University of California
Santa Barbara

Kernel-Centric Optimizations for Deep Neural Networks on GPGPUs

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

Doctor of Philosophy
in
Electrical and Computer Engineering

by

Zhaodong Chen

Committee in charge:

Professor Yuan Xie, Co-Chair
Professor Zheng Zhang, Co-Chair
Professor Timothy Sherwood
Professor Yufei Ding
Professor Yao Qin

March 2024

The Dissertation of Zhaodong Chen is approved.

Professor Timothy Sherwood

Professor Yufei Ding

Professor Yao Qin

Professor Zheng Zhang, Committee Co-Chair

Professor Yuan Xie, Committee Co-Chair

March 2024

Kernel-Centric Optimizations for Deep Neural Networks on GPGPUs

by

Zhaodong Chen

Acknowledgements

Pursuing a Ph.D. degree is never an easy task. Throughout my five-year journey as a Ph.D. student, I have experienced stressful moments - from sleepless nights striving to meet deadlines and anxiety about rebuttals, to moments of frustration upon receiving negative comments from reviewers. There were also moments of joy, such as the excitement of paper acceptances, weekend hotpot parties, and the unforgettable spring break trip to Alaska. I have been incredibly fortunate to be surrounded by my advisors, mentors, colleagues, lab mates, family, and friends during all these moments.

I have always been grateful to have Prof. Yuan Xie as my advisor. His insights have consistently guided me over the past five years and will continue to do so in my future career. He has given me the freedom to devote myself to research topics that interest me while offering unwavering respect, support, and all the necessary resources.

I also want to express my gratitude to Professor Yufei Ding for her invaluable advice and support regarding my research. She has always encouraged me to delve into deeper and more impactful topics, offering detailed and constructive suggestions from paper writing to rebuttal responses.

I also thank Professor Zhang, my co-chair, for his support of both my research and life over the last two quarters. Additionally, I appreciate my committee members, Professor Timothy Sherwood, and Professor Yao Qin, for their insightful input and support.

I want to extend my special thanks to Prof. Guoqi Li, my undergraduate supervisor, who has always had faith in me and provided unwavering support.

I consider myself fortunate to have been allowed to join NVIDIA's CUTLASS group as an intern for the past two summers, and I am sincerely grateful for the chance to return as a full-time employee. My manager, Alan Kaatz, has consistently provided encouragement and support during our weekly meetings. Additionally, my mentors, Dr.

Haicheng Wu, Dr. Andrew Kerr, Dr. Jack Kosaian, Richard Cai, as well as my colleague Dr. Yujia Zhai, have offered invaluable support and assistance throughout my internship.

I appreciate the opportunity to work with the hardworking, intelligent, and warm-hearted lab mates from SEAL Lab, PICASSO Lab, and Professor Zhang's Lab. Dr. Lei Deng helped me with my first first-author paper and continued to offer invaluable support thereafter. Memories of our daily gathering in Lei's apartment, where a few lab mates and I would come together to prepare dinner, will always hold a special place in my heart. Dr. Maohua Zhu is my mentor in CUDA programming and GPU microarchitecture, guiding me with expertise and patience. Dr. Ling Liang has been kindly driving me to the grocery stores during the pandemic, and he is also my indispensable research collaborator. I have also collaborated with Dr. Zheng Qu and Dr. Liu Liu on deep learning acceleration topics, resulting in the co-authorship of several remarkable papers, which could not haven been done without their dedication, intelligence, and talent. Furthermore, I extend my gratitude to Prof. Xing Hu, Prof. Mingyu Yan, Prof. Fengbin Tu, Prof. Jiayi Huang, Dr. Jilan Lin, Dr. Bangtan Wang, Prof. Nan Wu, Dr. Tianqi Tang, Prof. Gushu Li, Dr. Peng Gu, Dr. Abanti Basak, Dr. Shuangchen Li, Dr. Wenqin Huangfu, Dr. Xinfeng Xie, Dr. Anbang Wu, Dr. Boyuan Feng, Yuke Wang, Zheng Wang, Zhengyang Wang, Siqi Li, Alvin Liu, Hao Li, Guyue Huang. It has been a great time working with all my lab mates.

I am deeply grateful to my friends, whose unwavering encouragement, understanding, and companionship have been invaluable throughout this journey. I extend special thanks to Jianyu Xu, my comrade since our undergraduate days at Tsinghua University. We came to UCSB together, and he is always been the person I can pour out my heart.

I would like to express my deepest gratitude to my family. My parents have always been my strongest backing, giving me the confidence and courage I needed at every turning point along this journey. My girlfriend, Carina Quan, has enlightened my life since the day we met. She has stood by my side through both storms and sunshine, offering unwavering support and love.

Curriculum Vitæ
Zhaodong Chen

Education

2024	Ph.D. in Electrical and Computer Engineering (Expected), University of California, Santa Barbara.
2021	M.S. in Electrical and Computer Engineering, University of California, Santa Barbara.
2019	B.E. in Department of Precision Instrument, Tsinghua University

Publications

[1] Wenzhao Zheng, **Zhaodong Chen**, Jiwen Lu, and Jie Zhou. ”Hardness-aware deep metric learning.” In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 72-81. 2019.

[2] Liu Liu, Lei Deng, **Zhaodong Chen**, Yuke Wang, Shuangchen Li, Jingwei Zhang, Yihua Yang, Zhenyu Gu, Yufei Ding, and Yuan Xie. ”Boosting deep neural network efficiency with dual-module inference.” In International Conference on Machine Learning, pp. 6205-6215. PMLR, 2020.

[3] Mingyu Yan, **Zhaodong Che**n, Lei Deng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. ”Characterizing and understanding GCNs on GPU.” IEEE Computer Architecture Letters 19, no. 1 (2020): 22-25.

[4] **Zhaodong Chen**, Lei Deng, Guoqi Li, Jiawei Sun, Xing Hu, Ling Liang, Yufei Ding, and Yuan Xie. ”Effective and efficient batch normalization using a few uncorrelated data for statistics estimation.” IEEE Transactions on Neural Networks and Learning Systems 32, no. 1 (2020): 348-362.

[5] **Zhaodong Chen**, Lei Deng, Bangyan Wang, Guoqi Li, and Yuan Xie. ”A comprehensive and modularized statistical framework for gradient norm equality in deep neural networks.” IEEE Transactions on Pattern Analysis and Machine Intelligence 44, no. 1 (2020): 13-31.

[6] **Zhaodong Chen**, Mingyu Yan, Maohua Zhu, Lei Deng, Guoqi Li, Shuangchen Li, and Yuan Xie. ”fuseGNN: Accelerating graph convolutional neural network training on GPGPU.” In Proceedings of the 39th International Conference on Computer-Aided Design, pp. 1-9. 2020.

[7] **Zhaodong Chen**, Zheng Qu, Liu Liu, Yufei Ding, and Yuan Xie. ”Efficient tensor core-based gpu kernels for structured sparsity under reduced precision.” In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1-14. 2021.

[8] Ling Liang, Zheng Qu, **Zhaodong Chen**, Fengbin Tu, Yujie Wu, Lei Deng, Guoqi Li, Peng Li, and Yuan Xie. ”H2learn: High-efficiency learning accelerator for high-accuracy

spiking neural networks.” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 41, no. 11 (2021): 4782-4796.

[9] Boyuan Feng, Tianqi Tang, Yuke Wang, **Zhaodong Chen**, Zheng Wang, Shu Yang, Yuan Xie, and Yufei Ding. ”Faith: An Efficient Framework for Transformer Verification on GPUs.” In 2022 USENIX Annual Technical Conference (USENIX ATC 22), pp. 167-182. 2022.

[10] Zheng Qu, Liu Liu, Fengbin Tu, **Zhaodong Chen**, Yufei Ding, and Yuan Xie. ”Dota: detect and omit weak attentions for scalable transformer acceleration.” In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 14-26. 2022.

[11] **Zhaodong Chen**, Zheng Qu, Yuying Quan, Liu Liu, Yufei Ding, and Yuan Xie. ”Dynamic n: M fine-grained structured sparse attention mechanism.” In Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, pp. 369-379. 2023.

[12] Ling Liang, **Zhaodong Chen**, Lei Deng, Fengbin Tu, Guoqi Li, and Yuan Xie. ”Accelerating spatiotemporal supervised training of large-scale spiking neural networks on gpu.” In 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 658-663. IEEE, 2022.

[13] Liu Liu, Zheng Qu, **Zhaodong Chen**, Fengbin Tu, Yufei Ding, and Yuan Xie. ”Dynamic sparse attention for scalable transformer acceleration.” IEEE Transactions on Computers 71, no. 12 (2022): 3165-3178.

Abstract

Kernel-Centric Optimizations for Deep Neural Networks on GPGPUs

by

Zhaodong Chen

Deep learning has achieved remarkable success across various domains, ranging from computer vision to healthcare. General-Purpose Graphics Processing Unit (GPGPU) is one of the major driving forces behind this revolution. GPGPUs offer massive parallel computational power, enabling the training and deployment of large-scale neural networks within practical time and resource constraints. Their programmability also enables adaptability to emerging network architectures.

However, entering the post-Moore's Law era, the scaling of computational power offered by GPGPUs struggles to meet the demands of novel neural networks. On the other hand, existing GPGPUs face under-utilization challenges despite the computation power shortage.

This dissertation addresses the computation power shortage by improving the utilization of GPGPUs when running deep learning workloads. It presents a kernel-centric optimization approach with a focus on mapping neural networks to a more efficient set of kernels (parallel functions executed on GPGPUs) that ensures better utilization. This involves optimizations from multiple levels: algorithm level aiming to leverage more hardware-friendly formulations, operator level to harness on-chip high bandwidth on GPGPUs, and kernel implementation level that maximizes the utilization of computational resources.

Contents

Chapter 1

Introduction

Deep learning with Deep Neural Networks (DNNs) has witnessed ground-breaking success across various domains, revolutionizing fields such as computer vision, natural language processing, speech recognition, healthcare, finance, and autonomous vehicles [1, 2, 3, 4, 5, 6].

The General-Purpose Graphics Processing Unit (GPGPU) is one of the major driving forces behind this progress [7]. It provides immense parallel processing power, enabling the training and deployment of neural networks with billions of parameters within reasonable time frames and resources. The programmability of GPGPU facilitates support for novel operators introduced in evolving neural network architectures, executed in parallel across multiple threads through functions known as ”kernels” [8].

Entering the era of large language models and post-Moore’s Law, the scaling of computational power provided by GPGPUs is struggling to keep pace with the rocketing requirements of novel deep neural networks [9]. Despite the computation shortage, existing GPGPUs suffer from low utilization of deep learning workloads. Jeon et al., 2019 [10] reported that the average hardware utilization of GPUs in use for training jobs is only around 52%. Additionally, Xiao et al., 2020 [11] found that in GPU clusters, only 10% of the GPUs achieve higher than 80% GPU utilization concerning the usage of computation units. A recent study on Meta's data centers in 2022 [12] also reveals a vast majority of deep learning experimentation utilizes GPGPUs at only 30-50%.

1.1 Characterize the utilization of GPGPU

This dissertation considers a broader sense of utilization, encompassing the following dimensions:

- Algorithm-level Utilization. The proportion of effective computation that contributes to the output. Low algorithm-level utilization may result from expending computation on outcomes close to zero or from overhead incurred by auxiliary modules in the neural network architecture.

- Device-level Utilization. The percentage of time during which at least one kernel is executing on the GPU. The primary cause of low device-level utilization is the overhead associated with kernel launching under single-GPGPU settings and communication delays under multi-GPGPU settings.

- Streaming Multiprocessor (SM)-level Utilization. The percentage of SM cores that have at least one active thread block. Insufficient parallelism leading to a shortage of thread blocks typically leads to low SM-level utilization.

- Execution Unit (EU)-level Utilization. The percentage of computation throughput and memory bandwidth utilized. Suboptimal kernel implementations that cause pipeline stalls usually lead to underutilization of both computation and memory bandwidth. Additionally, insufficient data reuse can cause memory bound that limits computation utilization.

1.2 Kernel-Centric Optimization

This dissertation aims to improve the above dimensions of utilization through kernel-centric optimization. Motivated by the fact that the utilization of programmable accelerators is majorly determined by the kernels running on them, the kernel-centric optimization constructs a mapping from the neural network to a set of kernels that better utilize the computational resources of the accelerators.

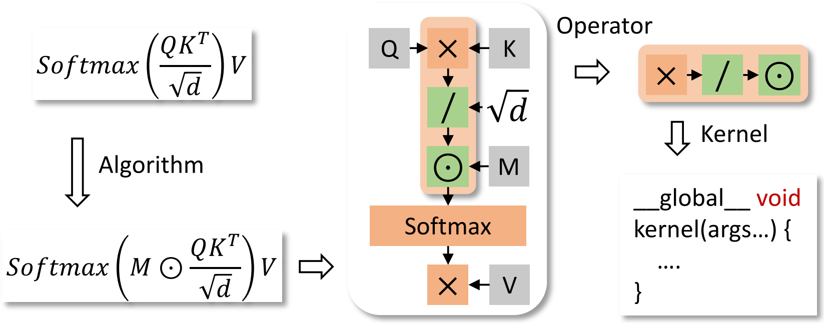

$Softmax\left(\frac{QK^T}{\sqrt{d}}\right)V$
Algorithm
$Softmax\left(M \odot \frac{QK^T}{\sqrt{d}}\right)V$
Q
K
$\sqrt{d}$
M
Softmax
V
Operator
Kernel
__global__ void
kernel(args…) {
....
}

Figure 1.1: Illustration of the kernel-centric optimization.

As illustrated in Figure 1.1, this dissertation demonstrates that the mapping can be constructed by considering optimizations on three hierarchical levels: algorithm, operator, and kernel.

Algorithm-level Optimization. The algorithm-level optimizations aim to find the representation of the original neural networks that have higher algorithm-level utilization. This is usually achieved by leveraging mathematical equivalences or the resilience to sparsity and low precision, and selecting only hardware-friendly operators. For example, the $M\odot$ in Figure 1.1 represents a sparse mask applied to leverage the sparsity resilience of the softmax function.

Besides algorithm-level utilization, it also opens up new optimization opportunities on lower levels of utilization. For instance, an algorithm composed of operators that are easy to parallelize could lead to higher SM-level utilization.

Operator-level Optimization. Representing the algorithm as a data-flow graph, an operator is defined by a subgraph/partition satisfying two conditions: 1) all internal edges can be realized through on-chip faster memories (shared memory and registers) in GPGPUs; 2) all nodes in the subgraph can be implemented under the same parallelism.

Operator-level optimization involves identifying the partition of the data-flow graph that improves the device and EU-level utilization. For example, the $\times$, $/$, and $\odot$ in Figure 1.1 are partitioned into the same operator. The kernel launching overhead regarding the latter two operators is eliminated which improves the device-level utilization. The intermediate data between these three operators are reused through the registers, which alleviates memory-bound constraints, thus enhancing computation utilization.

Kernel-level Optimization. Kernel-level optimization involves creating the optimal implementation of the kernel corresponding to each operator, as illustrated in Figure 1.1. Optimizations at the kernel level can effectively improve SM and EU-level utilization. The former can be achieved by parallelizing the operator across a sufficient number of thread blocks, while the latter involves leveraging data locality to cache data for reuse, thereby improving computation utilization. Additionally, optimizations that leverage accelerator-like features (e.g. Tensor Cores) or alleviate pipeline stalls can also be incorporated.

While optimizations on individual levels can effectively improve the utilization of GPGPUs on deep learning workloads, the thesis explores the codesign between optimizations at different levels both manually and with compiler techniques.

1.3 Manual Kernel-Centric Optimization

The kernel-centric optimization can be manually performed for specific neural network models and scenarios. The manual algorithm-level optimization typically involves simplifying the mathematical expression of the neural networks, identifying sparsity and low-precision resilience, and designing sparse patterns based on domain knowledge. Regarding operator-level optimization, the data-flow graph is manually examined to identify operators that can be partitioned together. The kernel-level optimization entails the manual design and optimization of efficient kernels in CUDA C++. This dissertation presents three case studies on manual kernel-centric optimizations as follows.

1.3.1 Low-precision static Sparsity with Tensor Core

Low precision and sparsity are two major techniques for improving algorithm-level utilization when deploying neural networks on GPGPUs. However, using them simultaneously can result in low EU-level utilization, leading to inferior performance compared with dense operators under low precision alone. Chapter 3 presents VecSparse to address this issue on two most important sparse operators, sparse matrix-matrix multiplication (SpMM) and sampled dense-dense matrix multiplication (SDDMM), under half precision. It improves the low EU-level utilization through codesign between algorithm and kernel-level optimizations.

At the algorithm level, a novel GPGPU-friendly structured sparse pattern, column vector sparse encoding, is introduced. This pattern offers higher data reuse while preserving model accuracy better than existing approaches, providing opportunities to alleviate low computation utilization caused by the memory bound without compromising accuracy.

At the kernel level, dedicated kernel designs under the proposed sparsity pattern are introduced. These designs innovatively map the sparse computations to Volta Tensor Cores at sub-warp (Octet) granularity, addressing numerous pipeline stall reasons and significantly improving EU-level utilization.

1.3.2 Dynamic Sparsity with Sparse Tensor Core

While it has long been recognized that sparsity exists in the attention weight matrices of the attention mechanism and could potentially enhance its algorithm-level utilization, few approaches could effectively leverage this opportunity. This is because the dynamic (input-dependent) and fine-grained nature of this sparsity is unfriendly to GPGPUs and typically leads to low algorithm and EU-level utilization. Chapter 4 presents a novel sparse attention mechanism, Dfss, that effectively accelerates the attention mechanism with sparsity and achieves high utilization through co-design between algorithm and EU-level optimizations.

At the algorithm level, Dfss carefully selects the position to introduce sparsity to maximize the benefits while avoiding the introduction of auxiliary or GPU-unfriendly operators. This ensures high algorithm-level utilization. It also innovatively leverages the N:M fine-grained structured sparsity, originally designed for static weight pruning, at this dynamic scenario. This pattern is proven to be more GPGPU-friendly than patterns used by existing approaches, despite high pruning overhead caused by low EU-level utilization.

The high pruning overhead of N:M sparsity is addressed by kernel-level optimization. A novel SDDMM kernel, the first of its kind, is introduced that prunes and encodes the output of a matrix multiplication into N:M sparsity without overhead. The encoded sparse matrix can be easily decoded by succeeding SpMM with sparse tensor core.

1.3.3 Efficient Graph Neural Network Training

The graph neural network (GNN) is an emerging type of neural network model designed to represent data with graph topology, which has significant applications in social networks and recommendation systems. However, the inherent sparsity in GNNs' adjacency matrix results in low utilization when deploying them on GPGPUs. Chapter 5 presents fuseGNN, a framework that combines optimizations at the algorithm, operator, and kernel levels to enhance the efficiency of GNN training on GPGPUs.

At the algorithm level, fuseGNN introduces the novel dual aggregation strategy to select the more efficient abstraction of the aggregation stage based on the average degree. This improves the algorithm-level utilization at the low average degree by avoiding the graph format transformation overhead, while improving the EU-level utilization at the high average degree through on-chip reduction.

At the operator level, fuseGNN identifies operators that can potentially be composed into a single kernel. This improves the device-level utilization as the kernel launching overhead is reduced. It also opens up opportunities to improve the EU-level utilization as intermediate results within each operator can be transferred through on-chip faster memory to relieve the memory bottleneck.

At the kernel level, an efficient fused kernel of each operator is developed that achieves higher EU-level utilization. Particularly, for the aggregation phase, fused kernels are developed for the two aggregation strategies presented in the algorithm-level optimization.

1.4 Compiler-based Kernel-Centric Optimization

The previous case studies present how kernel-centric optimization improves the utilization of deep learning workload on GPGPUs. However, these techniques require high engineering effort and are model-specific. Chapter 6 demonstrates the application of kernel-centric optimization in the realm of compilers. It introduces the Epilogue Visitor Tree (EVT) compiler, which automates kernel-centric optimization across a wide range of NN models during training.

At the algorithm level, EVT automatically identifies more efficient mathematical equivalents for neural network models through a series of transformations on the dataflow graph. This improves the algorithm-level utilization and reveals hidden optimization opportunities across other levels.

At the operator level, EVT integrates a novel integer linear programming (ILP)-based partitioner that efficiently solves the optimal and feasible partitions within complex joint forward-backward graphs.

At the kernel level, the EVT kernel-level compiler enables the construction of fused kernels with high-performance mainloops crafted by experts and the flexible compiler-generated epilogues. Utilizing handcrafted mainloops ensures state-of-the-art EU-level utilization by automatically incorporating dedicated optimizations, while integrating the compiler-generated epilogue enhances EU-level utilization by eliminating memory accesses caused by intermediate results within each partition.

1.5 Organization

This dissertation is organized as follows. Chapter 2 introduces the preliminaries and related work. Chapter 3, 4, 5 covers the manual kernel-centric optimizations, Chapter 6 covers the compiler-based kernel-centric optimizations. Chapter 7 concludes the thesis and discusses future research. The dissertation comprises work published elsewhere in conference papers:

- Chapter 3: Zhaodong Chen, Zheng Qu, Liu Liu, Yufei Ding, and Yuan Xie. ”Efficient tensor core-based gpu kernels for structured sparsity under reduced precision.” In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1-14. 2021.

- Chapter 4: Zhaodong Chen, Zheng Qu, Yuying Quan, Liu Liu, Yufei Ding, and

Yuan Xie. "Dynamic n: M fine-grained structured sparse attention mechanism." In Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, pp. 369-379. 2023.

- Chapter 5: Zhaodong Chen, Mingyu Yan, Maohua Zhu, Lei Deng, Guoqi Li, Shuangchen Li, and Yuan Xie. "fuseGNN: Accelerating graph convolutional neural network training on GPGPU." In Proceedings of the 39th International Conference on Computer-Aided Design, pp. 1-9. 2020.

- Chapter 6: Zhaodong Chen, Andrew Kerr, Richard Cai, Jack Kosaian, Haicheng Wu, Yufei Ding, and Yuan Xie. "EVT: Accelerating Deep Learning Training with Epilogue Visitor Tree.". To be appeared in Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2024.

Chapter 2

Background and Related Work

This chapter summarizes the background and related work of this dissertation. Firstly, Section 2.1 summarizes the background on transformers [13] and graph neural networks [14], two emerging neural network architectures accelerated in this dissertation. Section 2.2 introduces the background of general-purpose graphic processing units (GPGPUs). Then, Section 2.3, 2.4, and 2.5 discuss existing studies on optimizations at algorithm, operator, and kernel levels.

2.1 Neural Network Models

This section covers the background of transformers for Chapter 3, 4 and graph neural networks for Chapter 5.

2.1.1 Transformers and Attention Mechanism

The transformers [13] have achieved state-of-the-art performance across various domains such as natural language processing and computer vision. The architecture of transformers, as illustrated in Figure 2.1 (A), consists of stacked encoder and decoder

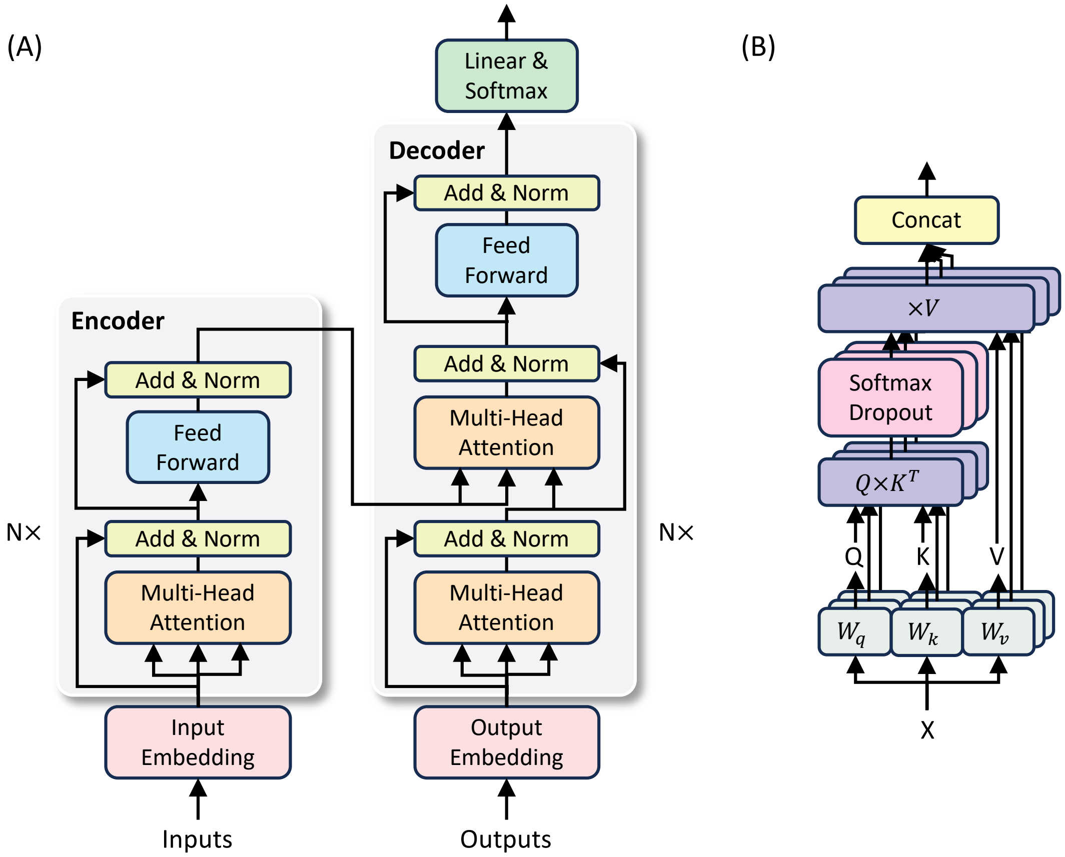
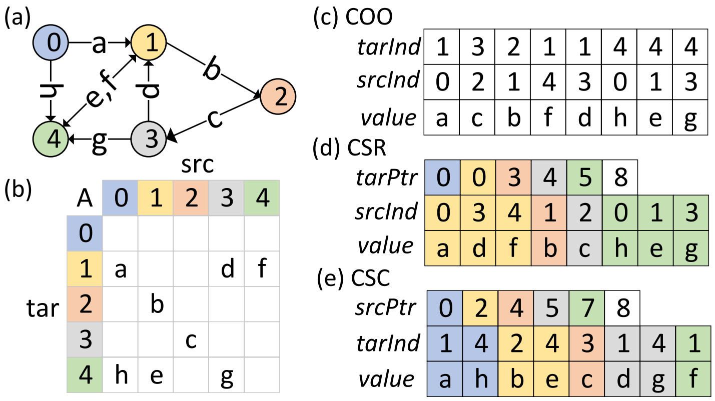


Figure 2.1: Transformer model architecture. (A) The encoder-decoder architecture of transformers; (B) The multi-head attention mechanism.

blocks. There are also encoder-only transformers like BERT [2] and decoder-only models like GPT-2 [15] designed for different applications.

The key strength of transformers, differentiating them from traditional neural networks, lies in the multi-head attention mechanism (MHA) illustrated in Figure 2.1 (B). The MHA provides a scalable solution to capture long-range dependencies, which allows building large-scale neural networks for real-world problems. The scalability comes from its architecture which is easy to parallel on GPGPUs. Given an input sequence $\boldsymbol{X} = (\boldsymbol{x}_1,..,\boldsymbol{x}_n) \in \mathbb{R}^{n\times d}$, each head of the attention mechanism can be defined as

$$\boldsymbol{O} = Softmax(\boldsymbol{Q}\boldsymbol{K}^T/\sqrt{d})\boldsymbol{V}, \tag{2.1}$$



(b)

A	0	1	2	3	4
0					
1	a			d	f
2		b			
3			c		
4	h	e		g	

(c) COO

tarInd	1	3	2	1	1	4	4	4
srcInd	0	2	1	4	3	0	1	3
value	a	c	b	f	d	h	e	g

(d) CSR

tarPtr	0	0	3	4	5	8		
srcInd	0	3	4	1	2	0	1	3
value	a	d	f	b	c	h	e	g

(e) CSC

srcPtr	0	2	4	5	7	8		
tarInd	1	4	2	4	3	1	4	1
value	a	h	b	e	c	d	g	f

Figure 2.2: Graph Structured Data. (a) Graph; (b) Adjacent matrix; (c) COO format; (d) CSR format; (e) CSC format.

Table 2.1: Dataset information

Dataset	#Vertex	Feature Len.	#Edge	Avg. Degree
Cora (CR)	$2,708$	$1,433$	$5,429 \times 2$	4.0
Citeseer (CS)	$3,327$	$3,703$	$4,732 \times 2$	2.8
Pubmed (PB)	$19,717$	500	$44,338 \times 2$	4.5
Reddit (RD)	$232,965$	602	$114,615,892$	492

where $\boldsymbol{Q} = \boldsymbol{X}\boldsymbol{W}_q$, $\boldsymbol{K} = \boldsymbol{X}\boldsymbol{W}_k$, and $\boldsymbol{V} = \boldsymbol{X}\boldsymbol{W}_v$ are query, key, and value matrices. $\boldsymbol{Q}\boldsymbol{K}^T$ forms a full-quadratic adjacency matrix, with edge weights being the dot-product similarity between all elements in the sequence. This adjacency matrix is standardized with $1/\sqrt{d}$ to maintain the unit second moment and then normalized with softmax. Finally, the row feature vectors in $\boldsymbol{V}$ are aggregated according to the normalized adjacency matrix by multiplying them together. Throughout this dissertation, $\boldsymbol{Q}\boldsymbol{K}^T$ is referred to as the attention score matrix, and $\boldsymbol{A} = Softmax(\boldsymbol{Q}\boldsymbol{K}^T/\sqrt{d})$ is named the attention weight matrix.

2.1.2 Graph Neural Networks

The Graph Neural Network (GNN) is an emerging type of neural network specially tailored for graph-structured data.

Graph Structured Data. A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ consists of two components: vertices (nodes) and edges. For a graph with N_v vertices and N_e edges, each vertex $v_i \in \mathcal{V}$ possesses a feature vector $\boldsymbol{x_i} \in \mathbb{R}^{1\times m}$ with these feature vectors organized into a matrix $\boldsymbol{X} \in \mathbb{R}^{N_v \times m}$. The edges $(v_i, v_j) \in \mathcal{E}$ can be directed or undirected, and may also have a feature e_{ij}. Table 2.1 summarizes popular datasets used by GNNs.

As illustrated in Figure 2.2 (a) & (b), edges can be represented as a sparse adjacency matrix $\boldsymbol{A} \in \mathbb{R}^{N_v \times N_v}$, where the row and column indices of each entry identify the target and source vertices. Three popular formats for the sparse matrix are shown in Figure 2.2(c), (d), and (e): Coordinate list (COO), Compressed Sparse Row (CSR), and Compressed Sparse Column (CSC).

GNN Models. The GNN models are designed under the neighborhood aggregation strategy [16]. With $\boldsymbol{h}_v^{(k-1)}$ represents the input feature vector of vertex v at layer k, the k-th layer of GNN is expressed as

$$\widetilde{\boldsymbol{h}_i^{(k-1)}} = MLP^{(k)}\left(\boldsymbol{h}_i^{(k-1)}\right), \boldsymbol{h}_v^{(k)} = AGG^{(k)}\left(\left\{\widetilde{\boldsymbol{h}_{u\in\mathcal{N}(v)}^{(k-1)}}, \widetilde{\boldsymbol{h}_v^{(k-1)}}\right\}\right), \tag{2.2}$$

where $\mathcal{N}(v)$ is a set of nodes adjacent to vertex v, $AGG^{(k)}$ is the aggregator in layer k. The aggregator updates each feature vector with the weighted sum of its neighbors.

$$\boldsymbol{h}_v^{(k)} = \sum_{u\in\mathcal{N}(v)\cup\{v\}} \widetilde{e_{vu}} \otimes \widetilde{\boldsymbol{h}_u^{(k-1)}}, \tag{2.3}$$

In Graph Convolutional Networks (GCN) [14], the weight $\widetilde{e_{ij}}$ is computed with $\widetilde{e_{ij}} = \frac{e_{ij}}{\sqrt{(d_i+1)(d_j+1)}}$, where the e_{ij} is the initial scalar edge weight provided by the dataset. In Graph Attention Networks (GAT) [17], attention score of each neighbor is computed based on the dot-product similarity between the feature vectors:

$$\widetilde{e_{ij}} = dropout\left(\frac{exp\left(lReLU\left([\widetilde{\boldsymbol{h}_i^{(k-1)}}||\widetilde{\boldsymbol{h}_j^{(k-1)}}]\boldsymbol{a}^{(k)}\right)\right)}{\sum_{q\in\mathcal{N}_i} exp\left(lReLU\left([\widetilde{\boldsymbol{h}_i^{(k-1)}}||\widetilde{\boldsymbol{h}_q^{(k-1)}}]\boldsymbol{a}^{(k)}\right)\right)}\right), \tag{2.4}$$

2.2 General Purpose Graphic Processing Units

Graphic Processing Units (GPUs) are hardware initially designed to accelerate computer graphics and image processing through their massive parallel processing power. In 2007, NVIDIA introduced CUDA, the widely adopted programming model for GPU computing, that allows GPUs to be programmed for general-purpose applications [18]. The programmability coupled with the parallel processing power of GPGPUs, quickly makes GPGPUs the most suitable and widely adopted accelerators for deep learning [7]. Throughout this dissertation, the terms GPU and GPGPU will be used interchangeably.

2.2.1 GPU Architecture

Figure 2.3 (A) illustrates the architecture of the GPGPUs. The GPGPU is composed of an array of streaming multiprocessors (SMs) that share the L2 cache and the off-chip High Bandwidth Memory (HBM). The HBM is usually referred to as global memory. Each SM has a private L1 data and instruction cache, and part of the L1 data cache can be configured as the shared memory managed by the programmer.

The SM is further partitioned into multiple sub-cores, each core contains a set of floating point and integer units, tensor cores, register file, and other components such as warp scheduler, dispatch unit, and L0 instruction cache.

For instance, the NVIDIA V100 GPU consists of 84 SMs sharing a 16GB HBM2 memory and 6144KB L2 cache. Each SM has a 128KB L1 data cache, 96 KB of which

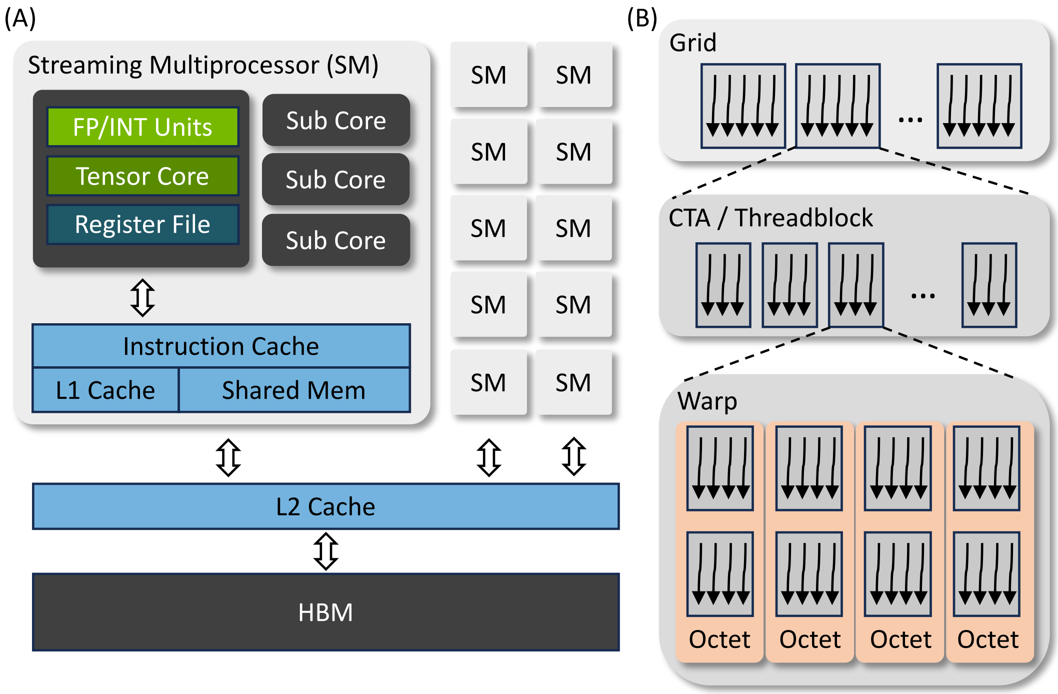


Figure 2.3: (A) GPU architecture and memory hierarchy; (B) Thread hierarchy.

can be configured as the shared memory. The SM contains 4 sub-cores, each sub-core has 16 FP32 Cores, 8 FP64 Cores, 16 INT32 Cores, 2 mixed-precision Tensor Cores, one L0 instruction cache, one warp scheduler, one dispatch unit, and 64 KB register file.

2.2.2 Programming Model

The programs of GPGPUs are organized into kernels: single instruction, multiple threads (SIMT) functions that are executed in parallel across thousands of threads. There are two key concepts in the GPGPU programming model: thread hierarchy and memory model. The former describes how the threads are dispatched to the SMs and sub-cores in GPGPU architecture, while the latter defines how the threads access different memory scopes including global memory, shared memory, and register files.

Thread Hierarchy. As illustrated in Figure 2.3 (B), the threads in a GPU kernel are organized as a hierarchical structure. At the top level, the kernel is composed of grids of cooperative thread arrays (CTAs), also known as thread blocks. The total number of CTAs is referred to as *grid size*. When launching the kernel, the CTAs are dispatched to the SMs in a round-robin fashion, and a large enough grid size is required to achieve high utilization of the SMs. The number of CTAs an SM can accommodate depends on its resource allocation, including registers and shared memory. The number of threads in a CTA is referred to as *CTA size*. Within a CTA, 32 consecutive threads are organized into warps, and dispatched to sub-cores within each SM.

Memory Model. The global memory loads and stores issued by threads of a warp are coalesced into 2-, 64-, or 128-byte transactions. Non-coalesced memory access from a warp, such as strided access, results in transactions containing redundant data not requested by threads, leading to memory access inefficiency. To achieve the best bandwidth utilization, data accessed by threads in a warp should fall in aligned 128-byte memory segments. Additionally, CUDA provides vectorized memory access, enabling each thread to load or store up to 128-bit data per instruction. Leveraging this vectorized approach reduces total instructions, lowers latency, and enhances overall bandwidth utilization.

The shared memory offers roughly 100× lower latency than uncached global memory. It is explicitly allocated per CTA and can be accessed by all threads in it. The shared memory is divided into 32 banks that can be accessed simultaneously. The banks are organized such that successive 32-bit data are assigned to successive banks. The bank conflict happens when the load and store from multiple threads of different data target the same bank, which leads to a performance penalty.

Each thread can use up to 255 32-bit registers. While the register is private to each thread, CUDA provides the warp shuffling instructions that allow threads in the same warp to access each other's data. Using more than 255 registers will lead to register spilling where the excess part will be stored in local memory, a part of global memory

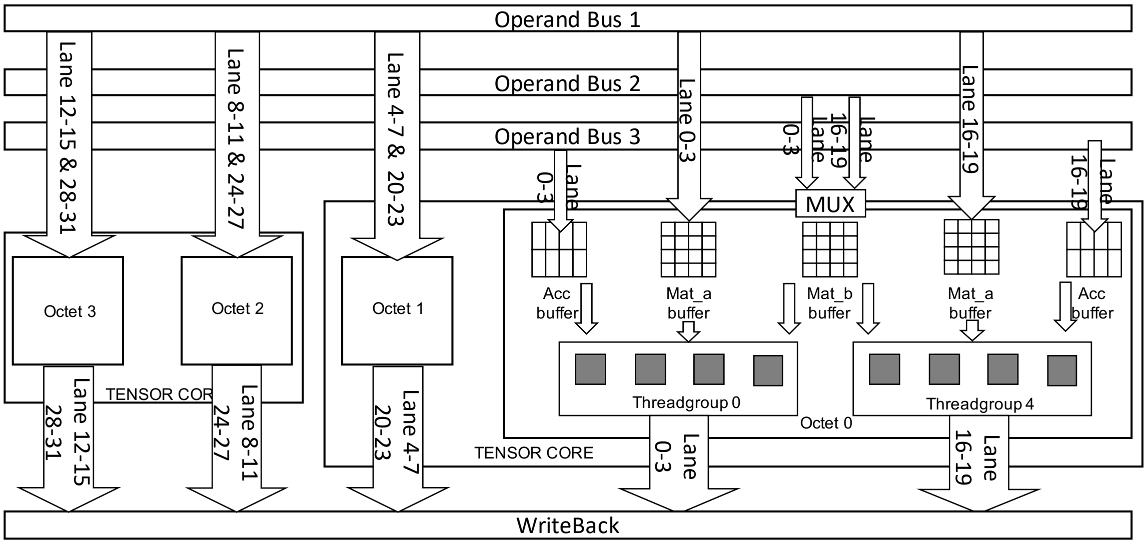


Figure 2.4: Volta TCU Architecture. [19]

private to each thread. The register spilling has a significant negative impact on kernel performance and has to be avoided.

2.2.3 Tensor Core

The Tensor Core Unit (TCU) is a specialized hardware unit designed to accelerate the matrix multiply-accumulate (MMA) operations. It significantly improves the performance of key operators in neural networks, such as matrix multiplication and convolution.

Volta Tensor Core. The Volta Tensor Core is the first generation tensor core introduced to GPGPUs by NVIDIA [20] that provides 8× peak FLOPs than floating point units. Figure 2.4 shows the Volta TCU architecture. A warp uses two TCUs at the same time. Within each warp, consecutive 4 threads form a *thread group*, the thread group id of a thread is $\lfloor \frac{threadIdx\%32}{4} \rfloor$. Furthermore, thread group i$\in\{0,1,2,3\}$ and thread group i+4 together form the *Octet* i. In this dissertation, thread group i and i+4 are referred to as low group and high group, respectively.

Each TCU is controlled by two octets. Each thread group in the octet has its own

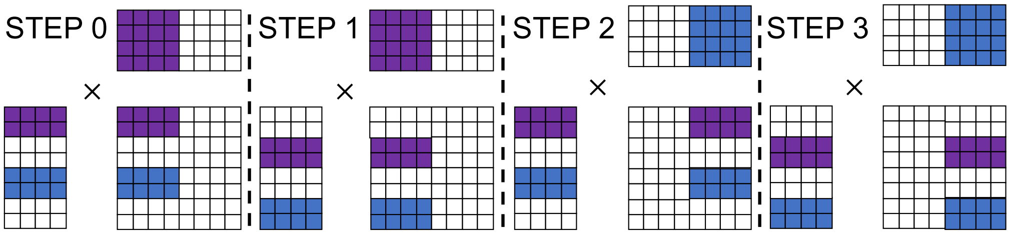


Figure 2.5: Visualization of the 4 steps in *mma.m8n8k4*. Purple blocks: low group; Blue blocks: high group.

buffer for storing the LHS matrix (Mat_a buffer) and accumulation result (Acc buffer), and each thread can access a four-by-four inner product unit (gray blocks). The RHS matrix buffer (Mat_b buffer) is shared by the two thread groups within each Octet. Its source is selected by a multiplexer.

CUDA provides two levels of APIs for Volta TCU. Firstly, warp-level matrix multiply and accumulate (WMMA) in C++ performs a dense matrix multiplication with a warp. For instance, *wmma.m8n32k16* computes a $8 \times 32 \times 16$ GEMM with a warp. Secondly, matrix multiply and accumulate (MMA) in PTX performs 4 dense matrix multiplications with a warp, one for each Octet. For instance, *mma.m8n8k4* completes four $8 \times 8 \times 4$ matrix multiplications. During the compilation, the *mma.m8n8k4* is further decomposed into four *HMMA* instructions when lowing to SASS: *HMMA.884.F32.F32.STEP*{*0,1,2,3*} in Figure 2.5. In each step, every thread group loads its LHS tile to its Mat_a buffer and the input accumulation values to its Acc_buffer. The multiplexer for RHS tile selects the low group in step 0&1, and high group in step 2&3.

Ampere Sparse Tensor Core. The third-generation tensor core in Ampere GPGPUs further supports the Sparse-Dense Matrix Multiplication (SpMM) under N:M fine-grained structured sparsity. The N:M sparsity preserves N elements in every $1 \times M$ vector from the original dense matrix and provides orders of magnitude more combinations than coarse-grained block sparsity while remaining GPU-friendly. NVIDIA introduces the 1:2 and 2:4 fine-grained structured pruning of weight matrices in neural networks [21]. As

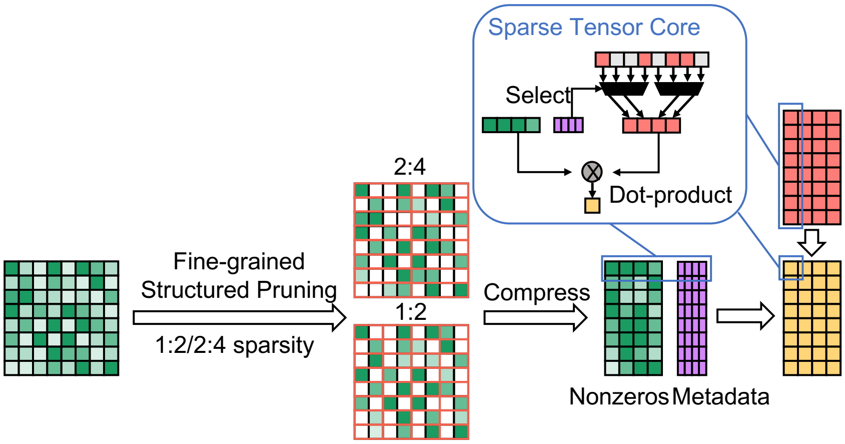


Figure 2.6: A100 GPU 1:2 and 2:4 Fine-Grained Structured Sparsity Pruning. [21]

shown in Figure 2.6, the dense weight matrix is pruned and then compressed with APIs in cuSPARSELt library offline. During inference, the compressed weight matrix is multiplied with input activation with sparse-dense matrix multiplication (SpMM) accelerated by Ampere sparse tensor core. This brings up to 1.9× speedup over the dense counterpart. Follow-up studies [22, 23, 24, 25, 26] propose algorithms to improve the accuracy, reduce training time, and accelerate the SpMM for N:M weight sparsity with new hardware designs.

2.3 Related Work on Algorithm-level Optimizations

This section summarizes existing studies on algorithm-level optimizations.

2.3.1 Low Precision and Sparsity

Low precision and sparsity are two major methods to improve the algorithm-level utilization by reducing the computational and memory footprint of large neural networks [27].

The low precision reduces the number of bits that represent each operand from 32 to

16 or even lower [28, 29]. Recent advances in GPGPU like Tensor Cores provide hardware acceleration for low-precision computation. Coupled with the lower memory requirement, the low precision significantly improves the efficiency of neural networks during training and inference.

The sparsity exploits the sparsity in the neural networks. For example, the dense weight matrices can be sparsified while maintaining comparable model quality [30, 27, 31, 32]. Existing studies [33, 34, 32, 35] have proposed different mechanisms to efficiently approximate and predict the zero values in the output feature map of CNNs, RNNs, and transformers to skip these computations during execution.

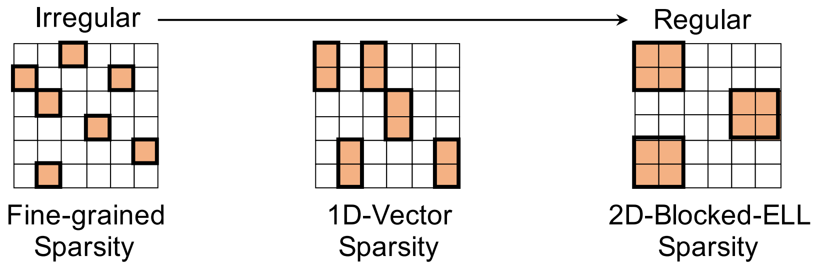


Figure 2.7: Different sparse structures.

Efforts have been made to construct hardware-friendly sparsity by adding structured constraints to the pattern. As shown in Figure 2.7, apart from fine-grained sparsity, structures like 2D-block [36] can be enforced to the sparse matrix to improve the locality and the computation for efficiency. Chapter 3 of this dissertation also presents a 1D-Vector sparsity that well balances the accuracy and data reuse.

2.3.2 Efficient Attention Mechanism

The high computation cost and memory footprint in the full attention mechanism come from the attention weight matrix $\boldsymbol{A}$, whose size grows quadratically with the sequence length n. To address this issue, various efficient attention mechanisms have been proposed [37].

Fixed Sparse Patterns. Beltagy et al., 2020 [38] and Zaheer et al., 2020 [39] apply a set of fixed sparse attention patterns on $\boldsymbol{A}$, like global attention and sliding window attention. These patterns are constructed from empirical observations and designed GPU-friendly to achieve speedup. The dynamic and fine-grained challenges are solved through costly retraining or finetuning to constrain the distribution of dominant entries.

Dynamic Sparse Patterns. Ham et al., 2021 [40] dynamically generate fine-grained sparse attention patterns on $\boldsymbol{A}$ with low-cost binary hashing. However, this technique requires specialized hardware to achieve speedup, so it is unavailable on GPGPU. Kitaev et al., 2020 [41], Tay et al., 2020 [37], and Roy et al., 2021 [42] apply various clustering methods and only compute the attention within each cluster. Although computing full attention in each cluster is more friendly to GPU compared with fine-grained sparsity, the clustering methods lead to low utilization as they contain several GPU-unfriendly operators like top-k and sorting that offset their benefits under moderate sequence length.

Low Rank / Kernel. Wang et al., 2020 [43] project $\boldsymbol{A}$ from $n \times n$ to $n \times k$ with linear projection. Choromanski et al., 2021 [44] introduce the FAVOR+ which approximates the softmax with the kernel method. This allows them to change the computation order and reduce the asymptotic complexity to linear. However, the low-rank projection and kernel construction also introduce considerable overhead. This makes these methods only effective under long sequence length. Besides, the low-rank projection drastically changes the attention mechanisms, tens of thousands of pre-training or finetuning steps are required to reach a comparable performance with the original full attention mechanism. So they require tremendous engineering effort to deploy.

