW kernel. Note that in RPW, adjacent filters overlap in channel dimensions.

which are detailed in our evaluation.

Implementation of New Kernel Design To implement the new rolling-pointwise convolution, we introduce two kind of implementation by compositing the existing Pytorch Operators. First, we can first extract the corresponding channels and concatenate them together. We will leverage the existing Pytorch operators, such as tensor slicing, concatenation, and standard group convolution. There are several steps, as shown in Listing 7.2. The second type of design is to let the convolution iterate through the input channel. The second implementation circumvents the "huge" concatenated tensor in the above implementation by applying convolution operation before concatenating. One major key insight is that the computation on the large concatenated tensor can be decomposed into the more effective computation on a set of small tensors. Instead of simply combining all the extracted features maps, we can pre-build a set of lightweight convolu-

Listing 7.2: Compositing RPW via PyTorch Operators.

```
width = int(input_channel/num_groups)
start, end, start_v, end_v= 0, width, 0, width
item_set, slice_li = set(), []
# input channel range for each kernel filter.
for fid in range(output_channel):
    item_set.add((start,end)); slice_li.append((start,end))
    start_v = end_v - int(overlap * width)
    end_v = start_v + width
    start, end= start_v%input_channel, end_v%input_channel
# define a groupwise convolution.
conv2D = nn.Conv2d(width*len(item_set), len(item_set),
                   kernel_size=1, groups=len(item_set))
# forward computation.
def forward(input):
    comb_unit = []
    for idx in range(len(item_set)):
        item = slice_li[idx]
        start, end = item[0], item[1]
        if start > end and start < input_channel:
            tmp = input[:, start:, :, :]
            tmp_1 = input[:, :end, :, :]
            new_tmp = torch.cat([tmp, tmp_1], dim=1)
            comb_unit.append(new_tmp)
        else:
            comb_unit.append(input[:, start:end, :, :])
    comb_tensor = torch.cat(combined_unit, dim=1)
    return conv2D(comb_tensor)
```

tions, each of which will generate the feature map for only one kernel filter. Finally, we concatenate these output feature map together. While this solution can largely overcome the third problem of the above channel-stack implementation, it is still hindered by the excessive inefficient Pytorch operations and lack of parallelization.

7.5 Evaluation

To validate the effectiveness of the architecture optimizer, we run comprehensive experiments on the state-of-the-art CNN models (VGG16 and VGG19 [147], MobileNet [143] and ResNet50 [124]. The major reason of choosing these CNN models are 1) VGG16 and VGG19 are two most classic CNNs with linearly stacked layers; 2) MobileNet is the representative lightweight model with DW+PW convolution block; 3) ResNet50 is the

representative model with the non-linearly stacked layers (residual connections).

Dataset: We use CIFAR-10 (CIFAR-100) [160] and ImageNet [161] dataset for evaluation. CIFAR-10 consists of 60,000 32×32 colour images in 10 classes, with 6,000 images per class. CIFAR-100 dataset is just like the CIFAR-10, except it has 100 classes containing 600 images each. ImageNet is a large dataset of over 14 million images with up to 1,000 output classes, and it is mainly used for computer vision research, such as image classification.

Training Settings: We follow the conventional settings [184] for training and testing on CIFAR-10 and CIFAR-100: learning rate starts from 0.1 and decays by the factor of 0.1 after 150 and 250 epochs, with 350 epochs in total. We adopt SGD with 0.9 momentum and 5e-4 for the weight decay. We apply normalization for the input image with (0.491, 0.482, 0.446) for each RGB channel as the mean and (0.247, 0.243, 0.261) for standard deviation, respectively. And we select two state-of-the-art Pytorch CNNs implementations on CIFAR-10 and CIFAR-100 , respectively. For ImageNet, we use the official Pytorch implementations [5] and choose learning rate starts with 0.1 with total 120 epochs. We adopt SGD with 0.9 momentum and 1e-4 weight decay. We also apply normalization for the input image with (0.485, 0.456, 0.406) for each RGB channel as the mean and (0.229, 0.224, 0.225) for standard deviation. We select the pre-trained model as the baseline from Pytorch official website.

7.5.1 Stage-Level Organizer

This experiment aims to demonstrate the effectiveness of our stage-level organizer. Specifically, we first use CIFAR-10 and CIFAR-100 for detailed analysis, and further leverage ImageNet to show our design applicability and scalability towards the challenging state-of-the-art large dataset. Table 7.3 exhibits the performance of various CNNs

[5] `github.com/pytorch/examples/tree/master/imagenet`

Table 7.3: Performance comparison (CIFAR-10) between original CNNs and reorganized structures.

Network	MFLOPs	Param.	Acc. (%)	$\Delta 3DRF$
VGG16	310	14.73M	92.64	-
VGG16-opt	370	**5.10M**	**92.95**	2.30
VGG19	400	20.04M	91.91	-
VGG19-opt	490	**8.09M**	**92.89**	3.13
MobileNet	50	3.22M	90.67	-
MobileNet-opt	50	**1.13M**	**92.05**	3.94
ResNet50	1,300	23.52M	93.75	-
ResNet50-opt	1,310	**17.24M**	**95.79**	0.76

Table 7.4: Performance comparison (**CIFAR-100**) between original CNNs and reorganized structures.

Network	MFLOPs	Param.	Acc. (%)	$\Delta 3DRF$
VGG16	330	34.02M	72.93	-
VGG16-opt	390	**24.39M**	**74.64**	2.30
VGG19	420	39.33M	72.23	-
VGG19-opt	500	**27.38M**	**74.00**	3.13
MobileNet	50	3.32M	65.98	-
MobileNet-opt	50	**1.23M**	**71.45**	3.94
ResNet50	1,310	23.71M	77.39	-
ResNet50-opt	1,380	**21.89M**	**78.25**	0.76

optimized by the stage-level organizer, including computation complexity (MFLOPs), parameter size, and accuracy. It is clear that the stage-level organizer can improve the accuracy of various state-of-the-art CNN models. On CIFAR-10 and CIFAR-100, stage-level organizer improves the accuracy of four evaluated models by 1.18% and 1.90% on average, while reducing model parameters by 54.15% and 32.33% on average, respectively. We also notice on the more complicated model, such as ResNet50, the accuracy improvement is notable (2.04% on CIFAR-10 and 0.86% on CIFAR-100). The original

Table 7.5: Performance comparison (**ImageNet**) between original CNNs and reorganized structures.

