Compiler and Runtime Systems for Generative AI Models

Zihao Ye

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

Doctor of Philosophy

University of Washington

2025

Reading Committee:

Luis Ceze, Chair

Tianqi Chen

Arvind Krishnamurthy

Baris Kasikci

Stephanie Wang

Luke Zettlemoyer

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Compiler and Runtime Systems for Generative AI Models

Zihao Ye

Chair of the Supervisory Committee:
Professor Luis Ceze
Computer Science & Engineering

Generative AI (GenAI) workloads have rapidly become the predominant data center GPU workload. However, designing efficient GPU kernels for GenAI presents significant challenges due to two central factors: (1) GenAI workloads are intrinsically dynamic—featuring variable sequence lengths and irregular sparsity patterns—and (2) they evolve at a rapid pace, with shifting model architectures and changing deployment requirements.

This dissertation addresses these challenges through a co-design approach spanning both compiler and runtime layers, presenting two complementary systems that collectively enable efficient GenAI acceleration.

SparseTIR is a tensor compiler specifically designed for sparse deep learning workloads. While sparsity is pervasive in GenAI models, developing high-performance sparse GPU kernels remains difficult due to heterogeneous sparsity patterns and their unique optimization requirements. SparseTIR introduces composable abstractions for both data formats and scheduling transformations, enabling complex optimization strategies with significantly reduced code complexity. It achieves performance competitive with hand-optimized libraries while improving modularity and developer productivity.

FlashInfer is a fast and adaptable attention engine tailored for large language model (LLM) inference. As attention increasingly dominates computational costs in modern GenAI models, scalable and customizable GPU kernels become essential. FlashInfer supports

block-sparse KV-cache layouts, Just-In-Time (JIT) compilation of parameterized attention templates, and dynamic load-balancing mechanisms compatible with CUDA Graphs. Building on this foundation, we are developing megakernels for low-latency inference and multiplexing inference scenarios. As an open-source project, FlashInfer has pioneered LLM inference kernel development, being among the first to explore techniques like split-KV, GQA packing, and cascade inference. It has been deployed at scale in production environments and fostered a vibrant community across academia and industry.

These systems form a cohesive framework for accelerating GenAI workloads through integrated compiler-runtime co-design. They demonstrate how principled systems approaches can achieve both high performance and adaptability in response to rapidly evolving machine learning demands, providing a foundation for future GenAI system development.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First and foremost, I thank the University of Washington and Seattle. UW's beautiful campus and Seattle's stunning environment have made Seattle an unforgettable place in my life, and I've even grown to love the rainy season. I can hardly imagine a more perfect city than Seattle in summer.

I am deeply grateful to my advisor, Professor Luis Ceze. More than a supervisor, Luis has been like a friend who is just slightly older than me–incredibly approachable and warm. One of my most cherished experiences during my PhD was discussing various ideas with Luis at the whiteboard in his office (though admittedly, most remained unexplored due to limited bandwidth, but surprisingly, many were later validated by others as excellent ideas–like CLAP. You know what I mean, Luis!). Luis is perpetually optimistic and transmits this spirit to his students. In contrast, I tend to be more conservative and occasionally pessimistic, but it was precisely Luis's character that helped sustain me through challenging times. Luis's research is cutting-edge and bold, encouraging innovation and supporting students in pursuing whatever they wish to explore. I remember several times during my PhD when I had seemingly unreliable ideas I wanted to try–Luis unhesitatingly helped me purchase the machines, and though those projects didn't pan out, it was this unrestricted environment that allowed me to fearlessly explore topics I was passionate about, such as SparseTIR and FlashInfer. Beyond research, Luis generously shares life's pleasures with his students. When I was writing GQA packing for FlashInfer, I told him I was using a technique from "Hacker's Delight" called fast divisor to reduce our binary size. Luis then told me that Henry S. Warren, the author of "Hacker's Delight," was actually his mentor during his IBM internship–what a small world! He gifted me a copy of "Hacker's Delight," inscribed with "I hope this delights you as much

as it delights me"–and indeed it does! I hope to continue passing on such delight. Despite his busy schedule, Luis always makes time for his students. I still remember our bike rides near Union Lake, and after graduation, we must find time to continue exploring together!