2.3.3 Automatic Algorithm-level Optimization

The algorithm-level optimization can be automated by regarding the neural networks as data-flow graphs and applying a series of transformations on the graph. Existing deep learning compilers have adopted various graph-level transformations to improve the efficiency of Deep Learning Systems. For instance, TVM [45] optimizes its graph-level IR, *relay*, with passes like constant-folding and data layout transformation.

2.4 Related Work on Operator-level Optimizations

Most existing studies employ simple heuristics when partitioning computation graphs. TVM[45], for instance, classifies the operators into four types: injective (element-wise), reduction, complex out-fusible (can fuse element-wise map to output), and opaque (cannot be fused). They define heuristic rules such as reduction can be fused with input injective operators and complex out-fusible nodes like GEMM can fuse element-wise operators to its output. However, as illustrated in Section 6.3, these heuristics would lead to infeasible partitions with cycles or suboptimal solutions.

There are also studies on the acyclic partitioning problem beyond the Deep Learning System community [46]. The ILP-based partitioner in Chapter 6 takes inspiration from Nosack et al., 2014 [47] which formulates the acyclic partitioning problem as ILP. The ILP-based formulation provides great extensibility to encode new constraints and heuristics through ILP constraints. However, the high complexity of the approach developed by Nosack et al., 2014 [47] makes it impractical to be directly used when solving large-scale neural networks with thousands of nodes, and their lower bound obtained from Kernighan's solution [48] is not valid when additional constraints are involved. In Section 6.3, a novel approach is developed that divides the large computation graph into smaller ones so that they can be solved in a reasonable amount of time.

2.5 Related Work on Kernel-level Optimizations

This section summarizes related kernel-level optimizations in existing studies.

2.5.1 Sparse Kernels on GPGPU

Existing sparse kernels on GPGPU focus on two key operators: Sparse Matrix-Matrix Multiplication (SpMM) and Sampled Dense-Dense Matrix Multiplication (SDDMM). The former multiplies a sparse matrix with a dense one. The latter multiplies two dense matrices, while sparsity is located in the output matrix of the equation to help reduce the required computations.

Efficient sparse kernel implementations have been proposed for different sparse patterns. NVIDIA introduces the cuSPARSE library that targets 95% or higher sparsity and provides the *cusparseSpMM* and *cusparseSDDMM* APIs. The former one supports half, single, or higher precision, and the sparse matrix can be either fine-grained sparsity or Blocked-ELL format. Gale et al., 2020 [49] introduce a library called Sputnik that targets fine-grained sparsity and outperforms cuSPARSE under relatively low sparsity. Sputnik achieves speedup over the dense baseline under $> 71\%$ sparsity with single precision.

2.5.2 Automatic Kernel Generation with Compiler

Kernel fusion is the key optimization applied in automated kernel-level optimization. During kernel fusion, multiple operators under the producer-consumer relationship are fused into a single one. The intermediate result can be directly cached in the register or shared memory to reduce memory access. The performance of the fused kernel is determined by the quality of the implementation of each operator, while the flexibility is determined by the ability of the operator compiler to align the loop structure of different operators.

Template metaprogramming (TMP)-based compilers, such as AITemplate [50] and Bolt [51], use expert-developed template libraries [52, 53, 54] like CUTLASS to achieve optimal performance. However, these compilers lack abstractions for aligning operators with the intricate loop structure of the core operation, they can only support fusion patterns defined by the template library.

Loop-based compilers, such as TVM [45], Tiramisu [55], Nvfuser, and Tensor Comprehensions [56], represent operations as loops and apply schedules, such as loop fission, fusion, parallel, and vectorization, to map them to GPUs. However, the performance of the core operators generated by these compilers is inferior to the expert-designed kernels.

Chapter 3

Algorithm-Kernel Codesign: Static Sparsity with Tensor Core

This chapter presents the kernel-centric optimization with a specific focus on the algorithm-kernel codesign. It improves the EU-level utilization of neural networks on GPGPUs when both sparsity and low precision are applied.

3.1 Introduction

In recent years, areas such as computer vision and natural language processing have witnessed remarkable advancement driven by deep neural networks. However, the achievements come at the expense of the enormous memory footprint and computation cost. To address this issue, a common strategy involves the application of low precision and sparsity [27, 57, 58, 59]. Low precision represents data with fewer bits to save storage and memory bandwidth, and specialized computational units like Tensor Cores [20] have been introduced to improve the computation throughput under low precision. On the other hand, sparsity involves storing the tensors in the neural networks with compressed encoding, recording only non-zero elements and their positions. This not only reduces the memory footprint but also significantly decreases the FLOPS by replacing dense matrix multiplications with sparse matrix-matrix multiplication (SpMM) and sampled dense-dense matrix multiplication (SDDMM).

However, applying sparsity together with low precision leads to inferior performance due to the low EU-level utilization. In detail, SpMM and SDDMM under fine-grained sparsity, despite lower FLOPS, face limitations in EU-level utilization caused by the memory bottleneck. The sparse operators have much fewer data reuse opportunities compared with their dense counterparts. Although structured sparsity, such as 2D blocked-ELL, offers more data reuse opportunities, their large grain size introduces challenges in maintaining the model accuracy [60].

This chapter presents the VecSparse to address these challenges.

At the algorithm level, this chapter balances the data reuse and granularity size with a novel structured sparse pattern named column vector sparse encoding in Section 3.3. Inspired by the widely used compressed sparse row (CSR) encoding, the new pattern associates each index with a short nonzero column vector. With vector length $V \times 1$, it provides the same data reuse rate as the $V \times V$ block sparsity in both SpMM and SDDMM operators, while maintaining the model accuracy with its smaller grain size.

Despite the data reuse opportunities offered by the new sparse pattern, existing floating-point unit (FPU) and tensor core unit (TCU)-based kernel implementations of SpMM and SDDMM face new challenges in achieving high EU-level utilization. The former's utilization is limited by pipeline stalls such as instruction cache miss, while the latter is constrained by shared memory bandwidth and waste of computations when mapping small Vs to TCUs.

At the kernel level, Section 3.2 presents five key guidelines for kernel-level optimization to improve the EU-level utilization in SDDMM and SpMM kernels, summarized from detailed profiling of existing implementations and the best practice of CUDA programming. A novel mapping between the warp tile and TCUs, namely TCU-based 1-D Octet Tiling (Section 3.4 and 3.5), is introduced that satisfies all the five guidelines simultaneously.

Section 3.6 extensively evaluates the proposed kernels on the sparse matrices from DLMC [61] dataset using different grain sizes, problem sizes, and sparsity ratios. In short, for SpMM, **1.71-7.19x** geometric mean speedup over the Blocked-ELL SpMM kernel from cuSPARSE and **1.34-4.51x** geometric mean speedup over an FPU-based kernel that directly extended from Sputnik [49] are achieved. For SDDMM, **1.27-3.03x** speedup over the FPU-based kernel extended from Sputnik [49] and **0.93-1.44x** speedup over the TCU-based kernel that uses the classic mapping between GEMM-like warp tile and TCU are achieved. Compared with the cuBLASHgemm, the proposed SpMM and SDDMM kernel achieve practical speedup under ≥70% and ≥90% sparsity with the tiny 4×1 grain size. Further evaluation of VecSparse on sparse transformer inference task demonstrates 1.41x end-to-end speedup and 13.37x peak memory reduction.

3.2 Motivation

This section discusses the limitations of existing sparsity patterns and corresponding SpMM and SDDMM kernels from the algorithm and kernel perspectives.

3.2.1 Algorithm: Limitation of Existing Sparse Patterns

As summarized in Section 2.3.1, existing sparse patterns can be classified into fine-grained sparsity and structured sparsity.

Fine-grained Sparsity. Figure 3.1 illustrates the speedup over the dense baseline (cuBLAS) achieved by the Sputnik [49] and cuSPARSE [62], two state-of-the-art libraries,

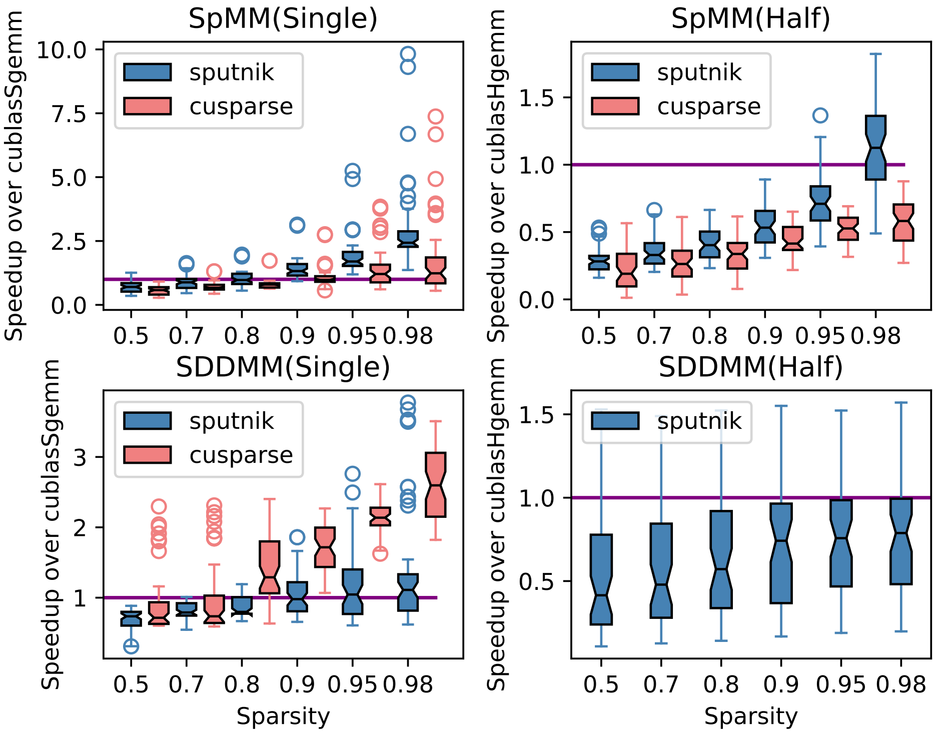


Figure 3.1: Speedup over cuBLAS with fine-grained sparsity.

under single and half-precision. Although considerable speedup is achieved under single precision [1], under half-precision, the performance of SpMM and SDDMM kernels is inferior to their dense counterpart, reflecting low EU-level utilization. The performance degradation has two major causes: memory bottleneck and ineligibility of using tensor cores.

To reduce the memory bottleneck, dense GEMM leverages the data locality and caches tiles of multiplicand and multiplier in shared memory, while SpMM and SDDMM depend on the sparsity to skip the memory access related to zeros. As shown in Figure 3.2, under half-precision, the L1$ Missed Sectors of GEMM is decreased by a factor of more than three times. This is from not only the halving of data size but also the improved locality as shared memory can cache more elements. In contrast, to compete

[1] The SDDMM in cuSPARSE is faster than Sputnik [49] under cuSPARSE v11.2.2.

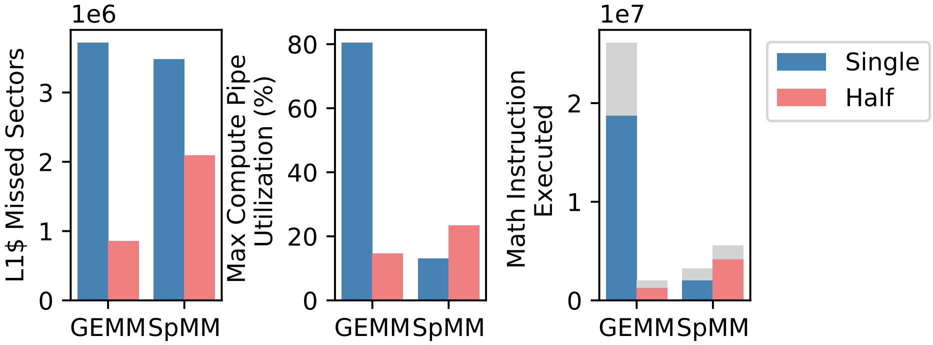


Figure 3.2: Profiling Metrics of GEMM and SpMM under different precision with problem size 2048 × 1024 × 256 with 90% sparsity.

with GEMM, SpMM and SDDMM can only increase their sparsity at the cost of model accuracy, as they lack data reuse opportunities.

The dense GEMM also benefits from the accelerated computation with tensor cores. Figure 3.2 shows that under half-precision, ”Max Compute Pipe Utilization” and ”Math Instruction Executed” of GEMM are significantly reduced, which indicates the relief of the computation bottleneck. The computations are offloaded to tensor cores that offer higher computation throughput with fewer instructions. Oppositely, the SpMM and SDDMM cannot be easily mapped to tensor cores, as the latter is only designed for dense matrix multiply accumulation.

Structured Sparsity. Existing structured sparse patterns like Blocked-ELL are usually composed of nonzero square blocks. With this pattern, the SpMM and SDDMM can be converted into a batch of small dense matrix multiplications. Although the blocked sparsity greatly improves the data locality and enables leveraging the tensor cores, its granularity increases quadratically with the data reuse ratio. Existing studies [60] have shown that large block size leads to accuracy degradation under the same sparsity. As a consequence, the design space is significantly limited due to the wrestling between kernel performance and model accuracy.

To conclude, a novel sparse pattern is required to be designed on the algorithm side

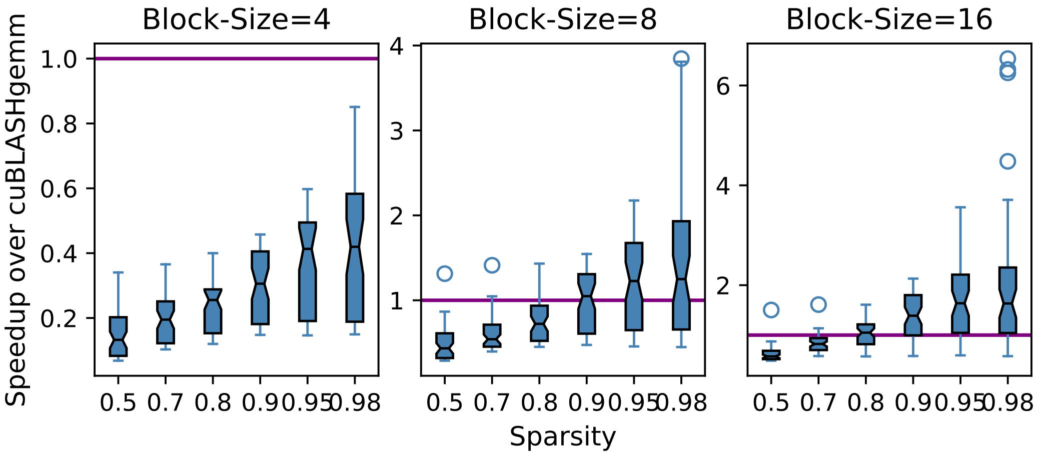


Figure 3.3: Speedup over cuBLAS with Blocked-ELL SpMM.

that better balances the data reuse opportunities and the granularity size, while enabling the use of tensor cores. This chapter introduces the column-vector sparse encoding in Section 3.3, which has finer granularity under the same data reuse rate as block sparsity.

3.2.2 Kernel: Limitation of Existing Kernel Implementation

Another limitation exists in the previous kernel implementation of structured sparsity. While the small structure sizes are desired to maintain model accuracy [60], Figure 3.3 illustrates that the Blocked-ELL-based SpMM exhibits suboptimal performance. It only surpasses the dense matrix multiplication with the block size greater than 8.

Table 3.1: Stall Reasons in Blocked-ELL based SpMM kernel

Block Size	**No Instruction**	**Wait**	**Short Scoreboard**
4	42.6%	21.0%	11.9%

Detailed profiling of Blocked-ELL SpMM kernel reveals that under small block size, the kernel suffers from low EU-level utilization of both memory bandwidth and computation units, which is primarily caused by pipeline stalls. Table 3.1 listed the top-3 stall reasons.

”No Instruction” is caused by the instruction cache miss. The SASS code of SpMM contains over 4600 lines of code, yet the 12 KB L0 instruction cache of each sub-core can only hold 768 instructions. This leads to L0 instruction cache capacity miss.

The ”Wait” happens when a warp is stalled waiting on a fixed latency execution dependency. The instruction statistics show the IMAD (Integer Multiply & Add) and IADD3 (3-input Integer Add) account for 27.4% of total executed instructions. These integer instructions compute addresses and predicates, contributing to the ”Wait” stall.

”Short Scoreboard” occurs when a warp is stalled, waiting for shared memory data loading. With block size 4, the $\frac{\#shared\ memory\ load\ requests}{\#global\ load\ requests}$ ratio of SpMM is 0.87, compared to 4.17 of dense matrix multiplication. This implies the data stored in shared memory is not reused frequently, whereas the shared memory causes other overhead such as more complex data path and synchronization overhead. Moreover, configuring part of the L1 cache into shared memory reduces implicit data reuse through L1 cache.

In summary, at the kernel level, with the analysis above and the best practices guide for CUDA kernel design, five key guidelines can be proposed for the kernel-level optimization. Guideline I and II improve the overall SM and EU-level utilization, III focuses on the computation utilization, while IV and V influence the memory utilization.

- **I**. Minimize program size to prevent overflowing the instruction cache.
- **II**. Increase the grid size to hide the latency through thread-level parallelism (TLP).
- **III**. Reduce fixed latency operations through looping unrolling, computing offset and constants at compile time, and merging floating point operations to *HMMA* with TCU.
- **IV**. Directly load data with limited reuse opportunities to the register file, bypassing shared memory.

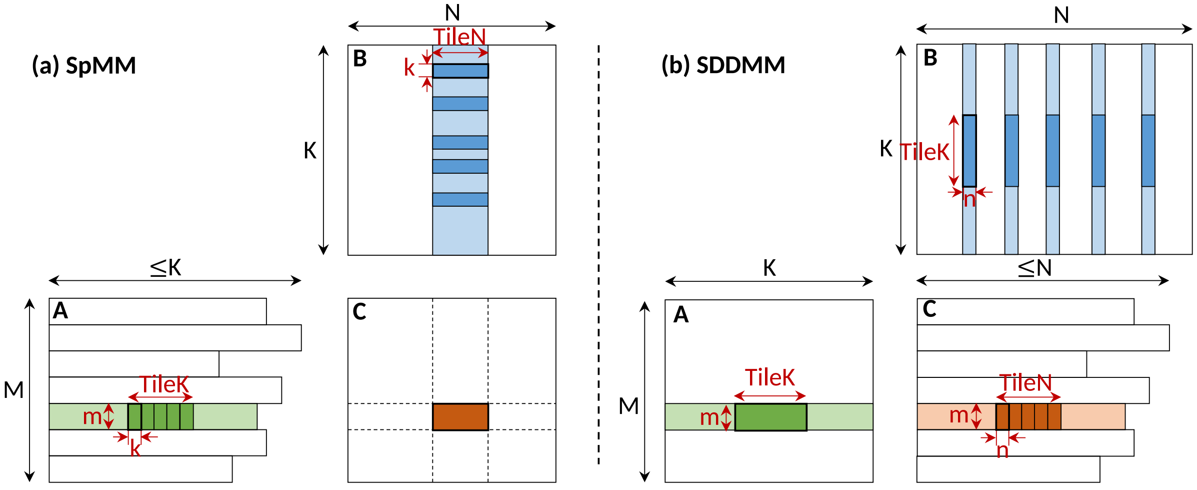


Figure 3.4: Generalized Block Sparse representation.

- **V**. Improve bandwidth utilization with 128B coalesced transactions and long vectorized memory access (*LDG.128*).

A novel TCU-based 1-D Octet Tiling in Section 3.4 and 3.5 is proposed to construct SpMM and SDDMM kernels that satisfy all the five guidelines simultaneously.

3.3 Algorithm-level Optimization: VecSparse

The key observation behind the algorithm-level optimization is that, for both SpMM and SDDMM, a block sparse matrix has the same data reuse regardless of the number of columns within each block. Building upon this observation, a novel sparsity pattern, namely column vector sparse encoding, is proposed that well balances the sparsity granularity and computation efficiency.

3.3.1 Data Reuse Analysis

Figure 3.4 shows the SpMM and SDDMM under block sparsity. The nonzero blocks are aligned in the vertical dimension. The problem size is $M \times N \times K$, $TileK$ and $TileN$

are the tiling sizes, which are constrained by the shared memory or register file capacity.

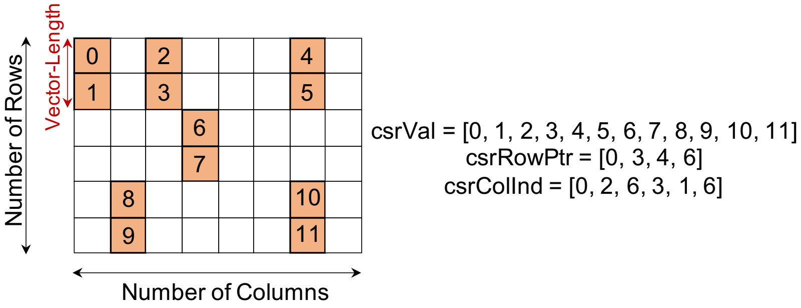
Regarding SpMM, each block represents an $m \times k$ matrix, where m and k are user-defined sizes for block sparsity. Following the workflow in Sputnik [49], each tile computes the partial sums in matrix C with $TileK/k$ consecutive nonzero blocks from A and a vector with width $TilesN$ from the corresponding rows in B.

For SDDMM, each block within matrix C is an $m \times n$ matrix, with user-defined grain size $m \times n$. In this scenario, each tile computes the partial sum of $TileN/n$ consecutive nonzero blocks in matrix C by extracting $TileK$ columns from the corresponding rows in A and $TileK$ rows from the corresponding columns in matrix B.

With the above settings, it is obvious that in both SpMM and SDDMM, each operand from the LHS matrix is reused for $TileN$ times, while each RHS operand is reused for m times. Therefore, the number of data reuse is determined by m and $TileN$, independent of the number of columns (k in SpMM and n in SDDMM) in each block.

3.3.2 Column Vector Sparse Encoding

As the data reuse is independent of the column number, the number of columns of each block can be reduced to 1 to minimize the sparsity granularity. This results in the column vector sparse encoding shown in Figure 3.5. The new pattern is equivalent to



csrVal = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
csrRowPtr = [0, 3, 4, 6]
csrColInd = [0, 2, 6, 3, 1, 6]

Figure 3.5: Column Vector Sparse Encoding.

replacing each nonzero scalar in the CSR sparse matrix with a nonzero column vector. The elements within each vector are consecutive in memory, and the vectors in the same row are sequentially arranged. With this encoding and small vector lengths such as $\{2, 4, 8\}$, the vectors can be loaded and stored with the CUDA vector types directly with coalesced memory access.

3.4 Kernel-level Optimization: SpMM

This section details the kernel-level optimization on SpMM under the proposed sparse pattern. It is organized as follows. Firstly, the weaknesses of two classic baseline designs are discussed under the five guidelines, including an FPU-based design following Sputnik [49], and a TCU-based design following the classic GEMM-like tiling approach. Then, a more efficient design, TCU-based 1-D Octet Tiling, is presented that satisfies all the five guidelines simultaneously.

3.4.1 Baseline I: FPU-based 1-D Subwarp Tiling

Gale et al., 2020 [49] propose the FPU-based 1-D subwarp tiling for the SpMM kernel under fine-grained sparsity that maximizes the memory access efficiency. It is called "1-D subwarp tiling" because under the fine-grained setup (V=1), as illustrated in Figure 3.6 (a), the LHS operand is a $1\times TileK$ 1-D vector handled by a subwarp of threads. A CTA tile contains multiple independent 1-D tiles that are assigned to subwarps ($Subwarp\ Size \leq 32$). Each 1-D tile is further decomposed to $Subwarp\ Size$ independent $(V{\times}TileK){\cdot}(\frac{TileK{\times}TileN}{Subwarp\ Size})$ thread tiles. The threads in the same subwarp first load the LHS fragment into the shared memory corporately. Then, each thread loads the RHS fragment corresponding to its tile and computes the MMA.

This tiling design opts for maximizing memory access efficiency. First, it satisfies the

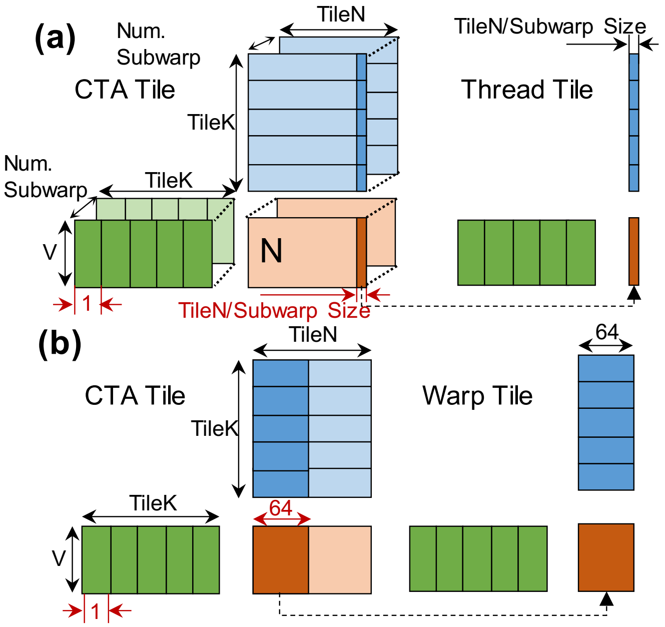
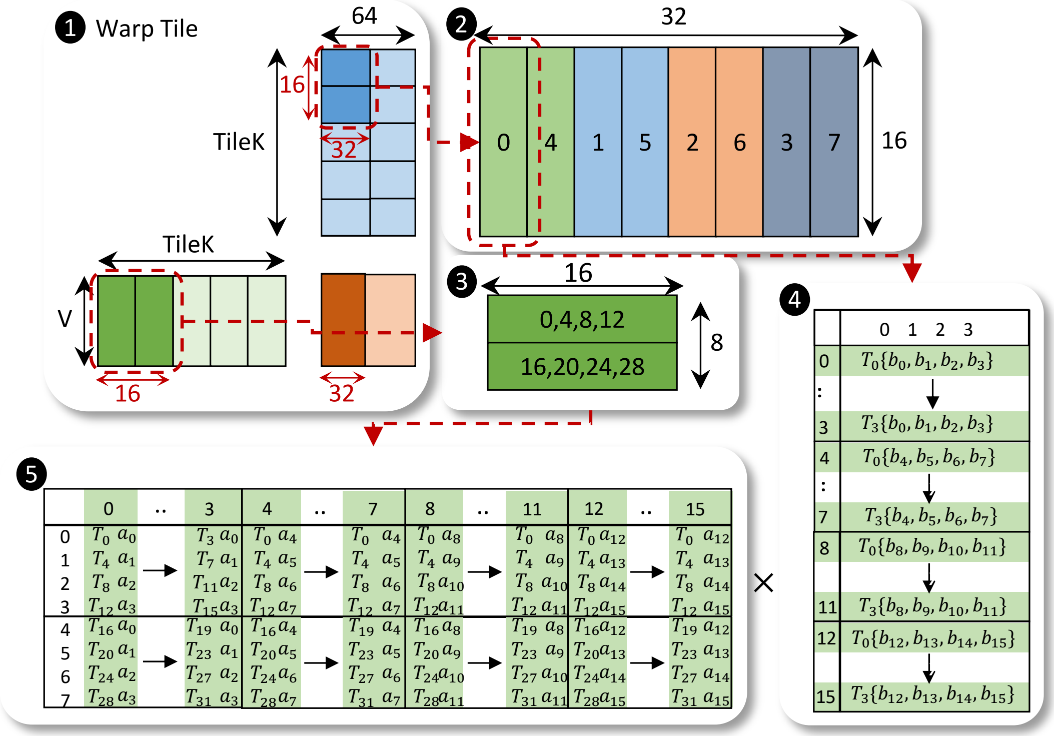


Figure 3.6: Decomposition of SpMM with 1-D tiling. (a) The FPU tiling extended from Sputnik[49]; (b) TCU-based tiling.

aforementioned guideline IV as the RHS operands are directly loaded to the register file. For guideline V, by choosing $TileN = 64$ and $Subwarp\ Size = 8$, each subwarp can load a row of consecutive 64 half operands from the RHS fragment with the vector memory operation *LDG.128* in a single 128B transaction.

Despite the benefits, this design compromises guidelines I, II, and III. For guideline I, computing each subwarp tile requires fully unrolling the loops along V, $TileK$, and $TileN$. This results in a significant amount of instructions. The unrolling is essential as it enables the compiler to determine the index to RHS operands at compile time, preventing the usage of local memory for indexing.

This design also leads to conflict between guideline II and V. While guideline V expects $\frac{TileN}{Subwarp\ Size} = 8$, this ratio only produces grid size up to $\frac{M \times N}{V \times \#Subwarp \times TileN} = \frac{M \times N}{256V}$. Yet, by compromising guideline V and having $\frac{TileN}{Subwarp\ Size} = 2$, the grid size can be improved to $\frac{M \times N}{64V}$ that is four times larger.



	0 1 2 3
0	$T_0\{b_0, b_1, b_2, b_3\}$
:	↓
3	$T_3\{b_0, b_1, b_2, b_3\}$
4	$T_0\{b_4, b_5, b_6, b_7\}$
:	↓
7	$T_3\{b_4, b_5, b_6, b_7\}$
8	$T_0\{b_8, b_9, b_{10}, b_{11}\}$
	↓
11	$T_3\{b_8, b_9, b_{10}, b_{11}\}$
12	$T_0\{b_{12}, b_{13}, b_{14}, b_{15}\}$
	↓
15	$T_3\{b_{12}, b_{13}, b_{14}, b_{15}\}$

	0	..	3	4	..	7	8	..	11	12	..	15
0	$T_0\ a_0$		$T_3\ a_0$	$T_0\ a_4$		$T_0\ a_4$	$T_0\ a_8$		$T_0\ a_8$	$T_0\ a_{12}$		$T_0\ a_{12}$
1	$T_4\ a_1$	→	$T_7\ a_1$	$T_4\ a_5$	→	$T_4\ a_5$	$T_4\ a_9$	→	$T_4\ a_9$	$T_4\ a_{13}$	→	$T_4\ a_{13}$
2	$T_8\ a_2$		$T_{11}\ a_2$	$T_8\ a_6$		$T_8\ a_6$	$T_8\ a_{10}$		$T_8\ a_{10}$	$T_8\ a_{14}$		$T_8\ a_{14}$
3	$T_{12}\ a_3$		$T_{15}\ a_3$	$T_{12}\ a_7$		$T_{12}\ a_7$	$T_{12}\ a_{11}$		$T_{12}\ a_{11}$	$T_{12}\ a_{15}$		$T_{12}\ a_{15}$
4	$T_{16}\ a_0$		$T_{19}\ a_0$	$T_{16}\ a_4$		$T_{19}\ a_4$	$T_{16}\ a_8$		$T_{19}\ a_8$	$T_{16}\ a_{12}$		$T_{19}\ a_{12}$
5	$T_{20}\ a_1$	→	$T_{23}\ a_1$	$T_{20}\ a_5$	→	$T_{23}\ a_5$	$T_{20}\ a_9$	→	$T_{23}\ a_9$	$T_{20}\ a_{13}$	→	$T_{23}\ a_{13}$
6	$T_{24}\ a_2$		$T_{27}\ a_2$	$T_{24}\ a_6$		$T_{27}\ a_6$	$T_{24}\ a_{10}$		$T_{27}\ a_{10}$	$T_{24}\ a_{14}$		$T_{27}\ a_{14}$
7	$T_{28}\ a_3$		$T_{31}\ a_3$	$T_{28}\ a_7$		$T_{31}\ a_7$	$T_{28}\ a_{11}$		$T_{31}\ a_{11}$	$T_{28}\ a_{15}$		$T_{31}\ a_{15}$

Figure 3.7: Classic mapping of the warp tile to TCU.

For computation efficiency, the design violates guideline III, as each thread computes its tile by a sequence of *HMUL* (FP16 Multiply) and *FADD* (FP32 Add) instructions.

3.4.2 Baseline II: TCU-based 1-D Warp Tiling

Guidelines I and III can be satisfied by mapping the computation to TCUs. The TCUs merge multiple *HMUL* and *FADD* into a single *HMMA* instruction, reducing the program size and fixed-latency instructions. Figure 3.6 (b) illustrates the TCU-based 1-D Warp Tiling that maps SpMM to TCUs. As TCUs are controlled by warps, the 1-D CTA tile is decomposed into warp-level tiles with size $V \times 64 \times TileK$. The 64 is chosen as it is the smallest number that perfectly fills the 128B transaction.

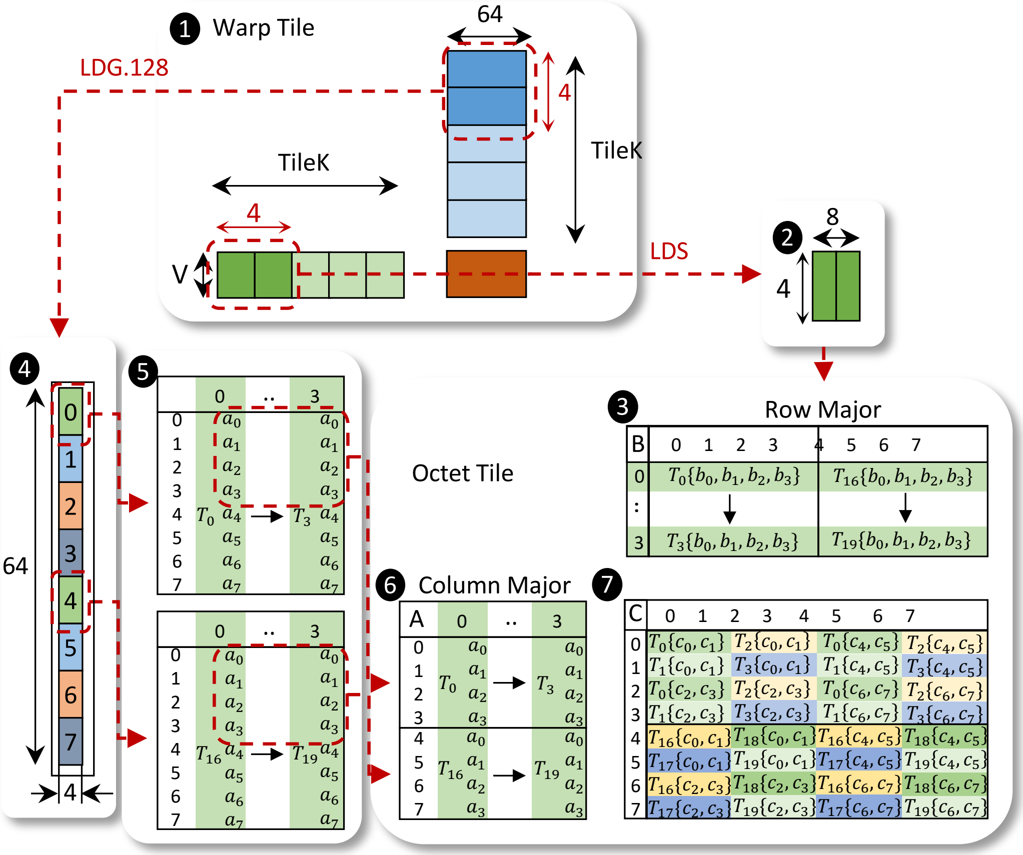
Figure 3.7 illustrates how the warp tile is further decomposed to each thread in the traditional way. In ❶, with $V \in \{2, 4, 8\}$, the warp tile is mapped to *wmma.m8n32k16* instruction to avoid the waste of computation. ❷ and ❸ show the mapping of the multiplicand and multiplier to the thread groups. In ❷, each thread group is responsible for a 16×4 block. An example of thread group 0 is illustrated in ❹, where T_i represents thread i and b_j indicates the register j of the thread. In ❸, each block represents a 4×16 block and the number on it indicates the thread groups that hold a copy, the detailed layout is shown in ❺.

Despite satisfying guidelines I and III, this design leads to conflict between guideline IV and V. When guideline IV is satisfied, the RHS fragment in ❷ is loaded directly from global memory to registers, as shown in ❹, each thread has 4 registers in each row, which prevents the use of *LDG.128* to achieve the optimal memory efficiency. On the other hand, to satisfy guideline V, the global memory access has to be coalesced through the shared memory, which violates guideline IV.

Besides, $TileK$ has to be the multiple of 16, which introduces additional overhead during residue handling when the number of nonzero in the current row is not divisible by $TileK$. At last, when V is smaller than 8, each computation step with $wmma.m8n32k16$ actually computes a $V \times 32 \times 16$ tile, which indicates a portion of wasted computation.

3.4.3 Solution: TCU-based 1-D Octet Tiling

Unlike the performance of the previous two designs limited by the wrestling between kernel/compute and memory access efficiency, this section presents a new design, TCU-based 1-D Octet Tiling, that satisfies all five guidelines simultaneously. The high-level idea is to map the SpMM to tensor cores to achieve good kernel and compute efficiency (guideline I, II, and III). The memory access efficiency (guideline IV and V) is achieved by redesigning the mapping between the warp tile and the TCU on the fine-grained Octet


1 Warp Tile
64
LDG.128
4
TileK
TileK
4
V
LDS
2
8
4
4
64
4
5
Octet Tile
3 Row Major
6 Column Major
7
B 0 1 2 3 4 5 6 7
0 $T_0\{b_0,b_1,b_2,b_3\}$ $T_{16}\{b_0,b_1,b_2,b_3\}$
3 $T_3\{b_0,b_1,b_2,b_3\}$ $T_{19}\{b_0,b_1,b_2,b_3\}$
A 0 .. 3
T_0 T_3 T_{16} T_{19}
a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7

C	0 1	2 3	4 5	6 7
0	$T_0\{c_0,c_1\}$	$T_2\{c_0,c_1\}$	$T_0\{c_4,c_5\}$	$T_2\{c_4,c_5\}$
1	$T_1\{c_0,c_1\}$	$T_3\{c_0,c_1\}$	$T_1\{c_4,c_5\}$	$T_3\{c_4,c_5\}$
2	$T_0\{c_2,c_3\}$	$T_2\{c_2,c_3\}$	$T_0\{c_6,c_7\}$	$T_2\{c_6,c_7\}$
3	$T_1\{c_2,c_3\}$	$T_3\{c_2,c_3\}$	$T_1\{c_6,c_7\}$	$T_3\{c_6,c_7\}$
4	$T_{16}\{c_0,c_1\}$	$T_{18}\{c_0,c_1\}$	$T_{16}\{c_4,c_5\}$	$T_{18}\{c_4,c_5\}$
5	$T_{17}\{c_0,c_1\}$	$T_{19}\{c_0,c_1\}$	$T_{17}\{c_4,c_5\}$	$T_{19}\{c_4,c_5\}$
6	$T_{16}\{c_2,c_3\}$	$T_{18}\{c_2,c_3\}$	$T_{16}\{c_6,c_7\}$	$T_{18}\{c_6,c_7\}$
7	$T_{17}\{c_2,c_3\}$	$T_{19}\{c_2,c_3\}$	$T_{17}\{c_6,c_7\}$	$T_{19}\{c_6,c_7\}$

Figure 3.8: TCU-based 1-D Octet Tiling for SpMM.

level.

In detail, the CTA and warp tiling in the TCU-based 1-D Warp tiling in Figure 3.6 (b) is directly used, while the new mapping of warp tile to threads is redesigned as shown in Figure 3.8. There are two major differences from the classic mapping in Figure 3.7.

Firstly, the LHS and RHS fragments are swapped to put V in the horizontal direction. This is motivated by four steps in Figure 2.5 where step 0&1 generates the left four columns in the outputs while step 2&3 produces the rest. When $V \leq 4$, step 2&3 can be skipped to reduce the waste of computation.

Secondly, the warp tile is further partitioned into octets to guarantee both efficient computation and memory access. Specifically, the warp tile is decomposed to $TileK/4$

steps to be processed in serial by each warp, and each step processes a $64 \times V \times 4$ sub-tile (after the swapping). With guideline IV, ❹ is directly loaded to the register file as it has few reuse opportunities. For guideline V, as each column of consecutive 64 half operands in ❹ are mapped to 8 different threads and each thread holds consecutive 8 half operands, ❹ can be loaded with a single $LDG.128$ instruction and coalesced into four 128B coalesced global memory transactions. For ❷, as the LHS fragment in the warp tile is reused many times, it is directly loaded into the shared memory at the beginning of the tile, so it does not influence the memory access efficiency. Besides, the new mapping only requires $TileK$ being the multiple of 4, which is more friendly to residual handling.

3.4.4 Implementation Details

For an SpMM with size $M \times N \times K$, $TileN = 64$ and $CTA\ size = 32$ are set to have as many CTAs as possible while maintaining the best memory access pattern. Therefore, $\lceil M/V \rceil \times \lceil N/64 \rceil$ CTAs are launched, each processes an $V \times 64$ output tile.

To generate the output tile, each CTA traverses all the nonzero vectors in its row with stride $TileK$ and accumulates the partial sums in the register file. For each stride, all the threads first work jointly to load the LHS fragment in Figure 3.8 ❶ to shared memory. Then, each thread group loads its share in the 64×4 RHS fragment in Figure 3.8 ❹. Next, an *mma.m8n8k4* is launched to compute a $64 \times V \times 4$ matrix multiplication (❻×❸=❼). This is repeated for $TileK/4$ times until the warp tile is done.

To improve the instruction level parallelism (ILP), a *__threadfence_block()* is inserted between the $TileK/4$ load instructions and the $TileK/4$ *mma.m8n8k4* instructions. This prevents the *nvcc* compiler from reusing the registers that store the source operand of each *mma.m8n8k4*, which increases the dependencies between *mma* instructions and hurt ILP.

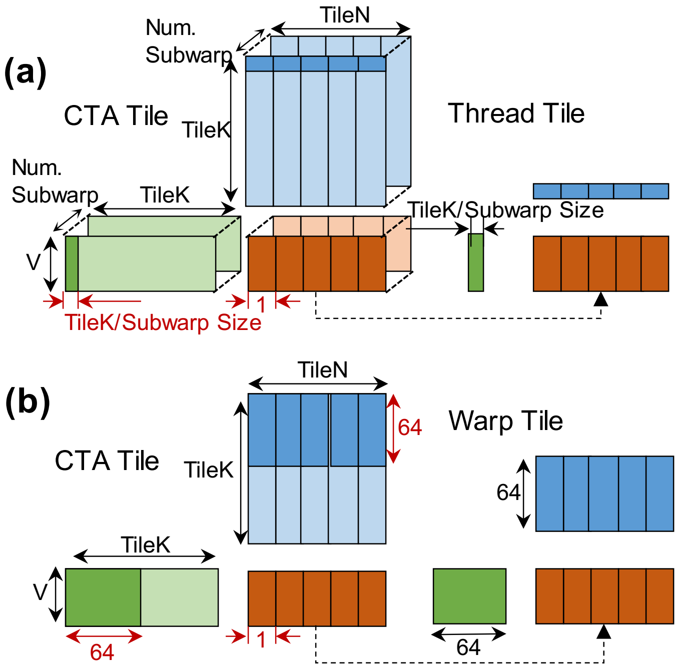


Figure 3.9: Decomposition of SDDMM with 1-D tiling. (a) The FPU tiling extended from Sputnik [49]; (b) TCU-based tiling.

When the last few nonzero vectors cannot fill the $TileK$ width, the load and computation of each $64 \times V \times 4$ tile will be interleaved until all the nonzero vectors are processed. This helps reduce the residual handling overhead. After all the nonzero vectors are processed, the data is reorganized with the warp shuffle primitives and then written to global memory with vectorized store.

3.5 Kernel-level Optimization: SDDMM

Similar to Section 3.4, this section first describes two baseline SDDMM designs. One is extended from Sputnik [49] that opts for memory access efficiency, and the other is based on the classic mapping between GEMM and TCU for high compute and kernel efficiency. Then, a novel design is present that achieves high EU-level utilization by satisfying all five guidelines.

3.5.1 Baseline I: FPU-based 1-D Subwarp Tiling

The FPU-based 1-D subwarp tiling is illustrated in Figure 3.9 (a). Similarly, each CTA tile contains multiple independent 1-D tiles assigned to subwarps. Each 1-D tile is decomposed to $Subwarp\ Size$ independent thread tiles with size $V \times TileN \times \frac{TileK}{Subwarp\ Size}$. Each thread loads its LHS and RHS tile into registers and computes partial sums. At last, partial sums stored by different threads in the same subwarp are reduced with warp shuffle.

This tiling design also has high memory access efficiency. Specifically, with $TileK = 64$ and $Subwarp\ Size = 8$, the rows in LHS fragment and columns in RHS fragment of the 1-D tile can be loaded with a single *LDG.128* instruction in the 128B coalesced pattern. Therefore, it satisfies guideline IV and V. However, kernel and compute efficiency are suboptimal due to the same reasons as in the FPU-based SpMM. Moreover, each thread holds a $V \times TileN$ array in the register file to store the partial sums. For instance, when $V = 8$ and $TileN = 32$, the partial sum consumes 256 registers of each thread, which exceeds the register file capacity and causes register spilling. Even without the register spilling, the large amount of registers reduces the occupancy.

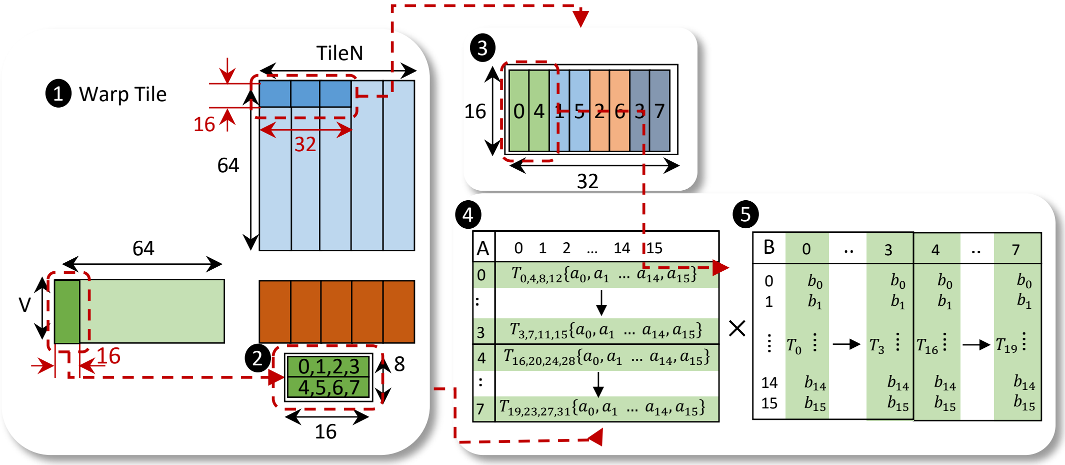

1 Warp Tile
TileN
16
32
64
64
V
16
2
0,1,2,3
4,5,6,7
8
16
3
16
0 4 1 5 2 6 3 7
32
4
A 0 1 2 ... 14 15
0 $T_{0,4,8,12}\{a_0, a_1 ... a_{14}, a_{15}\}$
3 $T_{3,7,11,15}\{a_0, a_1 ... a_{14}, a_{15}\}$
4 $T_{16,20,24,28}\{a_0, a_1 ... a_{14}, a_{15}\}$
7 $T_{19,23,27,31}\{a_0, a_1 ... a_{14}, a_{15}\}$
×
5
B 0 .. 3 4 .. 7
0 b_0
1 b_1
T_0 T_3 T_{16} T_{19}
14 b_{14}
15 b_{15}

Figure 3.10: Classic mapping of the warp tile to TCU.

3.5.2 Baseline II: TCU-based 1-D Warp Tiling

The TCU-based 1-D Warp Tiling for SDDMM is illustrated in Figure 3.9 (b). Each CTA tile has only one 1-D tile, which is further decomposed to warp tiles with size $V \times TileN \times 64$. Similarly, 64 is chosen because it is the smallest number that perfectly fills the 128B transaction. As shown in Figure 3.10 ❶, the warp tile is further processed with $\frac{TileK\times 64}{32\times 16}$ steps in serial, and each step is computed with a $wmma.m8n32k16$.

On the positive side, Similar to the SpMM kernel, it has high kernel and compution efficiency (guideline I, II, and III) for the same reasons. Besides, it uses fewer registers to store the partial sum. E.g., with $TileK = 64$, while the FPU-based implementation has *Subwarp Size* copies of the partial sums, this design only has one copy.

On the negative side, it has a suboptimal memory access pattern. Figure 3.10 ❷ and ❸ visualizes the operand layout for each *wmma.m8n32k16*. In ❷, each block represents a 4×16 block and the numbers on it indicate the thread groups that hold a copy. ❹ gives a detailed data layout for ❷. In ❸, each block represents a 16×4 block and the number on it represents the thread group that holds it. ❺ illustrates the data layout for thread group 0 and 4 as an example. If guideline IV is satisfied that the LHS fragment (❷) and RHS fragment (❸) are directly loaded into the register file, only 16B coalesced access can be achieved. In detail, the 16 operands in each row of the row-major ❷ and column of the column-major ❸ are consecutive, and they are mapped to the 16 registers of a thread. However, each $LDG.128$ can only load 8 of them. So it is 16B coalesced. On the other hand, To satisfy guideline V, the global memory access has to be coalesced through the shared memory, violating guideline IV.