Network	MFLOPs	Param.	Acc. (%)	$\Delta 3DRF$
VGG16	15,500	138.36M	71.59	-
VGG16-opt	16,900	**133.82M**	**72.17**	0.39
VGG19	19,670	143.67M	72.38	-
VGG19-opt	21,060	**141.34M**	**72.61**	1.09
MobileNet	580	4.23M	70.60	-
MobileNet-opt	570	**3.52M**	**71.05**	2.59
ResNet50	4,120	25.56M	76.15	-
ResNet50-opt	4,130	**23.67M**	**76.56**	0.47

ResNet50 model has 4 stages. Each stage contains {3, 4, 6, 3} bottleneck blocks respectively. Following the iterative optimization steps, the organizer moves the last two blocks from the third stage to the first stage and the last block from the last stage to the second stage to generate an optimized ResNet50 containing {5, 5, 4, 2} blocks in each stage. By improving the total $\Delta 3DRF$, this optimized architecture gets both higher accuracy and fewer model parameters. In addition, on the lightweight MobileNet model, which has factorized kernel designs (DW+PW) with the smallest number of parameters, our stage-level organizer also achieves a notable performance improvements (1.38% on CIFAR-10, and 5.47% on CIFAR-100). This is because our organizer finds five convolutions—four from the fourth stage and one from the last stage—which suffer from small $\Delta 3DRF$. By moving these convolutions to the first and second stage, we get a new architecture contains {4, 4, 2, 2, 1} convolutions in each stage, which offers a more efficient architecture in terms of less model parameters and higher accuracy. On the challenging ImageNet, our stage-level organizer can still effectively reduce the number of model parameters (up to 16.7%), meanwhile improving the testing accuracy (up to 0.58%) compared with baseline.

Table 7.6: Kernel-level design (**CIFAR-10**) on VGG16-opt.

Network	MFLOPs	Param.	Acc.(%)
Baseline	370	9.64M	92.95
DW+PW	50	1.11M	92.12
DW+GPW-g2	30	0.67M	92.35
DW+GPW-g4	20	0.36M	88.05
DW+GPW-g8	10	0.20M	86.41
DW+RPW-g2-o33%	**30**	**0.66M**	**92.52**
DW+RPW-g2-o50%	**30**	**0.66M**	**92.70**
DW+RPW-g4-o33%	**20**	**0.36M**	**91.61**
DW+RPW-g4-o50%	**20**	**0.36M**	**91.59**
DW+RPW-g8-o33%	**10**	**0.20M**	**89.86**
DW+RPW-g8-o50%	**10**	**0.20M**	**90.19**

7.5.2 Kernel-Level Decomposer

This experiment aims to demonstrate the benefits of our brand-new kernel design. We first use VGG16-opt (with stage-level optimization) on CIFAR-10 for a detailed study. We further highlight our new kernel scalability by applying it towards the complicated ResNet50-opt model on ImageNet. Table 7.6 shows that our new convolution block based on rolling-channel design achieve a better balance between the model efficiency and the prediction accuracy on VGG16-opt on CIFAR10, in contrast to DW+PW factorized kernel design. We tried three different group numbers g (2, 4, 8), as well as two overlapping ratios o (33%, 50%). Our model with DW+RPW-g2-o50% achieves a better accuracy compared to the high-performance DW+PW model while saving about 40.0% FLOPs and 40.5% parameters. With an increase in the group number, we observe a significant reduction in both computational cost and parameter usage, along with a slight degradation in prediction accuracy. This aligns well with our expectation that the group number

g determines the number of input channels that GPW/RPW would take, and thus also decides the number of computations and parameters of the model.

We also notice that our new convolution block design consistently outperforms with the ones without overlap (o) under the same number of groups (g). For example, our new design (DW+RPW-g4-o33%) outperform DW+RPW-g4 with 3.56% better accuracy. Under the settings with same number of group in RPW, such as DW+RPW-g2-o33% vs. DW+RPW-g2-o50%, the latter with higher overlap ratio offers higher accuracy, indicating the effectiveness of overlapping channels to improve model accuracy.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

In Chapter II, I propose QGTC, the first QGNN computing framework to support any-bit-width computation via GPU Tensor Core. Specifically, I introduce the first GNN-tailored any-bitwidth arithmetic design that can emulate different bitwidth computations to meet the end-user's demands. We craft a TC-tailored CUDA kernel design by incorporating 3D-stacked bit compression, zero-tile jumping, and non-zero tile reuse techniques to maximize the performance gains from GPU Tensor Core. We also incorporate an effective bandwidth-optimized subgraph packing strategy to maximize the data transferring efficiency. Finally, I integrate QGTC with the popular PyTorch framework for better programmability and extensibility. Extensive experiments show significant performance gains of QGTC in practice.

In Chapter III, I propose, GNNAdvisor, an adaptive and efficient runtime system for GNN acceleration on GPUs. Specifically, I explore the potential of GNN input-level information in guiding system-level optimizations. We further propose a set of GNN-tailored system-level optimizations (*e.g.*, 2D workload management, and specialized memory opti-

mizations) and incorporate them into our parameterized designs to improve performance and adaptability. Extensive experiments on a wide range of datasets and mainstream GNN models demonstrate the effectiveness of our design. Overall, GNNAdvisor provides users with a handy tool to accelerate GNNs on GPUs systematically and comprehensively.