I want to thank Professor Tianqi Chen (CMU). Tianqi is like my co-advisor (though not officially). One important reason I chose to pursue my PhD at UW was because of Tianqi–he had created many industrially impactful open-source projects here: XGBoost, MXNet, and TVM. I believed UW had the genes for producing excellent open-source projects–and indeed it does. Tianqi is an exemplary mentor with encyclopedic knowledge of deep learning systems and compilers, clear convictions (though we couldn't always grasp the full picture), and the ability to lead teams in advancing these directions. Often, he could accomplish tasks much faster himself, yet he was willing to spend time cultivating us (sometimes slowing down project progress), demonstrating unparalleled leadership and serving as a life role model. Working with Tianqi taught me how to drive large-scale projects, laying the foundation for later leading the FlashInfer project. With his rich experience in machine learning and systems, many of his casual ideas during our conversations later proved to be highly effective work. However, Tianqi maintains his research taste and doesn't chase trends simply because a topic is interesting–a virtue I need. In this restless era, having one's own convictions is rare and valuable. Tianqi is among the most approachable people I've met. Despite being the strongest engineer I know (and a professor and a scientist), he's always willing to share technical knowledge purely, never judging others regardless of their skill level. Having such a mentor has been a blessing during my PhD and in life. Tianqi is a deep believer in fully automated machine learning compilers, and I believe when that day truly comes, his persistence will be one of the most significant contributing factors.

I thank my two best friends at UW during my PhD–Lequn Chen and Chien-Yu Lin. Lequn was my undergraduate classmate and later worked in the same field at UW. Lequn is an expert in computer systems and networking–a true system hacker. From him, I learned not

only knowledge but also his hacker spirit. We collaborated extensively during our PhDs, and conversations with Lequn were always enlightening. Lequn introduced me to LLM inference. He's also someone who loves life, skilled in cooking, and often invited me to his home to enjoy delicious food. I regret not learning this skill during my PhD, but I promise to do more after graduation. Chien-Yu Lin, a fellow PhD under Luis, has been mutually supportive in both research and life. Chien-Yu introduced me to cycling and together we completed the 2024 Seattle to Portland ride. I can hardly imagine completing this feat without his continuous help (I wasn't previously good at endurance sports).

I thank UW SAMPL (now SyFI lab) and Syslab. UW has some of the world's best professors and graduate students in machine learning systems. Whether in GPUs, networking, compilers, or distributed systems, UW has corresponding talent. The professors actively explore new research directions and participate in discussions, while students are self-motivated and get along harmoniously–we're like a family, collaborating and sharing with each other. The weekly SAMPL Lunch was one of the most rewarding activities during my PhD. I thank Professor Arvind Krishnamurthy, Professor Baris Kasikci, and Professor Stephanie Wang. Your expertise has benefited me immensely. We've had many conversations, not just about research but also about life and personal development. No matter how busy you are, you always make time to chat with me. I thank fellow students Liangyu Zhao, Size Zheng, Aditya Kamath, Jaehong Min, Dedong Xie, Xiangfeng Zhu, Frank Zhao, Kan Zhu, Keisuke Kamahori, Yile Gu, Wei Shen, Weixin Deng, Pratyush Patel, Xieyang Xu, Tapan Chugh, Zezhou Wang, Vic Li, and others for your support along the way. I also thank the excellent undergraduate interns I mentored at UW: Yixin Dong, Yilong Zhao, Yifei Zuo, and Shanli Xing–it's my honor to work with you!