Furthermore, the LHS fragment ❷ is copied 4 times, which consumes additional registers and reduces occupancy. Also, $TileN$ has to be a multiple of 32, which introduces additional overhead in residual handling. At last, redundant computations occurred when

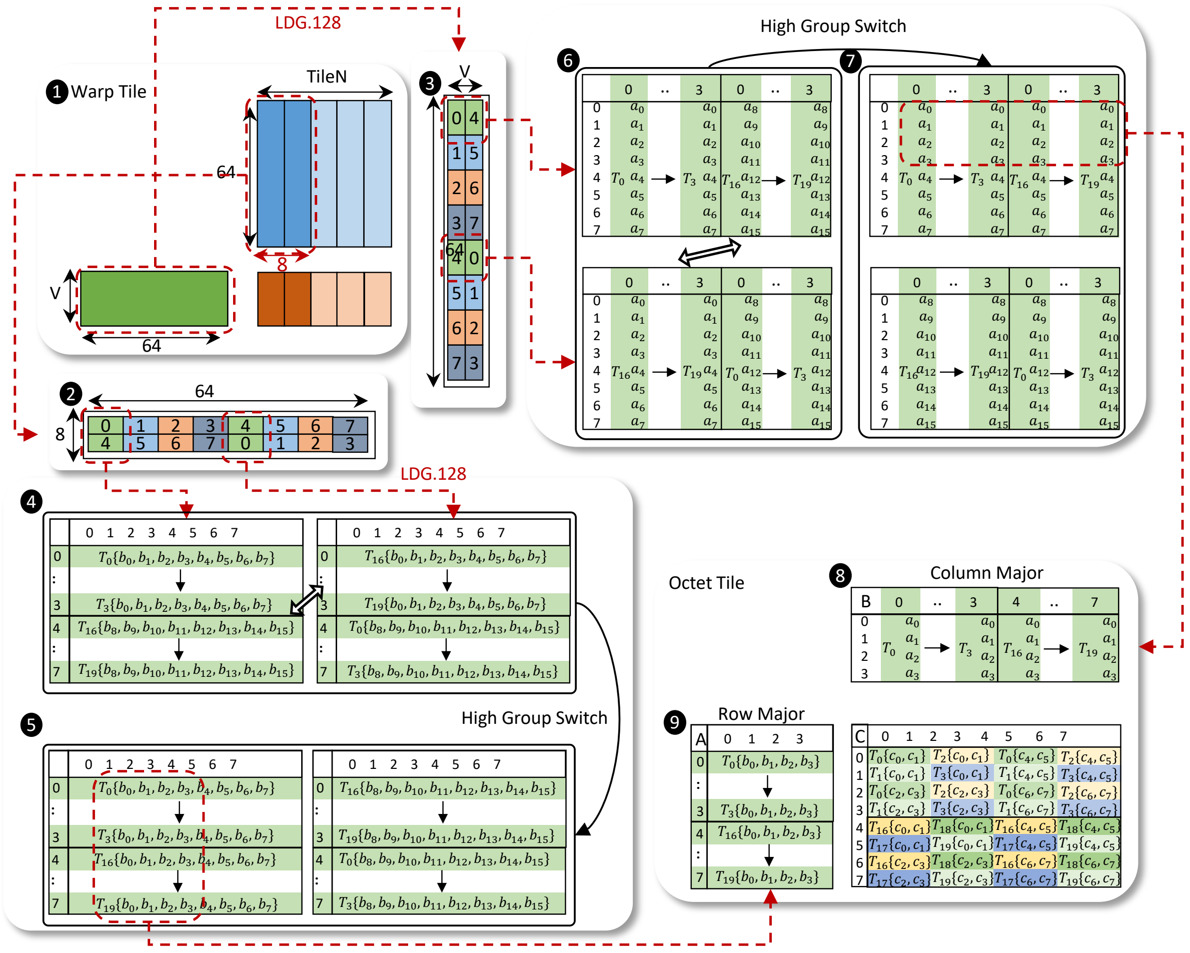


Figure 3.11: TCU based 1-D Octet Tiling for SDDMM.

V is smaller than 8.

3.5.3 Solution: TCU-based 1-D Octet Tiling

Based on the analysis above, a novel SDDMM kernel design is proposed that tackles the limitations of the previous two designs. Firstly, it adopts the same warp tiling as the TCU-based 1-D Warp Tiling in Figure 3.9 (b), which indicates good kernel and computation efficiency (guideline I, II, and III). To achieve optimal global memory access pattern, the mapping between the warp tile and the TCU is redesigned in the Octet granularity, namely TCU-based 1-D Octet Tiling.

The new mapping is shown in Figure 3.11. There are two major differences from the classic $m8n32k16$ mapping in Figure 3.10. Firstly, under the same motivation in Section 3.4.3, the LHS and RHS fragments are swapped to expose the opportunity of removing redundant *HMMA* in the SASS code when $V \leq 4$. Secondly, a novel warp tile partition is applied to guarantee efficient computation and memory access patterns. Specifically, the warp tile is decomposed to $TileN/8$ sub-tiles to be processed sequentially, and the size of each sub-tile is $8 \times V \times 64$ (after the switch). With guideline IV, the swapped LHS and RHS fragments are partitioned and stored in the register file of different thread groups, as shown in ❷ and ❸. Each block in ❷ and ❸ represents a $4{\times}8$ or $8{\times}4$ matrix held by a single thread group. Taking the thread group 0 and 4 as examples, the detailed data layouts are illustrated in ❹ and ❻. Under this setup, both ❷ and ❸ can be loaded with a *LDG.128* instruction and together generate eight 128B coalesced transactions. In detail, each row vector with length 64 of the row-major ❷ is partitioned to 8 sub-vectors with length 8, and different sub-vectors are loaded by different threads in the warp. This is the same in ❸. Besides, all the operands are only stored by a single thread group, whereas the LHS fragment in the TCU-based 1-D Warp tiling is copied 4 times.

The loaded operands in ❹ and ❻ cannot be directly used for *mma.m8n8k4*, as the register index of each row in ❹ and column in ❻ in thread group i and i+4 do not match. However, dynamically computing the indices based on the thread group index at runtime leads to local memory usage. To avoid this, the ”High Group Switch” is applied that switches the content in register j and $(j+8)Mod16$ in the thread group 4, 5, 6, and 7 (high group). The data layout after the High Group Switch is illustrated in ❺ and ❼, respectively. After the High Group Switching, each Octet has an $8 \times 8 \times 16$ tile to compute. While each $mma.m8n8k4$ can compute an $8 \times 8 \times 4$ tile, it takes 4 steps to finish the computation. Notably, the upper 4 rows in ❾ and the left 4 columns in ❽ are held by thread group 0 in step 1&2, but they are in thread group 4 in step 3&4. This

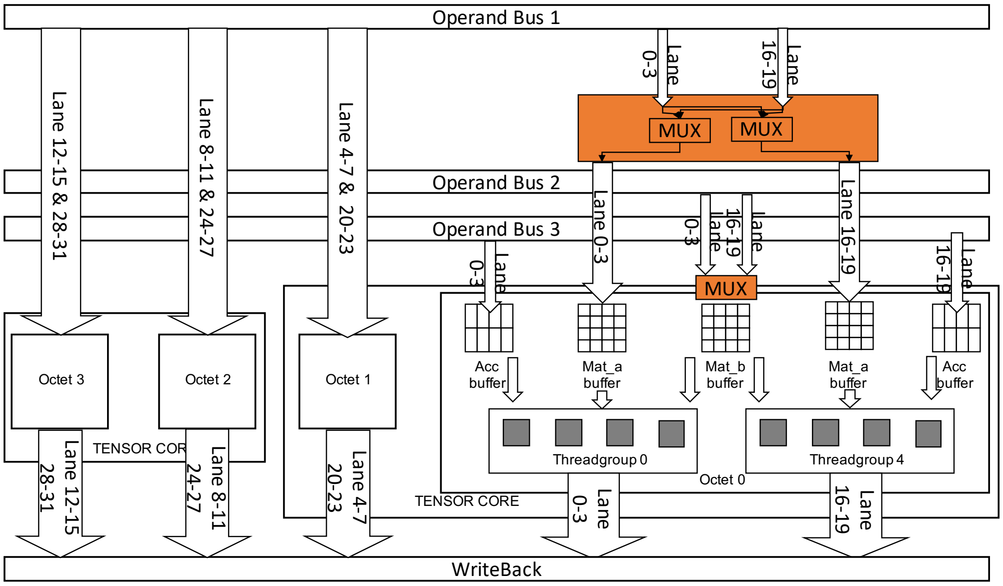


Figure 3.12: Proposed TCU Architecture.

inverted pattern of source operands also inverts the pattern of the output.

On the kernel side, the above problem can be solved by either shuffling the operands in thread group i and i+4 with the warp shuffle primitives before calling *mma.m8n8k4*, or using an additional set of registers to accumulate the partial sums from the last two steps. While the shuffle introduces additional overhead, the second solution reduces the occupancy as additional registers are used.

From the hardware perspective, an extension of the original *HMMA* instruction by adding a switch flag that directly switches the source operand in low and high groups within the TCU, i.e. *HMMA.884.F32.F32.STEP{0,1,2,3}.SWITCH*. To support this switch, as shown in Figure 3.12, a pair of multiplexers is added between the operand bus 1 and the Mat_a buffer of the two thread groups. This pair switches the source of the two Mat_a buffers if the switch is set. The source of the Mat_b buffer is switched by XORing the original control signal with the *SWITCH* bit.

3.5.4 Implementation Details

For an SDDMM with size $\boldsymbol{A}_{M\times K} \times \boldsymbol{B}_{K\times N} \odot \boldsymbol{D}_{M\times N} = \boldsymbol{C}_{M\times N}$, where $\boldsymbol{D}$ is a binary mask stored under the column vector sparse encoding, $TileK = 64$ and $CTA\ size = 32$ are set to reduce the residual processing overhead while maintaining the best memory access pattern. $TileN = 32$ is heuristically picked as it well balances the data reuse ratio and the number of CTA, but any multiple of 8 is acceptable. Therefore, $\lceil M/V \rceil \times \lceil N/32 \rceil$ CTAs will be launched, each processes an $V \times 32$ output tile.

To generate the output tile, each CTA traverses the dimension K with stride 64. Each octet holds the partial sums in its local registers. For each step, The LHS fragment in Figure 3.11 ❶ is loaded. Then, the warp takes four sub-steps, each sub-step load ❷ in Figure 3.11 and computes a $8 \times V \times 64$ tile (after the switching) as aforementioned.

This kernel has a tighter budget for the registers, as each octet holds at least one set of partial sums, and the compiler is relied on to determine the best strategy for register reusing. When K is traversed, the partial sums in different Octets are accumulated with warp shuffle primitives. The final result is reordered and written to global memory with the vectorized store.

3.6 Experiments

This section compares the performance of VecSparse with cuSPARSE and the FPU baseline extended from Sputnik [49]. Detailed profiling results on a representative benchmark are also discussed to justify the speedup and motivations.

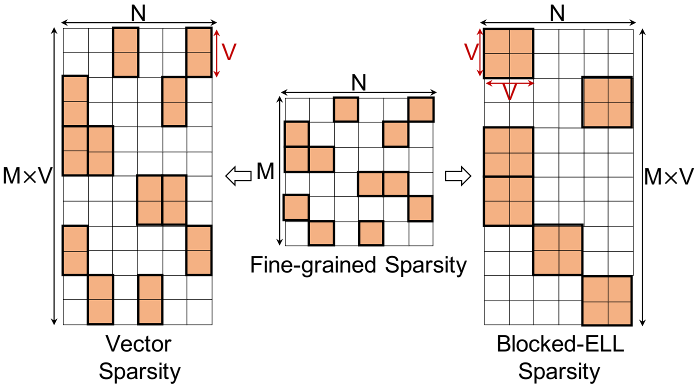


Figure 3.13: Benchmark Construction

3.6.1 Experiment Setup

Benchmark Construction. Figure 3.13 illustrates the procedure of constructing the benchmarks from ResNet 50 under magnitude pruning in the DLMC dataset [61]. Given a $M \times N$ sparse matrix from DLMC with sparsity S, as the column vector sparse encoding is equivalent to replacing the nonzero scalars with vector types, the *csrRowPtr* and *csrColInd* of the sparse matrices are used, coupled with randomly generate a nonzero vector with length V for each indexed position. To construct the blocked-ELL format sparse matrix, the block size is set to $V \times V$, and the number of blocks in each row is computed with $\lceil N/V \times S \rceil$. The column indices of the blocks are generated randomly under uniform distribution. In this way, the Blocked-ELL format has the same sparsity and problem size as the column vector sparse encoding.

Baseline Kernels. Both SpMM and SDDMM are compared with an FPU baseline and a TCU baseline. The FPU baseline is obtained by extending the kernels in Sputnik [49] following Section 3.4.1 and 3.5.1 to support the column vector sparse encoding. The tiling sizes are tuned on a subset of benchmarks to find a configuration that brings the highest geometric mean speedup. For the TCU baseline of SpMM, the Blocked-ELL-based SpMM kernel in cuSAPRSE is directly used. As the SDDMM under structured sparsity is not supported by off-the-shelf libraries, the kernel in Section 3.5.2 is used as the TCU baseline. Finally, the cuBLASHgemm kernel is selected as the dense baseline.

Removing HMMA instructions. While the tiling design in this chapter exposes opportunities for removing additional *HMMA*s, as existing SASS assemblers like [63] do not support this modification, it is left for future work.

3.6.2 SpMM

Figure 3.14 summarized the speedup achieved by the SpMM kernel. The distribution of the speedup achieved on different benchmarks is illustrated with the box plot. Furthermore, following Gale et al., 2020 [49], the geometric mean speedup is computed and visualized with the solid lines in Figure 3.14. The ”fpu”, ”blocked-ELL”, and ”mma” correspond to the FPU baseline extended from Sputnik[49], blocked-ELL based SpMM kernel in cuSPARSE, and the implementation with TCU-based 1-D Octet Tiling, respectively.

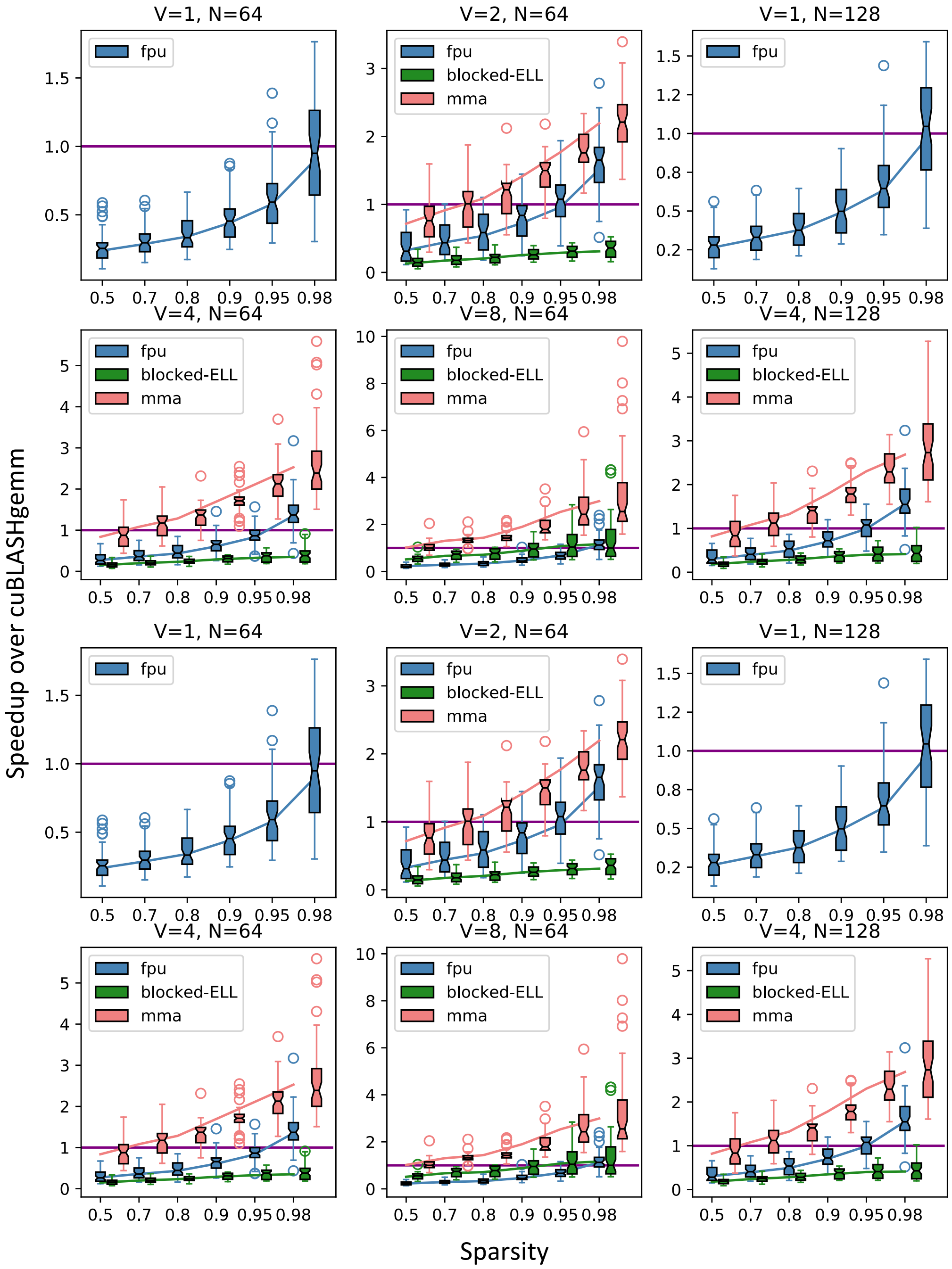


Figure 3.14: Speedup over cuBLASHgemm with SpMM under different configurations. The problem size is $M \times N \times K$. The size M and K are given in the benchmarks.

Overall Performance. Across all benchmarks, the TCU-based 1-D Octet Tiling (mma) achieves **1.34-4.51x** and **1.71-7.19x** geometric mean speedup over the FPU and

TCU baselines. Moreover, it outperforms cuBLASHgemm under > **80%**, > **70%**, and > **50%** sparsity under the tiny 2×1, 4×1, and 8×1 grain size. This illustrates that higher speedup can be achieved with larger vector length V, which justifies the motivation of achieving practical speedup under moderate sparsity with column vector sparse encoding.

Table 3.2: The five guidelines in different implementations.

Kernel	**No Inst**	**# CTA**	**Wait**	**Short Scoreboard**	**Sectors/Req**
SpMM, V=4					
MMA	1.1%	2048	4.7%	4.5%	12.56
CUDA	11.0%	2048	11.6%	2.6%	4.04
Blocked-ELL	42.6%	1024	21.0%	11.9%	14.92
SpMM, V=8					
MMA	1.1%	1024	6.2%	2.6%	13.22
CUDA	52.2%	1024	8.3%	2.0%	4.27
Blocked-ELL	35.1%	512	16.2%	12.1%	13.85

Utilization Analysis. To justify the above speedup, the SpMM kernels are further profiled and compared in terms of the five guidelines on $\boldsymbol{A}_{2048\times1024} \times \boldsymbol{B}_{1024\times256}$ under 90% sparsity, the results are summarized in Table 3.2. The percentage of pipeline stalls caused by "No Instruction", "Wait", and "Short Scoreboard" reflect guideline I, III, and IV, number of CTAs represents II, and Sector per Request to L1 cache reflects V. The results that are significantly worse than others are marked with red.

Compared with the TCU-based 1-D Octet Tiling, the FPU baseline suffers more from the "No Instruction" and "Wait" stalls. Examining the SASS code reveals that, under $V = 4$ and $V = 8$, the FPU baseline has 3776 and 6968 lines of code, and executes 3,402,752 and 3,407,872 *HUML+FADD* instructions, respectively. On the other hand, the proposed SpMM kernel has only 384 and 416 lines in the SASS code, with 429,504 and 215,104 executed HMMA instructions. This greatly reduces instruction cache miss and stalls for dependency on fixed latency instructions. Besides, the "Sectors/Req" of the FPU baseline is only around 4. This is because when tuning its tiling size, having

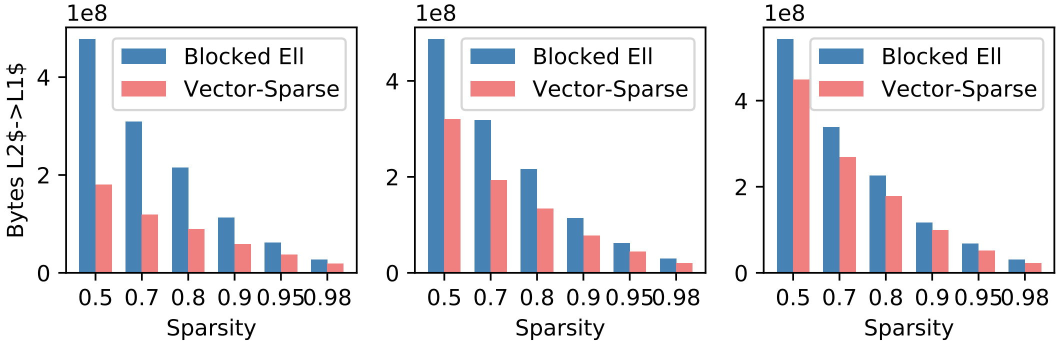


Figure 3.15: Total bytes loaded from L2 cache to L1 cache

$\#Subwarp = 1$ to improve the grid size generally improves the overall performance, while this comes at the cost of using shorter vector memory operation. The Blocked-ELL kernel suffers from ”No Instruction”, ”Wait”, and ”Short Scoreboard” stalls, which accords with the analysis in Section 3.2.

To justify the argument in Section 3.3 that the data reuse is independent of the column number in the block sparse matrix, Figure 3.15 shows the total number of bytes loaded from L2 cache to L1 cache under the VecSparse and blocked-ELL format, given the same problem size and sparsity. The VecSparse loads even fewer data from L2 to L1 cache than the Blocked-ELL format, across all the sparsity levels.

3.6.3 SDDMM

The speedup achieved by the proposed SDDMM kernels is summarized in Figure 3.16. The ”fpu” denotes the FPU baseline in Section 3.5.1, ”wmma” corresponds to the TCU baseline in Section 3.5.2. As three different methods are proposed to handle the inverted pattern, Figure 3.16 uses ”mma (reg)”, ”mma (shfl)”, and ”mma (arch)” to make the solution with additional accumulator buffers, shuffling the source operands, and new TCU architecture, respectively. For the last one, a fake kernel is developed to simulate the performance by assuming that the operands are swapped in TCUs.

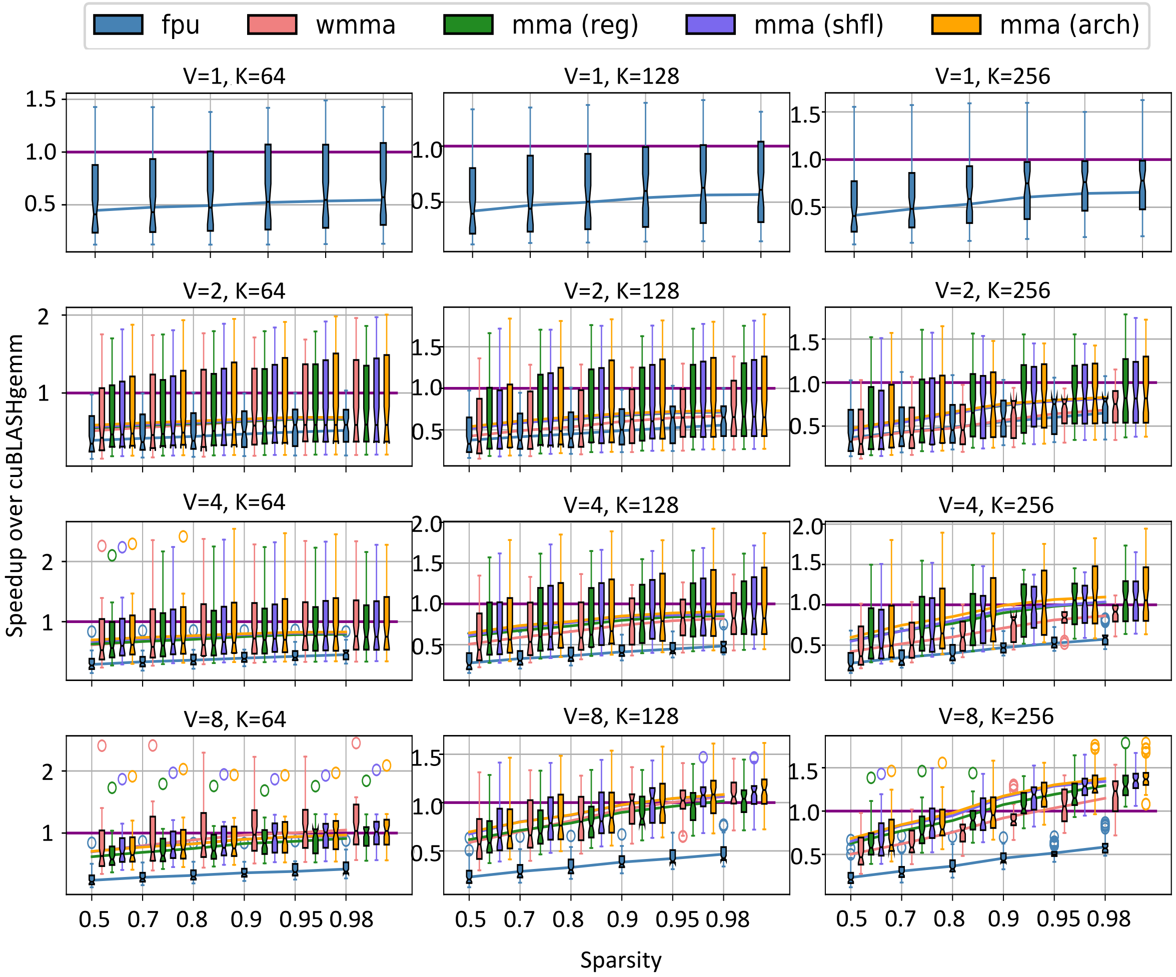


Figure 3.16: Speedup over cuBLASHgemm with SDDMM under different configurations. Problem size is $\boldsymbol{A}_{M\times K} \cdot \boldsymbol{B}_{K\times N} = \boldsymbol{C}_{M\times N}$, $\boldsymbol{C}$ is the sparse matrix under sparsity in {0.5,0.7,0.8,0.9,0.95,0.98}. The size M and N are given in the benchmarks, K is picked from $\{64, 128, 256\}$.

Overall Performance. The TCU-based 1-D Octet Tiling achieves considerable speedup over the baselines across all the setups except for $K = 64, V = 8$. Specifically, it achieves **1.27-3.03x** and **0.93-1.44x** geometric mean speedup over the FPU and TCU baselines. Besides, speedup at $>$ **90%** sparsity is achieved under $V = 8$ and $K = 256$. Moreover, with the modified TCU architecture, the mma (arch) consistently outperforms the mma (reg) and mma (shlf). This demonstrates that the proposed simple architecture optimization effectively improves performance.

The *SHFL* (Warp Wide Register Shuffle) + *FADD* accounts for 29.5% of the total instructions executed under $V = 8$ and $K = 64$ settings with the TCU-based 1-D Octet tiling. The percentage is reduced to 17.2% under $V = 8$ and $K = 256$. These instructions are primarily used for accumulating the partial sums of each Octet at the end of the kernel. This suggests that When K is small and V is large, the overhead of this reduction is not negligible and will offset the benefit of the dedicated tiling design. When K gets larger, it is obvious that the Octet tiling (mma) achieves significantly better performance.

Table 3.3: The 5 guidelines in different implementations.

Kernel	**No Inst**	**# CTA**	**Wait**	**Short Scoreboard**	**Sectors/Req**
SDDMM, V=4					
MMA	0.8%	16384	10.7%	2.1%	8.83
CUDA	6.1%	16384	28.1%	2.5%	3.53
WMMA	0.3%	16384	10.6%	14.4%	8.82
SDDMM, V=8					
MMA	1.0%	8192	11.0%	1.9%	9.25
CUDA	7.3%	16384	24.6%	3.1%	3.33
WMMA	0.4%	8192	9.5%	17.9%	9.26

Utilization Analysis. More detailed profiling results are summarized in Table 3.3 to justify the speedup achieved by the proposed SDDMM kernels. The setup is identical to 3.2 except that the benchmark size is $\boldsymbol{A}_{2048\times256} \times \boldsymbol{B}_{256\times1024} = \boldsymbol{C}_{2048\times1024}$ and $\boldsymbol{C}$ has 90% sparsity. As all three implementations of the MMAs are more or less similar in terms of these five guidelines, only the result of mma (reg) is listed.

Similarly, the FPU baseline suffers more from ”No Instruction” and ”Wait” stalls, and it has smaller ”Sector/req” than the other two implementations. The TCU baseline is limited by the shared memory bandwidth. These observations accord with the arguments in Section 3.5.

The mma (reg), mma (shfl), and mma (arch) are also compared to justify the architecture-level optimization. Given credit to our architecture modification, the mma (arch) uses 33% fewer registers and has 21.3% more active warps per scheduler than the mma (reg), as it removes the need for additional registers for partial sums from the inverted pattern. Besides, it has 10.4% fewer instructions, which is majorly contributed by the removed SHFL instructions for operands switching.

3.6.4 Application: Sparse Transformer

The SpMM and SDDMM kernels presented in this chapter can be leveraged to develop efficient sparse transformers. Under sequence length l and feature dimension m, the self-attention layer takes query, key, and value $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} \in \mathbb{R}^{l\times k}$ and computes the output with

$$\boldsymbol{A} = Softmax\left((\boldsymbol{Q}\boldsymbol{K}^T \odot \boldsymbol{C})/\sqrt{k}\right), Attention(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \boldsymbol{A}\boldsymbol{V}, \tag{3.1}$$

where $\boldsymbol{C}$ is an optional sparse mask that prunes most of the entries in the matrix $\boldsymbol{A}$. Many studies have been proposed to apply a fixed or learned sparse mask $\boldsymbol{C}$ [37]. Under this setup, $\boldsymbol{Q}\boldsymbol{K}^T \odot \boldsymbol{C}$ and $\boldsymbol{A}\boldsymbol{V}$ can be formulated as SDDMM and SpMM, respectively. However, without efficient GPU kernels, they could be much slower than their dense counterpart, thus previous studies [64] apply block sparsity with large block sizes like 32 or 64. In contrast, the SDDMM and SpMM kernels proposed in this chapter are capable of achieving speedup over the dense implementation under much smaller granularity, offering larger design space when constructing the sparse masks.

Experimental Setup: The transformer model is trained with a fixed sparse attention mask on the byte-level text classification task in Long-Range Arena (LRA) [65], a benchmark for transformers under long-sequence scenarios. In this task, the sequence length is 4000. The model configuration and training parameters are set to be the same as the original dense baseline. The 4-layer transformer model has four attention heads for each attention layer, and the feature dimension of each head is 64.

The sparse attention pattern is designed by adding the 8×1 vector sparsity constraints to the pattern proposed by Gale et al., 2020 [49]. Specifically, the fixed attention masks contain a dense band of size 256 along the diagonal and off-diagonal random attention. The overall sparsity is 90% and the attention mask can be expressed by the column-vector sparse encoding. A custom softmax kernel that works on column vector sparse encoding is also implemented. The half-precision models are directly quantized from the full-precision ones without fine-tuning. The inference throughput and peak memory usage under batch size 8 are evaluated. The result averaged over 10 runs is reported in Table 3.4.

Table 3.4: Sparse Transformer Results

Model	**Dense(float)**	**Dense(half)**	**Sparse(half)**
Accuracy	65.12%	65.09%	65.01%
Throughput (seq / s)	74.7	182.6	258
Peak Memory	4.44 GB	2.22 GB	170.03 MB

Results & Analysis: As shown in Table 3.4, the sparse model under half precision achieves 3.45x and 1.41x end-to-end speedup over the dense model under single and half-precision, respectively. The peak memory usage is reduced by 26.74x and 13.37x. With vector-wise sparsity, the accuracy degradation is only 0.11% of the dense baseline.

The speedup achieved in different parts of the attention layer is shown in Figure 3.17. In terms of the whole layer, $1.35 - 1.78\times$, $1.48 - 2.09\times$, and $1.57 - 2.30\times$ speedup are achieved under 90%, 95%, and 98% sparsity, respectively. The proposed SpMM and Softmax kernels effectively reduce the latency contributed by $Softmax$ and AV. The SDDMM kernel is slower than its dense counterparts when $k = 64$. It is because 64 is too small, which accords with the observation in Figure 3.16. Notably, the sparsity cannot be utilized in $Softmax$ and AV without the SDDMM kernel, and the SDDMM achieves

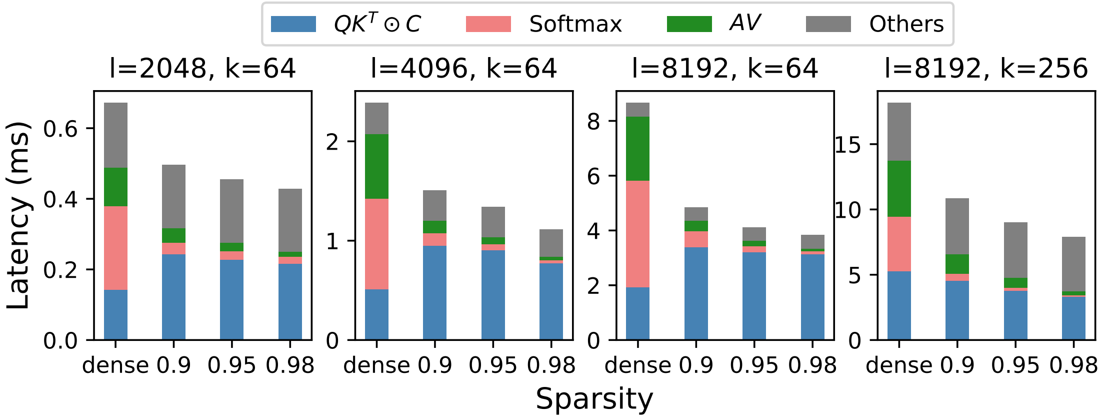


Figure 3.17: Latency of self-attention layer in various setups.

better performance than other baselines. Besides, as shown in the last figure in Figure 3.17, the SDDMM outperforms its dense counterpart when $k = 256$.

3.7 Discussion & Conclusion

Other Applications. While the SpMM and SDDMM kernels in this chapter are defined in terms of row-major matrices, they can also be applied to column-major matrices by mathematically transposing both LHS and RHS of the equation. In detail, the SpMM and SDDMM can be formulated as $\boldsymbol{D}^T = \boldsymbol{B}^T\boldsymbol{C}^T$ and $\boldsymbol{D}^T = (\boldsymbol{B}^T)^T\boldsymbol{A}^T \odot \boldsymbol{C}^T$, where $\boldsymbol{D}^T$, $\boldsymbol{A}^T$, and $\boldsymbol{B}^T$ are column-major dense matrices. $\boldsymbol{C}^T$ is a transposed sparse matrix under column-vector sparse encoding, which can be viewed as "row vector sparse encoding" that is composed of short row vectors aligned along the horizontal dimension. The position of these short-row vectors is encoded in the compressed sparse column (CSC).

Although VecSparse only requires the sparse matrix to be composed of short column vectors aligned along the vertical dimension, additional constraints can be added along the horizontal dimension in need. While additional adjustments can be applied to the way that operands are indexed, the CTA tile remains identical under different constraints, so the kernel designs are still applicable. Besides the case with the default setting provided in Section 3.6.4, it can also be applied to neural network training with sparse weight.

In detail, with the activations $\boldsymbol{X} \in \mathbb{R}^{n \times N}$ in row-major, where n is the input feature dimension and N is the batch size, the forward and backward passes can be formulated as

$$\text{❶}\ \boldsymbol{Y} = \boldsymbol{W}\boldsymbol{X}, \quad \text{❷}\ \frac{\partial \mathcal{L}}{\partial \boldsymbol{X}} = \boldsymbol{W}^T \frac{\partial \mathcal{L}}{\partial \boldsymbol{Y}}, \quad \text{❸}\ \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{Y}} \boldsymbol{X}^T. \tag{3.2}$$

❶ and ❷ in the above equation can be computed with the SpMM kernel, and SDDMM kernel is applicable in ❸. As both $\boldsymbol{W}$ and $\boldsymbol{W}^T$ are used, square nonzero blocks are required that are aligned in both vertical and horizontal dimensions. Both $\boldsymbol{W}$ and $\boldsymbol{W}^T$ can be encoded with the column vector sparse encoding.

Conclusions. This chapter presents an algorithm-kernel codesign solution that accelerates neural networks when sparsity and reduced precision are both present. On the algorithm size, the column vector sparse encoding is introduced. It achieves the same data reuse as block sparsity while delivering smaller granularity to help maintain neural network model quality. On the kernel side, a novel mapping strategy, namely TCU-based 1-D Octet Tiling, enables the design of SpMM and SDDMM kernels that achieve both efficient memory access and computation under tiny sparse granularity. Experiments on the DLMC sparse matrix benchmark illustrate that the proposed kernels achieve **1.71-7.19x** geometric mean speedup over the Blocked-ELL-based SpMM kernel. Moreover, the SpMM and SDDMM kernels achieve practical speedup over their dense counterparts, with $>$ **70%** and $>$ **90%** sparsity under the 4×1 grain size and half-precision. Benefiting from the proposed design, 1.41x end-to-end speedup and 13.37x peak memory reduction are achieved on the sparse transformer inference task.

Chapter 4

Algorithm-Kernel Codesign: GPGPU-friendly Sparse Attention

This chapter presents another algorithm-kernel codesign example of kernel-centric optimization. It improves the algorithm-level utilization of transformers on GPGPUs with DFSS by leveraging the inherent dynamic and fine-grained sparsity within the attention weight matrix. The GPGPU-friendly design of DFSS offers much higher algorithm and EU-level utilization compared with existing sparse attention mechanisms.

4.1 Introduction

Transformers [13] have achieved competitive performance across various domains like NLP [66] and Computer Vision [67]. The key feature that sets them apart from traditional neural network architectures is the attention mechanism [13], which allows the transformers to gather information from the embeddings of elements in the input sequence in an adaptive and learnable manner.

Nevertheless, the high computation cost and memory footprint brought by the attention mechanism make it difficult to apply transformers to latency-sensitive tasks. On the other hand, the inherent sparsity in the attention weight matrix leads to low algorithm-level efficiency, as only a few of the entries in the attention weight matrix are critical for model accuracy.

However, improving the algorithm-level utilization of attention through sparsity is challenging. Unlike traditional static weight sparsity, the sparsity in attention is dynamic and fine-grained. It is dynamic as the attention score matrix is computed from input activations, so it is different for every inference sample. It is fine-grained because any entries in it can have dominant value. Unfortunately, there is a dilemma between dynamic and fine-grained regarding the utilization of GPGPU. On one hand, dynamically pruning, encoding, and decoding the sparse attention score matrix on the fly require regular sparse patterns to achieve high utilization on GPGPU. On the other hand, an irregular sparse pattern is desired to catch the fine-grained distribution of dominant entries. Previous studies [37, 38, 39, 68, 42, 41, 69] use coarse-grained block sparsity, which allows for efficient dynamic encoding and decoding but scarifies the fine-grained property. Thus, they require training from scratch or retraining to force the dominant entries to follow the heuristic or learned patterns, which can be time-consuming even for small-scale transformer models. Other approaches, such as low-bit approximation or machine-learning-based prediction, are not GPGPU-friendly or guaranteed to be accurate, leading to low utilization.

This chapter directly tackles the dilemma between "dynamic" and "fine-grained" with an algorithm-kernel codesign approach called Dfss, which *dynamically* prunes the full attention score matrix using N:M fine-grained structured sparse patterns.

At the algorithm level, the key insight is that on the dynamic side, the N:M sparsity, originally designed for efficient decoding of sparse matrix only, makes the dynamic pruning and encoding easy to parallel on GPGPU. On the fine-grained side, the N:M sparsity preserves the largest entries in each row by preserving the local maximum of each M entries. DFSS carefully selects the position to introduce sparsity to maximize the benefits while avoiding the introduction of auxiliary or GPU-unfriendly operators.

At the kernel level, a new dynamic sampled dense-dense matrix multiplication (SDDMM) kernel is developed that multiplies the query and key matrices, dynamically prunes it to N:M sparsity with magnitude, and encodes the compressed nonzeros and metadata. Moreover, the compressed sparse kernel can be consumed by succeeding kernels for speedup and memory footprint reduction. It is the first kernel in the deep learning software stack that prunes and encodes sparse matrices without overhead.

Compared with previous studies, DFSS achieves speedup on GPGPUs in arbitrary sequence lengths. It only requires a few fine-tuning epochs (a few GPGPU-hours) from pre-trained dense models like BERT-large to achieve on-par accuracy. While this chapter focuses on 1:2 and 2:4 structured sparsity as decoding of them is supported by the sparse tensor cores on A100 GPGPUs [21], the techniques and observations can be applied equally to all N and Ms with the hardware support. The main **contributions** are as follows:

- At the algorithm level, this chapter proposes DFSS, a *dynamic* N:M sparse attention mechanism that is a drop-in replacement of the full attention mechanism and orthogonal to existing efficient attention mechanisms. Its effectiveness is justified by both empirical and theoretical evidence. It is the first *dynamic* pruning technique under N:M sparsity.

- At the kernel level, this chapter presents a dedicated CUDA kernel design to completely remove the pruning overhead. The pruning is implemented as an epilogue of the dense matrix multiplication, which produces the attention score matrix. It is the first kernel in the deep learning software stack that can be applied for dynamic and fine-grained sparse matrices and generates compressed sparse encoding with zero overhead.

- Dfss is evaluated on tasks across various domains and sequence lengths on A100 GPGPU. It achieves 1.38∼1.86× speedup over the full attention with no accuracy loss.

4.2 Algorithm-level Optimization: DFSS

Dfss addresses the "dynamic" and "fine-grained" challenges with two algorithm-level design choices: (1) location to generate the sparse pattern (2) structure of the sparse pattern. This section first gives an overview of the Dfss method. Then, two design choices are discussed in detail. Besides, Appendix A.1 offers a more theoretical analysis and Appendix A.4 discusses how to combine Dfss with existing linear attention.

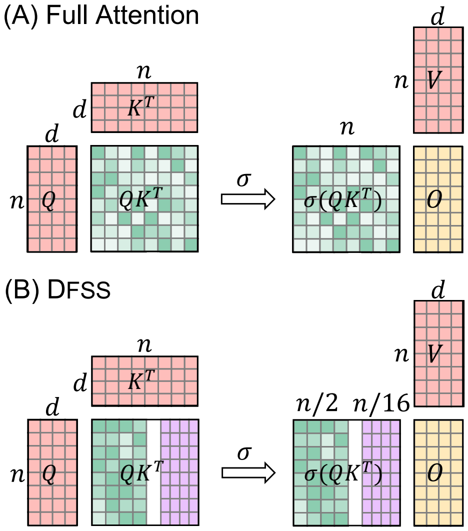


Figure 4.1: Overview of the Dynamic N:M Fine-grained Structured Sparse Attention Mechanism with N:M=1:2 or 2:4. σ represents *softmax*, $\frac{1}{\sqrt{d}}$ is omitted for simplicity.

4.2.1 Overview

The DFSS mechanism is simple and effective. Figure 6.1 illustrates the case under 1:2 or 2:4 sparsity. Compared with the full-quadratic attention mechanism, DFSS dynamically prunes attention scores without incurring storage or computation overhead while maintaining the effectiveness of attention. More importantly, it can achieve practical speedups of attention on existing GPU hardware with customized CUDA kernels. Listing 1 shows all the modifications to be made to use DFSS.

```
# Full attention mechanism
import torch.nn.functional as F
def full_attention(q,k,v):
    attn_weight = torch.bmm(
        q, k.transpose(1, 2))
    attn_weight = F.softmax(attn_weight, -1)
    return torch.bmm(attn_weight, v)

# DFSS attention mechanism
import dfss
def dfss_attention(q,k,v):
    attn_weight, metadata = dspattn.sddmm(q, k)
    attn_weight = F.softmax(attn_weight, -1)
    return dfss.spmm(
        attn_weight, metadata, v)
```

Listing 1: Example of using DFSS.

4.2.2 Design Choice I: Location to Generate Sparsity

The compute graph of the attention mechanism is illustrated in Figure 4.2. It can be considered as three stages: $\boldsymbol{QK}^T$, $Softmax$, and $\boldsymbol{AV}$. There are three locations where the sparse pattern can be generated: ⓿ before computing $\boldsymbol{QK}^T$. ❶ while computing $\boldsymbol{QK}^T$. ❷ while applying softmax.

At ⓿, the dense matrix multiplication between $\boldsymbol{Q}$ and $\boldsymbol{K}$ will be replaced with the traditional static sampled dense-dense matrix multiplication (SDDMM) which only

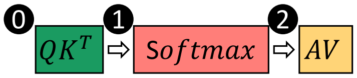


Figure 4.2: Attention Stages.

computes the entries identified by the input sparse pattern. The $Softmax$ only operates on the nonzero values in each row. The original dense matrix multiplication between $\boldsymbol{A}$ and $\boldsymbol{V}$ will be replaced with a sparse matrix-matrix multiplication (SpMM) which multiplies a sparse matrix with a dense matrix. However, it is not possible to exactly know which entry in $\boldsymbol{QK}^T$ has higher magnitude before computing $\boldsymbol{QK}^T$. Therefore, additional components that predict the location of important entries are required at ⓿. These auxiliary components would reduce the algorithm-level utilization. Besides, replacing the dense $\boldsymbol{QK}^T$ with traditional SDDMM offers limited speedup even at high sparsity. Chen et al. 2021 [53] show that it is difficult for traditional SDDMM to achieve speedup over its dense counterpart under 80% sparsity even with some structured design.

At ❶, a dynamic SDDMM kernel is required which generates the sparse pattern based on the magnitude of entries in $\boldsymbol{QK}^T$ and produces the pruned and compressed attention score matrix. The benefit is that the important entries can be explicitly selected from $\boldsymbol{QK}^T$ without prediction. As the softmax is a monotonically increasing function, starting from ❷ does not offer any benefits over ❶ but throws away the opportunity to accelerate $Softmax$.

Dfss selects ❶ based on two considerations. First, ❶ keeps the design simple such that it does not introduce additional overhead or hyperparameters to tune. Second, the dynamic SDDMM kernel developed in the next section runs faster than dense $\boldsymbol{QK}^T$ under even 50% sparsity.

4.2.3 Design Choice II: Structure of Sparsity

To achieve speedup while maintaining accuracy, structured constraints are required on the sparse pattern that makes it satisfy two requirements. First, it should be friendly to GPGPU while pruning, encoding, and consuming the sparse matrix, ensuring high EU-level utilization. Second, the pattern should preserve the largest values in each row.

Sparse patterns used by previous studies do not meet these two requirements. The unconstrained row-wise top-k sparsity violates the first requirement, as it requires comparisons between elements generated by different threadblocks in the GEMM kernel. So its pruning and encoding are hard to parallel. Besides, the popular compressed sparse row (CSR) based SpMM requires over 95% sparsity to be on par with its dense counterpart [53], which is too high to preserve accuracy. The block sparsity widely used in previous studies violates the second requirement, as it cannot guarantee the selected block contains the largest value of all the rows it covers.

The N:M fine-grained structured sparsity (Section 2.2.3), originally designed for static weight pruning, satisfies these two requirements. For requirement 1, it is easy to parallel during pruning and encoding. As the N:M selection is performed locally and the address to write nonzeros and metadata is deterministic. Speedup can be easily achieved when it is consumed by succeeding kernels. As the row length is reduced to N/M of the original attention score matrix, the softmax kernel achieves speedup without modification. Powered by the NVIDIA Sparse Tensor Core, the SpMM with value matrix also achieves 1.7× speedup with $N : M = 1 : 2$. For requirement 2, selecting N elements in each $1 \times M$ vector guarantees to preserve the largest entries in each row. While this chapter focuses on 1:2 and 2:4 sparsity in this paper as they are supported by the off-the-shelf GPUs, other N:M ratios are also supported given the hardware support for multiplication between an N:M sparse matrix and a dense matrix.

Table 4.1: F1 Score w/o Finetune on SQuAD v1.1

Full	**1:2**	**2:4**
93.17 ± 0.27	92.86 ± 0.22	93.00 ± 0.16

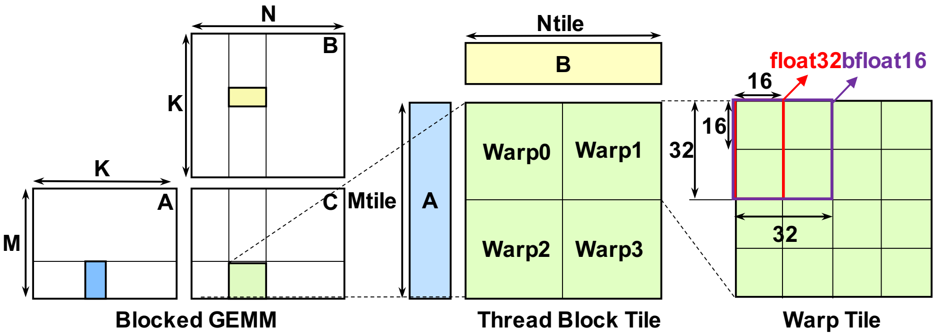


Figure 4.3: Dense Matrix Multiplication Tiling Design.

Empirically, the N:M sparsity can well approximate the full attention mechanism. Given a BERT-large model fine-tuned on SQuAD v1.1 under full attention, Table 4.1 summarizes the F1 score achieved by 1:2 and 2:4 sparsity without further fine-tuning. The accuracy loss is only around one sigma even without finetuning.

4.3 Kernel-level Optimization: Dynamic SDDMM

This section presents the design of the dynamic fine-grained SDDMM kernel. The proposed kernel fuses dense GEMM, magnitude-based pruning under N:M sparsity, and compression into a single kernel by developing a new dynamic pruning epilogue for CUTLASS [70] GEMM mainloop. The tiling is shown in Figure 4.3. The fusion of these three stages saves two round trips between global memory and the registers. Additionally, a dedicated design of thread and register mapping is proposed that ensures all memory access is 128B coalesced and reduces cross-thread shuffling. These optimizations allow for efficient implementation of dynamic SDDMM under N:M sparsity on GPGPU.