In Chapter IV, I present MGG, a novel multi-GPU system design, and implementation to exploit the potential of leveraging GPU intra-kernel software pipeline for accelerating GNNs. MGG consists of GNN-tailored pipeline construction and GPU-aware pipeline mapping to facilitate workload balancing and operation overlapping, and an intelligent runtime design to dynamically improve the GNN runtime performance. Experiments show the advantages of MGG over state-of-the-art solutions and its generality towards other DL applications.

In Chapter V, I introduce TC-GNN, the first GNN acceleration framework on TCU of GPUs. We design a novel sparse graph translation technique to gracefully fit the sparse GNN workload on dense TCUs. Our TCU-tailored GPU kernel design maximizes the TCU performance gains for GNN computing through effective CUDA core and TCU collaboration and a set of memory/data flow optimizations. Our seamless integration with the PyTorch framework further facilitates end-to-end GNN computing with high programmability. Extensive experiments demonstrate the performance advantage of TC-GNN over the state-of-the-art frameworks. across diverse GNN models and datasets.

Furthermore, our TC-GNN design could also inspire potential TCU-like hardware features that can support (i) the dynamic shape of TCU input tiles and (ii) the dynamic structural sparsity of input tiles to yield higher performance benefits at the runtime. These proposed hardware features will help reduce unnecessary computation in a more fine-grained and precise manner.

In Chapter VI, I introduce DSXplore, the first optimized design to explore the DSCs on CNNs. Specifically, at the algorithm-level optimization, DSXplore incorporates a

novel sliding-channel convolution (SCC), featured with the input-channel overlapping to capture cross-channel information that can effectively improve the accuracy while reducing FLOPs and parameter size across a board range of CNNs on mainstream image classification datasets. At the implementation level, I reduce the atomic operation during the backward phase by leveraging the input-centric back-propagation design. Moreover, I fully integrated DSXplore with the Pytorch to improve programmability. Overall, our work paves a new way of exploring DSCs systematically and comprehensively by combining both algorithmic and implementation innovations.

In Chapter VII, I propose *3D-Receptive Field (3DRF)*, an interpretable and easy-to-compute metric to guide the search of CNN designs. To illustrate the usefulness of *3DRF*, We build an optimizer and improve the CNN structure at the stage and kernel level. The stage-level optimization targets at reducing the model structural redundancy by improving the kernel organization, while the kernel-level optimization improves the individual kernel design by reducing the number of parameters without compromising the model accuracy. Experiments show models generated by our optimizer achieve higher efficiency and accuracy compared with state-of-the-art CNNs.

8.2 Future Work

Looking forward, I will deepen and strengthen my DL system research in several aspects. Specifically, I plan to develop a DL verification system that can effectively certify the robustness of the neural network models for safety-critical applications (e.g., autonomous driving). I will also work on expanding the existing system abstraction for new hardware features and DL workloads, and promoting system-level failure resiliency. My long-term goal is to facilitate the *efficiency*, *scalability*, and *safety* of DL systems to effectively support diverse DL applications in the future.

Secure DL Systems Despite the stunning success in accelerating DL model execution on GPUs, the performance of verifying the DL model's robustness (i.e., whether DL models can still make the right predictions under noisy/adversarial inputs) largely lagged behind. DL model verification involves neuros' variation (bound) derivation and propagation to quantify input variation on output prediction. Such bound-centric computation is highly irregular and memory-intensive in comparison to standard DL computation with regular dense GEMM computation. I will build an efficient **DL model verification system** on accelerating hardware (e.g., GPUs). The key of our design is to design and implement verification-tailored primitives and computation graph transformation on GPUs with model adaptability. Moreover, our prior experience [1, 4, 8] in leveraging new AI-tailored hardware features (e.g., GPU Tensor Cores) in irregular and sparse GNN workloads will provide us with good examples for exploring their uncovered potentials such as high computing throughput and quantization, in verifying and securing DL models. I also seek to demonstrate our design/optimization effectiveness by collaborating with faculty with research interests in software engineering and programming language.

Automated Novel Hardware Features Adaption: With the increasing popularity of GPUs in AI applications, GPU hardware also advances simultaneously. Numerous new hardware features have been introduced since the initial version of the GPU and CUDA programming model, tailoring for different needs of computation specialization. For instance, Ray tracing/Tensorcore units have been introduced for accelerating the computer graphics and AI workloads, respectively. The threadblock clusters provide new opportunities to manage the GPU kernel computation in a more efficient way. Register bypassing enables faster data loading from global memory to shared memory.

Unfortunately, it is impractical to rewrite all prior kernel designs to enjoy performance benefits from these new features and demands arduous design and engineering efforts. I will build a **holistic compiler stack and runtime orchestration strategy**

to automate the matching between the new hardware features and the emerging ML workloads. The key is to build the new computation/memory abstraction for new hardware features meanwhile reconstructing DL workloads to match such new abstraction. I will also explore the potential and design space of collaboration between existing and new hardware features to promote the overall performance delivery. Our prior experience [4, 8] in retargeting dense GPU Tensor Cores for sparse GNN workloads provides a good start to achieving our goal. Our collaboration with NVIDIA and Pacific Northwest National Laboratory can offer easier access to the latest hardware and valuable insights for kernel optimization and compiler stack implementation.

Failure Resilient DL Systems With the growing demands of Large Language Models (LLMs) training and inference, the failure of runtime execution is becoming more likely to happen at the software and hardware level. For instance, one of the commonly used communication libraries, NCCL, is vulnerable to a diverse range of errors including failures due to the crash of NICs (Network interface for inter-machine communication). Such failures would put the entire training on hold and take engineers/ML practitioners hours or even days to recover from those failures. I envision a **failure-resilient DL system** that can directly heal itself with several mechanisms, including agile failure detection, intelligent root cause diagnostic, and remediation generation, through the innovation of DL framework, runtime system, and accelerating hardware. I will develop the framework and runtime system support for failure-resilient DL systems. The design space for remediation solution generation will also be explored systematically to match or even outperform the default one used in existing DL frameworks.