I thank the CMU Catalyst research group. I spent a delightful six months there from January to June 2025. Although Pittsburgh was snowing when I arrived, I could feel everyone's passion for their research directions. Thank you for Tianqi's invitation and hosting, and

Professor Zhihao Jia–an incredibly brilliant systems and compilers researcher. I witnessed the birth of Mirage Persistent Kernel (partially beginning from our office conversations), and I learned about task-level programming models from you. Though I didn't stay longer at CMU, I hope to continue collaborating. I always enjoy working with excellent researchers like you. Professor Beidi Chen–an always energetic and smart efficient ML researcher, conversations with you always ignite my confidence in sparsity. I'm honored to collaborate with you and your PhD students! Thank you to CMU students for your hospitality, especially Ruihang Lai, Hongyi Jin, and Bohan Hou. We've collaborated since 2022, witnessing the birth of various projects like TensorIR/Relax and MLC-LLM, I enjoy the days having dinner, riding bikes and play DotA with you. Along with collaborators Mengdi Wu, Zhihao Zhang, Yonghao Zhuang, Zhuoming Chen, Xinhao Cheng, Charlie Ruan, Yingyi Huang, Aksara Bayyapu, Kathryn Chen, Yiyan Zhai, Alexander Jiang, and others.

I spent a full year of my PhD at NVIDIA. I thank my mentor Vinod Grover. As one of the original CUDA team members, Vinod understands extensive history about parallel computing, compilers, and GPUs. Every conversation with Vinod taught me much about this ancestral wisdom. Vinod has broad interests, continuous learning habits, and courage to challenge himself and accept new things–he's an exemplar for us to learn from.

I thank friends and open-source community members who supported me during FlashInfer development. Without you, this project couldn't have reached where it is today. Initially, FlashInfer was just a research prototype, and midway it was scooped multiple times (FlashAttention-2, Flash-Decoding, etc.). I once considered giving up, but with support from Tianqi, Luis, and others, I persevered. Special thanks to Ying Sheng and Lianmin Zheng from LMSys Org, who told me in late 2023 about their project called SGLang that could perfectly use our project. Thus, SGLang became one of our early and important users. I also thank Yineng Zhang for long-term support. In the crucial field of LLM inference, being scooped is normal. We need long-term persistence, discovering real-world needs, and continuously

breaking through ourselves. I thank all researchers in this field and the open-source community (SGLang, vLLM, FlashAttention, ThunderKittens, etc.) for making this field so active and innovative.

I thank my mentors Zheng Zhang and Minjie Wang from AWS Shanghai AI Lab before my PhD. They led me toward machine learning systems. I was fortunate to become a founding member of Deep Graph Library, learning from machine learning to distributed training and deep learning compilers, laying a solid foundation for my PhD. From Minjie Wang, I learned leadership and how to manage open-source communities. From Zheng Zhang, I experienced interest-driven research and good sense. I also thank Zheng Zhang for the atmosphere he created at AWS Shanghai AI Lab–relaxed, efficient, and collaborative. I also thank my colleagues and early team members on DGL: Mufei Li, Jinjing Zhou, Zhiqiang Xie, Chao Ma, Qipeng Guo, Quan Gan, Lingfan Yu, Da Zheng, Xiang Song, Tianjun Xiao, Tong He and many more, it's a great memory to work with you.

I thank "Heroes of Might and Magic III," a game I've been fascinated with since age 8, accompanying me through countless leisure hours. I can remember most maps and campaigns. Though this game is about as old as I am, I still enjoy it immensely. This sets one of my work standards–creating classics, not things people will quickly forget.

I thank "Doctor Who," especially the 12th Doctor. Doctor Who is one of my favorite shows. Whenever I want to give up, I watch "Heaven Sent," where the Doctor regenerates again and again, breaking through difficulties. "Never be cruel, never be cowardly. And never ever eat pears! Remember–hate is always foolish... and love is always wise. Always try to be nice and never fail to be kind." Thank you for your guidance. Now I'm also a Doctor, and like you, I'll do the right and decent thing, without hope, without witness, without reward.

I especially need to thank my parents and family, who supported me throughout my PhD. During my PhD, I didn't have many opportunities to return home to visit family, which is one of my regrets. I hope they always feel my love.