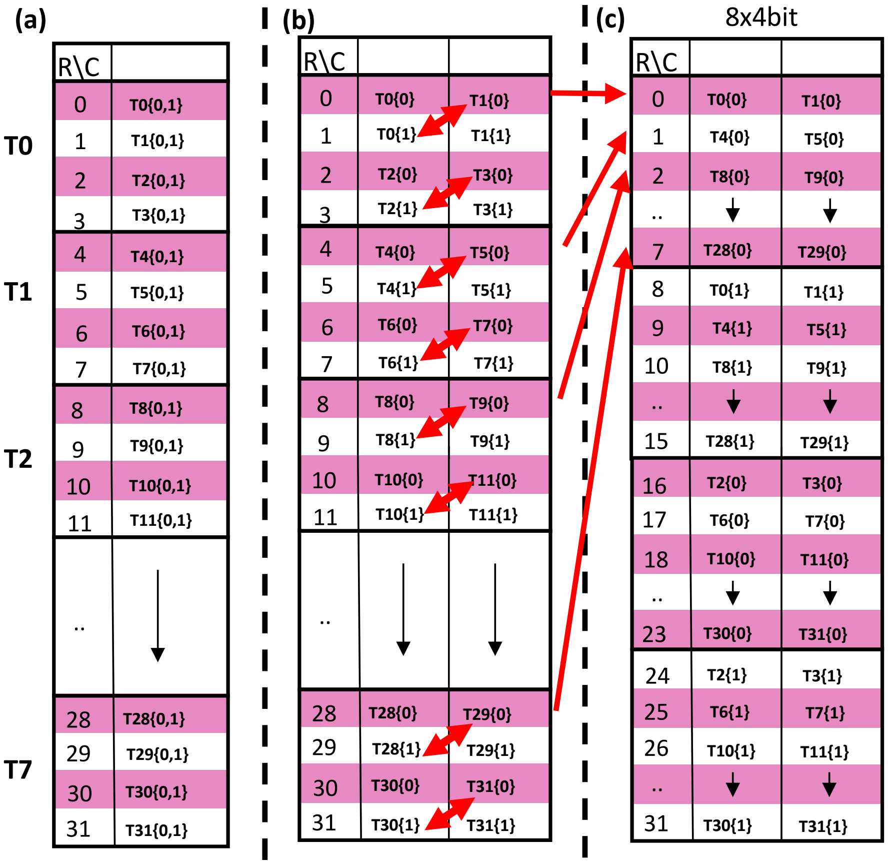


Figure 4.4: Encoding of metadata in CUTLASS [70]. $T\{threadIdx\}\{regIdx\}$ represents the $regIdx^{th}$ 16-bit register in thread $threadIdx$.

4.3.1 Decoding the Sparse Matrix in SpMM Kernel

The sparse matrix encoded should be easy for the succeeding SpMM kernel to decode. Figure 4.4 (c) shows the thread map of metadata used by two *mma.sp.m16n8k16* with tf32 type or *mma.sp.m16n8k32* with f16, bf16 types under the Tensor Core's hardware constraints. Each row has eight 4-bit metadata. The mapping is annotated with $T\{threadIdx\}\{regIdx\}$, representing the data is held in register $regIdx$ of thread $threadIdx$. Each $regIdx$ represents a 16-bit register. The threads satisfying $\lfloor(threadIdx \ mod \ 32)/2\rfloor mod \ 2 == 0$ hold the first 16 rows while the second 16 rows are stored in others. The *mma.sp* instruction has an argument to select between the first and second 16 rows.

Directly loading the metadata under this thread mapping will lead to an inefficient memory access pattern: each thread only loads 16 bits per instruction and the shared memory bank conflict occurs. To address this issue, CUTLASS [70] leverages the *ldmatrix* instruction. As shown in Figure 4.4 (a), thread $k \in [0, 7]$ loads 128-bit data, breaks it into four 32-bit data, and distributes them to thread 4k $\sim$ 4k+3 with a special data path in hardware. As shown in Figure 4.4 (b) and (c), to match the thread mapping of *ldmatrix* with *mma.sp*, two additional transformations are required: (1) switch the sub-diagonal entries in every 2×2 block (2) interleave the rows by 4 in every 32 rows. CUTLASS eliminates these two transformations by performing their counter-transformations when encoding the metadata offline: (1) interleave the rows by 8 in every 32 rows (2) switch the sub-diagonal entries in every 2×2 block. Therefore, only a single *ldmatrix* is required to load the metadata under desired thread mapping.

4.3.2 Dynamic Pruning Epilogue

The decoding process of the sparse matrix in SpMM kernel brings two constraints to Dfss's dynamic SDDMM Kernel. First,the two counter-transformations are performed when encoding the metadata. Second, As the counter-transformation occurs in a range of 32×16 with tf32 type and 32×32 with f16/bf16 types, the minimum pruning unit is 32×64-byte.

With these two constraints, the proposed dynamic pruning epilogue is shown in Figure 4.5. There are four major steps: ⓿ Prune 50% of each consecutive 8B data, generate nonzeros and metadata; ❶ counter-transformation (1): Interleave the metadata rows by 8; ❷ counter-transformation (2): Switch the metadata along sub-diagonal. ❸ Write metadata and nonzeros to global memory.

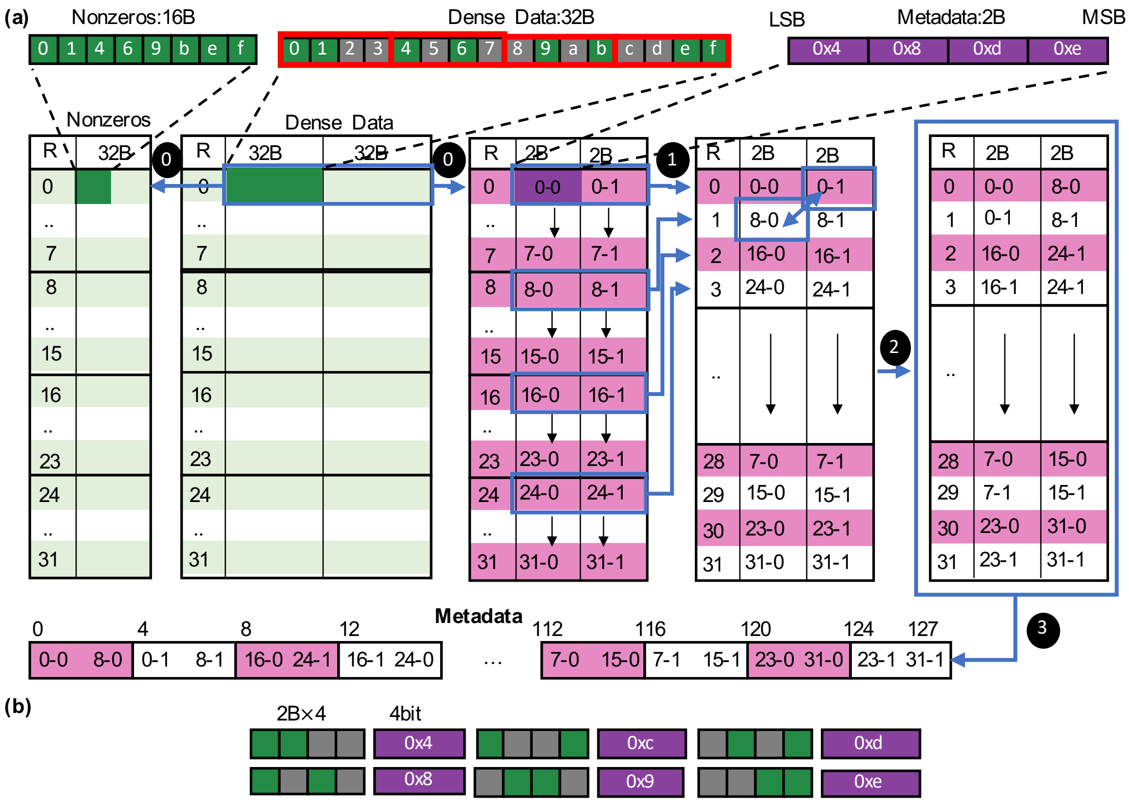


Figure 4.5: Prune dense data and generate nonzeros, metadata.

In detail, two out of four 2-byte data are selected based on their magnitude and a unique 4-bit metadata is assigned to each combination in ⓿. The corresponding metadata of the selection pattern is enumerated in Figure 4.5 (b). Notably, with float32 data type, each 32-bit data occupies two consecutive 2-byte slots. Therefore, it only supports the patterns under 0x4 and 0xe. After generating the 4-bit metadata, consecutive four of them are concatenated to a 2B metadata block. The rows of metadata are interleaved by 8 in ❶ with

$$dst\_row = \lfloor row/32 \rfloor \times 32 + (row\%8) \times 4 + \lfloor (row\%32)/8 \rfloor. \tag{4.1}$$

In ❷, the metadata blocks at the upper right and lower left of each 2 × 2 grid are switched. At last, in ❸, the metadata produced by ❷ is written into global memory under the interleaved column-major format with stride 4-byte. This can be realized by interpreting two consecutive metadata as an int object and then writing it to global memory in column-major. The nonzeros are written to global memory under row-major.

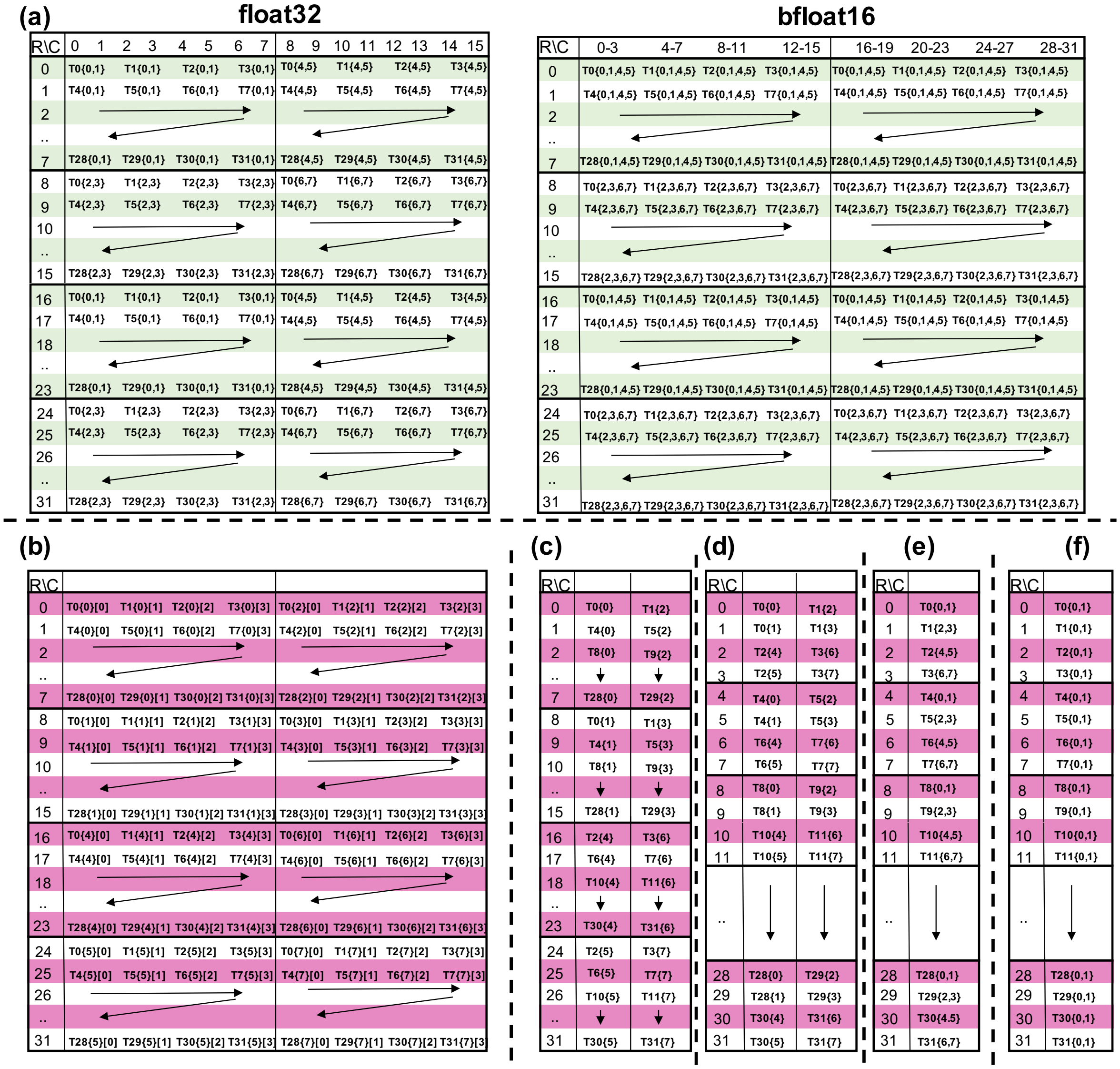
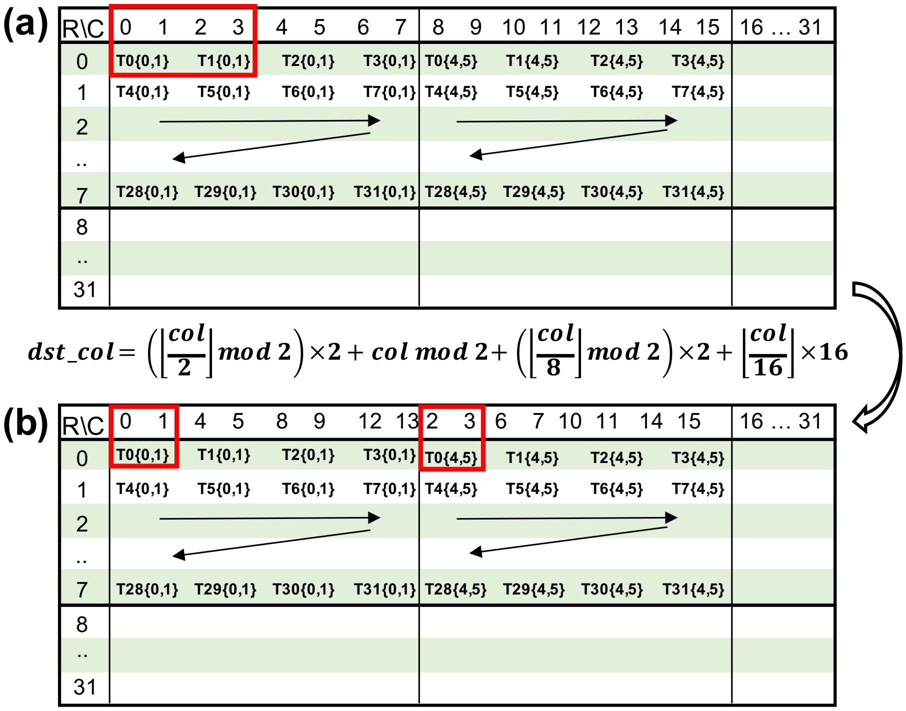

(a)
float32
bfloat16
R\C
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0-3 4-7 8-11 12-15 16-19 20-23 24-27 28-31
T0{0,1} T1{0,1} T2{0,1} T3{0,1} T0{4,5} T1{4,5} T2{4,5} T3{4,5}
T0{0,1,4,5} T1{0,1,4,5} T2{0,1,4,5} T3{0,1,4,5}
T0{2,3,6,7} T1{2,3,6,7} T2{2,3,6,7} T3{2,3,6,7}
(b)
T0{0}[0] T1{0}[1] T2{0}[2] T3{0}[3] T0{2}[0] T1{2}[1] T2{2}[2] T3{2}[3]
(c)
T0{0} T1{2}
(d)
T0{0} T1{2}
(e)
T0{0,1} T1{2,3}
(f)
T0{0,1} T1{0,1}

Figure 4.6: Mapping between the threads, registers, and data, metadata.

4.3.3 Thread and Register Mapping

In the pruning step, the warp tile is partitioned to a grid of $32 \times 64B$ blocks that are processed by the warp one at a time.

Pruning the GEMM result. In order to reduce cross-thread data access, all the comparisons are local to each thread during pruning. The thread and register mapping in the $32 \times 64B$ block to be pruned is shown in Figure 4.6 (a), $T threadId\{regId\}$ indicates the $regId$ 32-bit / 16-bit register of thread $threadId$ with float32 type and bfloat16 type, respectively. Under float32 type, the larger one is selected in the adjacent two entries. Under bfloat16 type, 2 larger ones are selected from the adjacent 4 entries.


(a)
R\C
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ... 31
0 T0{0,1} T1{0,1} T2{0,1} T3{0,1} T0{4,5} T1{4,5} T2{4,5} T3{4,5}
1 T4{0,1} T5{0,1} T6{0,1} T7{0,1} T4{4,5} T5{4,5} T6{4,5} T7{4,5}
2
..
7 T28{0,1} T29{0,1} T30{0,1} T31{0,1} T28{4,5} T29{4,5} T30{4,5} T31{4,5}
8
..
31
(b)
R\C
0 1 4 5 8 9 12 13 2 3 6 7 10 11 14 15 16 ... 31
0 T0{0,1} T1{0,1} T2{0,1} T3{0,1} T0{4,5} T1{4,5} T2{4,5} T3{4,5}
1 T4{0,1} T5{0,1} T6{0,1} T7{0,1} T4{4,5} T5{4,5} T6{4,5} T7{4,5}
2
..
7 T28{0,1} T29{0,1} T30{0,1} T31{0,1} T28{4,5} T29{4,5} T30{4,5} T31{4,5}
8
..
31

$$dst\_col = \left(\left\lfloor \frac{col}{2} \right\rfloor mod\ 2\right) \times 2 + col\ mod\ 2 + \left(\left\lfloor \frac{col}{8} \right\rfloor mod\ 2\right) \times 2 + \left\lfloor \frac{col}{16} \right\rfloor \times 16$$

Figure 4.7: Interleave the columns for matrix B to reduce cross-lane data sharing during pruning for bfloat16.

While the output mapping of *mma* in Ampere GPU matches Figure 4.6 (a) under float32 type, the output mapping under bfloat16 type in Figure 4.7 (a) is inconsistent with Figure 4.6 (b). Therefore, additional warp shuffles are required to first pass these

4 entries to the same thread, then compare them and obtain the 2 larger ones. This will introduce additional overhead. To solve this problem, when loading matrix B to shared memory, the kernel interleaves the columns by simply manipulating the pointer to the global memory at the beginning. The resulting mapping to the registers is shown in Figure 4.7 (b) which is equivalent to Figure 4.6 (a) bfloat16. After the interleaving, consecutive four data are naturally held by the same thread, and the larger twos are selected from them. To reduce branch divergence, the selection is done by comparing the sum of any two data.

Generate Metadata and Nonzeros. For both float and bfloat16 data types, each comparison produces a 4-bit metadata. Following Section 4.3.2, consecutive four metadata needs to be concatenated into a 16-bit metadata block through bitwise OR. As the adjacent four metadata are owned by different threads, the kernel first puts the 4-bit metadata of thread $4t+k$ to $[k\times 4{:}k\times 4+3]$ bits in the *int16* register with bit shifting as shown in Figure 4.7 (b), where "T*thread_id*{*register_id*}[*bit_id*]" denotes each 4-bit metadata. Then these int16 registers are shared cross threads with warp shuffle and produce Figure 4.6 (c). The destination thread of the warp shuffle is designed to make the whole warp busy. Figure 4.6 (d) and (e) illustrate the result after ❶ and ❷ in Figure 4.5. Notably, these two steps only change the logic mapping of the metadata, and the register allocation is not affected. So no code is required for these two steps. The last step is writing the metadata and nonzeros to global memory following ❸ in Figure 4.5. As shown in Figure 4.6 (e), each row is held by consecutive two *int16* registers of the same thread. Thus, it can be simply reinterpreted as an *=int32* object and write the metadata to global memory in column-major. On the other hand, the nonzeros are simply coalesced in the shared memory and then written to global memory in row-major.

4.4 Evaluation

This section first evaluates the accuracy of Dfss on tasks across different domains. Then, Dfss is profiled on NVIDIA A100 GPU under different sequence lengths from 256 to 4096 to show that practical speedup can be achieved in arbitrary sequence lengths. While this chapter focuses on 1:2 and 2:4 sparsity as their speedup can be directly evaluated on readily available hardware, other N:M combinations are left for future work.

4.4.1 Model Accuracy

To show that Dfss is effective in comprehensive scenarios, this section first evaluates the model accuracy on tasks in different domains and sequence lengths. For models under the "bfloat16" data type, the pre-trained model is first fine-tuned under the "float" data type, as "float" provides a more precise gradient that helps convergence. After the fine-tuning, all the parameters are directly cast to "bfloat16" and tested on the test dataset. For Question Answering and Masked Language Modeling tasks, the results reported are averaged over 8 runs under different random seeds.

Table 4.2: F1 score on BERT-large SQuAD v1.1 (Cl=95%)

Model	**w/o finetune**	**w/ finetune**
Transformer (float)	93.22 ± 0.15	$\underline{93.17 \pm 0.27}$
Transformer (bfloat16)	93.34 ± 0.31	93.18 ± 0.27
Dfss 1:2 (float)	92.86 ± 0.22	93.07 ± 0.17
Dfss 2:4 (bfloat16)	93.00 ± 0.16	$\mathbf{93.28 \pm 0.29}$

Question Answering. Table 4.2 summarizes the evaluation result of Dfss with BERT-large on SQuAD v1.1 under sequence length 384. The "bert-large-uncased-whole-word-masking" in Huggingface [71] is selected as the pre-trained model and fine-tuned with the default configuration in Huggingface [1]. The F1 scores of "1:2(float)" and

[1] https://github.com/huggingface/transformers/tree/master/examples/pytorch/question-answering

"2:4(bfloat16)" without fine-tuning are obtained by directly using the checkpoints from "Transformer(float)". The F1 scores of "Transformer (float)" and "Transformer (bfloat16)" without fine-tuning are obtained by directly using the checkpoints from "Dfss 1:2 (float)" and "Dfss 2:4 (bfloat16)" respectively, and running inference with the dense attention mechanism.

With fine-tuning, the 1:2 sparsity has only 0.1 F1 score loss, which is smaller than the standard deviation. The 2:4 sparsity even achieves a little bit of performance improvement over the dense baseline. One plausible explanation is that while the 2:4 sparsity can keep most of the important edges, it also occasionally drops a small fraction of important edges, which acts like the attention dropout technique [72]. Besides, directly applying Dfss to the dense transformer without fine-tuning also achieves comparable results. It justifies that Dfss can well approximate the dense attention mechanism.

Table 4.3: Perplexity on roBERTa-large (Cl=95%)

Model	Wikitext-2	
	w/o finetune	**w/ finetune**
Transformer (float)	2.85 ± 0.09	$\mathbf{2.83 \pm 0.09}$
Transformer (bfloat16)	2.85 ± 0.05	2.85 ± 0.07
Dfss 1:2 (float)	2.88 ± 0.06	2.88 ± 0.07
Dfss 2:4 (bfloat16)	2.88 ± 0.07	$\underline{2.84 \pm 0.04}$
Model	Wikitext-103	
	w/o finetune	**w/ finetune**
Transformer (float)	2.63 ± 0.03	$\underline{2.62 \pm 0.04}$
Transformer (bfloat16)	2.62 ± 0.08	2.63 ± 0.05
Dfss 1:2 (float)	2.64 ± 0.06	2.64 ± 0.06
Dfss 2:4 (bfloat16)	2.63 ± 0.03	$\mathbf{2.61 \pm 0.04}$

Masked Language Modeling. Dfss is also evaluated on the masked modeling task on Wikitext-2 and Wikitext-103 under sequence length 512. Similar to the question-answering tasks, the "roberta-large" is chosen as the pre-trained model and fine-tuned

Table 4.4: Accuracy of different transformer models on LRA benchmark. Dfss is trained following Tay et al. 2021[73] to reuse the results from this paper.

Model	ListOps (n=2048)	Text (n=2048)	Retrieval (n=4000)	Image (n=1024)	Avg
Transformer (float)	35.91	65.05	61.72	42.15	51.21
Transformer (bfloat16)	35.92	65.03	61.73	42.17	51.21
Local Attention[74]	15.82	52.98	53.39	41.46	40.91
Sparse Trans.[75]	17.07	63.58	59.59	**44.24**	46.12
Longformer[39]	35.63	62.85	56.89	42.22	49.40
Linformer[43]	35.70	53.94	52.27	38.56	45.12
Reformer[41]	**37.27**	56.10	53.40	38.07	46.21
Sinkhorn Trans.[68]	33.67	61.20	53.83	41.23	47.48
Synthesizer[37]	36.99	61.68	54.67	41.61	48.74
BigBird[38]	36.05	64.02	59.29	40.83	50.05
Linear Trans.[76]	16.13	**65.90**	53.09	42.34	44.37
Performer[44]	18.01	65.40	53.82	42.77	45.00
Dfss 1:2 (float)	36.85	64.95	61.83	42.02	51.41
Dfss 2:4 (bfloat16)	37.19	64.91	**62.26**	42.31	**51.67**

under the default configuration in Huggingface [2]. The results are summarized in Table 4.3. Similarly, the perplexities achieved by Dfss are on par with the dense transformer.

Long Range Arena. For sequence lengths longer than 512, Dfss is evaluated on four tasks from the Long Range Arena [73], including ListOps, Text Classification, Document Retrieval, and Image Classification under sequence lengths 2048, 2048, 4096, and 1024, respectively. Pathfinder (1K) task is omitted as the results of which cannot be reproduced, which was also reported in Lu et al., 2021 [77]. For a fair comparison with other efficient transformers, the model is trained from scratch under the default configurations. The results are summarized in Table 4.4. Dfss achieves comparable accuracy on all four benchmarks for long sequences.

[2] https://github.com/huggingface/transformers/tree/master/examples/pytorch/language-modeling

4.4.2 Speedup

This section demonstrates the speedup achieved by Dfss across different sequence lengths and compares it with existing studies.

Table 4.5: Kernel execution time under N:M=2:4 (millisecond)

Sequence Length	cuDNN GEMM	cuDNN GEMM+ cuSPARSELt	Our SDDMM
512	0.086	1.285	0.081
1024	0.291	4.611	0.284
2048	1.107	15.219	1.073
4096	4.338	N/A	4.221

Kernel Speedup. Table 4.5 compares the proposed dynamic SDDMM kernel with the state-of-the-art library, cuSPARSELt, which provides APIs for pruning and encoding input matrices into 2:4 sparsity. The batch size, number of heads, and embedding size are 8, 16, and 64, respectively. The proposed kernel accelerates the GEMM + pruning + encoding by 14.18 $\sim$ 16.24× compared with the cuSPARSELt. It is even 1.02 $\sim$ 1.06× faster than the dense GEMM in cuDNN. This demonstrates that the dynamic SDDMM kernel completely eliminates the pruning and encoding overhead.

End-to-end Speedup. Figure 4.8 compares the end-to-end speedup achieved by Dfss with previous studies. The evaluation takes the 4-layer dense transformer model of Text Classification task in Long Range Arena [73]. The dimension of each head is 64. Different combinations of numbers of heads (4, 8), sequence length (512, 1024, 2048, 4096), and hidden dimension of the feed-forward layer (256, 512, 1024) are explored. For models in previous studies, in case their implementations are not efficient, the PyTorch JIT script is applied.

Dfss achieves 1.11 $\sim$ 1.52× and 1.08 $\sim$ 1.47× end-to-end speedup over the dense transformer, it is the only method that delivers end-to-end speedup under all configurations. Under sequence length $\leq$ 2048, Dfss achieves higher speedup than most of

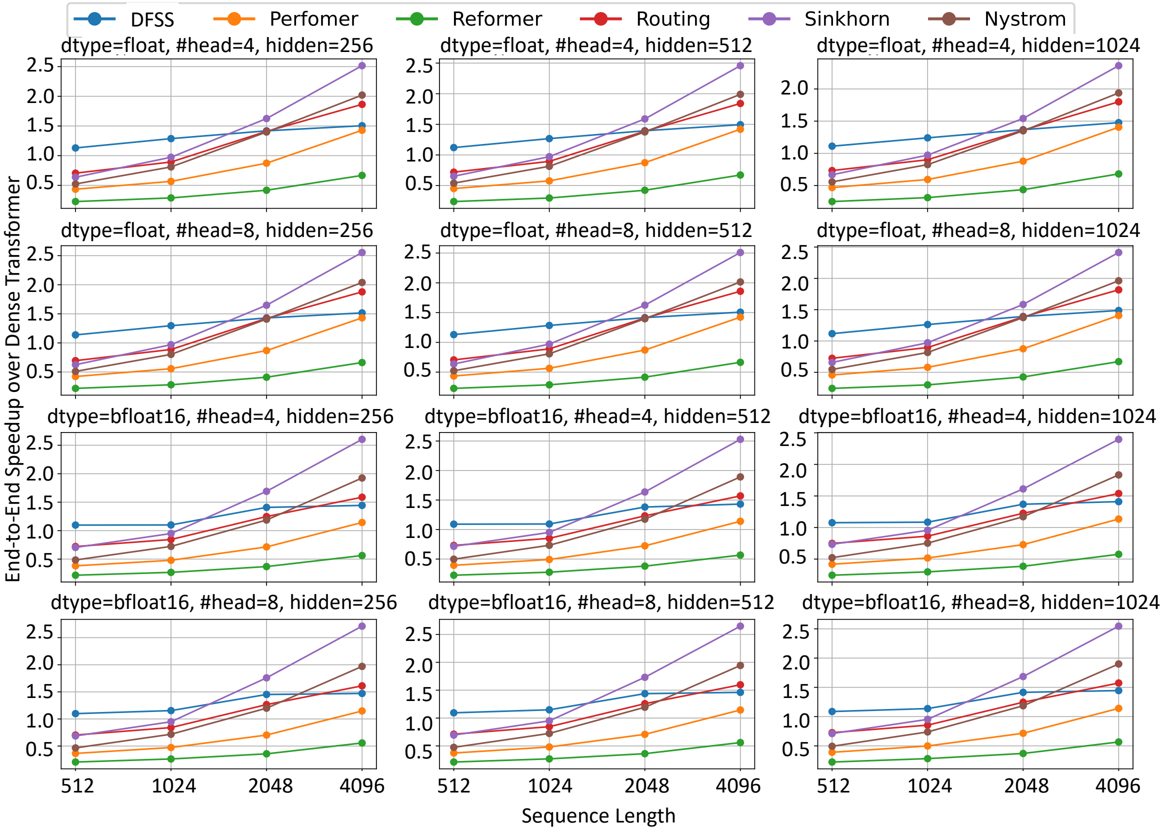


Figure 4.8: End-to-end inference speedup of different efficient transformers over dense transformer.

the baselines. Although Sinkhorn transformer [68] has a higher speedup than Dfss at sequence length 2048, as shown in Table 4.4, its accuracy is less satisfying. This result justifies that Dfss delivers good speedup under short and moderate sequence lengths. Additionally, Dfss has no hyperparameters and only requires a lightweight fine-tuning process, making it a promising solution for accelerating transformers' inference when a pre-trained model is available.

Attention Speedup Breakdown. As Dfss focuses on the attention mechanism and is orthogonal to techniques that accelerate the other parts of transformer models (e.g. static pruning, quantization), Figure 5.8 further shows the speedup achieved on the attention mechanism declared in Equation (2.1), including $\boldsymbol{QK}^T$, $Softmax$, $\boldsymbol{AV}$, and

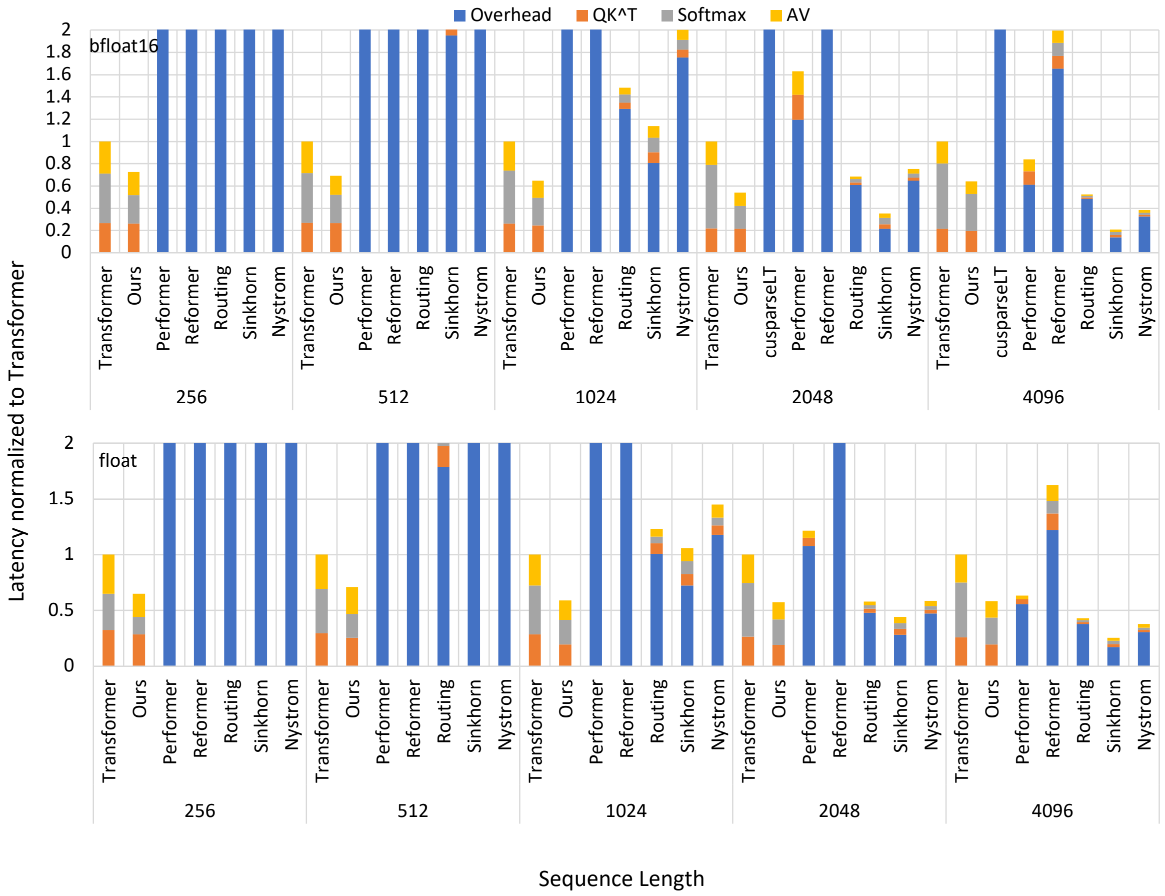


Figure 4.9: Latency breakdown of different attention mechanisms. For each configuration, the latency is normalized to the Transformer with the full attention mechanism and cut off the axis at 2 for clarity.

overhead introduced by the technique. The latency is normalized to the Transformer with the full attention mechanism under each configuration, and the y-axis is cut off at 2 for clarity.

Dfss achieves 1.38∼1.86x speedup over the transformer with full attention. It achieves speedup in all three stages with no overhead. In detail, the slight speedup in QK^T is because the dynamic SDDMM writes only nonzeros and metadata to global memory, which is only 9/16 of the dense output size. As Softmax only operates on the nonzeros, the number of entries in each row is reduced by half. This brings at least 2× speedup to the softmax kernel. Moreover, the halved row length allows the softmax kernel to switch from threadblock reduction to warp reduction in certain cases, which further boosts the performance. The AV is accelerated with CUTLASS SpMM kernel under N:M sparsity, which brings up to 1.7× speedup. Oppositely, although other approaches achieve higher speedup in QK, $Softmax$, and AV, their speedup is offset by the huge overhead, especially under sequence length ≤ 2048.

4.5 Conclusion and Discussion

This chapter presents Dfss, a novel sparse attention mechanism that dynamically prunes and encodes the attention weight matrix based on the magnitude. This compact design not only improves the utilization of GPGPUs but also simplifies fine-tuning and preserves accuracy as the dominant attention weights are accurately captured. The dynamic pruning and encoding overhead is eliminated through the algorithm-kernel codesign.

At the algorithm level, this chapter innovatively leverages the N:M fine-grained structured sparsity, originally designed for static weight pruning, to the attention weight matrix. Compared with sparse patterns of existing studies, N:M sparsity is much easier to parallel when pruning and encoding the sparse matrices at the moderate sparsity ratio.

With the benefits brought by the N:M sparsity, at the kernel level, this chapter develops a dynamic sampled dense-dense matrix multiplication kernel (SDDMM), the first of its kind, that multiplies the query and key matrices, prunes the result, and encodes the compressed sparse matrix without any overhead.

Dfss is also orthogonal to many existing efficient attention mechanisms and can potentially be applied jointly for further speedup. Additional discussions about Dfss are covered in Appendix A.4.

Chapter 5

Algorithm-Operator-Kernel Codesign: Efficient GNN Training

This chapter presents an example of kernel-centric optimization on Graph Neural Networks (GNNs). It improves the device and EU-level utilization of GPGPUs when training GNNs through the codesign across algorithm, operator, and kernel levels.

5.1 Introduction

In recent years, graph convolutional neural networks (GNN) that operate on graph-structured data have achieved convincing performance on tasks like node and graph classification [14, 78, 79]. Similar to other computation-intensive workloads [80, 81, 82], GNNs are usually trained on the GPGPUs that have high programmability and rich computation resources. Therefore, developing an efficient framework for GNNs on GPGPU is important.

Unlike existing studies that abstract GNN with two phases (Combination and aggregation), this chapter abstracts it as three distinct phases based on the computation pattern: Combination, Graph Processing, and Aggregation. Combination is usually a single- or multi-layer perceptron that updates feature vectors of each vertex. Graph Processing processes the graph to be used in Aggregation. Different from the other two phases, its computation pattern is similar to traditional graph analysis algorithms, such as computing degrees, updating or generating edge weights, and converting graphs between different sparse representations. Aggregation updates each feature vector by aggregating its neighbor feature vectors with some aggregators like max and sum [83].

While *Combination* phase is well-supported by *sgemm* (Single precision General Matrix Multiply) kernels in cuBLAS [84], the other two phases implemented with current APIs in PyTorch or TensorFlow lead to low utilization on GPGPUs. Profiling results show that the device-level utilization is limited by the kernel launching time, which could take up to 85% of the whole execution time due to the complex execution flow. The EU-level utilization is constrained by the memory bandwidth and high memory footprint. Besides, due to the large variance in graph structures such as average degree, there is no one-size-fits-all solution for all graphs. For instance, existing studies apply the *Gather-ApplyEdge-Reduce* (GAR) abstraction to the *Aggregation* phase to reduce memory traffic caused by the atomic reduction. However, this chapter observes that on graphs with low average degree, execution time saved by using GAR abstraction in *Aggregation* is offset by the format conversion overhead. Moreover, existing studies lack efficient support for more complicated GNN algorithms such as the Graph Attention Network [17], where the gradient flows through the edge weight.

This chapter proposes FUSEGNN, a highly optimized extension of PyTorch for GNNs on GPGPUs through kernel-centric optimization with Algorithm-Operator-Kernel Codesign. The key contributions are summarized below.

- **Algorithm Level**. Dual aggregation strategy that applies GAS to graphs with a low average degree and GAR to those with a high average degree.

- **Operator Level**. In the graph analysis and aggregation phases, operators that can potentially be combined into the same partition are identified.
- **Kernel Level**. Dedicated CUDA kernels are developed for each partition identified in the operator-level optimization. Various optimization strategies are applied to improve EU-level utilization.

Evaluation results with FUSEGNN on multiple benchmarks and compare it with the state-of-the-art frameworks including *PyG* [85], *DGL* [86], and *neuGraph* [87]. It achieves up to 5.3× end-to-end training speedup over *PyG*, and the memory footprint is reduced by nearly 500× on Reddit dataset. FUSEGNN makes it possible to train GAT on the entire Reddit with a single NVIDIA V100 GPU. The code is publicly available at https://github.com/apuaaChen/gcnLib.

5.2 Motivation

This section discusses the motivations inspiring the design of FUSEGNN. Existing implementations of GNNs on GPGPU suffer from low device/EU-level utilization and high memory footprint, which can potentially be significantly improved with the algorithm, operator, and kernel-level codesign.

5.2.1 Low Device-level Utilization

The low device-level utilization is caused by the high kernel launching overhead. As shown in Figure 5.1, when the dimension is small (e.g. 16), the GPU is idled for more than 85% of execution time, indicating low device-level utilization. The reason behind that is the suboptimal partition of the GNN's complex data flow graph leads to a handful

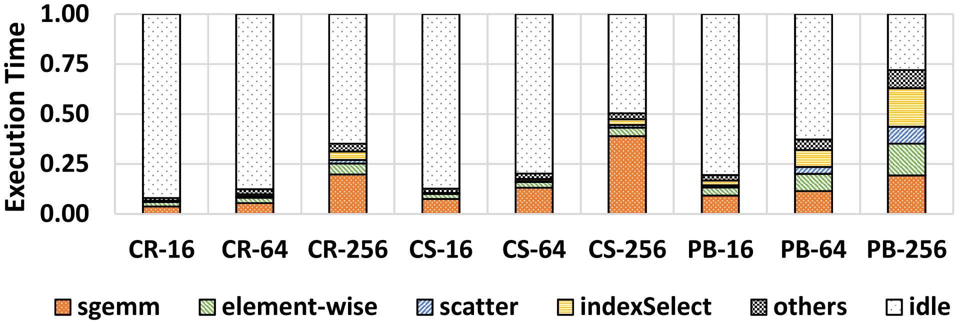


Figure 5.1: Execution time breakdown on V100 GPU. The results are collected on a single-layer GCN [14] on Cora (CR), Citeseer (CS), and Pubmed (PB). The detailed information on these datasets is summarized in Table 2.1. The input feature dimension is the native feature length of each dataset, and the output dimension is chosen from $\{16, 64, 256\}$ to cover various situations

of small kernels. The launching time of these kernels on the host (CPU) is even longer than the execution time on the GPUs.

5.2.2 Low EU and Algorithm-level Utilization

The Aggregation phase acts as the performance bottleneck of GNN training. Existing frameworks take either "Gather-ApplyEgde-Scatter" (GAS) and "Gather-ApplyEdge-Reduce" (GAR) abstractions for the *Aggregation* phase, yet the performance of both abstractions is limited by the EU-level utilization.

Memory Bottleneck of GAS. GAS used by PyTorch Geometric (*PyG*) takes an unsorted COO format graph. It splits the Aggregation into three distinct operators: *indexSelect*, *elementwise*, and *scatterAdd*. The *indexSelect* stacks the neighbor feature vectors of each vertex into an extended embedding with size $\#edges \times hidden\ dimension$. The *elementwise* applies elementwise operations such as scaling and biasing to the extended embedding, and the *scatterAdd* reduces each row of the extended embedding into the output embedding.

The GAS abstraction suffers from the memory bottleneck, the data movement statis-

Table 5.1: Data movement statistics of GCN's forward pass on the Pubmed dataset (88,676 edges, 19,717 vertices), with hidden dimension 256 (*: atomic transaction)

Operator	L2 $ Load	L2 $ Store	HBM Load	HBM Store
indexSelect	90.7 MB	86.6 MB	58.2 MB	86.3 MB
elementwise	87.7 MB	86.6 MB	87.1 MB	86.2 MB
scatterAdd	87.9 MB	90.77* MB	142.1 MB	58.3 MB

tics of which are summarized in Table 5.1. While the embedding of Pubmed dataset only consumes 20.2 MB, all three operators exhibit more than 3 times more traffic at the L2 cache in both load and store, due to the size of the extended embedding. Besides, comparing the L2 cache load/store and HBM load/store reveals that the operators have low locality, leading to high memory bandwidth requirements.

Transposing Overhead of GAR. *NeuGraph* [87] and Deep Graph Library (*DGL*) [86] choose GAR model, which takes a CSR format graph in the forward pass. As the neighbor of each vertex is ordered, the *scatterAdd* in GAS can be replaced with a *reduce* operator that accumulates the embeddings in on-chip faster memory to reduce bandwidth requirement.

Despite lower memory traffic, GAR introduces additional overhead when transposing the adjacency matrix during the backward pass. The CSR adjacency matrix has to be reformatted into CSC format during the transpose, which is GPU-unfriendly and leads to low algorithm-level utilization, significantly impacting on the end-to-end performance.

5.2.3 High Memory Footprint

The huge extended embedding in the *Aggregation* phase also leads to a high memory footprint. For instance, Reddit has 114,615,892 directed edges, which will consume over 58.7 GB when feature vector length is 128.

5.2.4 Solution: Kernel-Centric Optimization

The Kernel-centric optimization with codesign across the algorithm, operator, and kernel levels is a potential solution for all the limitations above.

Algorithm Level. At the algorithm level, instead of using a homogeneous abstraction, a dual aggregation strategy that automatically switches between GAS and GAR abstraction can potentially lead to better EU and algorithm-level utilization. Compared with GAS, GAR has lower store traffic with the cost of transposing overhead, and the store traffic reduction equals the average degree of each vertex. With this observation, when the average degree of a graph is too small to compensate for the transposing overhead, GAS is selected. Otherwise, GAR is chosen as a more suitable abstraction

Operator Level. With a better partitioning of the data flow graph into fewer operators, the device and EU-level utilization can both be improved. For the former, the kernel launching overhead is reduced along with the total number of operators. For the latter, the internal edge within each operator creates potential opportunities to pass intermediate results through faster memories such as registers to reduce global memory traffic.

Kernel Level. At the kernel level, a dedicated CUDA kernel for each operator is required to realize the opportunities created by the operator-level optimizations. Besides, due to the diversity of GNN embedding sizes, the kernels require sound logic for different hidden dimensions and residual handling.

5.3 Overview

With the motivations in the previous section, this section provides a brief overview of the FUSEGNN design. Different from the two-phase abstraction (*Combination-Aggregation*) of GNNs used by previous studies, a three-phase abstraction: *Combination-Graph Pro-*

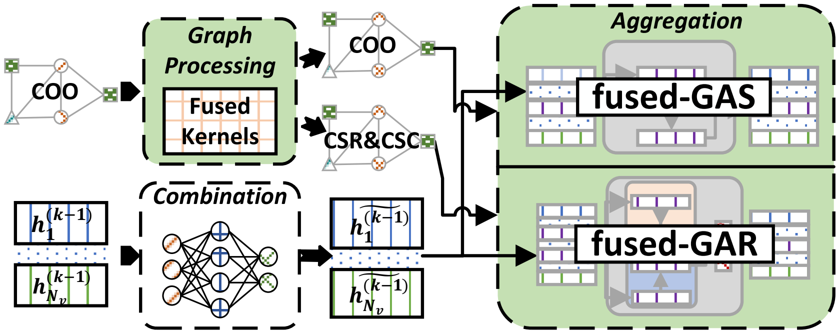


Figure 5.2: Design Overview of fuseGNN.

cessing-Aggregation is used, as shown in Figure 5.2. This abstraction is motivated by the distinct execution pattern in *Graph Processing* compared with the other two phases, such as edge weight computation and graph format transformations.

Input Graph Format. Compared with CSC and CSR, unsorted COO format has the lowest cost to construct or modify. So fuseGNN takes unsorted COO format graphs as input.

Design of Graph Processing. With the dual aggregation strategy, for graphs with a high average degree, the graph processing stage converts the input unordered COO graphs into CSR and CSC formats. Then, the edge weights are computed with the fused CUDA kernels created by the operator-kernel codesign. These fused kernels have lower kernel launching overhead that can be diminished by overlapping with other kernels. They also better data locality by caching intermediate results in registers to alleviate the memory bottleneck.

Design of Aggregation. With the algorithm-level optimization, the more suitable abstraction is selected between GAR and GAS, based on the average degree of the incoming graph. Dedicated fused kernels are created for both GAR and GAS in forward and backward passes. With the operator-kernel codesign, these fused kernels take the input embedding and adjacency matrix and directly generate the output embedding, without

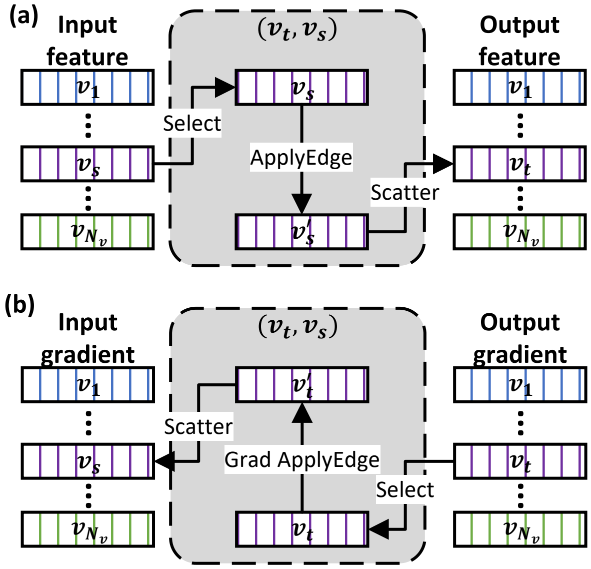


Figure 5.3: GAS Abstraction. (a) Forward; (b) Backward.

exposing the intermediate results.

5.4 Algorithm-level Optimization: Dual Aggregation Models

This section details the dual aggregation models and the underlying GAS/GAR abstractions.

5.4.1 Gather-ApplyEgde-Scatter

GAS takes an unsorted COO format graph and traverses all the edges within the edge list. As shown in Figure 5.3 (a), for edge (v_t, v_s) where v_t is the target vertex and v_s is the source vertex, the feature vector of v_s is selected from the input feature matrix. After applying the edge weight, the result is scattered to the corresponding row v_t of the output feature matrix. The backward pass is illustrated in Figure 5.3 (b). For each

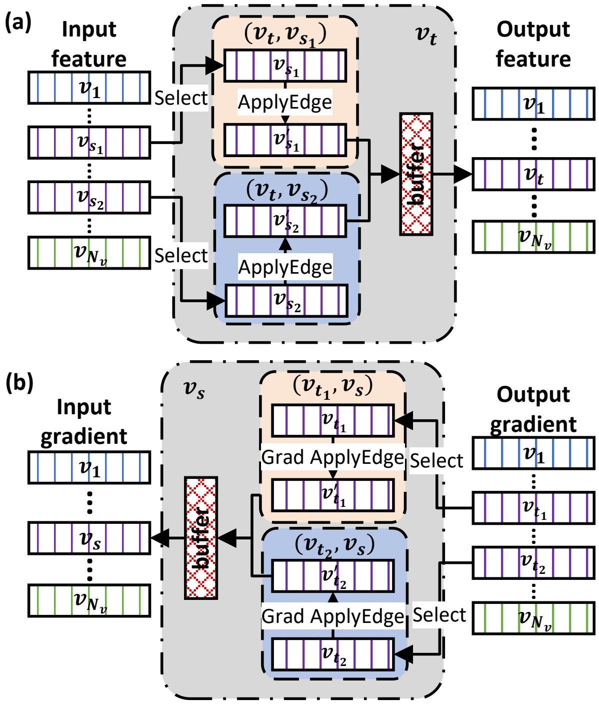


Figure 5.4: GAR Abstraction. (a) Forward; (b) Backward.

edge (v_t, v_s), the gradient on output feature vector of v_t is taken. It goes through the backward pass of the *ApplyEdge* function, and scattered to the gradient matrix of input feature matrix. The *Grad ApplyEdge* function will also generate the gradient of other operands in *ApplyEdge*, e.g., edge weight, if necessary.