Bibliography

[1] B. Feng, *Yuke Wang, T. Geng, A. Li, and Y. D. E. Contribution), *Apnn-tc: Accelerating arbitrary precision neural networks on ampere gpu tensor cores*, in *The International Conference for High Performance Computing, Networking, Storage, and Analysis. (SC'21)*, 2021.

[2] Yuke Wang, B. Feng, and Y. Ding, *Dsxplore: Optimizing convolutional neural networks via sliding-channel convolutions*, in *IEEE International Parallel and Distributed Processing Symposium (IPDPS'21)*, 2021.

[3] Yuke Wang, B. Feng, G. Li, S. Li, L. Deng, Y. Xie, and Y. Ding, *Gnnadvisor: An efficient runtime system for gnn acceleration on gpus*, in *USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*, 2021.

[4] Yuke Wang, B. Feng, and Y. Ding, *Qgtc: Accelerating quantized gnn via gpu tensor core*, in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. (PPoPP'22)*, 2022.

[5] Yuke Wang, B. Feng, and Y. Ding, *Tc-gnn: Accelerating sparse graph neural network computation via dense tensor core on gpus*, *USENIX Annual Technical Conference (USENIX ATC'23)* (2023).

[6] Yuke Wang, B. Feng, X. Peng, and Y. Ding, *An efficient quantitative approach for optimizing convolutional neural networks*, in *ACM International Conference on Information and Knowledge Management. (CIKM'21)*, 2021.

[7] Yuke Wang, B. Feng, G. Li, L. Deng, Y. Xie, and Y. Ding., *Stpacc: A compiler-based framework for accelerating distance algorithms on cpu-fpga platforms.*, in *IEEE Transactions on Computer Aided Design of Integrated Circuits & Systems. (TCAD'21)*, 2021.

[8] Wang, Yuke, B. Feng, Z. Wang, G. Huang, and Y. Ding, *Tc-gnn: Bridging sparse gnn computation and dense tensor cores on gpus*, in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023.

[9] Yuke Wang, B. Feng, Z. Wang, T. Geng, K. Barker, A. Li, and Y. Ding, *Mgg: Accelerating graph neural networks with fine-grained intra-kernel*

communication-computation pipelining on multi-gpu platforms, in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*, 2023.

[10] T. N. Kipf and M. Welling, *Semi-supervised classification with graph convolutional networks*, *International Conference on Learning Representations (ICLR)* (2017).

[11] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, *Graph attention networks*, in *International Conference on Learning Representations (ICLR)*, 2018.

[12] W. Hamilton, Z. Ying, and J. Leskovec, *Inductive representation learning on large graphs*, in *Advances in neural information processing systems (NeurIPS)*, 2017.

[13] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, *How powerful are graph neural networks?*, in *International Conference on Learning Representations (ICLR)*, 2019.

[14] L. Ma, Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou, and Y. Dai, *Neugraph: parallel deep neural network computation on large graphs*, in *USENIX Annual Technical Conference (ATC'19)*.

[15] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, Z. Huang, Q. Guo, H. Zhang, H. Lin, J. Zhao, J. Li, A. J. Smola, and Z. Zhang, *Deep graph library: Towards efficient and scalable deep learning on graphs*, *ICLR Workshop on Representation Learning on Graphs and Manifolds* (2019).

[16] M. Fey and J. E. Lenssen, *Fast graph representation learning with PyTorch Geometric*, in *ICLR Workshop on Representation Learning on Graphs and Manifolds (ICLR)*, 2019.

[17] Z. Huang, A. Silva, and A. Singh, *A broader picture of random-walk based graph embedding*, in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pp. 685–695, 2021.

[18] B. Feng, Y. Wang, X. Li, S. Yang, X. Peng, and Y. Ding, *Sgquant: Squeezing the last bit on graph neural networks with specialized quantization*, in *IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*, 2020.

[19] S. A. Tailor, J. Fernandez-Marques, and N. D. Lane, *Degree-quant: Quantization-aware training for graph neural networks*, *International Conference on Learning Representations* (2021).

[20] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and

S. Chintala, *Pytorch: An imperative style, high-performance deep learning library*, in *Advances in Neural Information Processing Systems (NeurIPS)*. 2019.

[21] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, *Tensorflow: A system for large-scale machine learning*, in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, (Savannah, GA, USA), 2016.

[22] G. Karypis and V. Kumar, "MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0." `http://www.cs.umn.edu/~metis`, 2009.

[23] W.-L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C.-J. Hsieh, *Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks*, in *Proceedings of the 25th ACM International Conference on Knowledge Discovery & Data Mining*, 2019.

[24] D. Liben-Nowell and J. Kleinberg, *The link-prediction problem for social networks*, *Journal of the American society for information science and technology* **58** (2007), no. 7 1019–1031.

[25] A. Grover and J. Leskovec, *node2vec: Scalable feature learning for networks*, in *Proceedings of the 22nd ACM international conference on Knowledge discovery and data mining (SIGKDD)*, 2016.

[26] Z. Huang, A. Silva, and A. Singh, *Pole: Polarized embedding for signed networks*, 2022.

[27] NVIDIA, *Programming tensor cores in cuda 9*, 2017.

[28] A. Abdelfattah, S. Tomov, and J. Dongarra, *Fast batched matrix multiplication for small sizes using half-precision arithmetic on gpus*, in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019.

[29] A. Li and S. Su, *Accelerating binarized neural networks via bit-tensor-cores in turing gpus*, *IEEE Transactions on Parallel and Distributed Systems (TPDS)* (2020).