Finally, special thanks to my fiancée, Yuhe(Lotus) Guo. No words can adequately express my love and gratitude. We've known each other for over five years, spending most of my PhD time in a long-distance relationship. You have always been my guide through the fog, showing me the way forward when I couldn't see clearly. Yuhe is the other side of my personality, sensing many feelings I overlook, but we're both fundamentally kind. When I am impulsive, Yuhe will calm me down, and vice versa, none of us is perfect, but the two of us are perfect for each other. During my PhD, I experienced many burnouts, and Yuhe's love helped me through these difficult times. In our time knowing each other, we've supported and listened to each other. Though working in different research directions, we share similar qualities–treating our research like our children. We'll continue supporting each other through the coming years of our lives together.

Chapter 1
INTRODUCTION

1.1 GenAI's Systems Gap

Transformer-based LLMs, Image and Video Generation models have turned *generative AI* (GenAI) into the dominant data-center workload. Yet GPUs rarely reach its theoretical throughput on GenAI because the software stack they inherit–dense, static, monolithic–clashes with three realities:

- **Irregular structure.** Pruning, mixture-of-experts (MoE) gating, long-context attention masks, and graph inputs introduce pervasive irregularity and sparsity. Meanwhile, hardware evolution trends toward increasingly dense tensor cores and bulk memory operations, creating a fundamental mismatch that challenges software stacks to develop efficient sparse kernels.

- **Runtime dynamism.** Inference workloads exhibit unpredictable variability through fluctuating batch sizes, dynamic sequence lengths, continuously evolving KV-cache layouts, and distinct operational phases (prefill, decode, and append) that dramatically alter computational patterns at runtime.

- **Fast evolution.** Novel attention mechanisms, KV-cache compression techniques, and emerging inference strategies like speculative decoding emerge at a pace that consistently outstrips vendor library development and performance optimization cycles.

Bridging this gap requires a fundamental rethinking that integrates *both* compiler and runtime layers, rather than relying on isolated optimizations. This dissertation presents such a co-design through two systems: *SparseTIR* and *FlashInfer*.

Tensor Compiler: SparseTIR. SparseTIR replaces "single-format, single-shot" sparse compilers with a *multi-stage IR* in which *formats* (e.g. CSR, BSR, ELL) and *schedules* (tiling, fusion, tensorization) are first-class, *independently composable* objects. This composability enables developers to decompose sparse matrices into hierarchical structures–blocks, bands, or hybrid layouts–and systematically explore the joint format-schedule design space to maximize hardware utilization. By leveraging a loop-level IR compatible with TVM in stage III, SparseTIR seamlessly integrates mature optimization techniques such as tensor-core tensorization and horizontal fusion with minimal adaptation.

For the diverse sparsity patterns in deep learning workloads, SparseTIR introduces a novel composable format approach that combines multiple hardware-friendly sparse representations to capture the original sparse pattern. This approach then applies composable transformations tailored to each specific format, systematically optimizing performance through format-aware scheduling.

In extensive evaluations, SparseTIR-generated CUDA kernels outperform vendor libraries by 1.20–2.34× for GNN operators, 1.05–2.98× for sparse attention, and up to 7.45× for sparse convolutions. End-to-end benchmarks demonstrate acceleration of GraphSAGE training by up to 1.52× and RGCN inference by up to 40×–all while reducing the volume of hand-written kernel code by an order of magnitude.

Kernel and System Co-Design: FlashInfer While compilers significantly accelerate GPU kernel development, they often fall short of optimal performance for critical use cases–a gap that remains unacceptable for industrial adoption. This performance deficit stems from either compilers lacking support for cutting-edge hardware features or intermediate representations that cannot adequately express advanced kernel optimizations. In our second work, we deliberately step back from fully automatic approaches to explore a more balanced methodology that sacrifices some automation for greater developer flexibility. This approach enables us to generate high-performance kernels specifically for scenarios where existing compilers prove inadequate.