5.4.2 Gather-ApplyEdge-Reduce

In forward pass, GAR works on the CSR format graphs where the rows indicate target vertex. Hence, all the edges with the same target vertex are contiguous in the edge list. For each target vertex v_t, all its incoming edges are traversed. Figure 5.4 (a) illustrates the forward pass of GAR model when v_t has two incoming edges. For each incoming edge (v_t, v_{s_i}), the procedure is similar to GAS model. The only difference is that instead of scattering the result to output feature matrix, the output of *ApplyEdge* is reduced in an on-chip buffer (registers or user-managed data cache). After all the incoming edges are reduced, the content of the buffer is written to the output feature matrix in global memory.

In backward pass, CSC format of the graph is required, so that all the edges with the same source vertex are contiguous in the edge list. As illustrated in Figure 5.4 (b), all the outgoing edges are traversed. For each edge (v_{t_i}, v_s), the gradient of v_{t_i} is selected, it goes through the *Grad ApplyEdge* function, and the result is reduced to on-chip buffer. *Grad ApplyEdge* function generates the gradient of other operands in *ApplyEdge* just like GAS. At last, the content in the buffer is written to gradient matrix of input features.

5.4.3 Selection Guideline

With the motivation in Section 5.2, the GAS and GAR abstractions complement each other under graphs with different average degrees.

GAS. GAS works on unsorted COO graphs in both forward and backward passes. This eliminates the graph format transformation overhead, and also saves global memory as only the COO format of the imput graph is required. Its disadvantage comes from the high global memory store traffic proportional to the number of edges under the lower bandwidth with atomic reduction. This disadvantage is only severe when the graph has a high average degree.

GAR. GAR addresses the disadvantage of GAS by performing the reduction in each SM, avoiding atomic transactions. It reduces the global memory store by the times equal to the average degree of the graph. Its disadvantage comes from its different requirements of graph format in forward and backward passes, which introduce transposing overhead with low algorithm-level utilization, and extra global memory footprint to store the graph in both CSR and CSC.

Selection between GAS and GAR. The average degree of the input graph can be easily calculated by dividing the edge list length by the number of rows in the embedding matrix. GAS is selected for graphs with low average degree, while GAR is preferred on high average degree graphs.

5.5 Operator-level Optimization

As different GNN models have distinct algorithms for Graph Processing and Aggregation phases, different operator-level optimizations are required. This section takes GAT [88] as an example.

5.5.1 Graph Processing

The *Graph Processing* of GAT is shown in Equation (2.4). With $\boldsymbol{a}^{(k)} \in \mathbb{R}^{2m\times 1}$ and embedding matrix $\widetilde{\boldsymbol{H}^{(k-1)}} \in \mathbb{R}^{N_v\times m}$, $\boldsymbol{a}^{(k)}$ can be firstly reshaped to an $m \times 2$ matrix. Then, $\widetilde{\boldsymbol{a}^{(k)}} = \widetilde{\boldsymbol{H}^{(k-1)}} \times \boldsymbol{a}^{(k)}$ is computed with dense matrix-matrix multiplication.

Next, the attention coefficient, leaky ReLU, exp, and the reduction that calculates the denominator in Equation (2.4) are partitioned into the same operator, as they are elementwise operators under the producer-consumer relationship. This optimization creates potential opportunities to alleviate memory bottlenecks by caching the intermediate results between these operators through registers and reducing kernel launching overhead. Similarly, the succeeding division and dropout are partitioned into the same operator.

5.5.2 Aggregation

In the forward pass, The *Gather*, *ApplyEdge*, *Scatter* under GAS abstraction and *Gather*, *ApplyEdge*, *Reduce* under the GAR abstraction are partitioned into the same operator. This saves two round trips that load and store the intermediate embedding between these operators.

In the backward pass, the *Gather*, *ApplyEdge*, *Scatter/Reduce* operators are partitioned to the same operator similar to the forward pass. As the gradients of the edge weights are required in GAT, the aforementioned operator also includes the computations related to the gradients of edge weight.

$$\frac{\partial L}{\partial A^T} = H_{out} \times H_{in}^T, \quad \frac{\partial L}{\partial H_{in}} = A^T \times H_{out}. \tag{5.1}$$

As shown in the equation above, the feature vectors of H_{out} can potentially be cached and reused for both $\frac{\partial L}{\partial A^T}$ and $\frac{\partial L}{\partial H_{in}}$.

5.6 Kernel-level Optimization

This section presents the detailed design of the CUDA kernels used in fuseGNN along with the key optimization strategies. It will focus on the optimizations used to construct the operator in the Aggregation phase.

Regarding the graph processing, it is more straightforward involving creating fused element-wise operators. Compared with the naive implementation with PyTorch that has more than a dozen kernels and N_e inner products, our new implementation only takes a much smaller *sgemm* kernel and two dedicated fused kernels.

5.6.1 Parallel Reductions

Two kinds of parallel reductions are used in *Aggregation* kernels to perform reductions of features and gradients in shared memory.

Group Reduce: For an $m \times r$ vector $\boldsymbol{v}$, each consecutive m entries form a group, so

```
Group reduce
Data: shared memory: buffer∈ $\mathbb{R}^T$, $r$, $m$, $tID$
begin
    while $r > 1$ do
        __syncthreads()
        if $tID < m$ && $r\%2 == 1$ then
            buffer[$tID$]+ =
                buffer[$tID + m \times (r - 1)$]
    $r$/= 2
    if $tID < m \times r$ then
        buffer[$tID$]+ = buffer[$tID + mr$]
```

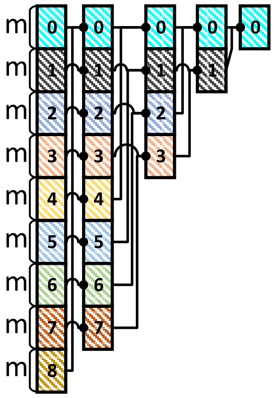


Figure 5.5: Group Reduce

there are total r groups. The target is to reduce the corresponding entries of each group into the first one: $\forall i < m,\ \boldsymbol{v}[i] += \sum_{j=1}^{r-1} v[i + mj]$. Figure 5.5 shows how group reduce works. When r is even, parallel reduction is performed to halve the number of groups. Otherwise, the last group is reduced to the first one, until there is only one left.

Block-wide Reduce: Given vector $\boldsymbol{v} \in \mathbb{R}^r$, block-wide reduce calculates $\sum_{i=1}^{r} \boldsymbol{v}[i]$. Following the implementation in Harris, Mark, 2017 [89], multiple optimization strategies including loop unrolling, and divergent avoiding are applied.

5.6.2 fused-GAS Forward and Backward Kernels

Fused-GAS partitions the workload to thread blocks in edge-centric way. For thread block size T and feature length m, each thread block handles the GAS of $max(\lfloor T/m \rfloor, 1)$ edges.

Forward Pass. Algorithm 1 shows the forward pass of *fused-GAS* model. It takes an $N_v \times m$ input feature matrix $\boldsymbol{H_{in}}$ and a COO format graph. T is set to 256 to maintain high occupancy.

If feature dimension m is smaller than block size T, as shown in line 3-10 in Algorithm 1, each thread block will handle $stride = \lfloor T/m \rfloor$ edges simultaneously. The consecutive

Algorithm 1: fused-GAS Forward Kernel

```
   Data: Input & output features: $\boldsymbol{H_{in}}, \boldsymbol{H_{out}} \in \mathbb{R}^{N_v \times m}$;
    COO Row & Col. Index: $tarInd, srcInd \in \mathbb{N}^{N_e}$;
    Edge weight.: $\boldsymbol{w_e} \in \mathbb{R}^{N_e}$; feature dim.: $m \in \mathbb{N}$;
    Block size $T \in \mathbb{N}$.
1  begin
2      $tID$ = thread ID, $bID$ = thread block ID.
3      if $m < T$ then
4          $stride = \lfloor T/m \rfloor$, $fId = tID\%m$
5          $B = \lfloor (N_e + stride - 1)/stride \rfloor$
6          if $tID < m \times stride$ then
7              for $eId = bID \times stride + \lfloor tID/m \rfloor$ to $N_e$
8                step $B \times stride$ do
9                  $atomicAdd\{\&\boldsymbol{H_{out}}[tarInd[eId]][fId],$
10                 $\boldsymbol{H_{in}}[srcInd[eId]][fId] \times \boldsymbol{w_e}[eId]\}$
11     else
12         $eId = bId$, $w = \boldsymbol{w_e}[eId]$.
13         for $fId = tID$ to $m$ step $T$ do
14             $atomicAdd\{\&\boldsymbol{H_{out}}[tarInd[eId]][fId],$
15             $\boldsymbol{H_{in}}[srcInd[eId]][fId] \times w\}$
```

entries in feature vector of each edge are handled by consecutive threads. Figure 5.6 (a) shows a toy example in which $T = 16$, $m = 5$. The thread block works on 3 edges: i, $i+1$, and $i+2$. Each of the first 15 threads loads the corresponding entry of source feature and multiplies it with the edge weight, then accumulates the result of multiplication to the address that stores the target feature vector with *atomicAdd*.

Otherwise, as shown in line 11-15 in Algorithm 1, each thread block only handles a single edge. At the beginning, the scalar edge weight is loaded into a register for reuse. Each iteration of the for loop at line 13 processes T consecutive entries of the feature vectors: it loads the source feature entry in, multiplies it with the edge weight in the register, and writes it to output target feature vector with *atomicAdd*. This process is illustrated in Figure 5.6 (b).

Backward Pass. When the gradient for edge weight is not required, the forward kernel can be directly used for the backward pass by replacing input $\boldsymbol{H_{in}}, \boldsymbol{H_{out}}$ with

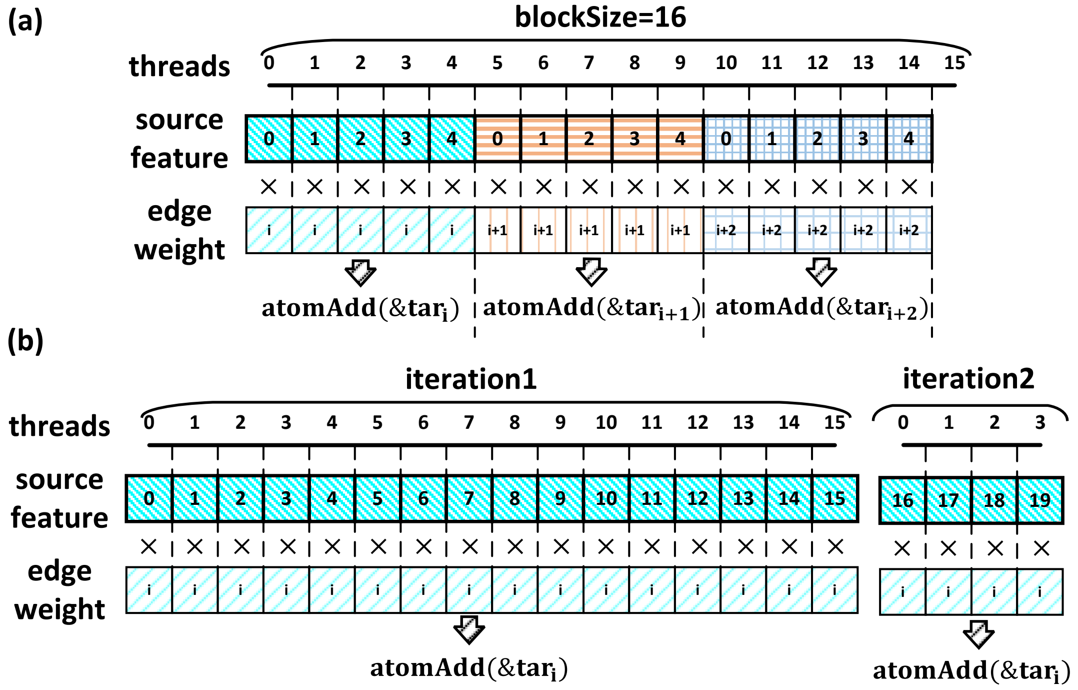


Figure 5.6: Forward kernel of fused-GAS module. (a): when feature length $\boldsymbol{m} < \boldsymbol{T}$; (b) when feature length $\boldsymbol{m} \geq \boldsymbol{T}$.

$\boldsymbol{G_{out}}, \boldsymbol{G_{in}}$ and switching the *srcInd* and *tarInd*. Otherwise, the kernel in Algorithm 2 is used. Line 9-14 and 24-26 calculate the gradient matrix of input features. They are the same as line 6-9 and 13-14 in Algorithm 1 with changed inputs and different looping way. Line 15-21 and line 27-31 calculates the gradient of each scalar edge weight with three steps:1) save the gradient contributed by each entry in a buffer in shared memory (line 15 & 27); 2) do reduction to generate the gradient of edge weight (line 17 & 29);3) write the result to DRAM (line 20 & 31).

As $\lfloor T/m \rfloor$ edges are handled simultaneously when $m < T$, the gradients are stored in an interleaved fashion (line 15 in Algorithm 2) as shown in Figure 5.7. Then the gradient of each edge weight is calculated with *group reduce* under group size $\lfloor T/m \rfloor$ and number of group m. The major benefit brought by the interleaved fashion is that in step 2) and 3), all the active threads are consecutive, therefore warp divergence is avoided (different

Algorithm 2: fused-GAS Backward Kernel

```
   Data: Output & input gradient: $\boldsymbol{G_{in}}, \boldsymbol{G_{out}} \in \mathbb{R}^{N_v \times m}$; Input feature: $\boldsymbol{H_{in}} \in \mathbb{R}^{N_v \times m}$; COO
         Row & Col. Index: $tarInd, srcInd \in \mathbb{N}^{N_e}$; Edge weight.: $\boldsymbol{w_e} \in \mathbb{R}^{N_e}$; Edge weight
         gradient: $\boldsymbol{g_e} \in \mathbb{R}^{N_e}$ feature dim.: $m \in \mathbb{N}$; Block size $T \in \mathbb{N}$.
1  begin
2      shared $buffer[T-1:0]$, $tID$ = thread ID, $bID$ = thread block ID, $buffer[tID] = 0$
3      if $m < T$ then
4          $stride = \lfloor T/m \rfloor$
5          $B = \lfloor (N_e + stride - 1)/stride \rfloor$, $step = stride \times B$
6          $N_s = \lfloor (N_e - bID \times stride + step - 1)/step \rfloor$
7          $fId = tID\%m$, $gId = \lfloor tID/m \rfloor$, $eId = bID \times stride + gId$
8          for $i = bID \times stride$ to $N_s \times step + bID \times stride$
9            step $step$ do
10             $\_\_syncthreads()$
11             if $tID < stride \times m$ && $eId < N_e$ then
12                 $g = \boldsymbol{G_{out}}[tarInd[eId]][fId]$
13                 $atomicAdd\{\&\boldsymbol{G_{in}}[srcInd[eId]][fId],\ g \times w\ \}$
14                 $buffer[gId + fId \times stride] = g \times \boldsymbol{H_{in}}[srcInd[eId]][fId]$
15             $\_\_syncthreads()$
16             $group\ reduce(buffer,\ m,\ stride)$
17             $\_\_syncthreads()$
18             if $tID < stride$ && $i + tID < N_e$ then
19                 $\boldsymbol{g_e}[i + tID] = buffer[tID]$
20             $buffer[tID] = 0$, $eId += stride$
21     else
22         $eId = bId$, $w = \boldsymbol{w_e}[eId]$
23         for $fId = tID$ to $m$ step $T$ do
24             $g = \boldsymbol{G_{out}}[tarInd[eId]][fId]$
25             $atomicAdd\{\&\boldsymbol{G_{in}}[srcInd[eId]][fId],\ g \times w\ \}$
26             $buffer[tID] += g \times \boldsymbol{H_{in}}[srcInd[eId]][fId]$
27         $\_\_syncthreads()$
28         $block\text{-}wide\ reduce(buffer)$
29         if $tID == 0$ then
30             $\boldsymbol{g_e}[eId] = buffer[0]$
```

threads of the same warp take different branch) [90] to the most extent. When $m \geq T$, as the thread block only handles a single edge, the gradient contributed by each entry consecutively (line 27 in Algorithm 2) is stored and reduced with block-wide reduction (line 29 in Algorithm 2).

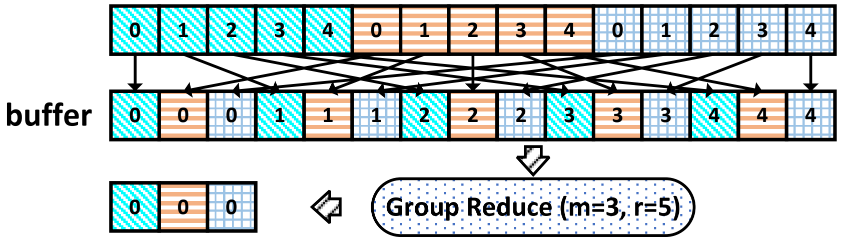


Figure 5.7: Illustration of line 15 & 17 in Algorithm 2

5.6.3 fused-GAR Forward and Backward Kernels

Fused-GAR kernel partitions the workload to thread blocks in the vertex-centric way. Each thread block will handle all the edges with the same target vertex in forward pass and all the edges with the same source vertex in backward pass.

Algorithm 3: fused-GAR Forward Kernel

```
Data: Input & output features: $\boldsymbol{H_{in}}, \boldsymbol{H_{out}} \in \mathbb{R}^{N_v \times m}$;
  CSR Row Ptr & Col. Index: $tarPtr \in \mathbb{N}^{N_v+1}, srcInd \in \mathbb{N}^{N_e}$;
  Edge weight.: $\boldsymbol{w_e} \in \mathbb{R}^{N_e}$, Self-loop weight: $\boldsymbol{w_{sl}} \in \mathbb{R}^{N_v}$;
  feature dim.: $m \in \mathbb{N}$; Block size $T \in \mathbb{N}$; Grid size: $B \in \mathbb{N}$.
1  begin
2      $tID$ = thread ID, $bID$ = thread block ID.
3      $start = tarPtr[bID]$, $stop = tarPtr[bID+1]$
4      if $m < T$ then
5          shared $buffer[T-1:0]$
6          $stride = \lfloor T/m \rfloor$, $fId = tID\%m$, $buffer[tID] = 0$
7          for $eId = start + \lfloor tID/m \rfloor$ to $stop$ step $stride$ do
8              $buffer[tID]{+}{=}\boldsymbol{H_{in}}[srcInd[eId]][fId] \times \boldsymbol{w_e}[eId]$
9          $group\ reduce(buffer, min(m, stop - start)$, m)
10         if $tID < m$ then
11             $\boldsymbol{H_{out}}[bID][fId] = \boldsymbol{H_{in}}[bID][fId] \times \boldsymbol{w_{sl}} + buffer[tID]$
12     else
13         $w = \boldsymbol{w_{sl}}[bId]$
14         for $fId = tID$ to $m$ step $T$ do
15             $buffer = \boldsymbol{H_{in}}[bID][fId] \times w$
16             for $eId = start$ to $stop$ step 1 do
17                 $buffer+= \boldsymbol{H_{in}}[srcInd[eId]][fId] \times \boldsymbol{w_e}[eId]$
18             $\boldsymbol{H_{out}}[bID][fId] = buffer$
```

Forward Pass. First of all, the thread block identifies the index to the first and last edge it handles based on $tarPtr$ (line 3). Similar to the *fused-GAS* forward kernel, $\lfloor T/m \rfloor$ edges can be processed simultaneously when $m < T$ (line 4-11).

As all the edges share the same target vertex, they can be reduced on-chip: consecutive m threads form a thread group, and the edges are partitioned evenly to the thread groups. Each thread group processes the edges assigned to it in sequence (loop at line 7). After all the partial results are stored in the buffer in shared memory, *group reduce* (line 9) is applied to the buffer under group size m and number of group $\lfloor T/m \rfloor$ to get the final output feature vector. When $m \geq T$, as all the threads work on the same edge in each iteration, a single register is used for each thread to do reduction (line 12-18).

Backward Pass. Similar to *fused-GAS*, the forward kernel can be used for backward when gradient on edge weights are not required. Otherwise, the same strategy in *fused-GAS* backward kernel is applied. Specifically, to generate gradient on edge weight, when $m < T$, the gradient contributed by each entry is stored in shared memory in an interleaved fashion and reduced with *group reduce*. Otherwise, the gradient are stored consecutively and reduced with block-wide reduction. The gradient on input feature vectors is calculated in the symmetric way of forward pass.

Besides, FUSEGNN fully exploit the data reuse opportunities. In backward pass, as all the edges share the same source vertex, the feature vector of which will be used to generate the gradient of all the edge weight. So it is cached at the beginning in registers when $m < T$ or shared memory otherwise. Besides, the gradient of the target vertex is required for both input feature gradient and edge weight gradient, so it is cached in a register for reuse.

5.6.4 Discussion on Kernel Design

Optimization Strategies. First of all, as consecutive threads work on consecutive entries in feature vectors, and each thread block handles consecutive edges, the kernels can achieve good global memory transaction coalescing and high atomic transaction bandwidth as they are in the same cache line [91]. Second, the global memory transactions are further reduced with extensive data reuse. For instance, as the three stages of Aggregation are fused in the single kernel, the data that will be used multiple times are cached with shared memory or registers. Moreover, giving the credit to interleaved fashion in Figure 5.7, reduction strategies, as well as the looping strategy in backward kernels (e.g. line 9 in Algorithm 2), the active threads are always kept consecutive to avoid warp divergence to the most extent. Last but not least, multiple edges can be handled concurrently so that our kernels can maintain high occupancy even with short feature vectors.

Flexibility vs. Performance. Although fused kernels have much lower latency and memory footprint, their re-usability are limited. As a result, it is impractical to produce libraries consisting of already-fused kernels [92]. Previous studies solve this problem by following "*Make the Common Case Fast*" idea. The "common case" in them refers to *Aggregation* phase in which *ApplyEdge* is element-wise operation and gradient on edge weight is not required. For example, models like GCN [14], GIN [16], SGC [93], and GraphSAGE [94] (except for LSTM aggregator) directly use a scalar edge weight. For these common cases, *neuGraph* provides the *Fused-Gather* kernel while *DGL* exploits *SpMM* in *cuSPARSE* library [62]. Other uncommon cases are still implemented with simple and re-usable kernels like *PyG*.

While "*Make the Common Case Fast*" strategy is also exploited in FUSEGNN, the "common case" in our work is relaxed to *Aggregation* phase in which *ApplyEdge* is element-wise operation, because gradient on edge weight is supported. Therefore, recent models with complex attention mechanism like GAT [17] and AGNN [88] treated as uncommon case in previous studies are included into "common case" in FUSEGNN.

Besides, unlike *DGL* that uses APIs in closed source library CUSPARSE, fused kernels in *Aggregation* phase are developed in neighborhood aggregation fashion [85] so that they can be used as templates when developing new *Aggregation* kernels.

5.7 Evaluation

This section evaluates the performance of FUSEGNN and compares it with state-of-the-art studies on a single NVIDIA V100 GPU [95]. The benchmarks are denoted as "Model-Dataset-Hidden". For "Model", GCN [14] and GAT [17] are picked to cover GNNs with simple and complex *Graph Processing.* "Dataset" includes Cora, Pubmed, and Reddit to cover various scale and average degree. "Hidden" (output dimension of *Combination* phase) is chosen from $\{16, 64, 128, 256, 512\}$. Both transductive learning and inductive learning are evaluated, where the *Graph Processing* phase is executed at each iteration or the result is cached in global memory in the first execution and reused in later iterations, respectively.

5.7.1 Latency

To demonstrate the efficiency of FUSEGNN in GNN training, this section first evaluate the it latency on several benchmarks.

GAS v.s. GAR. Figure 5.8 summarizes the relative end-to-end speedup over *PyG* [85] achieved under different configurations. It shows that the speedup provided by FUSEGNN is consistent and significant. On small graphs like Cora, *fused GAS* could achieve higher speedup compared with *fused GAR* abstraction. Figure 5.9 compares the

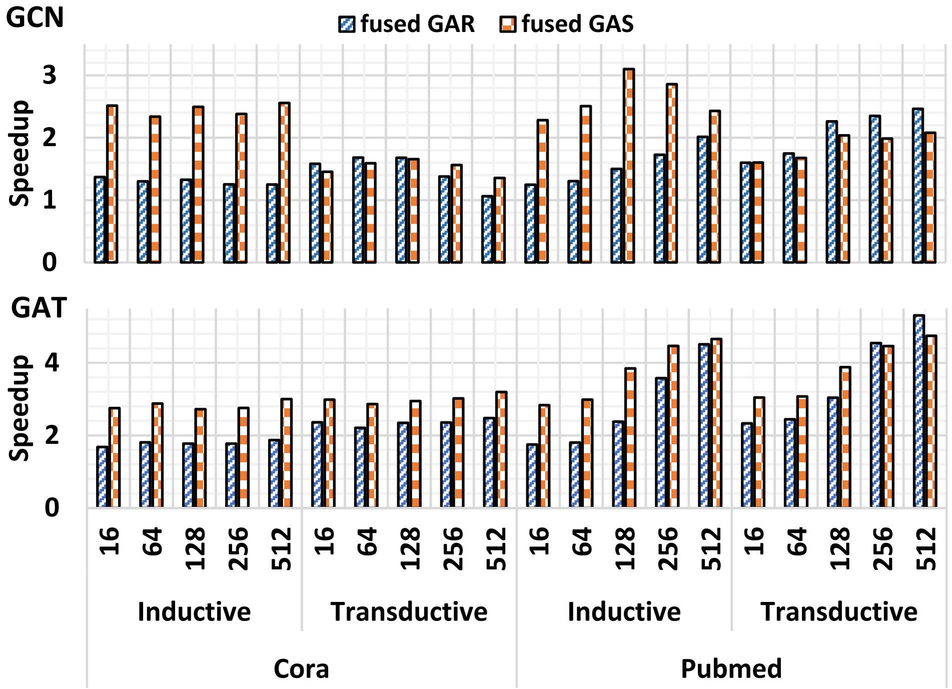


Figure 5.8: Speedup of fused GAR and fused GAS over PyG under different configurations.

latency of *fused GAR* and *fused GAS* on *Reddit*. It shows that *fused GAR* module is more effective than *fused GAS* on graphs with high average degree like *Reddit*.

To further justify this, Figure 5.10 shows the execution time of aggregation and graph format conversion on different benchmarks. First, on graphs with short feature vector and low average degree, *fused GAS* has lower *Aggregation* latency, this is because its kernel is simpler than *fused GAR* so that fewer registers are used and higher occupancy can be achieved. On graphs with long feature vectors and high average degree, *fused GAR* achieves lower *Aggregation* latency which outweighs the additional overhead of format conversion.

With all these observations, empirically, one should select *fused-GAR* for graphs with high average degree and *fused-GAS* for others. While a dedicated performance models can be built in future studies, as *fused-GAS* and *fused-GAR* share the same interface, the user can just try both of them and pick the better one.

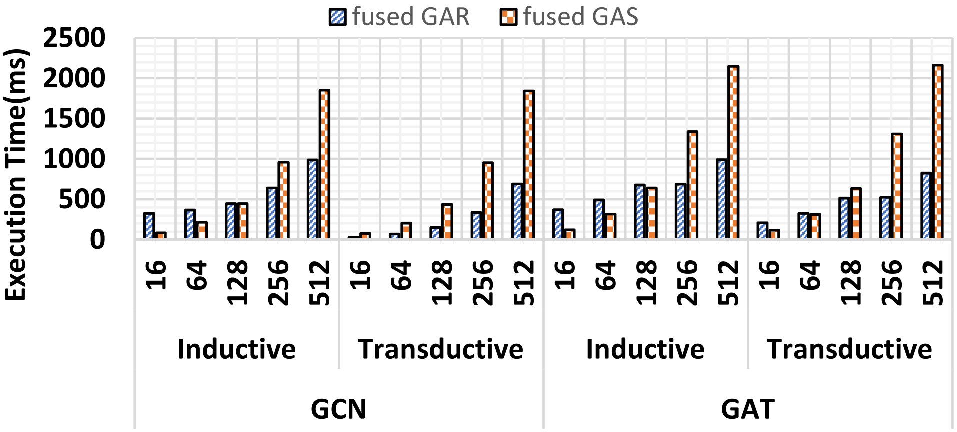


Figure 5.9: Latency of fused GAR and fused GAS on Reddit.

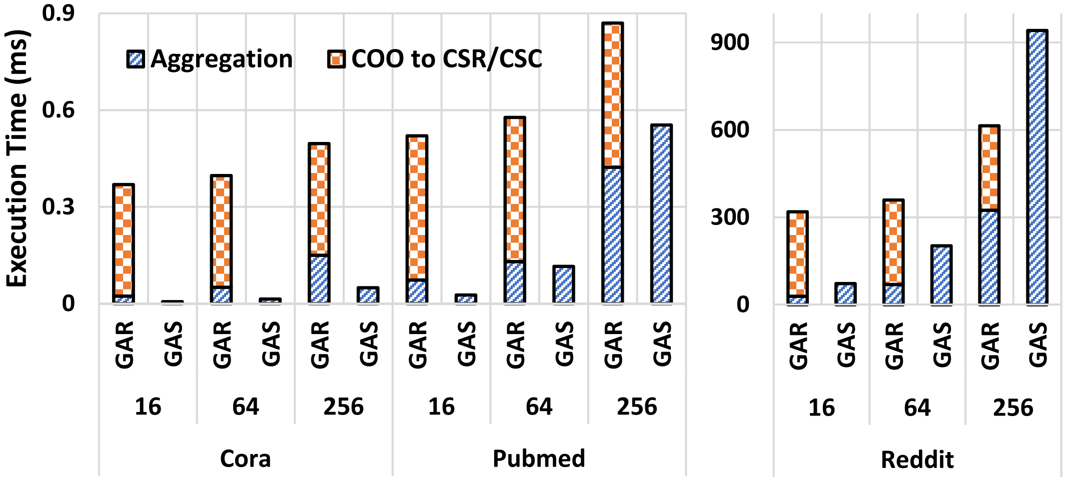


Figure 5.10: Latency of Aggregation and format conversion

Comparison to DGL [86]. The benchmark is composed of single layer GCN and GAT on Cora, Pubmed, and Reddit under dimension 16, 64, and 512. The results are summarized in Table 5.2. For FUSEGNN, the lower one in the latency of GAR and GAS is taken.

On small datasets like *Cora* and *Pubmed*, FUSEGNN implementation consistently achieves much lower latency. The major speedup comes from kernel-fusion applied to *Graph Processing* phase. For example, in *Cora-GAT*, 52 kernels are invoked in *DGL*, while *fused-GAS* only launches 24 kernels.

Table 5.2: Comparison of Training Latency (in millisecond) between DGL and fuseGNN.

Dataset	Model	hidden=16		hidden=64		hidden=512	
		DGL	fuseGNN	DGL	fuseGNN	DGL	fuseGNN
Cora	GCN	2.35	**1.05**	2.37	**1.09**	2.51	**1.10**
	GAT	7.46	**1.65**	7.59	**1.68**	8.07	**1.70**
Pubmed	GCN	2.69	**1.07**	2.79	**1.11**	3.61	**3.00**
	GAT	7.75	**1.72**	7.85	**1.72**	10.43	**3.74**
Reddit	GCN	**22.5**	29.2	**58.9**	71.9	**452.9**	690.4
	GAT	*OOM*	**120.0**	*OOM*	**314.3**	*OOM*	**825.3**

On *GCN-Reddit* where *Aggregation* phase becomes the major bottleneck, *DGL* has lower latency due to two major reasons. First, unlike fuseGNN, *DGL* does the COO to CSR/CSC conversion offline, so that this overhead is not included in their latency. Second, *DGL* directly exploits the *CSR SpMM* kernels in *cuSPARSE* library [62] that is optimized by more experienced experts of NVIDIA in SASS. However, compared with the closed source *cuSPARSE* library, kernels in fuseGNN can be easily modified to support new GNN algorithms.

On *GAT-Reddit*, as gradient on edge weight is not support by the *SpMM* implementation, *DGL* suffers from *OOM* with hidden=16. Oppositely, fuseGNN can even support hidden=512.

Table 5.3: Comparison on *GCN-Reddit-16*

Kernel	Latency	Active Warps	Occupancy	DRAM Bandwidth
fused-GAR	13.6 ms	63.13 / SM	98.6%	327.1 GB/s
fused-Gather[87]	23.5 ms	30.28 / SM	47.3%	196.1 GB/s

Comparison to NeuGraph [87]. As the authors haven't yet released their code, it is hard to have a thorough comparison across multiple benchmarks. However, first, the backward kernel for *fused-Gather* is not provided, so the high volume memory storage footprint and data movement remain unsolved for models like GAT. Second, their *fused-Gather* kernel is also less effective compared with *fused-GAR* kernel under certain

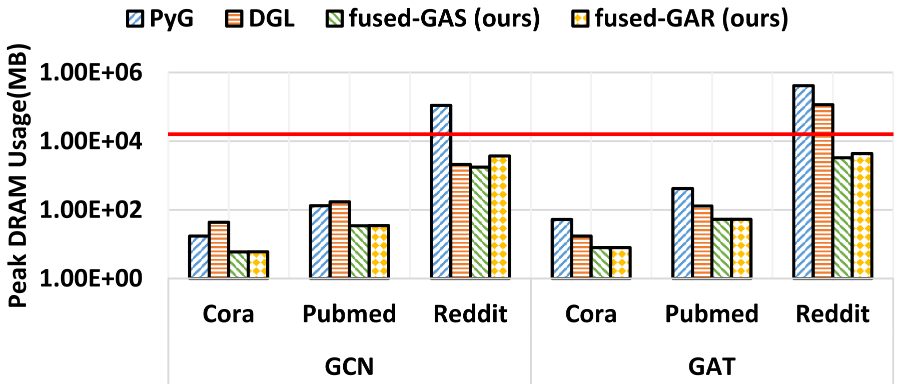


Figure 5.11: Peak global memory usage of the first layer of GCN/GAT with output dimension 128. The red line marks 16 GB.

Table 5.4: Data Movement (*: atomic transactions)

Kernel	L2 $ Read	L2 $ Write	HBM Read	HBM Write
GCN-Pubmed-256				
fused-GAS	92.5 MiB	90.78* MiB	107.8 MiB	57.0 MiB
fused-GAR	107.6 MiB	19.3 MiB	61.6 MiB	20.9 MiB
GCN-Reddit-128				
PyG(theoretical)	178 GiB	118 + 58* GiB	222 GiB	161 GiB
fused-GAS	59.4 GiB	58.7* GiB	100.9 GiB	50.2 GiB
fused-GAR	55.5 GiB	113.8 MiB	47.1 GiB	116.4 MiB

scenarios.

A *fused-Gather* kernel is implemented based on their description and compared with fuseGNN on benchmark *GCN-Reddit-16*. As shown in Table 5.3, when dimension is 16, *fused-GAR* kernel processes 8 edges simultaneously to fully exploit the thread block size 128. On the other hand, the *fused-Gather* kernel doesn't involve such design, so only 16 threads in each thread block are activated. As a result, it has much fewer active warps per SM which leads to lower thread-level parallelism and low global memory Bandwidth [96]. Besides, *fused-GAR* reduces the number of steps to process n edges from $O(n)$ to $O(\lceil n/r \rceil + \log r)$, $r = \lfloor T/m \rfloor$ where T is thread block size and m is the dimension.

5.7.2 Memory

This section evaluates the global memory footprint and data movement reduction with fuseGNN over existing studies.

Peak Memory Usage. Figure 5.11 compares the peak Memory Usage of different frameworks with hidden dimension 128. fuseGNN reduces the storage footprint by several orders. In particular, the peak memory usage is reduced by 95× and 26× on *GAT-Reddit-128* compared with *PyG* and *DGL*, respectively. This makes it possible to fit all these models in a single V100 GPU.

Data Movement. Table 5.4 summarizes the data movement in *fused-GAS/GAS* forward kernels under the same benchmark in Table 5.1. On *GCN-Pubmed-256*, the *fused-GAS* kernel reduces non-atomic L2 transactions (read and write) by 4.75× and global memory transactions by 3.14×. On the other hand, the non-atomic L2 and global memory transaction of *fused-GAR* are reduced by 3.46× and 6.28×, and the atomic transactions are eliminated. First, transaction related to the intermediate extended feature vectors are eliminated. Second, while the L2 cache hit rate of element-wise kernel is around 0%, the proposed kernels have around 30%, as a larger portion of the input feature matrix can be cached compared with the huge extended feature matrix. On *GCN-Reddit-128*, *fused-GAR* can reduce the L2 cache write transaction by more than $1,500\times$.

Workload Imbalance. In GNN, each edge introduces the same cost, so there is no workload imbalance problem in GAS abstraction. On the other hand, the imbalance degree would lead to imbalance workload on each thread block in GAR abstraction. However, As the number of thread block in *fused-GAR* equals to the number of vertices (from thousands to hundreds of thousands or even higher) while there are only 84 SMs per GPU, the workload imbalance of thread blocks can be compensated by assigning different amount of blocks to each SM, so that the resources of GPU are fully-utilized

[97]. For instance, on *GCN-Reddit-128*, the achieved occupancy per SM of *fused-GAR* is 93.4%, which shows that the workload imbalance of thread blocks doesn't affect the occupancy of GPU cores.

5.8 Conclusions

This chapter presents a highly optimized extension library of PyTorch for GNN training on GPGPU. With kernel-centric optimization, its dual aggregation strategy along with the fused CUDA kernels significantly improves the training throughput and reduces global memory footprint. FUSEGNN makes it possible to training GNN on larger graphs with the same hardware. The designs can be easily extended to multi-GPU or CPU+GPU scenarios, in which the graph is partitioned and assigned to each GPGPU.

Chapter 6

Kernel-Centric Optimization: Compiler Perspective

This chapter demonstrates the application of kernel-centric optimization in the realm of compiler. It introduces the Epilogue Visitor Tree (EVT) compiler, which automates kernel-centric optimization across a wide range of NN models to accelerate training workloads.

6.1 Introduction

The rapid evolution of deep learning has led to a surge in the complexity and quantity of deep learning models. These models, often implemented through deep learning frameworks like PyTorch [98] that target flexible programming, face suboptimal performance as they cannot efficiently utilize the resources provided by hardware like GPGPUs. Manual optimization of these models demands a high degree of engineering effort and expertise, and even experts may overlook hidden optimization opportunities within intricate models.

Deep learning (DL) compilers play a critical role in addressing this challenge. A typical DL compiler optimizes the user-defined model through three major steps [99], which align with kernel-centric optimization: algorithm-level optimizations through graph transformations, operator-level optimization with partition, and kernel-level optimization. In detail, during the algorithm-level optimization, the input model is first traced into the data-flow graph, with nodes representing operators and edges denoting dataflow. The graph transformations simplify the graph by folding constants or eliminating common sub-expressions. Subsequently, during the operator-level optimization, the partitioner divides the graph into individual partitions, and each partition is further optimized by kernel-level optimization, such as replacement by a fused kernel generated by the kernel-level compiler [100].

While considerable progress [45, 101, 98] has been made in optimizing models for inference efficiency, three key limitations remain in optimizing the training workload.

Algorithm-level Limitation.The training workload has a larger and more complex operator set. If mismanaged, these operators would significantly restrict potential optimizations such as kernel fusion. The training workload has intricate and non-fusible operators such as loss functions and gradient operators. Operators like batch normalization also exhibit more intricate behaviors during training. Moreover, a substantial number of reductions emerge in the training graph, which are challenging to fuse and sometimes not beneficial.

Operator-level Limitation. Partitioner in existing compilers cannot find feasible and optimal partitions in the training graph. Unlike the inference graph that contains a stream of operators and can be partitioned with simple heuristics, the training computation graph has additional edges passing saved activations to backward operators for gradient computation. Heuristically partitioning them can easily lead to an infeasible result that contains cycles. While arbitrarily separating the forward and backward graph

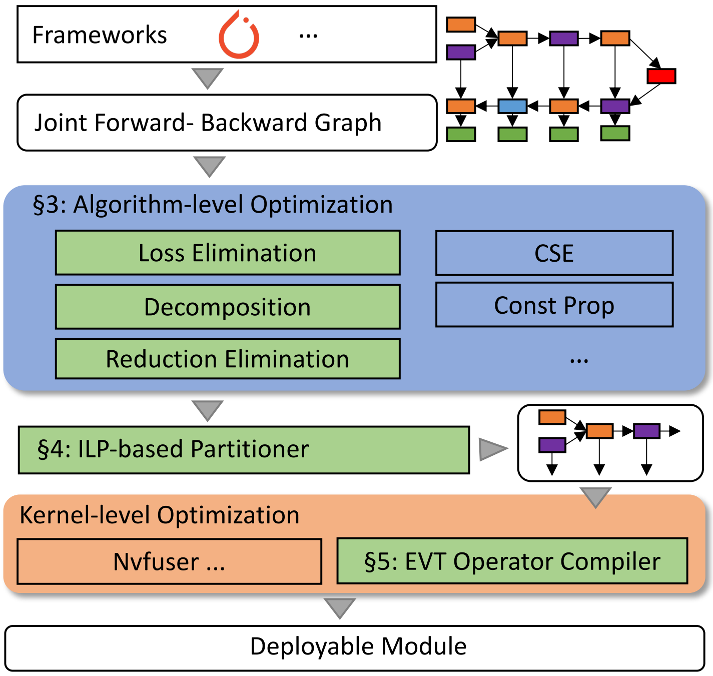

Frameworks
...
Joint Forward- Backward Graph
§3: Algorithm-level Optimization
Loss Elimination
Decomposition
Reduction Elimination
CSE
Const Prop
...
§4: ILP-based Partitioner
Kernel-level Optimization
Nvfuser ...
§5: EVT Operator Compiler
Deployable Module

Figure 6.1: Overview of the EVT System Design.

obscures opportunities for fusing operators from both forward and backward parts.

Kernel-level Limitation. Existing kernel-level compilers cannot generate fused kernels that deliver state-of-the-art performance while accommodating diverse fusion patterns. Loop-based compilers, like TVM [45], represent operators as nested loops. They face challenges in abstracting dedicated optimizations and leveraging the accelerator features and datapaths in modern GPUs for core operators like matrix multiplication (GEMM) and convolutions. On the other hand, template metaprogramming (TMP)-based compilers [51, 50] that leverage expert-developed template libraries lack the flexibility to support diverse fusion patterns in the training graphs.

6.1.1 The EVT Approach

This chapter present Epilogue Visitor Tree (EVT), a novel compiler designed to address all the aforementioned limitations. For the algorithm-level limitation, EVT facilitates graph transformations tailored for training workload. To tackle the operator-level limitation, EVT is also associated with an Integer Linear Programming (ILP)-based partitioner that can find the optimal and feasible partitions for subsequent operator-level optimizations. The name "Epilogue Visitor Tree" comes from the novel abstraction designed to address the kernel-level limitation. The key idea is leveraging the template libraries designed by GPU experts for core operators like GEMM to ensure state-of-the-art performance while presenting a flexible abstraction under which the operators to be fused can be easily assembled and integrated into the core operator. The detailed explanation is as follows:

- **Epilogue**: Create fused kernels with epilogue fusion.
- **Visitor**: The key abstraction proposed to provide the fusion pattern flexibility. It decouples operators in the epilogue so they can be developed and optimized individually, and assembled into the epilogue at compile time.
- **Tree**: Tree structure is used to assemble the epilogue in our operator compiler for best performance.

As shown in Figure 6.1, EVT directly takes models from frameworks like PyTorch [98] and converts them into a joint forward-backward graph, with training samples and model parameters as inputs and gradients as outputs.

EVT's pass manager then schedules a series of graph transformations directly on the joint forward-backward data-flow graph. In addition to standard transformations such as constant propagation that simplify the graph, EVT's graph compiler introduces three crucial passes to address the algorithm-level limitation and create more fusion opportunities for the kernel-level compiler: loss elimination, decomposition, and reduction elimination. The first one unlocks the opportunity to fuse the loss function and its backward pass by recognizing the loss value doesn't participate in the backward computation. The second pass breaks complicated non-fusible operators like *batch_norm_fw/bw* into fusible operators. The last one alleviates the reduction fusion issue by identifying cases where the reduction result is equivalent to a series of fusible operators.

Next, the ILP-based partitioner models the partitioning as an ILP problem to find the feasible global optimal solution. As the acyclic partitioning problem is NP-Hard [102], several simplification techniques are proposed to reduce the complexity, and large-scale neural networks with millions of decision variables can be solved in minutes.

Subsequently, partitions are dispatched to a suitable operator compiler to create the fused kernels. Particularly, partitions containing core operators are offloaded to EVT kernel-level compiler for better performance.

Finally, EVT returns a deployable model to the user, which is compatible with other optimization techniques such as CUDA graph to perform further orthogonal optimizations. EVT can be used as a backend for *torch.compile* that optimizes a model with a single line of code.

6.2 Algorithm-level Optimizations

This section first discusses the major optimizations exploited and then provides a running example to illustrate how these optimizations simplify the computation graph and expose new fusion opportunities.

6.2.1 Optimizations

Loss Elimination. Despite it sounding counter-intuitive, this chapter found that the loss value can be eliminated as a dead node from the joint forward-backward graph for two reasons: the backward pass can be performed without computing the loss value, and the loss value is not required for every iteration.

For the first reason, the loss value does not participate in the computation of the backward pass. For example, in the widely used softmax cross-entropy loss function, the gradient of logits X under target Y can be simplified to

$$\frac{\partial L}{\partial X} = -\frac{1}{n}OneHot(Y) + \frac{1}{n}Softmax(X). \tag{6.1}$$

With this equation, the gradient can be directly computed with higher numerical stability without the loss value.

For the second reason, the loss value is required only when the users need to observe it to assess the training process. However, the training process can take millions of iterations, and it is not necessary to check the loss of each iteration. Therefore, the loss can be computed only when it is required, and the accelerated version is used for the remaining iterations.

With this domain knowledge, loss elimination can be proposed that eliminates the loss value as a dead node. However, existing DNN frameworks like PyTorch still use the loss value to trigger the backpropagation, so it is achieved by simply replacing the loss value with a constant scalar tensor.

Decomposition. The deep learning frameworks contain many non-fusible operations like *logSoftmax*, *addmm*, designed to simplify programming and leverage existing fused kernels. However, these operations also become the major obstacle for kernel fusion, as each of them may require special rules to handle. To tackle this issue, decomposition rules are registered for these operations, breaking them down into basic fusible operations like element-wise operations and reductions through pattern matching and rewriting. Table 6.1 provides examples of these decomposition rules. The decomposition of Batch Normalization (BN) [103] is inspired by Jung et al., 2019 [104]. Unlike the original

Table 6.1: Decomposition Examples in EVT

Pattern	**Decomposition**
$logSoftmax(X)$	$log(Softmax(X))$
$nllLossBp(X, Y)$	$-\frac{1}{n}OneHot(Y)$
$logSoftmaxBp(dY, X)$	$dY - logSoftmax(X)\sum dY$
$dropout(X, p)$	$\frac{X \odot (rand\_like(X) > p)}{1-p}$
$dropoutBp(dY, M, p)$	$\frac{dY \odot M}{1-p}$
$reluBP(dY, X)$	$dY \odot (X >= 0)$
$addmm(X, W, b)$	$mm(X, W) + b$
$BN(X, \alpha, \beta)$[104]	$\mu = \sum X/n,\ m2 = \sum X^2/n$ $\sigma = \sqrt{m2 - \mu^2}$ $\frac{\gamma}{\sigma}(X - \mu) + \beta$
$BNBp(dY, X, \mu, \sigma, \gamma)$[104]	$d\beta = \sum dY,\ \hat{X} = \frac{X-\mu}{\sigma}$ $d\gamma = \sum(dY \odot \hat{X})$ $dX = \frac{\gamma}{\sigma}(dY - \frac{d\beta}{n}) - \frac{d\gamma\gamma\hat{X}}{n\sigma}$

paper that requires manual implementation of these fused kernels, the EVT kenel-level compiler, discussed later, can automatically generate them and achieve state-of-the-art performance.

Besides these registered decomposition rules, a simple interface is provided for users to register custom rules by declaring the pattern and replacement as normal Python functions, as illustrated by the example below:

```python
def pattern(bias, x, weight):
    return torch.ops.aten.addmm(bias, x, weight)

def replacement(bias, x, weight):
    mm = torch.ops.aten.mm(x, weight)
    return torch.ops.aten.add(mm, bias)
```

Reduction Elimination. Operator compilers generating fused kernels usually face challenges in fusing reductions with their consumers, restricting fusion opportunities. This limitation comes from the need for special parallelization that partitions all elements to be reduced to the same threadblock, which is typically incompatible with other operations.

In certain scenarios, a reduction operation can be transformed into a sequence of fusible element-wise operators when it involves operators with a constant reduction value. For instance, with integers $y_{0\leq i\leq m} \in [0, n)$ and a scalar α

$$\sum_{j=0}^{n-1} \alpha OneHot(\begin{bmatrix} y_0 \\ ... \\ y_m \end{bmatrix})_{:,j} = \alpha \begin{bmatrix} 1 \\ ... \\ 1 \end{bmatrix}. \tag{6.2}$$

EVT introduces reduction elimination that automatically identifies these opportunities and rewrites the graph to unlock additional fusion opportunities. A major challenge is that operations with constant reduction values may not always be the direct input to the reduction. There could be a sequence of associable arithmetic operations ($+,-,\times,\div$) between them. Following existing approaches in algebraic reassociation of expression, EVT addresses this challenge with 3 steps:

- Extract the subgraph generating the reduction, containing only operators associable with the reduction. When encountering a non-associable node, replace it with a placeholder and annotate it if it has constant reduction value.

- Perform expression reassociation following Briggs et al., 1994 [105], assigning rank -1 to reduction nodes such that they have the lowest rank. After reassociation, the reductions always take a placeholder as input. If the placeholder has a constant reduction value, fold it to a static tensor.