[30] A. Dakkak, C. Li, J. Xiong, I. Gelado, and W.-m. Hwu, *Accelerating reduction and scan using tensor core units*, in *Proceedings of the ACM International Conference on Supercomputing*.

[31] B. Feng, Y. Wang, G. Chen, W. Zhang, Y. Xie, and Y. Ding, *Egemm-tc: Accelerating scientific computing tensor cores with extended precision*, *ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)* (2021).

[32] Nvidia, "Dense linear algebra on gpus." `developer.nvidia.com/cublas`.

[33] Nvidia, "Cuda templates for linear algebra subroutines(cutlass)." `github.com/NVIDIA/cutlass.git`.

[34] Nvidia, "Warp matrix multiply-accumulate (wmma)."

[35] M. Bahri, G. Bahl, and S. Zafeiriou, *Binary graph neural networks*, *arXiv preprint arXiv:2012.15823* (2020).

[36] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, *Binarized neural networks*, in *Proceedings of the 30th international conference on neural information processing systems*, 2016.

[37] U. N. Raghavan, R. Albert, and S. Kumara, *Near linear time algorithm to detect community structures in large-scale networks*, *Physical review E* (2007).

[38] K. I. Karantasis, A. Lenharth, D. Nguyen, M. J. Garzaran, and K. Pingali, *Parallelization of reordering algorithms for bandwidth and wavefront reduction*, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.

[39] E. Cuthill and J. McKee, *Reducing the bandwidth of sparse symmetric matrices*, in *Proceedings of the 1969 24th National Conference*, 1969.

[40] M. Cowan, T. Moreau, T. Chen, J. Bornholt, and L. Ceze, *Automatic generation of high-performance quantized machine learning kernels*, in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pp. 305–316, 2020.

[41] R. Ying, J. You, C. Morris, X. Ren, W. L. Hamilton, and J. Leskovec, *Hierarchical graph representation learning with differentiable pooling*, in *The 32nd International Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

[42] K. Kersting, N. M. Kriege, C. Morris, P. Mutzel, and M. Neumann, *Benchmark data sets for graph kernels*, 2016.

[43] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, *Open graph benchmark: Datasets for machine learning on graphs*, *arXiv preprint arXiv:2005.00687* (2020).

[44] H. Zeng and V. Prasanna, *Graphact: Accelerating gcn training on cpu-fpga heterogeneous platforms*, in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2020.

[45] R. Kaspar and B. Horst, *Graph classification and clustering based on vector space embedding*. World Scientific, 2010.

[46] J. Gibert, E. Valveny, and H. Bunke, *Graph embedding in vector spaces by node attribute statistics*, *Pattern Recognition* (2012).

[47] A. G. Duran and M. Niepert, *Learning graph representations with embedding propagation*, in *Advances in neural information processing systems (NeurIPS)*, 2017.

[48] H. Chen, X. Li, and Z. Huang, *Link prediction approach to collaborative filtering*, in *Proceedings of the 5th ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL)*, IEEE, 2005.

[49] J. Kunegis and A. Lommatzsch, *Learning spectral graph transformations for link prediction*, in *Proceedings of the 26th Annual International Conference on Machine Learning (ICML)*, 2009.

[50] T. Tylenda, R. Angelova, and S. Bedathur, *Towards time-aware link prediction in evolving social networks*, in *Proceedings of the 3rd workshop on social network mining and analysis*, 2009.

[51] B. Perozzi, R. Al-Rfou, and S. Skiena, *Deepwalk: Online learning of social representations*, in *The 20th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2014.

[52] D. Luo, F. Nie, H. Huang, and C. H. Ding, *Cauchy graph embedding*, in *The 28th International Conference on Machine Learning (ICML)*, 2011.

[53] D. Luo, C. Ding, H. Huang, and T. Li, *Non-negative laplacian embedding*, in *Ninth IEEE International Conference on Data Mining (ICDM)*, 2009.

[54] D. Cheng, Y. Gong, X. Chang, W. Shi, A. Hauptmann, and N. Zheng, *Deep feature learning via structured graph laplacian embedding for person re-identification*, *Pattern Recognition* (2018).

[55] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, *Powergraph: Distributed graph-parallel computation on natural graphs*, in *The 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[56] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, *Gunrock: A high-performance graph processing library on the gpu*, in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2016.

[57] V. Balaji and B. Lucia, *When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs*, in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE.

[58] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, *Rabbit order: Just-in-time parallel reordering for fast graph analysis*, in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

[59] L. Page, S. Brin, R. Motwani, and T. Winograd, *The pagerank citation ranking: Bringing order to the web.*, tech. rep., 1999.

[60] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, *Cusha: vertex-centric graph processing on gpus*, in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing (HPDC)*, 2014.

[61] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, and O. Yakhnenko, *Translating embeddings for modeling multi-relational data*, in *Advances in Neural Information Processing Systems (NeurIPS)*, 2013.

[62] D. Duvenaud, D. Maclaurin, J. Aguilera-Iparraguirre, R. Gómez-Bombarelli, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams, *Convolutional networks on graphs for learning molecular fingerprints*, *arXiv preprint* (2015).

[63] A. H. Nodehi Sabet, J. Qiu, and Z. Zhao, *Tigr: Transforming irregular graphs for gpu-friendly graph processing*, in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[64] H. Liu and H. H. Huang, *Enterprise: breadth-first graph traversal on gpus*, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[65] H. Liu and H. H. Huang, *Simd-x: Programming and processing of graph algorithms on gpus*, in *USENIX Annual Technical Conference (ATC)*, 2019.

[66] A. Kyrola, G. Blelloch, and C. Guestrin, *Graphchi: Large-scale graph computation on just a pc*, in *The 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[67] A. Roy, I. Mihailovic, and W. Zwaenepoel, *X-stream: Edge-centric graph processing using streaming partitions*, in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[68] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et. al.*, *Tensorflow: A system for large-scale machine learning*, in *12th USENIX symposium on operating systems design and implementation (OSDI)*, 2016.