Attention dominates inference latency but must navigate a complex landscape of cache formats, masking rules, and precision modes. **FlashInfer** approaches attention with the same principled abstraction that BLAS provides for GEMM: a unified template with specialized implementations. It introduces a flexible block-sparse KV abstraction, dynamically generates optimized CUDA kernels just-in-time, and intelligently schedules computation tiles across streaming multiprocessors to minimize tail latency.

Drawing inspiration from the Inspector-Executor paradigm in high-performance computing, FlashInfer analyzes sequence length information before each generation step to develop optimal tile-scheduling strategies for each streaming multiprocessor. This approach ensures balanced workloads and maximizes GPU hardware utilization throughout the inference process.

FlashInfer has been successfully integrated into leading LLM serving frameworks including SGLang, vLLM, and MLC-Engine. Extensive evaluations at both kernel and end-to-end levels demonstrate FlashInfer's substantial performance improvements across diverse inference scenarios: compared to state-of-the-art LLM serving solutions, FlashInfer delivers 29-69% reduction in inter-token latency versus compiler backends on standard LLM serving benchmarks, 28-30% latency reduction for long-context inference, and 13-17% speedup for LLM serving with parallel generation.

1.2 Contributions

This section outlines the key contributions of this dissertation, which builds upon our published research on SparseTIR [183] and FlashInfer [182].

The primary contributions of this dissertation are:

1. **SparseTIR**: A novel framework featuring composable format and transformation abstractions that significantly simplifies the development of efficient sparse GPU kernels, enabling rapid creation and optimization of new sparse operators.

2. **FlashInfer**: A high-performance, open-source attention engine with just-in-time compilation that unifies diverse KV-cache layouts and supports emerging attention variants,

delivering exceptional performance for LLM inference workloads.

3. An integrated open-source ecosystem that fosters collaboration between LLM inference engine developers and kernel/compiler specialists, creating virtuous feedback loops that accelerate innovation and performance optimization.

Chapter 2
BACKGROUND

This chapter provides an background introduction across the key areas addressed in this dissertation: sparse tensor computation, attention mechanisms for large language models, and systems for efficient deep learning inference.

2.1 Sparse Tensor Computation

Sparse tensor computation plays a crucial role in modern machine learning systems, where efficiency is paramount due to the scale of operations. Sparse tensors–data structures where the majority of elements are zero–are ubiquitous in deep learning applications, from natural language processing to computer vision and graph neural networks. Sparsity arises naturally in many domains: adjacency matrices of large graphs are typically sparse, attention matrices often exhibit structured sparsity patterns, and model weights can be pruned to induce sparsity while maintaining accuracy.

Traditional dense tensor operations waste computational resources and memory bandwidth when operating on sparse data. Specialized sparse formats and algorithms address this inefficiency by storing and computing only with non-zero elements. Common sparse matrix formats include Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), and Coordinate Format (COO). These formats eliminate the need to store zero elements, reducing memory requirements and potentially accelerating computation.

More advanced sparse formats like Block Compressed Sparse Row (BSR) [1] group non-zero elements into contiguous blocks, enabling better hardware utilization on modern accelerators such as GPUs. BSR improves register reuse efficiency and demonstrates better compatibility with hardware matrix multiplication units. The efficiency of BSR is particularly evident when

subcomputations align with hardware matrix multiplication instructions, such as NVIDIA's tensor core operations.

Recent research has shown that efficient utilization of tensor cores can be achieved with smaller block sizes, such as (16,1) for matrix multiplication, challenging the traditional approach of using large block sizes. These vector-sparse formats are particularly beneficial for applications with fine-grained sparsity patterns, offering greater flexibility and efficiency.

2.2 Machine Learning Compilers

Machine learning compilers have emerged as essential tools for bridging the gap between high-level model descriptions and efficient hardware execution. Unlike traditional compilers that optimize general-purpose code, ML compilers specialize in translating tensor operations into optimized kernels for specific hardware targets. These compilers typically employ a multi-level intermediate representation (IR) that progressively lowers high-level operations into hardware-specific instructions.