- Conduct constant propagation, if the subgraph contains no reduction nodes, substitute it back to the original graph.

Other Graph Simplifications. In addition to the specialized optimizations dis-

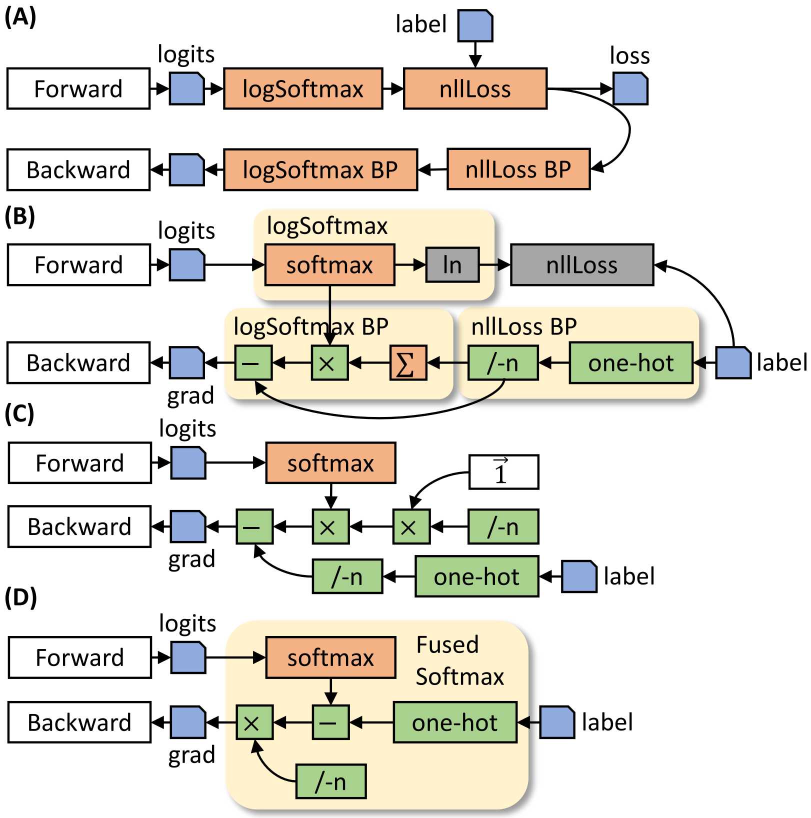


Figure 6.2: Optimization of Softmax Cross-Entropy Loss.

cussed earlier, EVT also included a set of standard graph transformations to further simplify the data-flow graph. These include conventional techniques such as constant propagation with expression reassociation, common factor extraction, common sub-expression elimination, and dead code elimination.

6.2.2 Example

To illustrate how aforementioned optimizations simplify the computation graph and uncover new fusion opportunities, *softmax_cross_entropy* and its backward operations are used as a running example, which can be simplified to a single fused softmax that not only enhances numerical stability but also improves the overall performance.

The *softmax_cross_entropy* is a standard loss function widely adopted by the deep learning community. It can be a performance bottleneck in many cases. For instance, profiling shows that when training Graph Convolutional Neural Networks (GCNs) [106] on ogbn-mag dataset, the loss and its backward pass contribute over 13% of the total iteration time. This inefficiency comes from that large reduction along the batch dimension, which is unfriendly to mixed precision training due to numerical stability concerns.

Figure 6.2 illustrates the traced data-flow graph. It contains four non-fusible nodes. After decomposition with patterns in Table 6.1, these non-fusible nodes are broken down into fundamental fusible nodes. Then, loss elimination eliminates the *nllLoss* as a dead node. While the $\sum$ node, decomposed from *logSoftmax BP*, initially hinders fusion with the decomposed ops from *nllLossBP*, our reduction elimination pass folds it to basic element-wise operations. Finally, the other graph simplification passes clean up the computation graph, eventually enabling it to be fused by the kernel-level compiler. As the optimized computation graph has no reductions, and all the intermediate results are stored in registers with full precision, it has much better numerical stability compared with the original version.

6.3 Operator-level Optimization

With the optimized graph in place, the next step is dividing it into disjoint partitions for fusion. This section first uses a simple example in Figure 6.3 to demonstrate heuristics used by existing studies may fail to find feasible and optimal partitions.

Example. Figure 6.3 shows the joint forward-backward graph of two layers. The element-wise operators are colored green, *mm*s stand for matrix multiplications that can fuse element-wise operators to its output. For feasibility, {*mm2*, *tanh*, *dtanh*} is valid under TVM[45]'s rule, yet it is infeasible due to the cycle. In terms of optimum, while

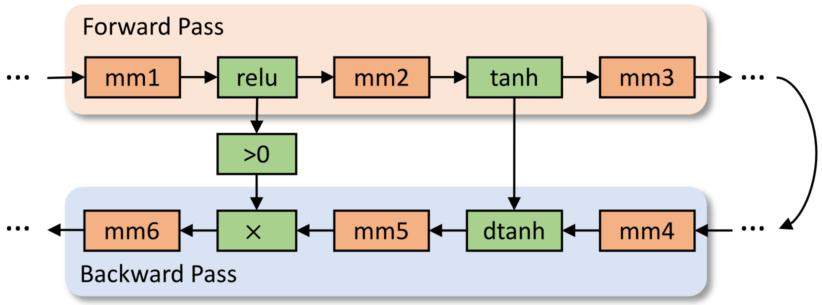


Figure 6.3: Illustrative Example for Partition.

{*mm1*, *relu*} and {*mm5*", > 0, ×} forms a valid partition, it is suboptimal as partition {*mm1*, *relu*, > 0} and {*mm5*, ×} saves more memory access with the output of "> 0" stored under bitmask.

EVT's Approach. EVT presents a novel algorithm that formulates partitioning as an integer linear programming problem that offers three key benefits:

- **Feasibility**: The requirements including acyclic are encoded as ILP constraints.
- **Optimum**: The objective function maximizes saved memory access through fusion in each partition.
- **Extendability**: Additional requirements and heuristics can be easily encoded as constraints.

This section is organized as follows. It first discusses the ILP formulation following Nossack et al., 2014 [47], then presents the novel techniques to reduce its solving time.

6.3.1 Integer Linear Programming Formulation

Let $D = (V, A)$ represents the topological sorted data-flow graph with node set $V = \{v_1, ..., v_n\}$ and edge set $A \subseteq \{(v_i, v_j)|v_i, v_j \in V\}$. The objective is to find disjoint partitions $\{V_1, ..., V_k\}$ of D under certain constraints and heuristics, such that each partition can be fused into a single kernel. The benefit is the sum of memory access caused by the internal edges within each partition. Therefore, each edge is assigned an edge weight $c_{ij} \in \mathbb{N}^+$ describing the size of the tensor passed long it. Following Nossack et al., 2014 [47], four types of decision variables are defined:

- $x_{is} \in \{0, 1\}$: $x_{is} = 1$ when node $v_i \in V$ in cluster V_s.
- $z_{ijs} \in \{0, 1\}$: $z_{ijs} = 1$ indicates both nodes $v_i, v_{j \neq i} \in V$ belong to cluster V_s.
- $y_{st} \in \{0, 1\}$: $y_{st} = 1$ identifies there is an edge between the nodes of cluster V_s and $V_{t \neq s}$.
- $\pi_s \in \mathbb{Z}$: auxiliary variable used to formulate the Miller-Tucker-Zemlin(MTZ) subtour elimination constraints that ensure acyclic partitioning [107].

The ILP problem can be formulated as follows. The object function (6.3a) maximizes the sum of edge weights internal to each partition. Constraint (6.3b) ensures that each node is exclusively partitioned to a partition. Constraints (6.3c) and (6.3d) encode additional constraints and heuristics. Constraints (6.3e) and (6.3f) connect the decision variables x, y, and z. Constraint (6.3g) formulates the acyclic constraint. Constraint (6.3h) reduces the symmetry of the solutions. At last, Constraints (6.3i) and (6.3j) are the bounds of each decision variable.

$$max \sum_{1\leq s\leq n} \sum_{1\leq i\leq j\leq n} (c_{ij} + c_{ji}) \cdot z_{ijs} \tag{6.3a}$$

$$s.t. \sum_{s=1}^{n} x_{is} = 1, \forall 1 \leq i \leq n \tag{6.3b}$$

$$x_{is} + x_{js} \leq 1, \forall 1 \leq s \leq n, v_i \text{ cannot fuse with } v_j \tag{6.3c}$$

$$\sum_{s=1}^{n} z_{ijs} = 1, v_i \text{ must fuse with } v_j \tag{6.3d}$$

$$z_{ijs} \leq x_{is}, z_{ijs} \leq x_{js}, \forall 1 \leq i < j \leq n, 1 \leq s \leq n \tag{6.3e}$$

$$x_{is} + x_{jt} - 1 \leq y_{st}, \forall (v_i, v_j) \in A, 1 \leq s \neq t \leq n \tag{6.3f}$$

$$\pi_s - \pi_t + n \cdot y_{st} \leq n - 1 \forall 1 \leq s \neq t \leq n \tag{6.3g}$$

$$\sum_{i=1}^{n} x_{is} \leq \sum_{i=1}^{n} x_{i,s-1} \forall 2 \leq s \leq n \tag{6.3h}$$

$$x_{is}, z_{ijs}, y_{st} \in \{0, 1\} \forall 1 \leq i < j \leq n, 1 \leq s \neq t \leq n \tag{6.3i}$$

$$\pi_s \in \mathbb{Z}, \forall 1 \leq s \leq n \tag{6.3j}$$

6.3.2 Reduce the Complexity

While Equation (6.3) offers feasibility, optimum, and high extendability, solving it directly is impractical. The four types of variables introduce $O(n^3)$ decision variables, and the branch and bound method further takes exponential time in the worst case to find the optimal solution. To address this challenge, inspired by the fact that neural networks are constructed by stacking similar layers, EVT divides D to disjoint components and solve Equation (6.3) separately in each component. Additionally, EVT caches the solutions to avoid solving the same ILP problem twice for components with similar structures. The

Algorithm 4: Dividing Node Set

```
Input: Computation Directed Acyclic Graph $D = (V, A)$.
A list of edges $A_{ne} \subset A$ such that $\forall (v_i, v_j) \in A_{ne}$, $v_j$ cannot be fused into $v_i$.
A list of nodes pairs $A_{nv}$, $(v_i, v_j) \in A_{nv}$ indicates $v_i$ and $v_j$ cannot be fused
Output: Disjoint components $\{C_1, ..., C_m\}$.
Create the directed graph $\bar{D} = (V, A/A_{ne})$
Construct a dict $R$ that maps each node $v_i \in V$ to all the nodes it can reach through directed
  edges in $D$.
for $v_i \in V$ do
    Unfusible Set $S_i = \{\}$
    for $v_j \in R[v_i]$ & $(v_i, v_j) \in A_{nv}$ do
        $S_i = S_i \cup \{v_j\} \cup R[v_j]$
    for $v_k \in S_i$ do
        remove edge $(v_i, v_k)$ from $\bar{D}$ if exists
$\{C_1, ..., C_m\}$ = weakly_connected_components($\bar{D}$)
```

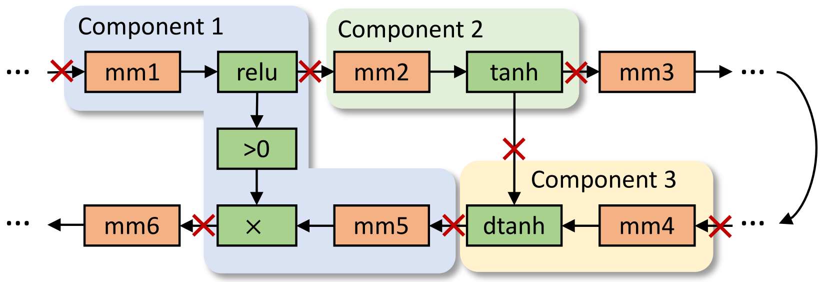


Figure 6.4: Motivating examples for Dividing Node Set.

major challenge is ensuring that the combined solution from each component remains both feasible and optimal. The two-step solution is presented to address this challenge:

- Divide node set V to disjoint components $\{C_1, ..., C_m\}$ without hurting optimum.
- Reconstruct the edges in each component that ensures the feasibility of the solution and solve Equation (6.3).

Dividing Node Set. The node set is divided with Algorithm 4. The insight is that in the optimal solution, nodes belong to the same partition only if they are weakly connected by fusible edges. Thus, all non-fusible edges can be cut to divide D into weakly connected components. Figure 6.4 shows that this approach divides D into multiple small

Algorithm 5: Reconstructing Edges and Solving ILP of Each Component

```
Input: Computation Graph $D = (V, A)$. $\bar{D} = (V, \bar{A})$ and components $\{C_1, ..., C_m\}$ from
       Algorithm 4.
Output: Partitions $\{V_1, ..., V_k\}$, Bool indicator $is\_optimal$
Create graph $\tilde{D} = (V, A + \bar{A}^T)$.
Partitions = {}, $is\_optimal$ = true
for $C_i \in \{C_1, ..., C_m\}$ do
    Get the directed graph $\tilde{D} \backslash C_i$ that excludes nodes in $C_i$.
    Get the one-hop ancestors $N_i^a$ and descendants $N_i^d$
      $N_i^a = \{v_j | v_j \notin C_i, \exists v_k \in C_i, (v_j, v_k) \in A\}$ $N_i^d = \{v_j | v_j \notin C_i, \exists v_k \in C_i, (v_k, v_j) \in A\}$.
    Extract the subgraph $D_i$ containing $V_i' = C_i \cup N_i^a \cup N_i^d$:
      $D_i = \{V_i', A_i' = \{(v_i, v_j) | (v_i, v_j) \in A, v_i, v_j \in V_i'\}\}$
    for $v_d \in N_i^a$ do
        for $v_s \in N_i^d \neq v_d$ do
            if has_path($\tilde{D} \backslash C_i$, $v_d$, $v_s$) then
                if not $\exists$ path $v_s \to v_d$ in $D$ including nodes from at least two components then
                    add_edge($D_i$, ($v_d$, $v_s$))
                    if !has_path($D \backslash C_i$, $v_d$, $v_s$) then
                        $is\_optimal$ = false
    Solve Equation (6.3) with additional constraint that each node in $N_i^a$ and $N_i^b$ form a
      partition with size 1.
    for $n \in [1, |C_i|]$ do
        $V_n = \{v_n | x_{nk} == 1\}$
        if $V_n \neq \emptyset$ then
            Partitions.append($V_n$)
```

components with few nodes while leaving the partition of component 1 to the ILP solver.

Two types of edges are identified as non-fusible in Algorithm 4. The first (line 1) includes edges directly connecting two non-fusible nodes, such as the incoming edges to *mm*s. The second type (line 2-8) involves edges that, if fused, would create cycles. A necessary condition from Theorem 3.3 in Nossack et al., 2014 [47] is leveraged: if a directed path from v_i to v_j exists in D and v_i, v_j belong to different partitions, then all nodes that can be reached from node v_j cannot be in the same partition with v_i. In Figure 6.4, the edge between "*tanh*" and "*dtanh*" is cut based on this rule, as "*tanh*" cannot be fused with "*mm3*" while "*mm3*" can reach "*tanh*".

Reconstructing Edges and Solving ILP. With Algorithm 5, for each component,

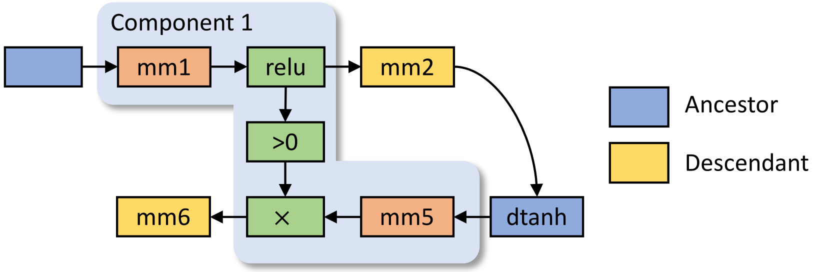


Figure 6.5: Example of Reconstructing Edges for Feasibility.

the edges are constructed to ensure feasibility before solving the ILP. Theorem 6.3.1 below guarantees its solutions are always feasible, and they are optimal under an easily verifiable condition, which always holds during our evaluation of various DL models.

Theorem 6.3.1 *Algorithm 5 returns the global optimal and feasible solution when is_optimal is true. Otherwise, it returns a feasible solution. (Proof: See Appendix B.1).*

Figure 6.5 uses component 1 from Figure 6.4 as an illustrative example. A node is a descendant if it has an incoming edge from the component (e.g. "*mm2*"), and it is an ancestor if it has an outgoing edge to the component (e.g. "*dtanh*"). The edges can be constructed in 3 steps:

- Identify all the ancestors and descendants of the current component (line 5,6).
- Modify D by replacing any edges that can be fused with bidirectional edges (line 4, 10).
- Add edges from any descendants to any ancestors if a path exists in the modified graph between them(line 11), e.g. the edge *mm2*→*dtanh* in Figure 6.5.

The intuition is that the solutions from Figure 6.5 are feasible and optimal if the acyclic constraint is neither loosened nor tightened. In other words, a cycle exists in

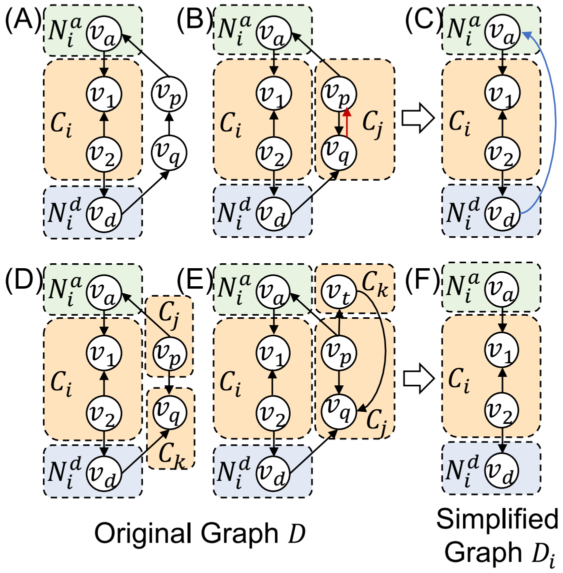


Figure 6.6: Adding edges from descendants to ancestors.

Figure 6.4 involving nodes in component 1 is a sufficient and necessary condition for a cycle exists in Figure 6.5.

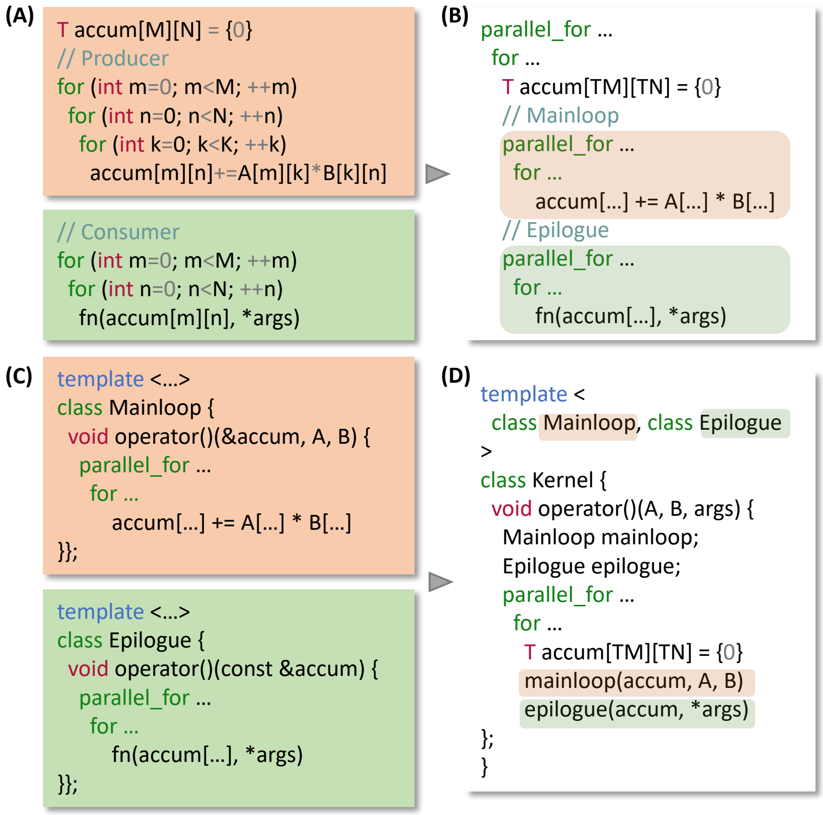
Ancestors and descendants serve as auxiliary nodes. If a fusion in component 1 creates a cycle involving nodes outside the component, the cycle must contain pairs of descendants and ancestors. The partitioner then uses these as boundaries to break the cycle into paths within and outside the component. The condition becomes: a path exists in Figure 6.4 between any pair of descendants and ancestors is a sufficient and necessary condition for a path to exist in Figure 6.5 with the same source and target.

Internal paths from ancestors to descendants are ensured sufficient and necessary by preserving all internal edges. However, external paths from descendants to ancestors can be influenced by fusion in other components. In detail, while fusion does not remove paths, it creates new ones equivalent to changing fused edges to bidirectional. For example, fusing v_p and v_q in Figure 6.6 (B) creates a path from $v_d \to (v_p, v_q) \to v_a$ equivalent to adding the red arrow. To ensure feasibility, edges from any descendant to any ancestor in Figure 6.5 are added when a path could exist between them. Figure 6.6 illustrates four different scenarios. Figure 6.6 (A,B) shows scenarios when the edge should be added. In (A), there is a natural path $v_d \rightarrow v_a$. In (B), v_p and v_q belong to the same component and could be fused, which creates the path $v_d \rightarrow (v_p, v_q) \rightarrow v_a$. Oppositely, Figure 6.6 (D,E) shows cases when the edge is not needed. In (D), v_p and v_q belong to different components, so they cannot be fused. In (E), although v_p and v_q belong to the same component, fusing them is infeasible as it creates a cycle $(v_p, v_q) \leftrightarrow v_k$. While this method ensures the condition is necessary, it can also be sufficient (optimal) if whenever a path exists in the modified graph, there is always a path that exists in the original graph. Line 12-13 checks this condition and returns indicator *is_optimal*.

Caching the Solutions. The ILP solutions are stored in a database so that they can be retrieved for components with isomorphic structures. Particularly, keys are created with the Weisfeiler Lehman graph hashing [108] so that isomorphic components will have identical hash.

6.4 Kernel-level Optimization

This section first explains the key abstraction under our operator compiler, ”Epilogue Visitor”. This abstraction allows fusing operators under the producer-consumer relationship by directly leveraging expert-designed implementations for the producer part and ensuring optimal performance while assembling the consumer part at compile time to accommodate to different fusion patterns. Under the abstraction, the EVT kernel-level compiler is presented, which automatically generates fused kernels with state-of-the-art performance while supporting diverse patterns.



```
T accum[M][N] = {0}
// Producer
for (int m=0; m<M; ++m)
 for (int n=0; n<N; ++n)
  for (int k=0; k<K; ++k)
   accum[m][n]+=A[m][k]*B[k][n]
```

```
// Consumer
for (int m=0; m<M; ++m)
 for (int n=0; n<N; ++n)
  fn(accum[m][n], *args)
```

```
parallel_for ...
 for ...
  T accum[TM][TN] = {0}
  // Mainloop
  parallel_for ...
   for ...
     accum[...] += A[...] * B[...]
  // Epilogue
  parallel_for ...
   for ...
     fn(accum[...], *args)
```

```
template <...>
class Mainloop {
 void operator()(&accum, A, B) {
  parallel_for ...
   for ...
     accum[...] += A[...] * B[...]
}};
```

```
template <...>
class Epilogue {
 void operator()(const &accum) {
  parallel_for ...
   for ...
     fn(accum[...], *args)
}};
```

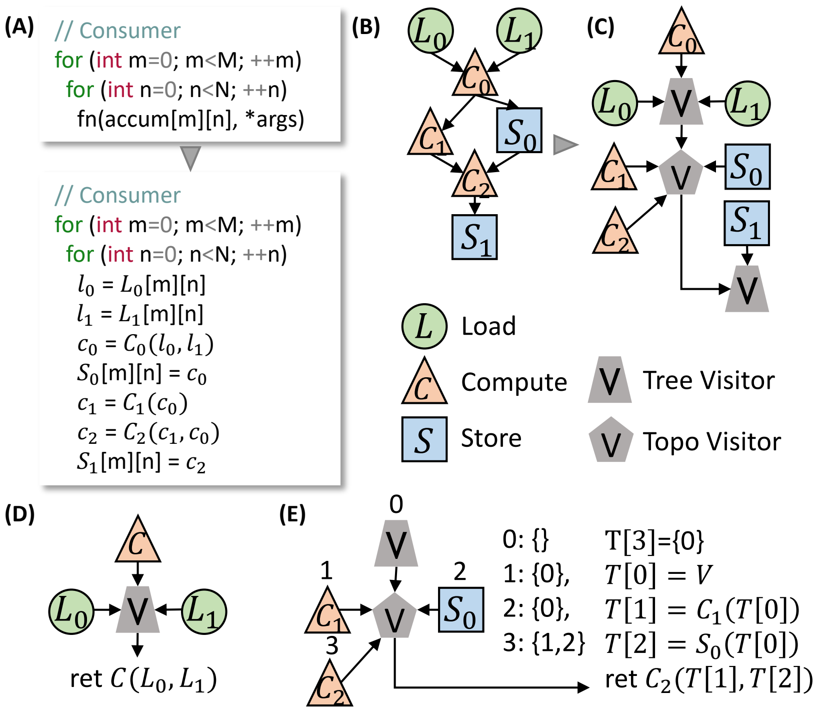
```
template <
 class Mainloop, class Epilogue
>
class Kernel {
 void operator()(A, B, args) {
  Mainloop mainloop;
  Epilogue epilogue;
  parallel_for ...
   for ...
    T accum[TM][TN] = {0}
    mainloop(accum, A, B)
    epilogue(accum, *args)
};
}
```

Figure 6.7: Mainloop-Epilogue Abstraction

6.4.1 Epilogue Visitor Abstraction

The Epilogue Visitor abstraction originates from the Mainloop-Epilogue abstraction widely adopted by existing template libraries and TMP-based compilers [54].

Mainloop-Epilogue Abstraction and Its Limitation. As shown in Figure 6.7(A), the Mainloop-Epilogue abstraction fuses operators under the producer-consumer relationship. It requires the consumer to have the same set of spatial loops as the producer so that they can be fused by injecting the statement of the consumer into the end of the producer's spatial loops. Subsequently, As shown in Figure 6.7(B), the loops are restructured to achieve optimal performance through techniques such as tiling, parallelization, and software pipelining. The Mainloop-Epilogue abstraction takes the producer as the



```
// Consumer
for (int m=0; m<M; ++m)
  for (int n=0; n<N; ++n)
    fn(accum[m][n], *args)
```

```
// Consumer
for (int m=0; m<M; ++m)
  for (int n=0; n<N; ++n)
    $l_0$ = $L_0$[m][n]
    $l_1$ = $L_1$[m][n]
    $c_0$ = $C_0(l_0, l_1)$
    $S_0$[m][n] = $c_0$
    $c_1$ = $C_1(c_0)$
    $c_2$ = $C_2(c_1, c_0)$
    $S_1$[m][n] = $c_2$
```

Figure 6.8: Epilogue Visitor Tree Abstraction

mainloop and the consumer as the epilogue. As shown in Figure 6.7(C), different mainloops and epilogues are developed separately as different C++ classes, and a universal kernel template in Figure 6.7 (D) takes the mainloop and epilogue as template arguments. This modular approach allows assembling different mainloop-epilogue pairs at compilation time and supports |mainloop| × |epilogue| patterns.

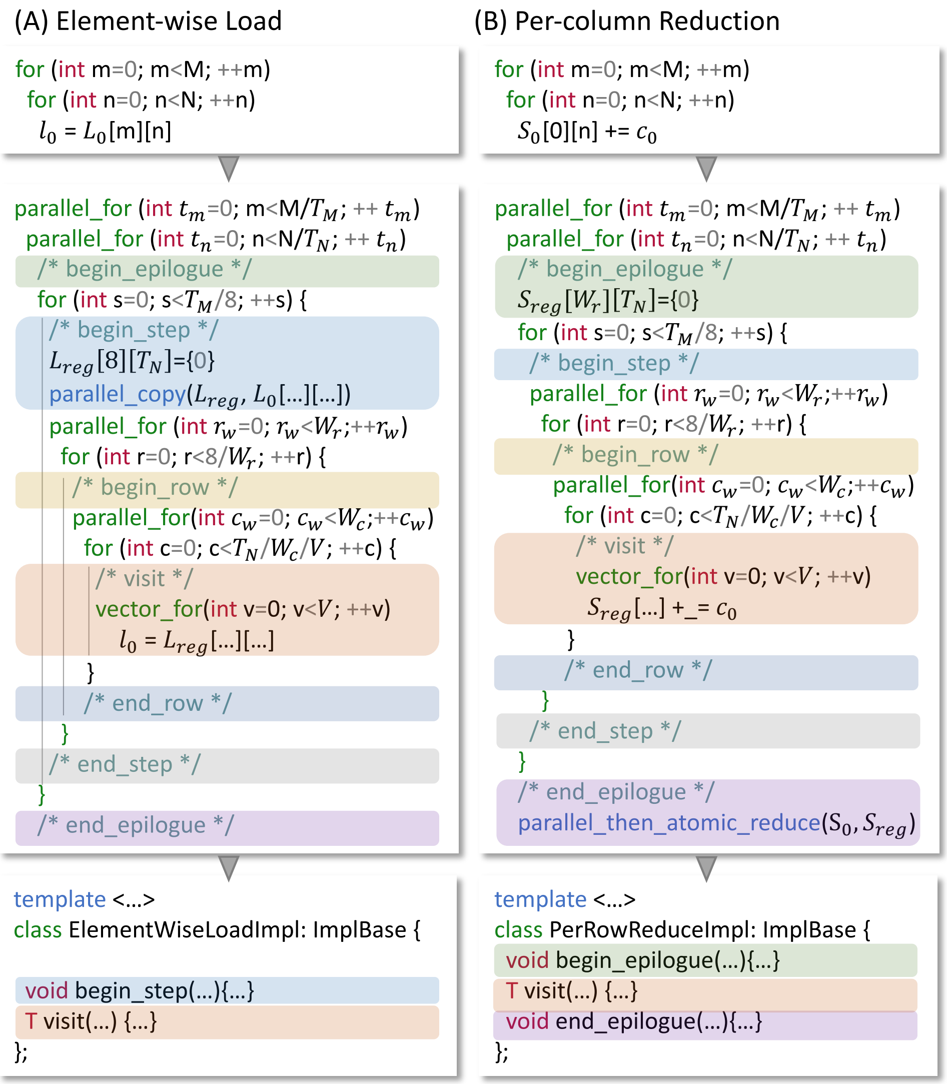
However, the "Epilogue" of the Mainloop-Epilogue abstraction is not scalable as the number of consumer operators grows rapidly. It is impractical to implement all potential epilogues manually, optimizing and verifying intricate epilogues also poses significant challenges.

Epilogue Visitor Abstraction. The EVT abstraction preserves the mainloops and further modularizes the epilogue to address its scalability issue. As shown in Figure 6.8 (A,B), the consumer is modeled as a computation graph instead of a single operator, the key insight is that the rapid growth of potential consumer operators occurs on the

edge set instead of the **node set**, i.e. different consumers have different combinations of the same set of basic operators. This insight motivates the design to manage a library of high-performance implementation of basic operators decoupled from each other, and develop a compiler to assemble them at compile time for different consumers. The detailed explanation is as follows.

Supported Patterns. EVT supports any consumers that can be written as a perfectly nested loop, where all the statements are in the spatial loops of the producer (Figure 6.8 (A)). This structural constraint has proven to be key in simplifying transformations and has been widely adopted by many state-of-the-art compilers like the *linalg* dialect of MLIR [109]. This requirement is loose enough to capture most scenarios in neural networks, including element-wise load/store, broadcast, reduction (atomic), and element-wise computations.

Decouple Epilogue Operators with Visitor. Figure 6.8 shows the perfectly nested loop representation of the epilogue, where the original function can be decomposed into a series of load, store, and compute operators. Figure 6.8(B) visualizes it as a graph. Notably, the store is modeled as a transparent node that forwards its input. To decouple the implementation of each node, the ”visitor” is introduced to rewrite the graph to Figure 6.8(C), where all nodes become leaves. Two types of visitors are introduced. The tree visitor in Figure 6.8 (D) handles non-leave nodes with output degree 1. The topological visitor in Figure 6.8 (E) takes other general cases. It sorts the inputs by topological order and creates an edge list describing the dependency between them. At runtime, the topological visitor holds intermediate results from each input node with a temporary register buffer.



(A) Element-wise Load

```
for (int m=0; m<M; ++m)
 for (int n=0; n<N; ++n)
  l_0 = L_0[m][n]
```

```
parallel_for (int t_m=0; m<M/T_M; ++ t_m)
 parallel_for (int t_n=0; n<N/T_N; ++ t_n)
  /* begin_epilogue */
  for (int s=0; s<T_M/8; ++s) {
   /* begin_step */
   L_reg[8][T_N]={0}
   parallel_copy(L_reg, L_0[…][…])
   parallel_for (int r_w=0; r_w<W_r;++r_w)
    for (int r=0; r<8/W_r; ++r) {
     /* begin_row */
     parallel_for(int c_w=0; c_w<W_c;++c_w)
      for (int c=0; c<T_N/W_c/V; ++c) {
       /* visit */
       vector_for(int v=0; v<V; ++v)
        l_0 = L_reg[…][…]
      }
     /* end_row */
    }
   /* end_step */
  }
  /* end_epilogue */
```

```
template <…>
class ElementWiseLoadImpl: ImplBase {

 void begin_step(…){…}
 T visit(…) {…}
};
```

(B) Per-column Reduction

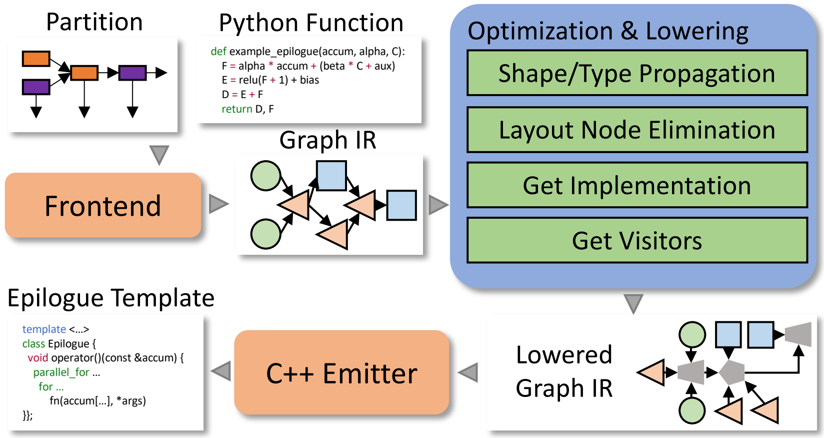
```
for (int m=0; m<M; ++m)
 for (int n=0; n<N; ++n)
  S_0[0][n] += c_0
```

```
parallel_for (int t_m=0; m<M/T_M; ++ t_m)
 parallel_for (int t_n=0; n<N/T_N; ++ t_n)
  /* begin_epilogue */
  S_reg[W_r][T_N]={0}
  for (int s=0; s<T_M/8; ++s) {
   /* begin_step */
   parallel_for (int r_w=0; r_w<W_r;++r_w)
    for (int r=0; r<8/W_r; ++r) {
     /* begin_row */
     parallel_for(int c_w=0; c_w<W_c;++c_w)
      for (int c=0; c<T_N/W_c/V; ++c) {
       /* visit */
       vector_for(int v=0; v<V; ++v)
        S_reg[…] +_= c_0
      }
     /* end_row */
    }
   /* end_step */
  }
  /* end_epilogue */
  parallel_then_atomic_reduce(S_0, S_reg)
```

```
template <…>
class PerRowReduceImpl: ImplBase {
 void begin_epilogue(…){…}
 T visit(…) {…}
 void end_epilogue(…){…}
};
```

Figure 6.9: Implementation of Epilogue Operators

Implementation of Epilogue Operators. Figure 6.9 illustrates how the decoupled operators can be optimized and implemented separately. The original loop structure of all the operators is reconstructed into the same nested loops for optimal performance, with *parallel_for* tied to GPU thread hierarchies and *vector_for* vectorized. For convenience, the code blocks at the start of the sequential loops are named as *begin_xxx*, and those at the end as *end_xxx*. The code block of the innermost sequential loop is called *visit*.



```
def example_epilogue(accum, alpha, C):
  F = alpha * accum + (beta * C + aux)
  E = relu(F + 1) + bias
  D = E + F
  return D, F
```

```
template <...>
class Epilogue {
 void operator()(const &accum) {
  parallel_for ...
   for ...
     fn(accum[...], *args)
}};
```

Figure 6.10: Overview of the EVT Operator Compiler

While all operators can be implemented by only filling the *visit* code block because of the perfectly nested loop constraint, better performance can be achieved by leveraging other blocks. For instance, in Figure 6.9 (A), the element-wise load can be optimized by first copying a matrix fraction into registers at *begin_step* and then accessing them in *visit*. This ensures coalesced memory access and minimized address/predicate arithmetic. In Figure 6.9 (B), the per-column reduction can be accelerated by initiating a temporary buffer at *begin_epilogue* to store partial results accumulated during *visit*, then perform the parallel reduction in *end_epilogue* before atomically reducing the result to global memory. Finally, as shown at the bottom of Figure 6.9, each operator can be implemented as a template class providing member functions defining the non-empty code blocks. These functions will be called by the visitors at runtime.

6.4.2 EVT Kernel-level Compiler

With the Epilogue Visitor abstraction, a novel kernel-level compiler is presented: Epilogue Visitor Tree (EVT). The ”Tree” comes from the fact that the transformed graph in Figure 6.8 (C) becomes a tree whose root is the last visitor. The overview of the compiler is shown in Figure 6.10.

The core of the compiler is a graph intermediate representation (IR) shown in Figure 6.11. It models the epilogue as a graph, where each node represents an operator. The nodes are classified into four types: load, compute, store, and layout. Each node also has four fields: name, shape, stride, and data type. The shape and stride are tuples following the cute library of CUTLASS [1]. For instance, shape (m, n) and stride $(n, 1)$ represent a $m \times n$ matrix stored in the row-major layout. The index to the storage of the tensors in memory can be computed via inner product between shape and stride.

To support different input formats, the IR is constructed with minimum information and constraints by frontends. Then, the optimization and lowering schedules passes that fill in missing information and canonicalize the IR to meet the epilogue visitor abstraction. The C++ emitter then traverses the IR and emits the template class for the epilogue. This design allows adding new frontends for other compilation workflows easily while reusing the rest of the logic.

Frontend. EVT supports diverse user inputs from partitions generated by the previous section to Python functions. An example of the Python frontend is shown below, which defines the epilogue in Figure 6.11 (A).

```
def example_epilogue(accum, bias):
    add = reshape(accum, new_shape=(128, 4, 64) + bias
    permute_1 = permute(add, indices=(2,1,0))
    reduce = sum(permute_1, dim=[0, 1])
    out = relu(add) * permute(permute_1, indices=(2,1,0))
    return reduce, out

tensors = {
  "accum": Tensor(torch.float32, shape=(128, 256)),
  "bias": Tensor(torch.float16, shape=(4, 64)),
  "reduce": Tensor(torch.float32, shape=(128,)),
  "out": Tensor(torch.float16, shape=(128, 4, 64))}
epilogue = python_ast.trace(example_epilogue, tensors)
```

[1] https://github.com/NVIDIA/cutlass/blob/main/media/docs/cute/01_layout.md

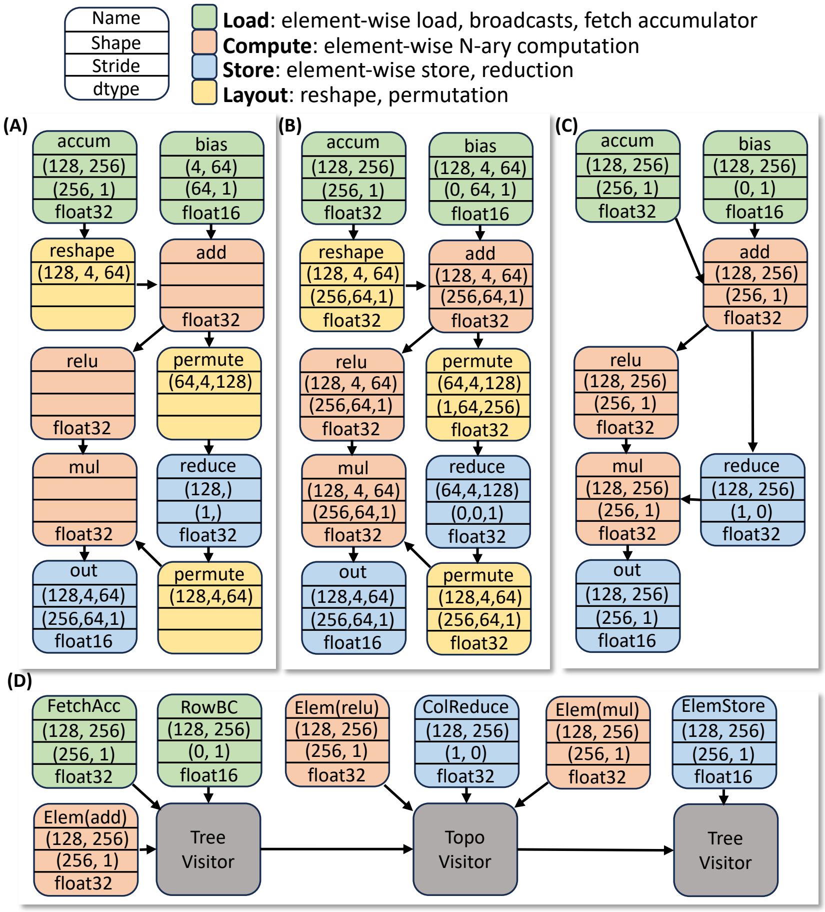

Name
Shape
Stride
dtype
Load: element-wise load, broadcasts, fetch accumulator
Compute: element-wise N-ary computation
Store: element-wise store, reduction
Layout: reshape, permutation
(A)
accum (128, 256) (256, 1) float32
bias (4, 64) (64, 1) float16
reshape (128, 4, 64)
add float32
relu float32
permute (64,4,128)
mul float32
reduce (128,) (1,) float32
out (128,4,64) (256,64,1) float16
permute (128,4,64)
(B)
accum (128, 256) (256, 1) float32
bias (128, 4, 64) (0, 64, 1) float16
reshape (128, 4, 64) (256,64,1) float32
add (128, 4, 64) (256,64,1) float32
relu (128, 4, 64) (256,64,1) float32
permute (64,4,128) (1,64,256) float32
mul (128, 4, 64) (256,64,1) float32
reduce (64,4,128) (0,0,1) float32
out (128,4,64) (256,64,1) float16
permute (128,4,64) (256,64,1) float32
(C)
accum (128, 256) (256, 1) float32
bias (128, 256) (0, 1) float16
add (128, 256) (256, 1) float32
relu (128, 256) (256, 1) float32
mul (128, 256) (256, 1) float32
reduce (128, 256) (1, 0) float32
out (128, 256) (256, 1) float16
(D)
FetchAcc (128, 256) (256, 1) float32
RowBC (128, 256) (0, 1) float16
Elem(relu) (128, 256) (256, 1) float32
ColReduce (128, 256) (1, 0) float32
Elem(mul) (128, 256) (256, 1) float32
ElemStore (128, 256) (256, 1) float16
Elem(add) (128, 256) (256, 1) float32
Tree Visitor
Topo Visitor
Tree Visitor

Figure 6.11: Example of EVT transformations.

Shape Type Propagation. This pass takes two stages. The first stage visits the nodes in topological order to fill in missing strides, shapes, and data types. The second stage, in reversed topological order, updates the shape and stride to remove implicit broadcast. For example, the shape and stride of *bias* are updated from (4,64):(64,1) to (128,4,64):(0,64,1), with the first dimension broadcasted.

Layout Node Elimination. This pass aims to canonicalize the IR by removing layout nodes. In the context of the epilogue visitor abstraction, all operators must align with the spacial loops of the producer, which can be translated to all the nodes having the same shape with the *accum* node with our IR. However, as shown in Figure 6.11

(B), layout nodes alter shapes and violate this requirement. EVT removes layout nodes by propagating them away from the *accum* through swapping with its neighbors. These layout nodes will eventually merge into a load or store node by updating their shape and stride. The result is shown in Figure 6.11 (D), all the operators now share the same shape.

Get Implementation and Visitor. This pass lowers the canonical IR to specific implementations. While the proposed IR only has four types of nodes for simplicity, each type can have multiple underlying implementations for different strides. For instance, the *reduce* has stride $(1, 0)$ that reduces the matrix into a column vector, while the *out* has stride $(256, 1)$ that stores the matrix to memory in row-major. This pass infers the implementation of each node based on its stride and injects the visitors for each non-leaf node based on its output degree. The result is shown in Figure 6.11 (D).

C++ Emitter. The proposed compiler provides a high-performance operator library including popular epilogue operators. The C++ Emitter emits the template class of the epilogue by simply traversing the IR in Figure 6.11 (D) and emitting each node in post-order. Notably, for operators not covered by our library, with our epilogue visitor abstraction, they can be implemented easily or generated by compiler techniques.

6.4.3 Additional Features

This section summarizes additional features EVT provides to improve flexibility and performance.

Dynamic Shapes. The epilogue operators take the shape and stride as runtime parameters so that the same kernel compiled can support inputs with dynamic shapes.

Performance. EVT is compatible with the StreamK[110] for better workload balance. It also introduces the ping-pong buffer to the epilogue on Ampere GPU, enabling overlapping the computation and memory access in the epilogue.

Mainloop Fusion. While EVT focuses on epilogue fusion, it also offers the capability of fusing permutations of mainloop arguments for particular producers, such as matrix multiplication. In detail, CUTLASS indices the multiplicand and multiplier in matrix multiplication using the strides provided through its runtime arguments. EVT leverages this feature and recomputes the stride of permuted inputs, which ensures the multiplicand and multiplier are accessed correctly without actually permuting their data in memory.

6.5 Evaluation

This section evaluates the performance of EVT on five real-world NN architectures and compares it with existing compilers. Section 6.5.2 evaluates the end-to-end training performance. Section 6.5.3 provides additional evaluations on representative layers of each architecture to explain the rationale behind EVT's speedup. Besides, it also enables comparison with inference compilers such as TVM [45].

6.5.1 Experiment Setup

Benchmarks. Five models are selected including BERT-Large [111], VIT [112], ResNet-50 [113], XML-CNN [114], and GCN [106] to cover diverse architectures and domains. Details on the rationale behind benchmark selection are available in supplemental material.

Platform. Single NVIDIA A100 GPU (40 GB) with CUDA 12.1 and NGC docker `nvcr.io/nvidia/pytorch:23.07-py3`.

Baselines. In the end-to-end benchmark, EVT is compared with Torch Inductor [98] and NVFuser. The former integrates various state-of-the-art compiler techniques from Triton [101] to CUTLASS[54]. The mode is set to *max-autotune* for optimal performance.

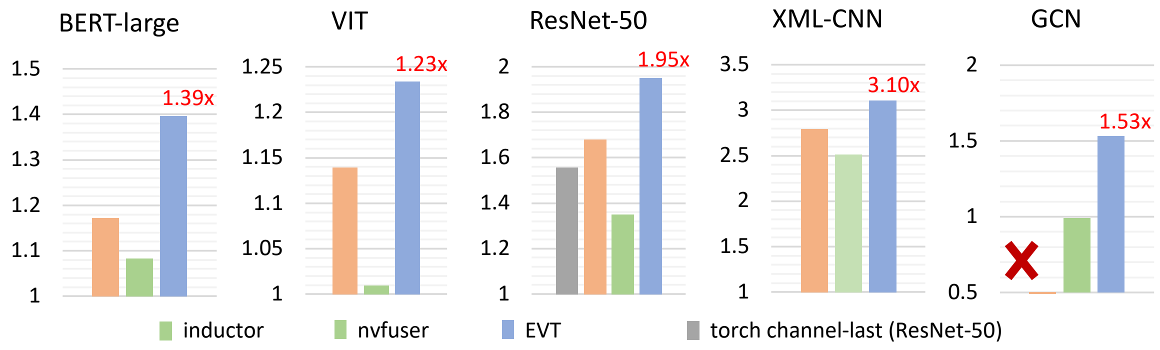


Figure 6.12: End-to-end Training Speedup over the PyTorch Implementation.

The NVFuser is an operator compiler for GPU that just-in-time compiles fast and flexible GPU-specific code. In layer-wise benchmarks, EVT is compared with Triton[101] and TVM [45]. For TVM, GEMM and convolutions are tuned by *autotvm* to leverage Tensor Cores, while others are generated through Ansor [115]. The CUDA Graph and AMP are applied to all methods for fair comparison.

6.5.2 End-to-End Benchmarks

The end-to-end training speedup for the five benchmarks is summarized in Figure 6.12 with results normalized to the naive PyTorch Implementation. EVT achieves 1.23∼3.10× end-to-end training speedup across all benchmarks, outperforming existing compilers.

6.5.3 Layer-Wise Benchmarks

This section evaluates EVT on representative layers from the five models. For each benchmark, the computation graph is visualized from Triton, TVM, and our EVT. Operators fused are grouped with boxes, with orange indicating heavy ops (e.g. GEMM, Softmax) and green indicating light ones (element-wise / reductions). The normalized latency over the PyTorch is also shown, which is divided into heavy and light ops. The speedups are also annotated for convenience.