[69] M. Fey and J. E. Lenssen, *Pytorch extension library of optimized scatter operations*, 2019.

[70] Nvidia, "Cuda sparse matrix library (cusparse)." `developer.nvidia.com/cusparse`.

[71] A. Sala, H. Zheng, B. Y. Zhao, S. Gaito, and G. P. Rossi, *Brief announcement: Revisiting the power-law degree distribution for social graph analysis*, in *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, 2010.

[72] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, *Mizan: A system for dynamic load balancing in large-scale graph processing*, in *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, 2013.

[73] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin, *An experimental comparison of pregel-like graph processing systems*, *The VLDB Endowment* (2014).

[74] S. Fortunato, *Community detection in graphs*, *Physics reports* (2010).

[75] A. Lancichinetti, S. Fortunato, and F. Radicchi, *Benchmark graphs for testing community detection algorithms*, *Physical review E* (2008).

[76] M. E. Newman, *Spectral methods for community detection and graph partitioning*, *Physical Review E* (2013).

[77] B. Hendrickson and T. G. Kolda, *Graph partitioning models for parallel computing*, *Parallel computing* (2000).

[78] P. Boldi, M. Rosa, M. Santini, and S. Vigna, *Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks*, in *Proceedings of the 20th international conference on World wide web (WWW)*, 2011.

[79] Nvidia, "Tesla v100." `https://www.nvidia.com/en-us/data-center/v100/`.

[80] Nvidia, "Quardo p6000 gpu." `https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/productspage/quadro/quadro-desktop/quadro-pascal-p6000-data-sheet-us-nv\-704590-r1.pdf`.

[81] C. Yang, A. Buluç, and J. D. Owens, *Design principles for sparse matrix multiplication on the gpu*, in *European Conference on Parallel Processing*, 2018.

[82] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection." `http://snap.stanford.edu/data`, 2014.

[83] Intel, "Xeon sliver 4110." https://ark.intel.com/content/www/us/en/ark/products/123547/intel-xeon-silver-4110-processor-11m-cache\-2-10-ghz.html.

[84] Nvidia, "Dgx-1." https://www.nvidia.com/en-us/data-center/dgx-1/.

[85] Nvidia, "Profiling tools." docs.nvidia.com/cuda/profiler-users-guide/index.html.

[86] Nvidia, "Tesla p100." https://www.nvidia.com/en-us/data-center/tesla-p100/.

[87] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, *Hygcn: A gcn accelerator with hybrid architecture*, 2020.

[88] Y. Zhang, X. Yu, Z. Cui, S. Wu, Z. Wen, and L. Wang, *Every document owns its structure: Inductive text classification via graph neural networks*, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)* (2020).

[89] Nvidia, "Nvidia dgx a100." https://nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-dgx-a100-datasheet.pdf.

[90] Nvidia, "Dgx superpod." https://nvidia.com/en-us/data-center/dgx-superpod/.

[91] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken, *Improving the accuracy, scalability, and performance of graph neural networks with roc*, in *Proceedings of the3rd MLSys Conference, 2020*.

[92] S. W. Min, K. Wu, S. Huang, M. Hidayetoğlu, J. Xiong, E. Ebrahimi, D. Chen, and W.-m. Hwu, *Large graph convolutional network training with gpu-oriented data communication architecture*, *Proc. VLDB Endow.* (2021).

[93] S. Gandhi and A. P. Iyer, *P3: Distributed deep graph learning at scale*, in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[94] Nvidia, "Nvshmem communication library." https://developer.nvidia.com/nvshmem.

[95] M. Naumov, D. Mudigere, H. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy, *Deep learning recommendation model for personalization and recommendation systems*, *arXiv preprint arXiv:1906.00091* (2019).

[96] Nvidia, "Nvidia collective communication library (nccl)." https://developer.nvidia.com/nccl.

[97] T. Schroeder, "Peer-to-peer & unified virtual addressing." https://developer.download.nvidia.com/CUDA/training/cuda_webinars_GPUDirect_uva.pdf.

[98] Z. Cai, X. Yan, Y. Wu, K. Ma, J. Cheng, and F. Yu, *Dgcl: an efficient communication library for distributed gnn training*, in *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys)*, 2021.

[99] Z. Lin, C. Li, Y. Miao, Y. Liu, and Y. Xu, *Pagraph: Scaling gnn training on large graphs via computation-aware caching*, in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020.

[100] J. Yang, D. Tang, X. Song, L. Wang, Q. Yin, R. Chen, W. Yu, and J. Zhou, *Gnnlab: a factored system for sample-based gnn training over gpus*, in *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys)*, 2022.

[101] S. Polisetty, J. Liu, K. Falus, Y. R. Fung, S.-H. Lim, H. Guan, and M. Serafini, *Gsplit: Scaling graph neural network training on large graphs via split-parallelism*, *arXiv preprint arXiv:2303.13775* (2023).

[102] J. Chen, T. Ma, and C. Xiao, *FastGCN: Fast learning with graph convolutional networks via importance sampling*, in *International Conference on Learning Representations (ICLR)*, 2018.

[103] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, *Legion: Expressing locality and independence with logical regions*, in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[104] M. Alle, A. Morvan, and S. Derrien, *Runtime dependency analysis for loop pipelining in high-level synthesis*, in *Proceedings of the 50th Annual Design Automation Conference (DAC)*, 2013.

[105] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, *Optimization principles and application performance evaluation of a multithreaded gpu using cuda*, in *The 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP)*, 2008.

[106] NVIDIA, "Unified memory for cuda beginners." https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.

[107] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, *Pipedream: generalized pipeline parallelism for dnn training*, in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[108] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia, *Memory-efficient pipeline-parallel dnn training*, in *International Conference on Machine Learning (ICML)*, 2021.

[109] K. Andreev and H. Räcke, *Balanced graph partitioning*, in *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures (SPAA)*, 2004.

[110] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker, *Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect*, *IEEE Transactions on Parallel and Distributed Systems (TPDS)* (2019).

[111] wikipedia, "Nvidia gpu micro-architecture." `https://en.wikipedia.org/wiki/CUDA`.

[112] T. A. Davis and Y. Hu, *The university of florida sparse matrix collection*, *ACM Transactions on Mathematical Software (TOMS)* (2011).

[113] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, *Open graph benchmark: Datasets for machine learning on graphs*, *Advances in neural information processing systems (NeurIPS)* **33** (2020).

[114] S. W. Min, K. Wu, S. Huang, M. Hidayetoğlu, J. Xiong, E. Ebrahimi, D. Chen, and W.-m. Hwu, *Pytorch-direct: Enabling gpu centric data access for very large graph neural network training with irregular accesses*, *arXiv preprint arXiv:2101.07956* (2021).

[115] H. Kim, J. Sim, P. Gera, R. Hadidi, and H. Kim, *Batch-aware unified memory management in gpus for irregular workloads*, in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[116] AMD, "Rocm openshmem." `https://github.com/ROCm-Developer-Tools/ROC_SHMEM`.

[117] S. Zhang, L. Yao, A. Sun, and Y. Tay, *Deep learning based recommender system: A survey and new perspectives*, *ACM Computing Surveys (CSUR)* (2019).

[118] Z. Wang, Y. Wang, B. Feng, D. Mudigere, B. Muthiah, and Y. Ding, *El-rec: efficient large-scale recommendation model training via tensor-train embedding table*, in *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2022.

[119] U. Gupta, C. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. M. Hazelwood, M. Hempstead, B. Jia, H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang, *The architectural implications of*

facebook's dnn-based personalized recommendation, in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[120] Criteo, "Criteo display ad challenge." `https://kaggle.com/c/criteodisplay-ad-challenge`.

[121] K. K. Thekumparampil, C. Wang, S. Oh, and L.-J. Li, *Attention-based graph neural network for semi-supervised learning*, .

[122] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et. al.*, *In-datacenter performance analysis of a tensor processing unit*, in *Proceedings of the 44th annual international symposium on computer architecture (ISCA)*, 2017.

[123] AMD, "All-new matrix core technology for hpc and ai." `https://amd.com/en/technologies/cdna`.

[124] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2016.

[125] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, *Bert: Pre-training of deep bidirectional transformers for language understanding*, *arXiv preprint arXiv:1810.04805* (2018).

[126] Nvidia, "Accelerating matrix multiplication with block sparse format and nvidia tensor cores." `https://developer.nvidia.com/blog/accelerating-matrix-multiplication-with\-block-sparse-format-and-nvidia-tensor-cores/`.

[127] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1999.

[128] W. Bosma, J. Cannon, and C. Playoust, *The Magma algebra system. I. The user language*, *J. Symbolic Comput.* (1997). Computational algebra and number theory (London, 1993).

[129] *Intel Math Kernel Library. Reference Manual*. Intel Corporation. Santa Clara, USA.

[130] G. Huang, G. Dai, Y. Wang, and H. Yang, *Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks*, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

[131] S. E. Kurt, A. Sukumaran-Rajam, F. Rastello, and P. Sadayyapan, *Efficient tiled sparse matrix multiplication through matrix signatures*, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

[132] M. Zhang and Y. Chen, *Link prediction based on graph neural networks*, *Advances in neural information processing systems (NeurIPS)* **31** (2018).

[133] NVIDIA, "Improved tensor core operations." `https://docs.nvidia.com/cuda/ampere-tuning-guide/index.html#tensor-operations`.

[134] O. Zachariadis, N. Satpute, J. Gómez-Luna, and J. Olivares, *Accelerating sparse matrix–matrix multiplication with gpu tensor cores*, *Computers & Electrical Engineering* (2020).

[135] P. Tillet, H. T. Kung, and D. Cox, *Triton: An intermediate language and compiler for tiled neural network computations*, in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*, 2019.

[136] Nvidia, "Nvidia blocked-sparse api." `https://docs.nvidia.com/cuda/cusparse/index.html#cusparse-generic-function-spmm)`.

[137] Z. Chen, M. Yan, M. Zhu, L. Deng, G. Li, S. Li, and Y. Xie, *fusegnn: accelerating graph convolutional neural network training on gpgpu*, in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020.

[138] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken, *Improving the accuracy, scalability, and performance of graph neural networks with roc*, *Proceedings of Machine Learning and Systems (MLSys)* (2020).

[139] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, *Rabbit order: Just-in-time parallel reordering for fast graph analysis*, in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

[140] E. Cuthill and J. McKee, *Reducing the bandwidth of sparse symmetric matrices*, in *Proceedings of the 1969 24th national conference*, 1969.

[141] S. Gandhi and A. P. Iyer, *P3: Distributed deep graph learning at scale*, in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[142] Y. Wang, B. Feng, Z. Wang, T. Geng, A. Li, K. Barker, and Y. Ding, *Mgg: Accelerating graph neural networks with fine-grained intra-kernel communication-computation pipelining on multi-gpu platforms*, in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2023.

[143] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*, *arXiv* (Apr., 2017).

[144] F. Chollet, *Xception: Deep learning with depthwise separable convolutions*, in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2017.

[145] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, *Inception-v4, inception-resnet and the impact of residual connections on learning*, in *Thirty-First AAAI Conference on Artificial Intelligence (AAAI)*, 2017.

[146] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, *Aggregated residual transformations for deep neural networks*, in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[147] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, *ICLR* (2015).

[148] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, *Mobilenetv2: Inverted residuals and linear bottlenecks*, in *IEEE Computer Vision and Pattern Recognition (CVPR)*, 2018.