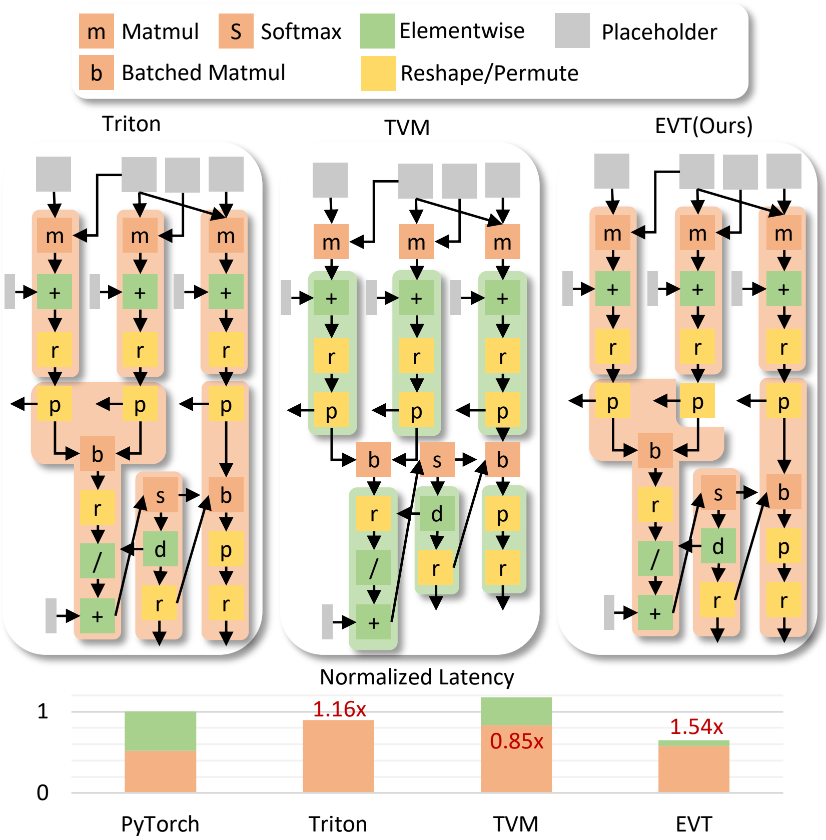


Figure 6.13: The Self-Attention Layer. (Bert, VIT)

Self-Attention Layer. Accelerating the self-attention layer requires the compiler to 1) efficiently handle different permutations and reshapes of the same tensor, and 2) generate high-performance fused kernels. EVT handles 1) through the *cute*-based IR design and passes in Section 6.4.2 and guarantees 2) through the Epilogue Visitor abstraction. In contrast, fused kernels generated by Triton are less efficient than ours, while TVM does not handle 1) efficiently.

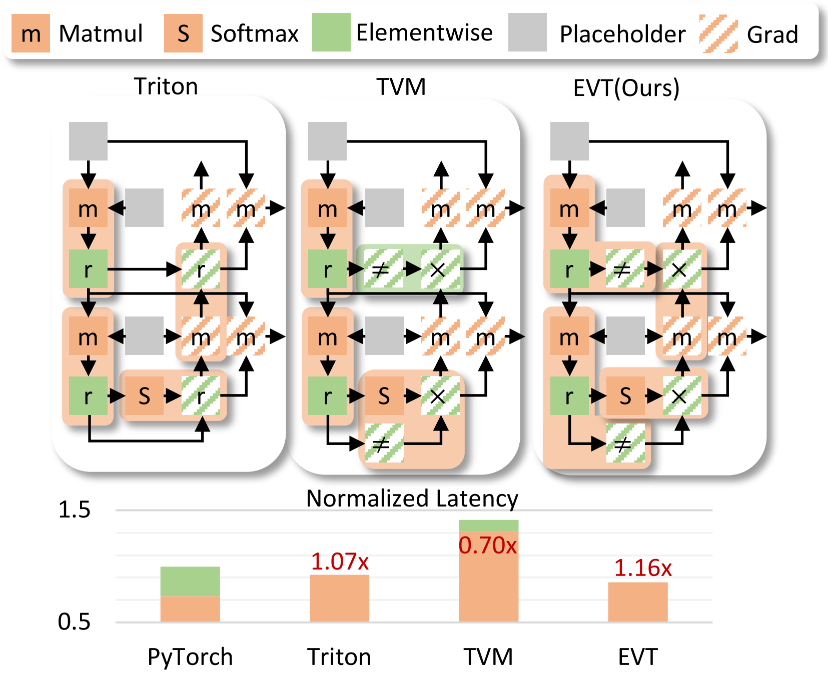


Figure 6.14: The Multilayer Perceptron (MLP). (Bert, VIT)

MLP. Optimizing the MLP requires the compiler to find the feasible and optimal partition of the computation graph. In Figure 6.14, EVT identifies the optimal cut between ≠ and × decomposed from the ReLU backward through the partitioner in Section 6.3, while Triton and TVM generate suboptimal partitions.

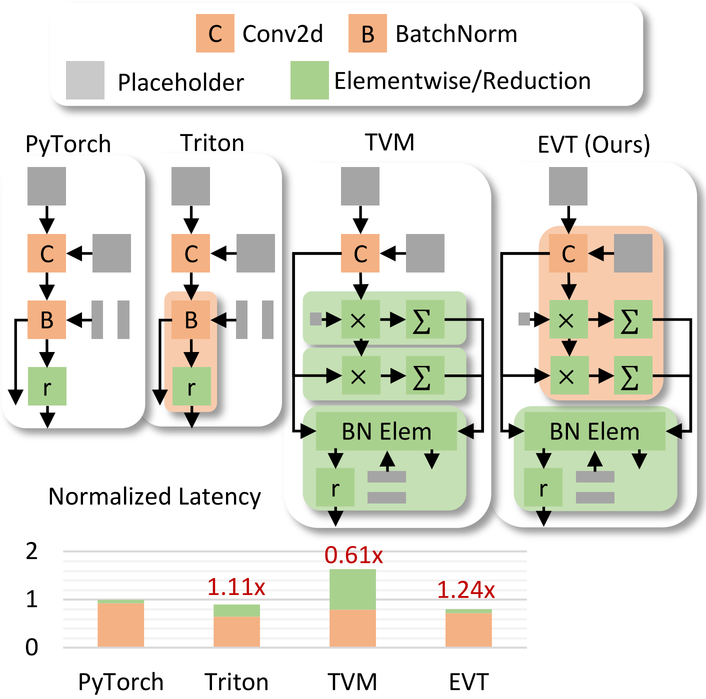


Figure 6.15: The Conv-BN-ReLU Layer. (ResNet)

Conv-BN-ReLU. The major challenge of BN under the channel-last layout is the strided access to the reduction dimension, so optimizing this layer requires compilers to 1) decompose the batch norm operator to create new fusion opportunities, and 2) generate high-performance kernels under the channel-last layout. EVT addresses this by decomposing BN to reduction and element-wise, and the reduction is fused to the preceding convolution efficiently. In contrast, Triton's fused BN-ReLU is less performant due to the layout, and TVM fails to fuse the reductions.

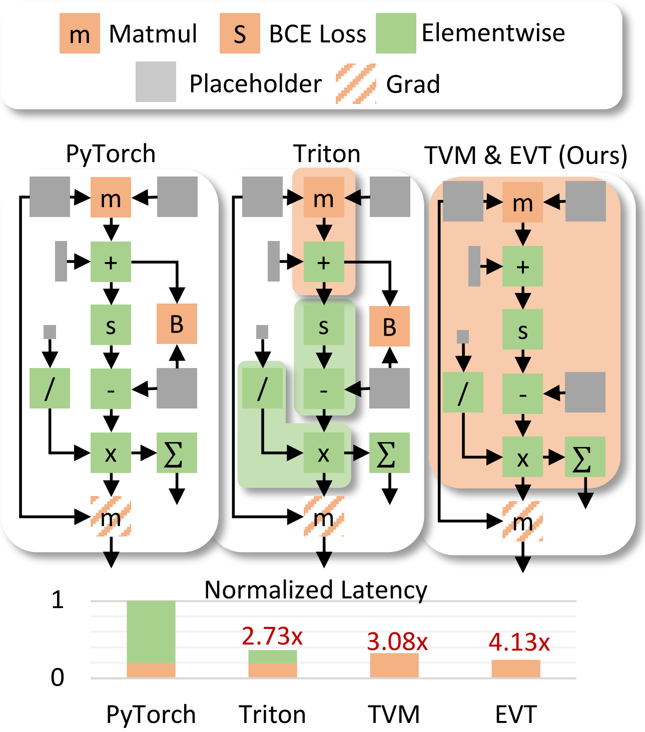


Figure 6.16: The Binary Cross-Entropy Loss. (XML-CNN)

Binary Cross-Entropy Loss. Accelerating BCE loss involves 1) identifying fusion opportunities on the fence of the forward and backward pass and 2) generating high-performance fused GEMM with diverse epilogue operators, including broadcast, various element-wise operators, and reductions. EVT accomplishes 1) through its loss elimination pass and achieves 2) through the Epilogue Visitor abstraction. In contrast, Triton failed to cross the boundary between forward and backward graphs, while TVM creates less performant kernels.

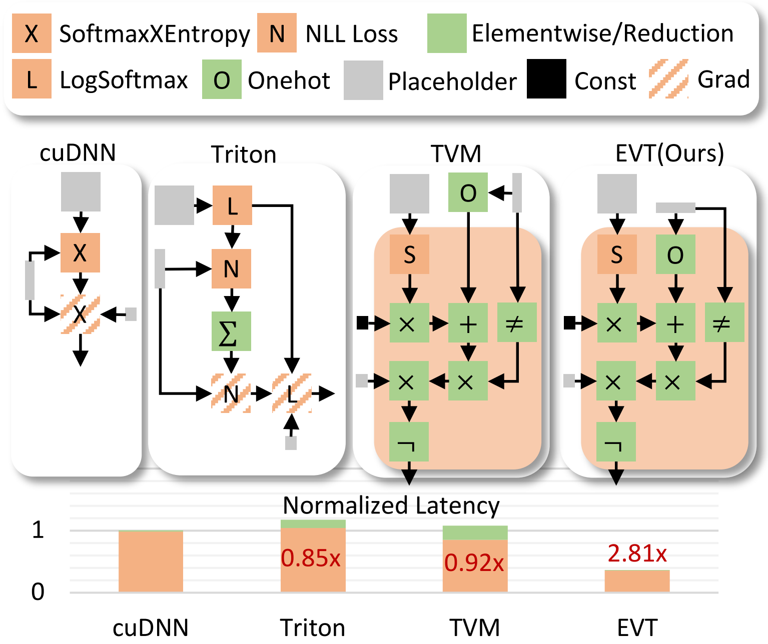


Figure 6.17: Softmax Cross-Entropy Loss. (GCN)

Softmax Cross-Entropy Loss. This graph shares similar optimization requirements with BCE loss. With the graph-level optimizations and operator compiler, EVT fuses the entire graph into a single kernel that offers 2.81× speedup over cuDNN. Oppositely, Triton failed to identify fusion opportunities between forward and backward graphs, while TVM creates less efficient kernels.

6.6 Conclusion & Discussion

EVT enables generating fused kernels with state-of-the-art performance while accommodating diverse fusion patterns. Along with the algorithm-level optimizations and partitioner that fully unleash its potential, EVT optimizes the deep learning training workload automatically and achieves state-of-the-art performance.

Impact on Multi-GPU/Node Training. While this chapter majorly focuses on a single GPU in this paper, EVT can be easily leveraged to accelerate the multi-GPU/node training. Particularly, a recent trend is fusing GEMMs with succeeding communication kernels that allow overlapping between computation and communication at the tile granularity [116, 117]. While this technique originally required manually implementing these fused kernels by experts, with EVT, the communication operator can be implemented as a special type of store node while reusing the rest of the EVT infrastructure.

Chapter 7

Conclusion

This dissertation presents how the utilization of GPGPU when executing deep learning workloads can be effectively improved through kernel-centric optimization.

7.1 Summary of Contributions

The above four chapters demonstrate that the kernel-centric optimization is not only about implementing high-performance kernels alone, but also about codesigns with upper stacks from algorithm design to operator definitions.

With codesign between algorithm and kernel levels, the vecSparse in Chapter 3 achieves 1.41× end-to-end speedup and 13.37× peak memory reduction on the sparse transformer inference task. The new sparse pattern designed on the algorithm level provides higher data reuse opportunities with linear granularity scaling, as opposed to the quadratic scaling seen in previous studies. The TCU-based 1-D Octet tiling strategy proposed in kernel-level optimization addresses numerous pipeline stall reasons, which are responsible for the low EU-level utilization of kernels regarding the new sparse pattern. Compared with existing implementations, the new kernel design achieves 1.71×-7.19×

speedup.

Also benefitting from algorithm and kernel-level codesign, the Dfss in Chapter 4 achieves 1.38 $\sim$ 1.86× speedup over the dense attention mechanism. The algorithm design, which focuses on the use of N:M sparsity and the position to introduce it, ensures that Dfss only contains GPGPU-friendly operators. The kernel design significantly improves the EU-level utilization when pruning and encoding the N:M sparsity, eliminating all the pruning overhead. This enables Dfss to achieve speedup at arbitrary sequence lengths and allows for easy combination with existing efficient transformers.

With the codesign approach encompassing algorithm, operator, and kernel levels, fuseGNN in Chapter 5 achieves 5.3× end-to-end speedup over state-of-the-art frameworks when training GNNs, accompanied by the reduction in global memory footprint by several orders of magnitude on large datasets. This advancement is attributed to the dual aggregation strategy proposed at the algorithm level, potential fusible operators identified at the operator level, and the dedicated kernels designed at the kernel level. All these optimizations together improves the utilization of GPGPU when training GNNs.

Finally, the EVT in Chapter 6 brings kernel-centric optimization to the realm of deep learning compilers. It delivers 1.23 $\sim$ 3.10× speedup across five real-world neural network models with diverse architectures. By translating algorithm-level optimizations to graph transformations, operator-level optimizations into the data flow graph partitioning, and kernel-level optimization into a novel compiler that composes fused kernels with handcrafted mainloops and compiler-generated epilogue under the epilogue visitor tree abstraction, EVT automates kernel-centric optimization on a wider range of neural network models with minimal engineering effort and expertise.

7.2 Future Research

This dissertation only scratched the surface of kernel-centric optimizations of deep learning on GPGPUs. Future studies can be conducted on the following aspects.

7.2.1 Multi-device/node Training

While the approaches presented in this dissertation primarily focus on single GPGPU scenarios, the rapid advancements in large language models in recent years have brought forth new challenges when training across multiple devices or nodes, where device-level utilization is limited by the communication overhead of passing activations and gradients between different GPGPUs.

At the algorithm level, novel training algorithms can be designed that are easier to parallel, such as asynchronous parameter updates between different GPGPUs, would enable more overlapping between communication and computation.

At the kernel level, beyond the partitioning of the data flow graph into fusible operators, exploration of novel partitioning strategies can optimize the data flow graph across multiple devices, with a specific focus on minimizing inter-GPGPU communication overhead.

At the kernel level, dedicated kernel designs can be proposed that enable the fine-grained overlapping between computation and communication at tile levels, which could significantly reduce communication overhead, leading to improved device-level utilization.

7.2.2 Combination of MLIR and EVT

The integration of EVT in Chapter 6 into the MLIR ecosystem is another promising direction. Currently, EVT's kernel-level compiler addresses the scaling issue of epilogue fusion in existing template-based compilers, yet the process remains somehow semiautomatic, requiring expert implementation of visitor nodes. Additionally, the compiler's standalone design makes it challenging to reuse, extend, connect with existing compilers, or compile and debug efficiently.

These issues can be addressed by leveraging the MLIR ecosystem. By designing EVT as a formal series of dialects, corresponding transformations, and lowering rules, the resulting compiler could leverage the high-quality compiler infrastructure offered MLIR, which makes it reusable and extensible. For example, an MLIR-based EVT could take the Linalg dialect as input, allowing it to be connected by existing MLIR-based compilers. Then, the fusion can be performed that aligns the spatial loops of the epilogue to the mainloops. Subsequently, tensors could be lowered to a Cute-inspired dialect based on affine maps, coupled with loop transformations that apply various transformations to automatically generate a high-performance epilogue, ultimately connecting with the handcrafted mainloop.

Appendix A

Supplemental Materials for DFSS

A.1 Theoretical Results

This section provides more theoretical and empirical evidence that justifies Dfss as a good replacement of the full attention. It first derives the theoretical value of 1) quality of the approximation with different sparse patterns 2) speedup can be achieved under certain sparsity. Then, the quality of different methods is compared under the same speedup.

A.1.1 Attention Lottery Ticket

The lottery ticket hypothesis [118] can be extended to the attention mechanism. The last step $\boldsymbol{AV}$ in the attention mechanism can be viewed as the aggregation in the graph neural network. Following the Generalized Attention Mechanism [38], it can be described with a weighted directed graph $\mathcal{G} = (\boldsymbol{A}, \boldsymbol{X})$. $\boldsymbol{A}$ is the adjacent matrix and $\boldsymbol{A}_{u,v} > 0$ indicates that element $\boldsymbol{x}_u$ attends to $\boldsymbol{x}_v$. Inspired by the *Graph Lottery Tickets* [119], the *Attention Lottery Ticket* can be formulated as follows.

Attention Lottery Ticket (ALT). Given a fully connected d graph $\mathcal{G} = \{\boldsymbol{A}, \boldsymbol{X}\}$

constructed from the full quadratic attention mechanism [13], the associated sub-graph can be defined as $\mathcal{G}_s = \{\boldsymbol{m} \odot \boldsymbol{A}, \boldsymbol{X}\}$, where $\boldsymbol{m}$ is a binary mask. If a $\mathcal{G}_s$ has the performance matching the original full quadratic attention mechanism, then we define the sparse attention mechanism with $\mathcal{G}_s$ as an *attention lottery ticket*.

Zaheer et al. 2020[38] have proved the existence of lottery tickets by showing 1) sparse attention mechanisms are universal approximators of sequence to sequence functions when being used as the encoder 2) sparse encoder-decoder transformers are Turing Complete. So the remaining problem is how to identify the winning tickets $\mathcal{G}_s$ at runtime.

A.1.2 Quality of the Lottery Ticket

A popular strategy that empirically works well is selecting the top-k neighborhood in $\mathcal{G}$ based on the magnitude of edge weight, which can be refered as *Top-k Sparsity*. Intuitively, this strategy is based on the hypothesis that the edges with larger edge weight are more important. It has been widely adopted in existing studies [118, 119, 40, 120] and demonstrated its ability to preserve model accuracy at a high sparsity ratio. Following this trend of work, Quality of Attention Lottery Ticket can be defined as follows:

Definition A.1.1 ***(L^p-Quality of Attention Lottery Ticket)*** *The quality of attention lottery ticket $\mathcal{G}_s = \{\boldsymbol{m} \odot \boldsymbol{A}, \boldsymbol{X}\}$ under density $s = \frac{1}{n^2}\sum_{j=1}^{n}\sum_{i=1}^{n}\boldsymbol{m}_{j,i}$ is defined as*

$$\mathcal{Q}^p = \frac{1}{n}\sum_{j=1}^{n}\frac{\sum_{i=1}^{n}(\boldsymbol{m} \odot \boldsymbol{A})^p_{j,i}}{\sum_{i=1}^{n}\boldsymbol{A}^p_{j,i}}. \tag{A.1}$$

The above definition computes the expectation of normalized L^p norm in each row of the attention score matrix. The p is a task-dependent factor that indicates how the accuracy depends on the edges with higher magnitude. In this section, the L^p-Quality of tickets yield by three types of sparse patterns are compared: Top-K, fixed, and our dynamic 1:2 and 2:4 sparse pattern. Particularly, the proposition below can be proved:

Proposition A.1.2 *Under the assumption that the entries in $\boldsymbol{QK}^T/\sqrt{d}$ follow i.i.d. $\mathcal{N}(\mu, \sigma)$, the following equations holds*

$$\begin{aligned} &\mathcal{Q}^p_{topk}|_s \approx \frac{1 + erf\left(\frac{p\sigma}{\sqrt{2}} - erfinv(1-2s)\right)}{2}, \\ &\mathcal{Q}^p_{fix}|_s = s, \quad \mathcal{Q}^p_{2:4} \geq \mathcal{Q}^p_{1:2} = \frac{1 + erf\left(\frac{p\sigma}{2}\right)}{2} \end{aligned} \tag{A.2}$$

(Proof: Appendix A.6)

It is obvious that the $\mathcal{Q}^p_{topk}$ achieves the upper bound of $\boldsymbol{Q}^p$ under s. Besides, the $p\sigma$ is always positive, leading to $\mathcal{Q}^p_{2:4} \geq \mathcal{Q}^p_{1:2} > \mathcal{Q}^p_{fix}|_{s=0.5} = 1/2$.

A.1.3 Efficiency of the Lottery Ticket

A lottery ticket with high quality does not necessarily mean that it is also efficient to execute for actual speedup on GPGPUs. In this section, the efficiency of the three sparse patterns is discussed.

Top-K Sparsity. Zhao et al. 2019[121] explicitly select k neighbors in each row of $\boldsymbol{A}$ based on their magnitude. However, as shown in their Table 4, the explicit sparse transformer has lower inference throughput despite $k \ll n$. On one hand, the top-k operator is difficult to parallel and introduces high overhead. On the other hand, even if an oracle top-k sparsity mask $\boldsymbol{m}$ were provided with zero overhead, it would still be difficult for the explicit Top-K sparse attention to beat its dense counterpart. A theoretical upper bound for density s is provided in Proposition A.1.3 below.

Proposition A.1.3 *Given embedding size d and the maximum tiling size T supported by GPU, the upper bound of the speedup achieved by Top-K Sparsity under density s is*

(Proof: Appendix A.7)

$$Speedup < \frac{4d + 3T}{2d + T + (d + 2T + dT)s}. \tag{A.3}$$

As typical values for the dimension d and tiling size T are $d = 64, T = 128$, $s < 4.5\%$ is a necessary and insufficient condition to have $Speedup > 1$. Notably, this is not a strict upper bound as the overhead of identifying top-k entries is not taken into consideration. Therefore, the strict upper bound should be even smaller.

Fixed Sparsity. As the fixed sparse pattern are designed or learned before inference, they can be designed to be GPU-friendly and have the same tiling size as the dense matrix multiplication. Therefore, the upper bound of the speedup under density s can be derived with the same strategy in Proposition A.1.3:

$$\begin{aligned} Speedup &= \frac{n^2\left(\frac{2d}{T}+1\right) + 2n^2 + nd\left(\frac{2n}{T}+1\right)}{sn^2\left(\frac{2d}{T}+1\right) + 2n^2 s + nd\left(\frac{(1+s)n}{T}+1\right)} \\ &\overset{n\ggg d}{=} \frac{4d + 3T}{(1+3s)d + 3sT}. \end{aligned} \tag{A.4}$$

Dynamic 1:2 / 2:4 Sparsity. Similarly, the theoretical speedup with 1:2 and 2:4 sparsity can be derived as follows:

$$\begin{aligned} Speedup &= \frac{n^2\left(\frac{2d}{T}+1\right) + 2n^2 + nd\left(\frac{2n}{T}+1\right)}{n^2\left(\frac{2d}{T}+\frac{1}{2}+\frac{1}{16}\right)+n^2+nd\left(\frac{n}{T}+\frac{n}{2T}+\frac{n}{16T}+1\right)} \\ &\overset{n\ggg d}{=} \frac{64d + 48T}{57d + 25T}. \end{aligned} \tag{A.5}$$

A.1.4 Quality of the Lottery Tickets under the same Efficiency

With the theoretical conclusions above, the quality of the lottery ticket can be compared between dynamic 1:2 and 2:4 sparsity and the other two methods under the same efficiency.

Comparison with Top-K Sparsity. The Top-K sparsity achieves the same efficiency with Dfss at

$$s < \frac{(4d+3T)(57d+25T)}{(64d+48T)(d+2T+dT)} - \frac{2d+T}{(d+2T+dT)}. \tag{A.6}$$

With typical values $T = 128, d = 64$, $s < 0.02$. It can be substituted it to Proposition A.1.2 and get $\mathcal{Q}^p_{topk} < \mathcal{Q}^p_{1:2}$ when $p\sigma < 7$. On the other hand, when $p\sigma > 7$, although the Top-K sparsity produces tickets with higher quality, $\mathcal{Q}^p_{1:2}|_{p\sigma=7} \approx 0.9999996$ is already very close to 1.

Comparison with Fixed Sparsity. The fixed sparsity achieves the same efficiency with Dfss when

$$s = \frac{(4d+3T)(64d+48T)}{(57d+25T)(3d+3T)} - \frac{d}{3d+3T}. \tag{A.7}$$

With typical values $T = 128, d = 64$, $s \approx 0.63$. In addition, the theoretical values of $\sigma \approx 1$ and $p \geq 1$. The $p \geq 1$ is based on the observation that the edges with higher magnitude are more influential. Therefore, $p\sigma \geq 1$ and $\mathcal{Q}^p_{1:2} \geq 0.76 > 0.63 = \mathcal{Q}^p_{fix}|_{s=0.63}$.

To conclude, compared with both top-k sparsity and fixed sparsity, Dfss can always yield lottery tickets with higher quality under the same efficiency. To support this conclusion, further empirical studies are provided in Appendix A.2. Besides, Dfss is a good complementary to the kernel-based transformers like Performer [44]. More discussions about it are included in Appendix A.3.

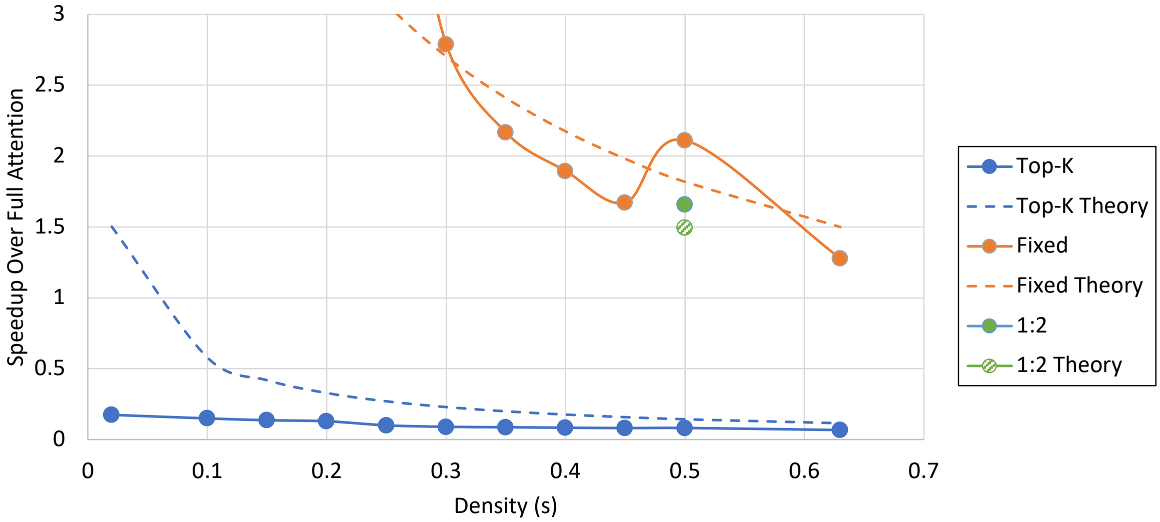


Figure A.1: Theoretical and actual speedup achieved by different sparse patterns on A100 GPU.

A.2 Empirical Results

This section provides more empirical evidence to support conclusions in Appendix A.1. It first compares the theoretical speedup of different sparsity predicted in Equation (A.3), (A.4), and (A.5) and the actual speedup measured on A100 GPU in Figure A.1.

First of all, the Top-K sparsity is well bounded by the theoretical value, and Dfss achieves better speedup than the Top-K sparsity when the density $s > 0.02$. This is because gathering top-k elements in each row of the attention weight matrix and sorting them to compressed row format introduce huge overhead.

Second, the speedup achieved by the fixed sparsity is well predicted by the theoretical value. The speedup it achieved is lower than Dfss when density $s \geq 0.63$, which accords with the theoretical conclusion. Notably, the speedup of fixed sparsity here simply truncates the number of columns of the attention weight matrix based on the density. The actual speedup will be even lower when more fine-grained pattern is involved.

Dfss delivers speedup a little bit higher than the theoretical value. This is because

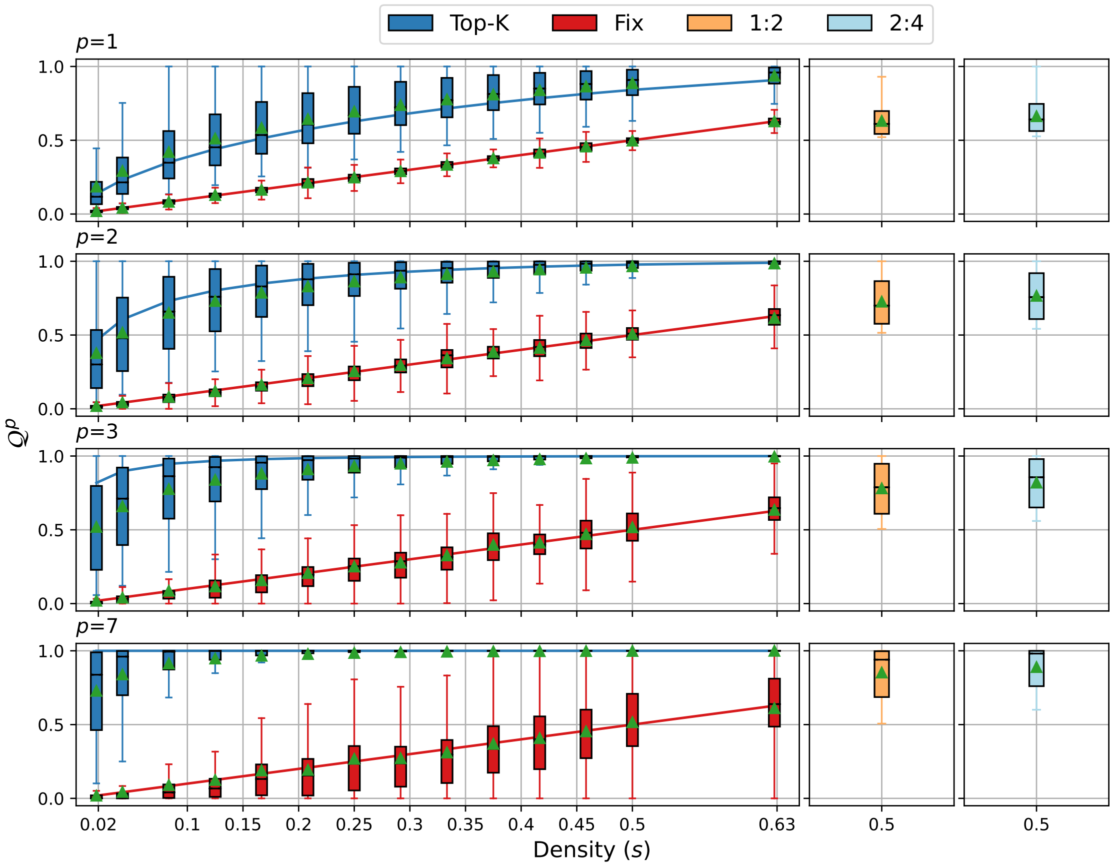


Figure A.2: $\mathcal{Q}^p$ under different density s and sparsity strategies. Box plot: Empirical results from BERT-large on SQuAD v1.1; Solid line: Theoretical results from Proposition A.1.2.

the softmax kernel has different implementations under different sequence length. When the sequence length is moderate, as mentioned in Appendix A.7, the data loaded from the attention score matrix can be explicitly cached in fast memory like registers or shared memory for reuse. When sequence length too long for the fast memory to cache, it has to be implicitly reused through lower-level cache or even global memory. The second implementation is slower than the first one as lower-level cache has longer access latency and lower throughput. As DFSS reduces the sequence length by half, it can use the implementation for moderate sequence length while the full attention is handled by the long sequence version.

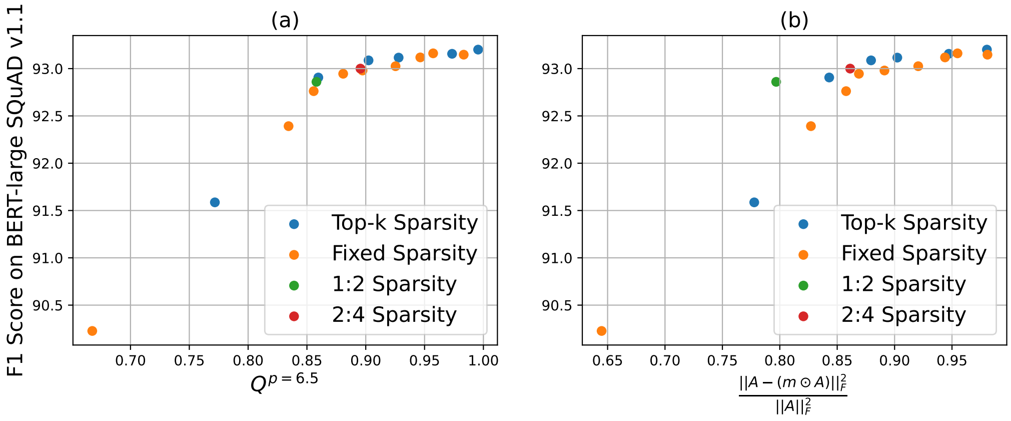


Figure A.3: $\mathcal{Q}^p$ under different density s and sparsity strategies. Box plot: Empirical results from BERT-large on SQuAD v1.1; Solid line: Theoretical results from Proposition A.1.2.

Figure A.2 shows the theoretical value (solid line) and empirical value (box plot) of $\mathcal{Q}^p$ over attention matrix $\boldsymbol{A}$ in BERT-Large on SQuAD v1.1. The p is sweeped through several typical values, as it is a task-dependent value that is hard to obtain.

Compared with the top-k sparsity, when $p < 7$, 1:2 and 2:4 sparsity always achieve better performance than the top-k sparsity when $s < 0.05$. Besides, when $p = 7$, the $\mathcal{Q}^p_{1:2}$ and $\mathcal{Q}^p_{2:4}$ are very close to 1. These observations accord with the conclusion that 1:2 and 2:4 sparsity can obtain tickets with better quality than Top-K sparsity at the same efficiency.

Compared with the fixed sparsity, $\mathcal{Q}^p_{1:2}$ and $\mathcal{Q}^p_{2:4}$ are also similar or better than $\mathcal{Q}^p_{fix}$ across different ps. This supports the conclusion that DFSS achieves better performance than the fixed sparsity patterns under the same efficiency.

To show that Q^p is a good metric to compare the performance of different sparse patterns, Figure A.3 shows the Q^p and F1 score on BERT-large SQuAD v1.1. As mentioned before, p is a task-specific value used to model tasks with different degree of dependency on the largest few elements. In order to identify the p for target task, the value of p is tuned until the data points from Top-K sparsity and Fixed sparsity form a monotonically increasing line. It demonstrates that $p = 6.5$ is a good choice. This large p accords the observation that the Top-K sparsity works well even under 5.4% density. After anchoring the p, the data points from 1:2 and 2:4 sparsity are put into the plot to verify if the line is still monotonically increasing. Figure A.3 shows that the data points from 1:2/2:4 sparsity perfectly fills in the monotonically increasing line. Oppositely, The traditional F-norm based metric cannot explain why the 1:2 sparsity has better F1-score than some Fixed Sparsity even though it has lower score. This demonstrates that Q^p is a better metric than existing metrices.

A.3 Comparison with Performer

This section adds more discussions on how Dfss compared with kernel-based transformer, i.e. Performer [44]. As Definition A.1.1 is designed to characterize how well the sparse pattern could reserve the important edges in $\boldsymbol{A}$, it is not suitable for kernel-based attention mechanisms that do not involve sparsity. For example, an approximation of $\boldsymbol{A}$ with high positive approximation error can have $\mathcal{Q}^P \geq 1$ under Definition A.1.1. Therefore, this section instead compares the mean squared error (MSE) following Choromanski et al. 2021[44]. Given the query and two adjacent key vectors $\boldsymbol{q}$, $\boldsymbol{k}$, and $\boldsymbol{k'} \in \mathcal{N}(0, \boldsymbol{I}_d)$, the softmax kernel between them is denoted as $SM(\boldsymbol{q}, \boldsymbol{k}) = exp(\boldsymbol{q}^T\boldsymbol{k}/\sqrt{d})$. And the softmax approximated by dynamic 1:2 sparsity $\widehat{SM_{1:2}}(\boldsymbol{q}, \boldsymbol{k})$ is defined as

$$\widehat{SM_{1:2}}(\boldsymbol{q}, \boldsymbol{k}) = \begin{cases} exp\left(\frac{\boldsymbol{q}^T\boldsymbol{k}}{\sqrt{d}}\right) & if \ \ \boldsymbol{q}^T\boldsymbol{k} > \boldsymbol{q}^T\boldsymbol{k'} \\ 0 & else \end{cases}. \tag{A.8}$$

Then, its MSE can be computed as follows

$$\begin{aligned}&MSE(\widehat{SM_{1:2}}(\boldsymbol{q},\boldsymbol{k})) = \\ &\int_{\boldsymbol{q}^T\boldsymbol{k}<\boldsymbol{q}^T\boldsymbol{k'}} exp\left(\frac{2\boldsymbol{q}^T\boldsymbol{k}}{\sqrt{d}}\right) 2\pi^{-d/2} exp\left(-\frac{||\boldsymbol{k'}||_2^2}{2}\right) d\boldsymbol{k'}.\end{aligned} \tag{A.9}$$

Because $\boldsymbol{q}^T\boldsymbol{k'} = \sum_{i=1}^{d} \boldsymbol{q}_i\boldsymbol{k}'_i$ is the weighted sum of i.i.d variables following $\mathcal{N}(0,1)$, $x = \boldsymbol{q}^T\boldsymbol{k'} \sim \mathcal{N}(0, ||\boldsymbol{q}||_2^2)$. It can be substituted into Equation (A.9) and get

$$\begin{aligned}&MSE(\widehat{SM_{1:2}}(\boldsymbol{q},\boldsymbol{k})) \\ &= SM^2(\boldsymbol{q},\boldsymbol{k})\frac{1 - erf\left(\frac{\sqrt{d}}{||\boldsymbol{q}||_2\sqrt{2}} ln\left(SM(\boldsymbol{q},\boldsymbol{k})\right)\right)}{2}.\end{aligned} \tag{A.10}$$

With Lemma 2 and Theorem 2 in Choromanski et al. 2021[44], the MSE of their positive softmax kernel with orthogonal random features has an upper bound as follows

$$\begin{aligned}&MSE\left(\widehat{SM_m^{ort+}}(\boldsymbol{q},\boldsymbol{k}))\right) \\ &\leq \frac{1}{m} exp\left(\frac{2\boldsymbol{q}^T\boldsymbol{k}}{\sqrt{d}}\right)\left[exp\left(\frac{||\boldsymbol{q}+\boldsymbol{k}||^2}{\sqrt{d}}\right) - 1 - \left(1-\frac{1}{m}\right)\frac{2}{d+2}\right] \\ &= \frac{SM^2(\boldsymbol{q},\boldsymbol{k})}{m}\left[exp\left(\frac{||\boldsymbol{q}||_2^2 + ||\boldsymbol{k}||_2^2}{\sqrt{d}}\right) SM^2(\boldsymbol{q},\boldsymbol{k}) - 1 - \frac{2\left(1-\frac{1}{m}\right)}{d+2}\right]\end{aligned} \tag{A.11}$$

First of all, when $SM(\boldsymbol{q},\boldsymbol{k}) \to 0$, both $MSE(\widehat{SM_{1:2}}(\boldsymbol{q},\boldsymbol{k}))$ and $MSE\left(\widehat{SM_m^{ort+}}(\boldsymbol{q},\boldsymbol{k}))\right)$ converge to 0. However, for large $SM(\boldsymbol{q},\boldsymbol{k})$s that are potentially be critical for the model accuracy, the $exp\left(\frac{||\boldsymbol{q}||_2^2+||\boldsymbol{k}||_2^2}{\sqrt{d}}\right) SM^2(\boldsymbol{q},\boldsymbol{k})$ term in the positive softmax kernel in Performer could greatly increases the MSE. Oppositely, the $1 - erf\left(\frac{\sqrt{d}}{||\boldsymbol{q}||_2\sqrt{2}} ln\left(SM(\boldsymbol{q},\boldsymbol{k})\right)\right)$ term in DFSS reduces the MSE. To conclude, while both the positive softmax kernel and ours has low MSE error when approximating small edge weights, DFSS can better approximate the edges with high magnitude.

From the empirical perspective, as shown in Table 4.2 and 4.3, DFSS can achieve good accuracy even without finetuning. Whereas the Performer still requires tens of thousands steps of finetuning (e.g. Figure 5 in Choromanski et al. 2021[44]). Table 4.4 also reveals that Performer has poor accuracy on certain tasks like byte-level document retrieval, while Dfss consistently achieves accuracy on par with the dense transformer. All these observations suggest that Dfss can better approximate the full attention mechanism than Performer.

In terms of speedup, Figure 5.8 illustrates that the Performer can only achieve good speedup at long sequence length. The similar phenomenon is also observed in multiple online forums [1]. Certainly, the PyTorch JIT script does not yield the optimal implementation of the computation graph, but it reveals that tremendous engineering efforts are required for Performer to achieve good speedup under moderate sequence length.

Following Section A.1.3, the theoretical speedup achieved by Dfss and the Performer are also compared as follows.

$$\begin{aligned}
&\boldsymbol{T}^{\mathbf{(1)}}_{n\times m} = \frac{\boldsymbol{Q}_{n\times d}}{\sqrt[4]{d}}\boldsymbol{P}_{d\times m},\quad \boldsymbol{T}^{\mathbf{(2)}}_{n\times 1} = \frac{1}{2\sqrt{d}}\sum_{i=1}^{d}\left[\boldsymbol{Q}_{n\times d}\odot \boldsymbol{Q}_{n\times d}\right]_{:,i}\\
&\boldsymbol{T}^{\mathbf{(3)}}_{n\times 1} = max_i[\boldsymbol{T}^{\mathbf{(1)}}_{n\times m}]_{:i},\\
&\phi(\boldsymbol{Q}_{n\times m}) = \frac{1}{\sqrt{m}}exp\left(\boldsymbol{T}^{\mathbf{(1)}}_{n\times m} - \boldsymbol{T}^{\mathbf{(2)}}_{n\times 1} - \boldsymbol{T}^{\mathbf{(3)}}_{n\times 1} + \epsilon\right)\\
&\boldsymbol{T}^{\mathbf{(4)}}_{n\times m} = \frac{\boldsymbol{K}_{n\times d}}{\sqrt[4]{d}}\boldsymbol{P}_{d\times m},\quad \boldsymbol{T}^{\mathbf{(5)}}_{n\times 1} = \frac{1}{2\sqrt{d}}\sum_{i=1}^{d}\left[\boldsymbol{K}_{n\times d}\odot \boldsymbol{K}_{n\times d}\right]_{:,i}\\
&\boldsymbol{T}^{\mathbf{(6)}}_{n\times 1} = max_i[\boldsymbol{T}^{\mathbf{(4)}}_{n\times m}]_{:i},\\
&\phi(\boldsymbol{K}_{n\times m}) = \frac{1}{\sqrt{m}}exp\left(\boldsymbol{T}^{\mathbf{(4)}}_{n\times m} - \boldsymbol{T}^{\mathbf{(5)}}_{n\times 1} - \boldsymbol{T}^{\mathbf{(6)}}_{n\times 1} + \epsilon\right)\\
&\boldsymbol{T}^{\mathbf{(7)}}_{m\times 1} = \sum_{i=1}^{n}[\phi(\boldsymbol{K})_{n\times m}]_{i,:},\quad \boldsymbol{T}^{\mathbf{(8)}}_{n\times 1} = 1/\left(\phi(\boldsymbol{Q})_{n\times m}\times \boldsymbol{T}^{\mathbf{(7)}}_{m\times 1}\right)\\
&\boldsymbol{T}^{\mathbf{(9)}}_{m\times d} = \phi(\boldsymbol{K})^{T}_{n\times m}\times \boldsymbol{V}_{n\times d},\quad \boldsymbol{T}^{\mathbf{(10)}}_{n\times d} = \phi(\boldsymbol{Q})_{n\times m}\times \boldsymbol{T}^{\mathbf{(9)}}_{m\times d}\odot \boldsymbol{T}^{\mathbf{(8)}}_{n\times 1}.
\end{aligned}\tag{A.12}$$

[1] https://github.com/huggingface/transformers/issues/7675

The computation steps of Performer are listed in Equation (A.12) where each equation denotes a sub computation graph that can potentially be fused. Notably, this is more complex than the original mathematical expression to handle the numerical instability of *exp*. The total memory access can be computed with

$$\begin{aligned}&\mathit{Speedup}{=}\{2\left[nm\left(\frac{2d}{T}+1\right){+}n(d{+}1){+}n(m+1){+}n(m+3)\right]\\&+m(n+1){+}n(\frac{m}{T}+m+1){+}md\left(\frac{2n}{T}+1\right){+}nd\left(\frac{2m}{T}+1\right)+n\}\\&/\left[n^2\left(\frac{2d}{T}+1\right)+2n^2+nd\left(\frac{2n}{T}+1\right)\right].\end{aligned} \tag{A.13}$$

With Theorem 4 in Choromanski et al. 2021[44], $m = dln(d)$. The $m = 266$, $d = 64$, and $T = 128$ can be substituted into Equation (A.13) and get $\mathit{Speedup} > 1$ when $n > 672$. On the other hand, the performer achieves the same speedup with ours with $n > 1002$.

To conclude, DFSS is a good complementary to performer. With delicately optimized computation graph, performer can achieve good speedup and relatively good accuracy under long sequence scenario. In contrary, DFSS has better speedup and accuracy under moderate and short sequence length. Besides, DFSS delivers lower approximation error on important edges so it is more friendly to finetuning.

A.4 Combination with the Existing Efficient Transformers

Existing efficient transformers usually sparsify the full attention mechanism to densely connected clusters [68, 42, 41, 38] or approximate it with low-rank projection [43]. As DFSS is a good approximation of the full attention mechanism and brings wall time speedup at arbtrary sequence length, it can potentially be combined with the existing

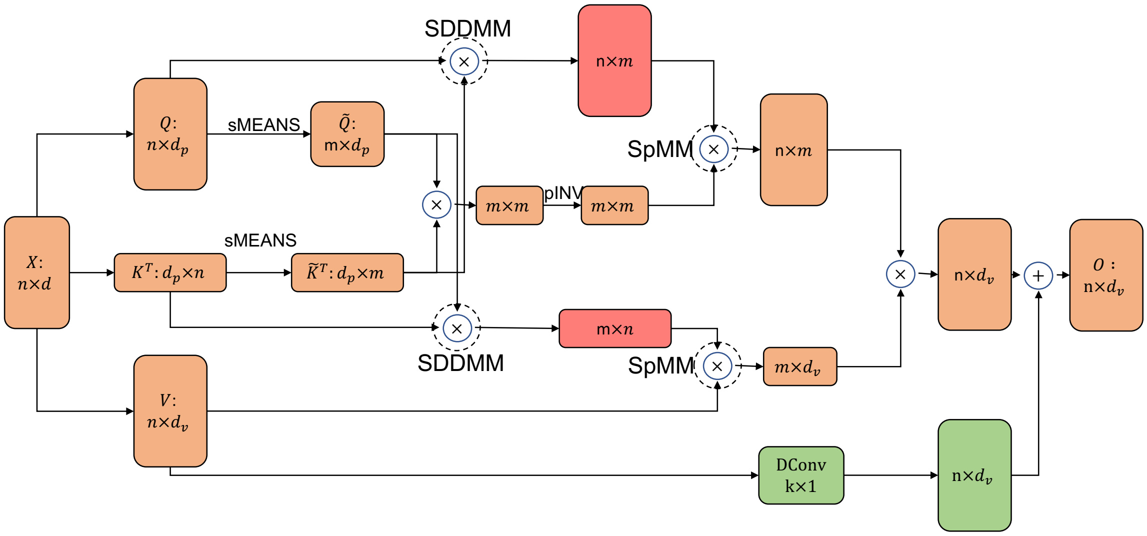


Figure A.4: Combination of our method with Nystromformer [69]. The two red matrices are stored under 1:2/2:4 structured sparsity.

efficient transformers.

This section first demonstrates the combination of our method with Xiong et al. 2021[69]. Xiong et al. 2021[69] propose a Nystrom-based self-attention mechanism that approximate standard self-attention with $O(n)$ complexity. The Nystromformer is illustrated in Figure A.4. It shows that the computation circled in Figure A.4 is identical to the standard attention mechanism, so it can be further accelerated with Dfss. More importantly, the two matrix multipliation involved are the two of the three largest $m \times n$ matrices. It will be very beneficial to reduce their complexity.

The accuracy on Image (1K) on LRA [73] is reported in Table A.1. A standard Nystromformer is first trained from scratch for 35,000 iterations following Xiong et al. 2021[69]. Then, it is fine-tuned for 3,500 iterations (1/10 of the training process) under standard Nystromformer, Nystromformer + Dfss 1:2, and Nystromformer + Dfss 2:4. It is obvious that by combining Dfss and Nystromformer, higher accuracy can be achieved on LRA with lightweight finetuning.

Then a complexity analysis of the combination following Xiong et al. 2021[69] is

Table A.1: Accuracy on Image (1K) on LRA [73] under the combination of Dfss and Nystromformer [69].

	Pretraining	Finetuning
Nystromformer (float)	41.17	41.52
Nystromformer (bfloat16)	-	41.59
Nystromformer + Dfss 1:2 (float)	-	<u>41.91</u>
Nystromformer + Dfss 2:4 (bfloat16)	-	**42.54**

as follows. The landmark selection with segement-means takes $O(n)$, iterative approximation of the pseudoinverse takes $O(m^3)$. The matrix multiplication complexity of the standard Nystromformer takes $O(nm^2+mnd_v+m^3+nmd_v)$. After applying our method, it can be reduced to $O(\frac{nm^2}{2}+\frac{nmd_v}{2}+m^3+nmd_v)$. The memory footprint can be reduced from $O(md_q+nm+m^2+nm+nd_v)$ to $O(md_q+nm+m^2+nd_v)$. Given $n \gg m > d_v \approx d_p$, this could be a significant improvement that allows us to use more landmarks m to better approximate the full attention mechanism.

Besides Nystromformer, this section also illustrates two possible combinations with BigBird [38] and Linformer [43] that can be explored in future work.