[149] A. Krizhevsky, I. Sutskever, and G. E. Hinton, *Imagenet classification with deep convolutional neural networks*, in *Advances in Neural Information Processing Systems (NeurIPS)* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.). 2012.

[150] L. Sifre and S. Mallat, *Rigid-motion scattering for image classification*. PhD thesis, Citeseer, 2014.

[151] X. Zhang, X. Zhou, M. Lin, and J. Sun, *Shufflenet: An extremely efficient convolutional neural network for mobile devices*, in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[152] B. Wu, A. Wan, X. Yue, P. Jin, S. Zhao, N. Golmant, A. Gholaminejad, J. Gonzalez, and K. Keutzer, *Shift: A zero flop, zero parameter alternative to spatial convolutions*, in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018.

[153] W. Niu, X. Ma, S. Lin, S. Wang, X. Qian, X. Lin, Y. Wang, and B. Ren, *Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning*, in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[154] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, *Scalpel: Customizing dnn pruning to the underlying hardware parallelism*, in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[155] T. Zhang, S. Ye, K. Zhang, J. Tang, W. Wen, M. Fardad, and Y. Wang, *A systematic dnn weight pruning framework using alternating direction method of multipliers*, in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018.

[156] X. Ma, F.-M. Guo, W. Niu, X. Lin, J. Tang, K. Ma, B. Ren, and Y. Wang, *Pconv: The missing but desirable sparsity in dnn weight pruning for real-time execution on mobile devices.*, .

[157] N. Liu, X. Ma, Z. Xu, Y. Wang, J. Tang, and J. Ye, *Autocompress: An automatic dnn structured pruning framework for ultra-high compression rates*, in *AAAI Conference on Artificial Intelligence*, 2020.

[158] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et. al.*, *Tensorflow: A system for large-scale machine learning*, in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[159] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, *cudnn: Efficient primitives for deep learning*, *arXiv preprint arXiv:1410.0759* (2014).

[160] A. Krizhevsky and G. Hinton, *Learning multiple layers of features from tiny images*, tech. rep., Citeseer, 2009.

[161] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, *Imagenet: A large-scale hierarchical image database*, in *2009 IEEE conference on computer vision and pattern recognition (CVPR)*, 2009.

[162] R. Girshick, J. Donahue, T. Darrell, and J. Malik, *Rich feature hierarchies for accurate object detection and semantic segmentation*, in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, pp. 580–587, 2014.

[163] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, *Large-scale video classification with convolutional neural networks*, in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June, 2014.

[164] N. Wang and D.-Y. Yeung, *Learning a deep compact image representation for visual tracking*, in *Advances in Neural Information Processing Systems (NeurIPS)*

(C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, eds.), pp. 809–817. 2013.

[165] J. Long, E. Shelhamer, and T. Darrell, *Fully convolutional networks for semantic segmentation*, in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, pp. 3431–3440, 2015.

[166] A. Toshev and C. Szegedy, *Deeppose: Human pose estimation via deep neural networks*, in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, pp. 1653–1660, 2014.

[167] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, *Shufflenet v2: Practical guidelines for efficient cnn architecture design*, in *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 116–131, 2018.

[168] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. V. Le, and A. Kurakin, *Large-scale evolution of image classifiers*, in *Proceedings of the 34th International Conference on Machine Learning (ICML)*, pp. 2902–2911, 2017.

[169] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, *Learning efficient convolutional networks through network slimming*, in *The IEEE International Conference on Computer Vision (ICCV)*, Oct, 2017.

[170] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, *Progressive neural architecture search*, in *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 19–34, 2018.

[171] B. Baker, O. Gupta, N. Naik, and R. Raskar, *Designing neural network architectures using reinforcement learning*, *ICLR 2017* (2016).

[172] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, *Learning transferable architectures for scalable image recognition*, in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, pp. 8697–8710, 2018.

[173] D.-Q. Zhang, *clcnet: Improving the efficiency of convolutional neural network using channel local convolutions*, in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 7912–7919, 2018.

[174] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, *Regularized evolution for image classifier architecture search*, *AAAI 2019* (2018).

[175] T. Elsken, J.-H. Metzen, and F. Hutter, *Simple and efficient architecture search for convolutional neural networks*, *arXiv preprint arXiv:1711.04528* (2017).

[176] T. Elsken, J. H. Metzen, and F. Hutter, *Efficient multi-objective neural architecture search via lamarckian evolution*, *ICLR2019* (2018).

[177] Y. Zhou and G. Diamos, *Neural architect: A multi-objective neural architecture search with performance prediction*, in *Proc. Conf. SysML*, 2018.

[178] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean, *Efficient neural architecture search via parameters sharing*, in *International Conference on Machine Learning*, pp. 4095–4104, PMLR, 2018.

[179] X. Jin, J. Wang, J. Slocum, M.-H. Yang, S. Dai, S. Yan, and J. Feng, *Rc-darts: Resource constrained differentiable architecture search*, *arXiv preprint arXiv:1912.12814* (2019).

[180] H. Liu, K. Simonyan, and Y. Yang, *Darts: Differentiable architecture search*, *arXiv preprint arXiv:1806.09055* (2018).

[181] J. Yu, P. Jin, H. Liu, G. Bender, P.-J. Kindermans, M. Tan, T. Huang, X. Song, and Q. Le, *Scaling up neural architecture search with big single-stage models*, *arXiv preprint* (2019).

[182] J. Mei, Y. Li, X. Lian, X. Jin, L. Yang, A. Yuille, and J. Yang, *Atomnas: Fine-grained end-to-end neural architecture search*, in *International Conference on Learning Representations (ICLR)*, 2020.

[183] C. S. Sherrington, *Observations on the scratch-reflex in the spinal dog*, *The Journal of physiology* **34** (1906), no. 1-2 1–50.

[184] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, *Pruning filters for efficient convnets*, *ICLR 2017* (2016).