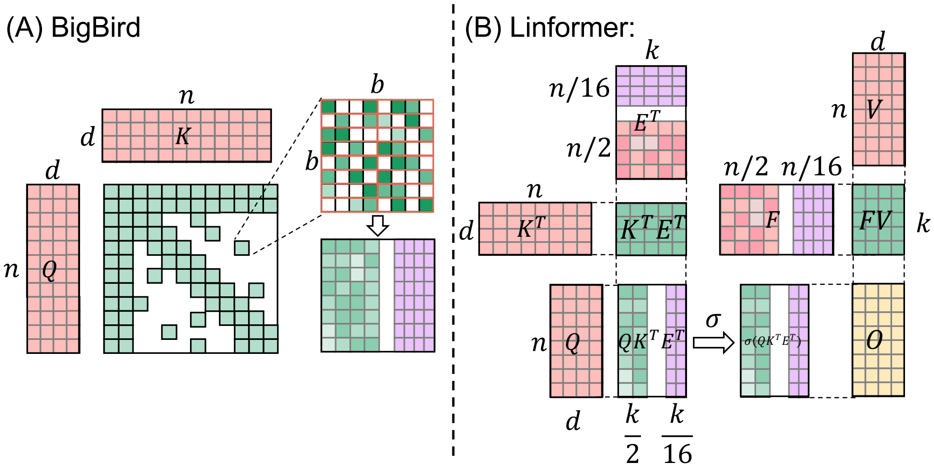


Figure A.5: Combination of our method with BigBird [38] and Linformer [43]

As shown in Figure A.5 (A), Zaheer et al. 2020[38] use block sparsity with block size 64 and compute a full attention within each block. The 1:2 or 2:4 sparsity can be applied within each block to bring further speedup.

Figure A.5 (B) gives another example on how to combine DFSS with Linformer [43]. Linformer uses low-rank approximation on the attention mechanism as follows:

$$\boldsymbol{O} = softmax\left(\frac{\boldsymbol{Q}\left(\boldsymbol{E}\boldsymbol{K}\right)^T}{\sqrt{d}}\right)\boldsymbol{F}\boldsymbol{V}, \tag{A.14}$$

where $\boldsymbol{E}, \boldsymbol{F} \in \mathbb{R}^{n\times k}$ are linear projection matrices and $k \ll n$. $\boldsymbol{E}$ and $\boldsymbol{F}$ can be first pruned along with other weight matrices to have 1:2 or 2:4 sparsity offline following Mishra et al. 2021[24]. Then, $\boldsymbol{EK}$ and $\boldsymbol{FV}$ are computed with Sparse Matrix-Matrix multiplication. Next, $\boldsymbol{Q}$ is multiplied with $(\boldsymbol{EK})^T$ and the result is pruned to 50% structured fine-grained sparsity on the fly. After applying softmax to the nonzeros, it is multiplied with $\boldsymbol{FV}$.

A.5 Visualize Attention Distribution

To illustrate that DFSS can well capture the fine-grained sparsity in attention, this section visualizes the attention weight matrices in BERT-large on SQuAD v1.1 in Figure A.6.

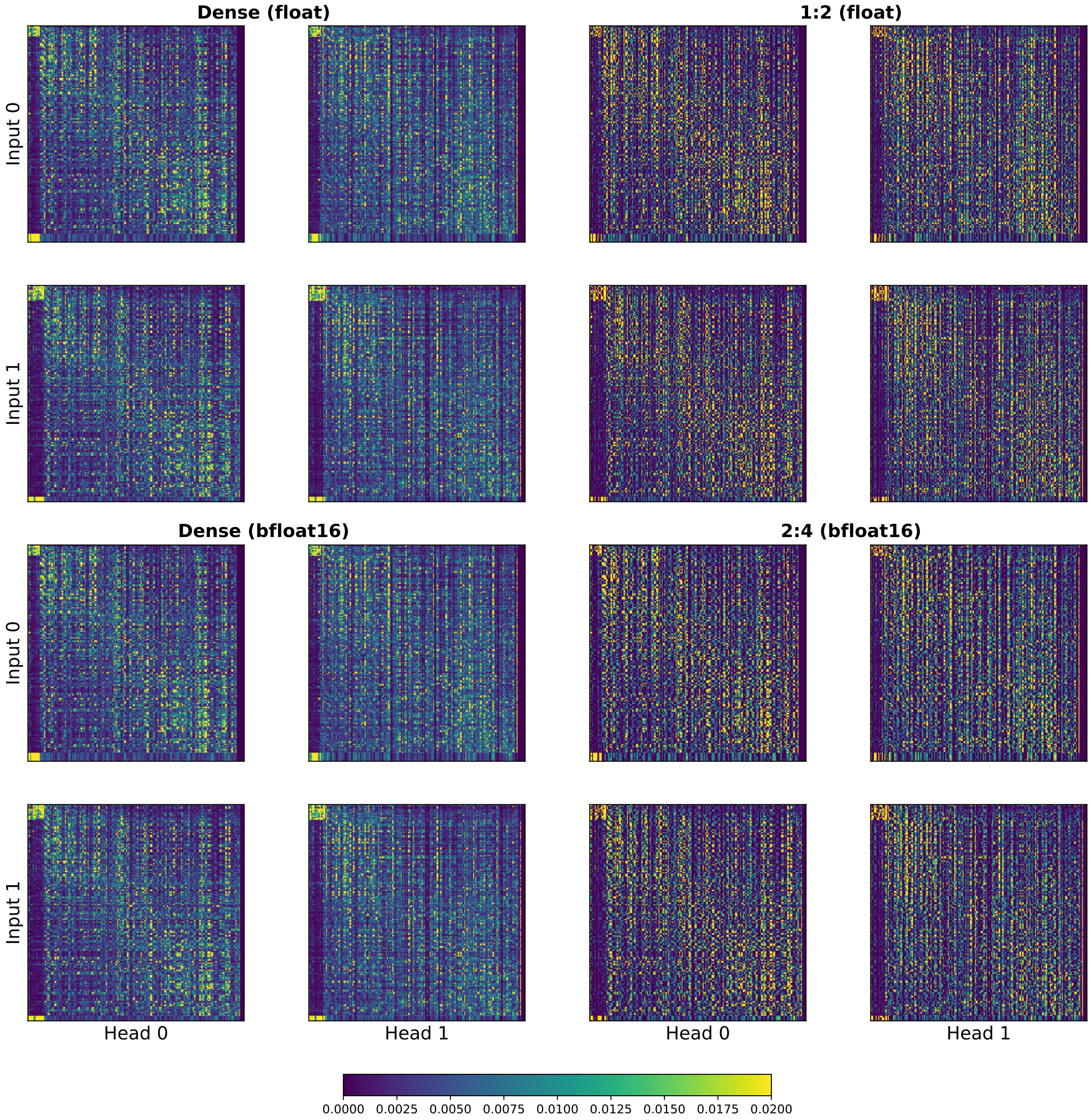


Figure A.6: Visualization of attention weight in dense transformer and Dfss

In detail, the inference of the same input sample in BERT-large model pretrained under dense, 1:2, and 2:4 settings is launched, then the attention weight matrix is collected in the first layer. It is obvious that the pattern in dense transformer and Dfss are quite similar. The magnitude of nonzero values in Dfss are a little bit higher than dense attention. This is because the softmax normalizes the values in each row with the exponential sum of each entry. After removing 50% smaller entries, the magnitude of remaining entries would be relatively higher. Nevertheless, this does not influence the model accuracy, as the forthcoming normalization layers will take care of it.

A.6 Proof of Proposition A.1.2

Proof: Under the assumption that the entries in $\boldsymbol{Q}\boldsymbol{K}^T/\sqrt{d}$ follow i.i.d. $\mathcal{N}(\mu, \sigma)$, we denote $x_{i,j} = e^{\mu+\sigma z_{i,j}}$, where $z \sim i.i.d.\ \mathcal{N}(0,1)$. Then it can be substituted into the definition of the softmax and get

$$\boldsymbol{A}_{u,v} = \frac{x_{u,v}}{\sum_{i=1}^{n} x_{u,i}}. \tag{A.15}$$

The above equation is then substituted into the definition of L^p-Quality and get

$$\mathcal{Q}^p = \frac{1}{n}\sum_{j=1}^{n}\frac{\sum_{i=1}^{n}\left(\boldsymbol{m}\odot\boldsymbol{A}\right)_{j,i}^{p}}{\sum_{i=1}^{n}\boldsymbol{A}_{j,i}^{p}} = \frac{1}{n}\sum_{j=1}^{n}\frac{\frac{1}{n}\sum_{i=1}^{n} m_{j,i}x_{j,i}^{p}}{\frac{1}{n}\sum_{i=1}^{n} x_{j,i}^{p}} \tag{A.16}$$

With $n \to \infty$, the denominator can be approximated with

$$\frac{1}{n}\sum_{i=1}^{n} x_{j,i}^{p} \approx \int_{-\infty}^{\infty}\frac{e^{p\mu+p\sigma z}}{\sqrt{2\pi}} exp\left(-\frac{z^2}{2}\right)dz = exp\left(p\mu + \frac{p^2\sigma^2}{2}\right) \tag{A.17}$$

Top-K Sparsity. When the sequence is long enough such that $n \to \infty$, the numerator can be approximated with

$$\begin{aligned}\frac{1}{n}\sum_{i=1}^{n} m_{j,i}x_{j,i}^{p} &\approx \int_{\sqrt{2}erfinv(1-2s)}^{\infty}\frac{e^{p\mu+p\sigma z}}{\sqrt{2\pi}} exp\left(-\frac{z^2}{2}\right)dz \\ &= exp\left(p\mu + \frac{p^2\sigma^2}{2}\right)\frac{1+erf\left(\frac{p\sigma}{\sqrt{2}} - erfinv(1-2s)\right)}{2}\end{aligned} \tag{A.18}$$

Therefore, the L^p-Quality of Top-K sparsity is

$$\mathcal{Q}^p_{topk} \approx \frac{1 + erf\left(\frac{p\sigma}{\sqrt{2}} - erfinv(1-2s)\right)}{2}. \tag{A.19}$$

Fixed Sparsity. Without any assumption on the distribution of important edges in $\boldsymbol{A}$, applying a fixed pattern is equivalent with uniformly sampling with probability s, leading to

$$\frac{1}{n}\sum_{i=1}^{n} m_{j,i}x^p_{j,i} \approx exp\left(p\mu + \frac{p^2\sigma^2}{2}\right)s. \tag{A.20}$$

Therefore, the L^p-Quality of the fixed sparsity is

$$\mathcal{Q}^p_{fix} \approx s. \tag{A.21}$$

2-to-1 Sparsity: This sparsity pattern select the larger one in every two elements. The adjacent two elements are denoted with

$$X = e^{\mu+\sigma Z_1}, Y = e^{\mu+\sigma Z_2}; Z_1, Z_2 \sim \mathcal{N}(0,1), \tag{A.22}$$

Z_1 and Z_2 are independent. Then it can be proved that

$$\begin{aligned}&\frac{1}{n}\sum_{i=1}^{n} m_{j,i}x^p_{j,i} \approx \frac{1}{2}\mathbb{E}\left[max(X^p, Y^p)\right] = \\ &\frac{1}{2}\left[\iint_{z_1\geq z_2} e^{p\mu+p\sigma z_1}\frac{1}{2\pi}exp\left(-\frac{z_1^2+z_2^2}{2}\right)dz_1dz_2\right. \\ &\left.+\iint_{z_1<z_2} e^{p\mu+p\sigma z_2}\frac{1}{2\pi}exp\left(-\frac{z_1^2+z_2^2}{2}\right)dz_1dz_2\right]\end{aligned} \tag{A.23}$$

By denoting

$$x = \frac{z_1 - z_2}{\sqrt{2}}, \quad y = \frac{z_1 + z_2}{\sqrt{2}}, \tag{A.24}$$

It leads to

$$
\begin{aligned}
&\iint_{z_1 \geq z_2} e^{p\mu + p\sigma z_1} \frac{1}{2\pi} exp\left(-\frac{z_1^2 + z_2^2}{2}\right) dz_1 dz_2 \\
&= \int_{-\infty}^{\infty} \int_{0}^{\infty} e^{p\mu + \frac{p\sigma}{\sqrt{2}}(x+y)} \frac{1}{2\pi} exp\left(-\frac{x^2 + y^2}{2}\right) dxdy \\
&= \int_{-\infty}^{\infty} \int_{0}^{\infty} e^{p\mu + \frac{p^2\sigma^2}{2}} \frac{1}{2\pi} exp\left[-\frac{\left(x - \frac{p\sigma}{\sqrt{2}}\right)^2 - \left(y - \frac{p\sigma}{\sqrt{2}}\right)^2}{2}\right] dxdy \\
&= e^{p\mu + \frac{p^2\sigma^2}{2}} \int_{0}^{\infty} \frac{1}{\sqrt{2\pi}} exp\left[-\frac{1}{2}\left(x - \frac{\sigma}{\sqrt{2}}\right)^2\right] dx \\
&= \frac{e^{p\mu + \frac{p^2\sigma^2}{2}}}{2} \left[1 + erf\left(\frac{p\sigma}{2}\right)\right].
\end{aligned}
\tag{A.25}
$$

With the conclusion above, we have

$$
\frac{1}{n} \sum_{i=1}^{n} m_{j,i} x_{j,i}^{p} = exp\left(p\mu + \frac{p^2\sigma^2}{2}\right) \frac{1 + erf\left(\frac{p\sigma}{2}\right)}{2}. \tag{A.26}
$$

The L^P-Quality of 1:2 sparsity can be computed with

$$
\mathcal{Q}_{1:2}^{p} = \frac{1 + erf\left(\frac{p\sigma}{2}\right)}{2}. \tag{A.27}
$$

2:4 Sparsity: This sparsity pattern select the largest two elements in consecutive four elements. While it is more challenging to find an explicit expression for $\mathcal{Q}_{4-to-2}^{p}$, a trivial lower bound can be found with

$$
\begin{aligned}
&\frac{1}{n} \sum_{i=1}^{n} m_{j,i} x_{j,i}^{p} \approx \frac{1}{4}\mathbb{E}\left[max(X^p + Y^p, X^p + U^p, X^p + V^p, Y^p + U^p, Y^p + V^p, U^p + V^p)\right] \\
&\geq \frac{1}{4}\left(\mathbb{E}[max(X^p, Y^p)] + \mathbb{E}[max(U^p, V^p)]\right) \\
&= \frac{1}{2}\mathbb{E}[max(X^p, Y^p)],
\end{aligned}
\tag{A.28}
$$

where

$$X = e^{\mu+\sigma Z_1}, Y = e^{\mu+\sigma Z_2}, U = e^{\mu+\sigma Z_3}, \\ V = e^{\mu+\sigma Z_4}; Z_1, Z_2, Z_3, Z_4 \sim \mathcal{N}(0,1), \tag{A.29}$$

$Z_1, ..., Z_4$ are independent. Therefore, the lower-bound of $\mathcal{Q}^p_{2:4}$ is

$$\mathcal{Q}^p_{2:4} \geq \frac{1 + erf\left(\frac{p\sigma}{2}\right)}{2}. \tag{A.30}$$

■

A.7 Proof of Proposition A.1.3

Proof: First of all, thanks to the Tensor Core in latest GPUs, the latency of matrix multiplication operations, both sparse and dense, are bounded by the memory access. Therefore, instead of counting the number of MACs (multiply-accumulate operations), the amount of memory access is a better metric to estimate the latency.

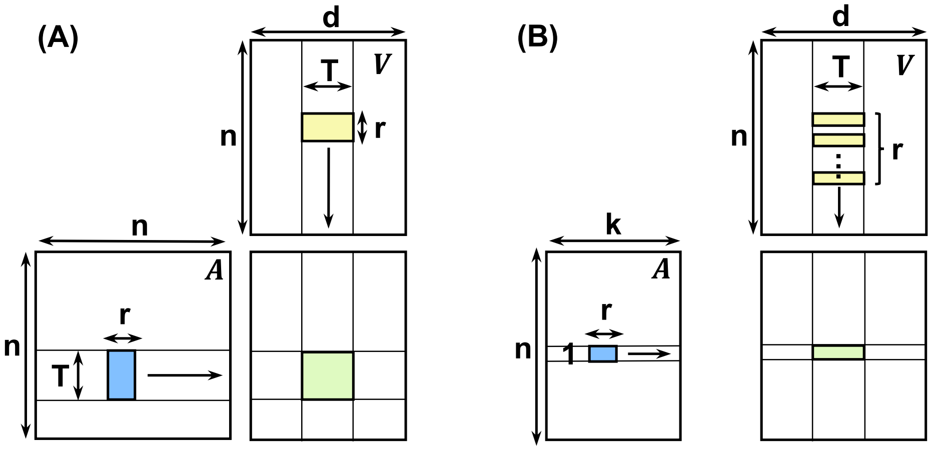


Figure A.7: Tiling Matrix-Matrix Multiply

Tiling is a basic optimization applied to optimize matrix matrix multiply on GPU. As shown in Figure A.7 (A), the original $n \times n$ output is partitioned to independent blocks with size $T \times T$. When computing each block, operands with size $T \times r$ and $r \times T$ are loaded from $\boldsymbol{A}$ and $\boldsymbol{V}^T$ to the fast memory, respectively. Then, these two operand are multiplied and accumulated to the partial sum stored in the registers. After applying the top-k, as shown in Figure A.7 (B), the k elements in each row of $\boldsymbol{A}$ correspond to different rows in $\boldsymbol{A}$. Therefore, the output can only be partitioned to independent vectors with size $1 \times T$. During the computation, operands with size $1 \times r$ and $r \times T$ are loaded from $\boldsymbol{A}$ and $\boldsymbol{V}^T$ to the fast memory, respectively. Then, the loaded operands are multiplied and accumulated to the partial sum stored in the registers. With the tiling strategy mentioned above, the table below summarizes the amount of memory access in different attentions.

Table A.2: Amount of Memory Access in Different Operations in Attention. $s = k/n$: density of the sparse attention; T: tiling size.

	$\boldsymbol{QK}^T$	Softmax	$\boldsymbol{AV}$
Full Attention	$n^2\left(\frac{2d}{T}+1\right)$	$2n^2$	$nd\left(\frac{2n}{T}+1\right)$
Explicit Top-k Attention	$n^2\left(\frac{2d}{T}+1\right)$	$2n^2s$	$nd\left(sn+\frac{sn}{T}+1\right)$

For $\boldsymbol{QK}^T$, as it requires to compute all of it before getting the top-k elements, it is a dense matrix matrix multiplication for both full and explicit top-k attention. The Softmax needs to read the $n \times n$ $\boldsymbol{QK}^T$ in, normalizes it, and write the result $\boldsymbol{A}$ back. As the intermediate values can be stored in registers, it only requires to count reading $\boldsymbol{QK}^T$ in and writing $\boldsymbol{A}$ out. Therefore, its memory access is $2n^2$ for full attention and $2n^2s$ for explicit top-k attention. For $\boldsymbol{AV}$ in full attention, the output size is nd. As each output element is generated from the inner product between two vectors with length n, the total data read equals $nd \times 2n$. However, with the tiling in Figure A.7 (A), each operand is reused for T times. Therefore, the total memory access for $\boldsymbol{AV}$ in full attention is $nd(\frac{2n}{T}+1)$. For $\boldsymbol{AV}$ in explicit top-k attention, as shown in Figure A.7 (B), each left-hand-side data is reused for T times while each right-hand-side data is used only for once. Therefore, its memory access equals to $nd(\frac{sn}{T} + sn + 1)$. The theoretical speedup can be computed with

$$\begin{aligned} Speedup &\leq \frac{n^2\left(\frac{2d}{T}+1\right)+2n^2+nd\left(\frac{2n}{T}+1\right)}{n^2\left(\frac{2d}{T}+1\right)+2n^2s+nd\left(sn+\frac{sn}{T}+1\right)} \\ &\overset{n\gg d}{\approx} \frac{4d+3T}{2d+T+(d+2T+dT)s} \end{aligned} \tag{A.31}$$

■

Appendix B

Supplemental Materials for EVT

B.1 Proofs

Theorem 4.1. *Algorithm 5 returns the global optimal and feasible solution when is_optimal is true. Otherwise it returns a feasible solution.*

Proof: The proof is organized as follows. For clarity, the notations used during the proof are provided. Then, it proves the feasibility of the solution. At last, it proves the optimum of the solution when *is_optimal* is true.

Notations. A path from node v_i to v_j in graph G is denoted as $v_i \overset{G}{\Longrightarrow} v_j$, and a direct edge from v_i to v_j in graph G as $v_i \overset{G}{\rightarrow} v_j$.

Proof of Feasibility. A sufficient condition for the feasibility of the solution is that for each component C_i, consider all feasible partitions $\{V_1, ..., V_k\}$, such that $\cup_{s=1,...,k} V_s = V/C_i$, and $V_s \cap V_t = \emptyset, \forall s, t = 1, ...k, s \neq t$, and there is no cycle in $C_i \cup \{V_1, ..., V_k\}$, if partition $\{V_{k+1}, ..., V_n\}$ in C_i results in a cycle in the whole graph D, then it also creates a cycle in D_i. In this way, all infeasible partitions in D can be detected by D_i.

Given the condition that a cycle is created because of partition $\{V_{k+1}, ..., V_n\}$ in C_i, the cycle must contain at least one partition in $\{V_{k+1}, ..., V_n\}$. Otherwise, the original

$\{V_1, ..., V_k\}$ is infeasible. The partitions on the cycle is denoted as $V^c = \{V_p, ..., V_q\}$, leading to $|V^c| >= 1$. Regarding any two partitions $V_i^c, V_j^c \in V^c$ on the cycle, such that all the partitions between them are not in V^c (When $|V^c| = 1$, the two partition are the same one), there are two cases.

Case 1: there are no other partitions between these two partitions in the cycle: $V_i^c \xrightarrow{D} V_j^c$. In this case, the path does not go through any nodes outside of C_i, thus there must be a path $V_i^c \xrightarrow{D_i} V_j^c$ in the subgraph.

Case 2: there are at least one partition between these two partitions in the cycle. Without loss of generality, it can be assumed that the path is $V_i^c \xrightarrow{D} V_d \overset{D}{\Longrightarrow} V_a \xrightarrow{D} V_j^c$, where $V_d, V_a \notin V^c$. It is required to prove that there must be a path $V_i^c \overset{D_i}{\Longrightarrow} V_j^c$ in the subgraph.

Case 2.1: $V_d = V_a := V_*$. In this case, $V_i^c \xrightarrow{D} V_* \xrightarrow{D} V_j^c$. There must $\exists v_d, v_a \in V_*$, such that $V_i^c \xrightarrow{D} v_d$, $v_a \xrightarrow{D} V_j^c$. As N_i^a contains all the one-hop ancestors of C_i and N_i^d contains all the one-hop descendants of C_i, $v_a \in N_i^a$ and $v_d \in N_i^d$.

Case 2.1.1: $v_d = v_a$. $V_i^c \xrightarrow{D_i} v_d \xrightarrow{D_i} V_j^c$ exists as these nodes and edges are included when constructing D_i.

Case 2.1.2: $v_d \neq v_a$. According to Algorithm 5, there is an edge $v_d \xrightarrow{D_i} v_a$ added to graph D_i when it satisfies two conditions:

- $v_d \overset{\tilde{D}\backslash C_i}{\Longrightarrow} v_a$ exists in graph $\tilde{D}\backslash C_i$.
- There is no path $v_a \overset{D}{\Longrightarrow} v_d$ exists that contains nodes from two components.

In this case, for the first condition, as both v_a and v_d are in the same partition V_*, there must be a bidirectional path between them in graph $\tilde{D}$: $v_d \overset{\tilde{D}}{\Longleftrightarrow} v_a$. Moreover, as there are no partitions from V^c between the V_d and V_a on the cycle, it is safe to conclude that $v_d \overset{\tilde{D}\backslash C_i}{\Longleftrightarrow} v_a$ exists. The second condition can be proved by contradiction. Assuming that $v_a \overset{D}{\Longrightarrow} v_d$ exists and it contains nodes from at least two components, it suggests that there exists a $v_t \notin V_*$, such that path $v_d \overset{D}{\Longrightarrow} v_t \overset{D}{\Longrightarrow} v_a$ exists in the original graph D. With partition V_*, this creates a cycle $V_* \overset{D}{\Longrightarrow} v_t \overset{D}{\Longrightarrow} V_*$, which is contradict to the requirement that there is no cycle in $C_i \cup \{V_1, ..., V_k\}$. To conclude, under this case there is an edge $v_d \xrightarrow{D_i} v_a$ in the subgraph D_i. And the path $V_i^c \xrightarrow{D_i} v_d \xrightarrow{D_i} v_a \xrightarrow{D_i} V_j^c$ exists, which proves $V_i^c \overset{D_i}{\Longrightarrow}$ exists.

Case 2.2: $V_d \neq V_a$. In this case, path $V_i^c \xrightarrow{D} V_d \overset{D}{\Longrightarrow} V_a \xrightarrow{V_j^c}$ exists. Similar to Case 2.1, there $\exists v_d \in V_d, v_a \in V_a$, such that $V_i^c \xrightarrow{D_i} v_d$ and $v_a \xrightarrow{D_i} V_j^c$, and $v_a \in N_i^a$ and $v_d \in N_i^d$. Similar to Case 2.1.2, the following proves there is an edge $v_d \xrightarrow{D_i} v_a$ in graph D_i.

For the first condition, because of $V_d \overset{D}{\Longrightarrow} V_a$, there must $\exists v'_d \in V_d, v'_a \in V_a$, such that $v'_d \overset{D}{\Longrightarrow} v'_a$. As there is no partition from V^c between V_d and V_a, $v'_d \overset{\tilde{D}\backslash C_i}{\Longrightarrow} v'_a$. On the other hand, as v'_d and v_d are fused in the same partition, there exists a bidirectional path $v_d \overset{\tilde{D}}{\Longleftrightarrow} v'_d$. And because the path is internal to partition V_d, it's equivalent to $v_d \overset{\tilde{D}\backslash C_i}{\Longleftrightarrow} v'_d$. In symmetry, $v_a \overset{\tilde{D}\backslash C_i}{\Longleftrightarrow} v'_a$. By combining these paths, $v_d \overset{\tilde{D}\backslash C_i}{\Longrightarrow} v'_d \overset{\tilde{D}\backslash C_i}{\Longrightarrow} v'_a \overset{\tilde{D}\backslash C_i}{\Longrightarrow} v_a$, and the first condition holds. The second conditional can also be proved by contradiction. Assuming that $v_a \overset{D}{\Longrightarrow} v_d$ exists (it always contains nodes from at least two components as v_a and v_d are from different components), as v_a and v_d are fused int V_a and V_d, $V_a \overset{D}{\Longrightarrow} V_d$. However, in this case, $V_d \overset{D}{\Longrightarrow} V_a$, this results in a cycle $V_a \overset{D}{\Longrightarrow} V_d \overset{D}{\Longrightarrow} V_a$ and leads to contradiction.

With the two conditions hold, it can be concluded that edge $v_d \xrightarrow{D_i} v_a$ exists in D_i, which can be combined with $V_i^c \xrightarrow{D_i} v_d$ and $v_a \xrightarrow{D_i} V_j^c$ to create path $V_i^c \xrightarrow{D_i} v_d \xrightarrow{D_i} v_a \xrightarrow{D_i} V_j^c$.

With the discussion on all the cases above, it is proved that for any path $V_i^c \overset{D}{\Longrightarrow} V_j^c$ on the cycle, there is always a path $V_i^c \overset{D_i}{\Longrightarrow} V_j^c$ in the simplified graph D_i. Therefore, if a cycle exists in the whole graph D, the paths can be replaced to find a cycle in the simplified graph D_i. This proves the feasibility of the solutions obtained from the simplified graphs.

Proof of Optimum. When the *is_optimal* is true, line 7-15 in Algorithm 5 are equivalent to adding edge $v_d \xrightarrow{D_i} v_a$ when $v_d \overset{D}{\Longrightarrow} v_a$ exists. To prove that optimal solution can be found given the condition, it is required to prove that given a cycle in D_i, there must be a cycle in D as well. Without lost of generality, the cycle found in D_i can be denoted as $V_p^c \overset{D_i}{\Longrightarrow} V_{p+1}^c \overset{D_i}{\Longrightarrow} ... \overset{D_i}{\Longrightarrow} V_{q+1}^c$. As fusion does not remove paths, for each pair of $V_s^c \overset{D_i}{\Longrightarrow} V_{s+1}^c$, there is always a path in D. This proves that optimum can be ensured when *is_optimal* is true. ■

B.2 Benchmarks

This section elaborates the choice of benchmarks in detail.

BERT-large [111] for language modeling and VIT [112] for computer vision is constructed using stacked self-attention layers, which have become an indispensable building block in modern neural networks. These two benchmarks cover multiple mainloops including GEMM, Batched GEMM, Softmax, and LayerNorm operations. Their epilogue consists of various element-wise operations, activation functions, reductions, reshapes, and permutations that are challenging to fuse.

ResNet-50 [113] is one of the most popular backbone networks used in computer vision tasks. In addition to GEMM, ResNet-50 includes special heavy operations like Conv2dFprop and Conv2dDgrad. Additionally, the ”Conv-BN-ReLU” pattern of ResNet-50 makes it challenging to be fused due to the complexity of BN operation.

XML-CNN [114] is a representative example of extreme classification tasks, where the label space contains millions of categories. Its performance is critical for applications such as language modeling and recommendation systems. Due to the enormous number of categories, XML-CNN presents new challenges in optimizing the loss function, which uses binary cross-entropy loss, its gradient kernel, as well as surrounding element-wise functions and reductions.

GCN [106] for node classification is selected as an example of graph neural networks (GNN) used to model graph-structured data, such as protein and social networks. GCN introduces a new mainloop operation, SpMM, for its aggregation stage. Additionally, the large batch size of whole graph training can pose challenges for U-turn optimization.

Bibliography

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, *Imagenet classification with deep convolutional neural networks*, *Advances in neural information processing systems* **25** (2012).

[2] 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).

[3] W. Chan, N. Jaitly, Q. V. Le, and O. Vinyals, *Listen, attend and spell*, *arXiv preprint arXiv:1508.01211* (2015).

[4] E. Callaway, *What's next for the ai protein-folding revolution*, *Nature* **604** (2022) 234–238.

[5] W. Bao, J. Yue, and Y. Rao, *A deep learning framework for financial time series using stacked autoencoders and long-short term memory*, *PloS one* **12** (2017), no. 7 e0180944.

[6] Z. Chen and X. Huang, *End-to-end learning for lane keeping of self-driving cars*, in *2017 IEEE intelligent vehicles symposium (IV)*, pp. 1856–1860, IEEE, 2017.

[7] R. Raina, A. Madhavan, and A. Y. Ng, *Large-scale deep unsupervised learning using graphics processors*, in *Proceedings of the 26th annual international conference on machine learning*, pp. 873–880, 2009.

[8] D. Guide, *Cuda c programming guide*, *NVIDIA, July* **29** (2013) 31.

[9] G. Manso, *The computational limits of deep learning*, in *2024 AAAS Annual Meeting*, AAAS, 2024.

[10] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, *Analysis of {Large-Scale}{Multi-Tenant}{GPU} clusters for {DNN} training workloads*, in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 947–960, 2019.

[11] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia, *{AntMan}: Dynamic scaling on {GPU} clusters for deep learning*, in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 533–548, 2020.

[12] C.-J. Wu, R. Raghavendra, U. Gupta, B. Acun, N. Ardalani, K. Maeng, G. Chang, F. Aga, J. Huang, C. Bai, *et. al.*, *Sustainable ai: Environmental implications, challenges and opportunities*, *Proceedings of Machine Learning and Systems* **4** (2022) 795–813.

[13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, *Attention is all you need*, in *Advances in neural information processing systems*, pp. 5998–6008, 2017.

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

[15] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, *et. al.*, *Language models are unsupervised multitask learners*, *OpenAI blog* **1** (2019), no. 8 9.

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

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

[18] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming.* Addison-Wesley Professional, 2010.

[19] M. A. Raihan, N. Goli, and T. M. Aamodt, *Modeling deep learning accelerator enabled gpus*, in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 79–92, IEEE, 2019.

[20] T. NVIDIA, *V100 gpu architecture: The world's most advanced datacenter gpu*, *NVIDIA Corporation* (2017).

[21] NVIDIA, "Nvidia a100 tensor core gpu architecture." `https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf`, 2020.

[22] A. Zhou, Y. Ma, J. Zhu, J. Liu, Z. Zhang, K. Yuan, W. Sun, and H. Li, *Learning n:m fine-grained structured sparse neural networks from scratch*, in *International Conference on Learning Representations*, 2021.

[23] J. Pool and C. Yu, *Channel permutations for n: M sparsity*, *Advances in Neural Information Processing Systems* **34** (2021).

[24] A. Mishra, J. A. Latorre, J. Pool, D. Stosic, D. Stosic, G. Venkatesh, C. Yu, and P. Micikevicius, *Accelerating sparse deep neural networks*, *arXiv preprint arXiv:2104.08378* (2021).

[25] M. Zhu, T. Zhang, Z. Gu, and Y. Xie, *Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus*, in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 359–371, 2019.

[26] Z.-G. Liu, P. N. Whatmough, and M. Mattina, *Systolic tensor array: An efficient structured-sparse gemm accelerator for mobile cnn inference*, *IEEE Computer Architecture Letters* **19** (2020), no. 1 34–37.

[27] S. Han, H. Mao, and W. Dally, *Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding*, 10, 2016.

[28] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, *Quantization and training of neural networks for efficient integer-arithmetic-only inference*, in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2704–2713, 2018.

[29] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, *et. al.*, *Mixed precision training*, *arXiv preprint arXiv:1710.03740* (2017).

[30] S. Han, J. Pool, J. Tran, and W. J. Dally, *Learning both weights and connections for efficient neural networks*, *CoRR* **abs/1506.02626** (2015) [arXiv:1506.0262].

[31] S. Narang, G. F. Diamos, S. Sengupta, and E. Elsen, *Exploring sparsity in recurrent neural networks*, *CoRR* **abs/1704.05119** (2017) [arXiv:1704.0511].

[32] R. Child, S. Gray, A. Radford, and I. Sutskever, *Generating long sequences with sparse transformers*, *CoRR* **abs/1904.10509** (2019) [arXiv:1904.1050].

[33] L. Liu, L. Deng, Z. Chen, Y. Wang, S. Li, J. Zhang, Y. Yang, Z. Gu, Y. Ding, and Y. Xie, *Boosting deep neural network efficiency with dual-module inference*, in *Proceedings of the 37th International Conference on Machine Learning* (H. D. III and A. Singh, eds.), vol. 119 of *Proceedings of Machine Learning Research*, pp. 6205–6215, PMLR, 13–18 Jul, 2020.

[34] V. Akhlaghi, A. Yazdanbakhsh, K. Samadi, R. K. Gupta, and H. Esmaeilzadeh, *Snapea: Predictive early activation for reducing computation in deep convolutional neural networks*, in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 662–673, 2018.

[35] S. Sukhbaatar, E. Grave, P. Bojanowski, and A. Joulin, *Adaptive attention span in transformers*, *CoRR* **abs/1905.07799** (2019) [arXiv:1905.0779].

[36] M. Zaheer, G. Guruganesh, A. Dubey, J. Ainslie, C. Alberti, S. Ontanon, P. Pham, A. Ravula, Q. Wang, L. Yang, and A. Ahmed, *Big bird: Transformers for longer sequences*, 2021.

[37] Y. Tay, M. Dehghani, D. Bahri, and D. Metzler, *Efficient transformers: A survey*, *arXiv preprint arXiv:2009.06732* (2020).

[38] M. Zaheer, G. Guruganesh, K. A. Dubey, J. Ainslie, C. Alberti, S. Ontanon, P. Pham, A. Ravula, Q. Wang, L. Yang, *et. al.*, *Big bird: Transformers for longer sequences.*, in *NeurIPS*, 2020.

[39] I. Beltagy, M. E. Peters, and A. Cohan, *Longformer: The long-document transformer*, *arXiv preprint arXiv:2004.05150* (2020).

[40] T. J. Ham, Y. Lee, S. H. Seo, S. Kim, H. Choi, S. J. Jung, and J. W. Lee, *Elsa: Hardware-software co-design for efficient, lightweight self-attention mechanism in neural networks*, in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 692–705, IEEE, 2021.

[41] N. Kitaev, L. Kaiser, and A. Levskaya, *Reformer: The efficient transformer*, in *International Conference on Learning Representations*, 2020.

[42] A. Roy, M. Saffar, A. Vaswani, and D. Grangier, *Efficient content-based sparse attention with routing transformers*, *Transactions of the Association for Computational Linguistics* **9** (2021) 53–68.

[43] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, *Linformer: Self-attention with linear complexity*, *arXiv preprint arXiv:2006.04768* (2020).

[44] K. M. Choromanski, V. Likhosherstov, D. Dohan, X. Song, A. Gane, T. Sarlos, P. Hawkins, J. Q. Davis, A. Mohiuddin, L. Kaiser, D. B. Belanger, L. J. Colwell, and A. Weller, *Rethinking attention with performers*, in *International Conference on Learning Representations*, 2021.

[45] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, *et. al.*, *Tvm: an automated end-to-end optimizing compiler for deep learning*, in *Proceedings of the 13th USENIX conference on Operating Systems Design and Implementation*, pp. 579–594, 2018.

[46] J. Herrmann, J. Kho, B. Uçar, K. Kaya, and Ü. V. Çatalyürek, *Acyclic partitioning of large directed acyclic graphs*, in *2017 17th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGRID)*, pp. 371–380, IEEE, 2017.

[47] J. Nossack and E. Pesch, *A branch-and-bound algorithm for the acyclic partitioning problem*, *Computers & operations research* **41** (2014) 174–184.

[48] B. W. Kernighan and S. Lin, *An efficient heuristic procedure for partitioning graphs*, *The Bell system technical journal* **49** (1970), no. 2 291–307.

[49] T. Gale, M. Zaharia, C. Young, and E. Elsen, *Sparse GPU kernels for deep learning*, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020*, 2020.

[50] B. Xu, Y. Zhang, H. Lu, Y. Chen, T. Chen, M. Iovine, M.-C. Lee, and Z. Li, *AITemplate*, 10, 2022.

[51] J. Xing, L. Wang, S. Zhang, J. Chen, A. Chen, and Y. Zhu, *Bolt: Bridging the gap between auto-tuners and hardware-native performance*, *Proceedings of Machine Learning and Systems* **4** (2022) 204–216.

[52] G. Huang, G. Dai, Y. Wang, and H. Yang, *Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks*, in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, IEEE, 2020.

[53] Z. Chen, Z. Qu, L. Liu, Y. Ding, and Y. Xie, *Efficient tensor core-based gpu kernels for structured sparsity under reduced precision*, .

[54] V. Thakkar, P. Ramani, C. Cecka, A. Shivam, H. Lu, E. Yan, J. Kosaian, M. Hoemmen, H. Wu, A. Kerr, M. Nicely, D. Merrill, D. Blasig, F. Qiao, P. Majcher, P. Springer, M. Hohnerbach, J. Wang, and M. Gupta, *CUTLASS*, 1, 2023.

[55] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, *Tiramisu: A polyhedral compiler for expressing fast and portable code*, in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 193–205, IEEE, 2019.

[56] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, *Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions*, *arXiv preprint arXiv:1802.04730* (2018).

[57] G. Venkatesh, E. Nurvitadhi, and D. Marr, *Accelerating deep convolutional networks using low-precision and sparsity*, in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 2861–2865, IEEE, 2017.

[58] B. Li, W. Wen, J. Mao, S. Li, Y. Chen, and H. Li, *Running sparse and low-precision neural network: When algorithm meets hardware*, in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 534–539, IEEE, 2018.

[59] M. S. Park, X. Xu, and C. Brick, *Squantizer: Simultaneous learning for both sparse and low-precision neural networks*, *arXiv preprint arXiv:1812.08301* (2018).

[60] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. J. Dally, *Exploring the regularity of sparse structure in convolutional neural networks*, *arXiv preprint arXiv:1705.08922* (2017).

[61] G. Research, "Deep Learning Matrix Collection." `https://github.com/google-research/google-research/tree/master/sgk`.

[62] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, *Cusparse library*, in *GPU Technology Conference*, 2010.

[63] D. Yan, W. Wang, and X. Chu, *Optimizing batched winograd convolution on gpus*, in *25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '20)*, (San Diego, CA, USA), ACM, 2020.

[64] M. Zaheer, G. Guruganesh, K. A. Dubey, J. Ainslie, C. Alberti, S. Ontanon, P. Pham, A. Ravula, Q. Wang, L. Yang, and A. Ahmed, *Big bird: Transformers for longer sequences*, in *Advances in Neural Information Processing Systems* (H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, eds.), vol. 33, pp. 17283–17297, Curran Associates, Inc., 2020.

[65] Y. Tay, M. Dehghani, S. Abnar, Y. Shen, D. Bahri, P. Pham, J. Rao, L. Yang, S. Ruder, and D. Metzler, *Long range arena: A benchmark for efficient transformers*, *arXiv preprint arXiv:2011.04006* (2020).

[66] M. Ott, S. Edunov, D. Grangier, and M. Auli, *Scaling neural machine translation*, in *Proceedings of the Third Conference on Machine Translation: Research Papers*, pp. 1–9, 2018.

[67] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, *An image is worth 16x16 words: Transformers for image recognition at scale*, in *International Conference on Learning Representations*, 2021.

[68] Y. Tay, D. Bahri, L. Yang, D. Metzler, and D.-C. Juan, *Sparse sinkhorn attention*, in *International Conference on Machine Learning*, pp. 9438–9447, PMLR, 2020.

[69] Y. Xiong, Z. Zeng, R. Chakraborty, M. Tan, G. Fung, Y. Li, and V. Singh, *Nyströmformer: A nyström-based algorithm for approximating self-attention*, in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, pp. 14138–14148, 2021.

[70] A. Kerr, "Cutlass." `https://github.com/NVIDIA/cutlass`, 2021.

[71] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, *Transformers: State-of-the-art natural language processing*, in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, (Online), pp. 38–45, Association for Computational Linguistics, Oct., 2020.

[72] L. Zehui, P. Liu, L. Huang, J. Chen, X. Qiu, and X. Huang, *Dropattention: A regularization method for fully-connected self-attention networks*, *arXiv preprint arXiv:1907.11065* (2019).

[73] Y. Tay, M. Dehghani, S. Abnar, Y. Shen, D. Bahri, P. Pham, J. Rao, L. Yang, S. Ruder, and D. Metzler, *Long range arena : A benchmark for efficient transformers*, in *International Conference on Learning Representations*, 2021.

[74] N. Parmar, A. Vaswani, J. Uszkoreit, L. Kaiser, N. Shazeer, A. Ku, and D. Tran, *Image transformer*, in *International Conference on Machine Learning*, pp. 4055–4064, PMLR, 2018.

[75] R. Child, S. Gray, A. Radford, and I. Sutskever, *Generating long sequences with sparse transformers*, *arXiv preprint arXiv:1904.10509* (2019).

[76] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret, *Transformers are rnns: Fast autoregressive transformers with linear attention*, in *International Conference on Machine Learning*, pp. 5156–5165, PMLR, 2020.

[77] J. Lu, J. Yao, J. Zhang, X. Zhu, H. Xu, W. Gao, C. Xu, T. Xiang, and L. Zhang, *Soft: Softmax-free transformer with linear complexity*, *arXiv preprint arXiv:2110.11945* (2021).

[78] M. Yan, Z. Chen, L. Deng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, *Characterizing and understanding gcns on gpu*, *IEEE Computer Architecture Letters* (2020).

[79] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, *Hygcn: A gcn accelerator with hybrid architecture*, in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb, 2020.

[80] H. Zhang, T. Yan, M. D. Wong, and S. J. Patel, *Accelerating aerial image simulation with gpu*, in *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 178–184, IEEE, 2011.

[81] H. Qian and Y. Deng, *Accelerating rtl simulation with gpus*, in *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 687–693, IEEE, 2011.

[82] Y. Deng, B. D. Wang, and S. Mu, *Taming irregular eda applications on gpus*, in *Proceedings of the 2009 International Conference on Computer-Aided Design*, pp. 539–546, 2009.

[83] G. Li, M. Müller, G. Qian, I. C. Delgadillo, A. Abualshour, A. Thabet, and B. Ghanem, *Deepgcns: Making gcns go as deep as cnns*, *arXiv preprint arXiv:1910.06849* (2019).

[84] C. Nvidia, *Cublas library*, *NVIDIA Corporation, Santa Clara, California* **15** (2008), no. 27 31.

[85] M. Fey and J. E. Lenssen, *Fast graph representation learning with pytorch geometric*, *arXiv preprint arXiv:1903.02428* (2019).

[86] 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).

[87] 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 *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pp. 443–458, 2019.

[88] K. K. Thekumparampil, C. Wang, S. Oh, and L.-J. Li, *Attention-based graph neural network for semi-supervised learning*, *arXiv preprint arXiv:1803.03735* (2018).

[89] M. Harris *et. al.*, *Optimizing parallel reduction in cuda*, *Nvidia developer technology* **2** (2007), no. 4 70.

[90] T. D. Han and T. S. Abdelrahman, *Reducing divergence in gpgpu programs with loop merging*, in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pp. 12–23, 2013.

[91] L. Nyland and S. Jones, *Understanding and using atomic memory operations*, in *4th GPU Technology Conf.(GTC'13), March*, 2013.

[92] J. Filipovič, M. Madzin, J. Fousek, and L. Matyska, *Optimizing cuda code by kernel fusion: application on blas*, *The Journal of Supercomputing* **71** (2015), no. 10 3934–3957.

[93] F. Wu, A. Souza, T. Zhang, C. Fifty, T. Yu, and K. Weinberger, *Simplifying graph convolutional networks*, in *International Conference on Machine Learning*, pp. 6861–6871, 2019.

[94] W. Hamilton, Z. Ying, and J. Leskovec, *Inductive representation learning on large graphs*, in *Advances in neural information processing systems*, pp. 1024–1034, 2017.

[95] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, *Dissecting the nvidia volta gpu architecture via microbenchmarking*, *arXiv preprint arXiv:1804.06826* (2018).

[96] C. CUDA, "Best practices guide-cuda toolkit documentation."

[97] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, *Improving gpgpu resource utilization through alternative thread block scheduling*, in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 260–271, IEEE, 2014.

[98] 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 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[99] M. Li, Y. Liu, X. Liu, Q. Sun, X. You, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian, *The deep learning compiler: A comprehensive survey*, *IEEE Transactions on Parallel and Distributed Systems* **32** (2020), no. 3 708–727.

[100] G. Wang, Y. Lin, and W. Yi, *Kernel fusion: An effective method for better power efficiency on multithreaded gpu*, in *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pp. 344–350, IEEE, 2010.

[101] 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*, pp. 10–19, 2019.

[102] M. Y. Özkaya and Ü. V. Çatalyürek, *A simple and elegant mathematical formulation for the acyclic dag partitioning problem*, *arXiv preprint arXiv:2207.13638* (2022).

[103] S. Ioffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, in *International conference on machine learning*, pp. 448–456, pmlr, 2015.

[104] W. Jung, D. Jung, B. Kim, S. Lee, W. Rhee, and J. H. Ahn, *Restructuring batch normalization to accelerate cnn training*, *Proceedings of Machine Learning and Systems* **1** (2019) 14–26.

[105] P. Briggs and K. D. Cooper, *Effective partial redundancy elimination*, *ACM SIGPLAN Notices* **29** (1994), no. 6 159–170.

[106] T. N. Kipf and M. Welling, *Semi-supervised classification with graph convolutional networks*, in *International Conference on Learning Representations.*

[107] C. E. Miller, A. W. Tucker, and R. A. Zemlin, *Integer programming formulation of traveling salesman problems*, *Journal of the ACM (JACM)* **7** (1960), no. 4 326–329.

[108] N. Shervashidze, P. Schweitzer, E. J. Van Leeuwen, K. Mehlhorn, and K. M. Borgwardt, *Weisfeiler-lehman graph kernels.*, *Journal of Machine Learning Research* **12** (2011), no. 9.

[109] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, *Mlir: A compiler infrastructure for the end of moore's law*, *arXiv preprint arXiv:2002.11054* (2020).

[110] M. Osama, D. Merrill, C. Cecka, M. Garland, and J. D. Owens, *Stream-k: Work-centric parallel decomposition for dense matrix-matrix multiplication on the gpu*, in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pp. 429–431, 2023.

[111] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, *Bert: Pre-training of deep bidirectional transformers for language understanding*, in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, 2019.

[112] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, *et. al.*, *An image is worth 16x16 words: Transformers for image recognition at scale*, in *International Conference on Learning Representations.*

[113] 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*, pp. 770–778, 2016.

[114] J. Liu, W.-C. Chang, Y. Wu, and Y. Yang, *Deep learning for extreme multi-label text classification*, in *Proceedings of the 40th international ACM SIGIR conference on research and development in information retrieval*, pp. 115–124, 2017.

[115] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, *et. al.*, *Ansor: Generating high-performance tensor programs for deep learning*, in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pp. 863–879, 2020.

[116] T. Chakraborti, B. McCane, S. Mills, and U. Pal, *Coconet: A collaborative convolutional network*, *arXiv preprint arXiv:1901.09886* (2019).

[117] K. Punniyamurthy, B. M. Beckmann, and K. Hamidouche, *Gpu-initiated fine-grained overlap of collective communication with computation*, *arXiv preprint arXiv:2305.06942* (2023).

[118] J. Frankle and M. Carbin, *The lottery ticket hypothesis: Finding sparse, trainable neural networks*, in *International Conference on Learning Representations*, 2019.

[119] T. Chen, Y. Sui, X. Chen, A. Zhang, and Z. Wang, *A unified lottery ticket hypothesis for graph neural networks*, in *International Conference on Machine Learning*, pp. 1695–1706, PMLR, 2021.

[120] H. Wang, Z. Zhang, and S. Han, *Spatten: Efficient sparse attention architecture with cascade token and head pruning*, in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 97–110, IEEE, 2021.

[121] G. Zhao, J. Lin, Z. Zhang, X. Ren, Q. Su, and X. Sun, *Explicit sparse transformer: Concentrated attention through explicit selection*, *arXiv preprint arXiv:1912.11637* (2019).