Pioneering frameworks like TVM [25, 26], MLIR [85], and XLA [43] have established the standard approach of using domain-specific IRs to represent and optimize tensor computations. These systems typically separate the computation description (what to compute) from scheduling directives (how to compute it), allowing for systematic exploration of implementation strategies without changing the algorithm specification.

The scheduling process involves applying transformations such as tiling, fusion, vectorization, and parallelization to optimize memory access patterns and computational throughput. For dense computations, these techniques have proven highly effective, often generating code that rivals or exceeds hand-tuned libraries. However, extending these capabilities to sparse computations presents unique challenges.

2.2.1 Sparse Tensor Compilers

Sparse tensor compilers aim to automate the generation of efficient code for sparse operations, but face several distinctive challenges not present in dense compilation:

- **Format complexity**: Different sparse formats (CSR, COO, BSR, etc.) require specialized access patterns and algorithms, creating a combinatorial explosion of implementation variants.

- **Irregular memory access**: Indirect memory accesses through index arrays complicate prefetching, caching, and vectorization strategies.

- **Load imbalance**: The non-uniform distribution of non-zero elements can lead to workload imbalance across parallel processing units.

- **Format-schedule coupling**: The optimal execution schedule often depends heavily on the specific sparse format being used, limiting the reuse of optimization strategies.

Existing approaches to sparse compilation, such as the Sparse Tensor Algebra Compiler (TACO) [81], MLIR's sparse tensor support [12], and COMET [157], have made significant progress but typically handle only a limited set of formats or require format-specific schedule implementations. This leads to increased development effort when supporting new formats or combinations of formats.

2.2.2 SparseTIR: A Novel Approach to Sparse Compilation

SparseTIR [183] addresses these limitations through a fundamental redesign of the sparse compilation approach. Instead of treating formats and schedules as tightly coupled entities, SparseTIR introduces a multi-stage IR where formats and schedules are first-class, independently composable objects. This composability enables several key innovations:

- **Format composition**: SparseTIR allows decomposing sparse matrices into hierarchical structures–blocks, bands, or hybrid layouts–that better match both the natural sparsity pattern of the data and the capabilities of the target hardware.

- **Format-schedule decoupling**: By separating format specification from scheduling directives, SparseTIR enables systematic exploration of the joint format-schedule design space without requiring manual implementation of each combination.

- **Hardware-friendly abstractions**: SparseTIR's architecture facilitates the integration of advanced hardware features such as tensor cores through format-aware tensorization, enabling sparse operations to leverage specialized hardware acceleration.

A key insight of SparseTIR is that no single format is optimal for all sparsity patterns. Instead, by composing multiple hardware-friendly sparse representations (e.g., combining BSR for dense blocks with vector-sparse formats for scattered elements), the system can better capture the original sparse pattern while maximizing hardware utilization.

This approach has proven remarkably effective in practice. SparseTIR-generated CUDA kernels consistently outperform vendor libraries across a range of applications, from graph neural networks (1.20–2.34×) to sparse attention mechanisms (1.05–2.98×) and sparse convolutions (up to 7.45×). Moreover, SparseTIR significantly reduces development effort by eliminating the need for format-specific kernel implementations, enabling rapid exploration of novel sparse operator designs.

The challenge in sparse tensor computation ultimately lies in balancing format flexibility, computational efficiency, and ease of programming. This dissertation explores how SparseTIR's approach to sparse tensor computation can be applied to critical workloads in generative AI, with particular focus on attention mechanisms in large language models, where both structured and unstructured sparsity patterns can be exploited for significant performance gains.

2.3 Attention Mechanisms and LLM Inference

Attention mechanisms have become the cornerstone of generative AI models, particularly in large language models (LLMs). First introduced in the Transformer architecture by Vaswani et al. [161], attention allows models to dynamically focus on relevant parts of the input

when producing each element of the output. This capability has proven crucial for handling long-range dependencies in sequential data.

At its core, attention computes weighted sums of value vectors, where the weights are derived from the compatibility between query and key vectors. Formally, for a query vector $\mathbf{q}$, key vectors $\{\mathbf{k}_i\}$, and value vectors $\{\mathbf{v}_i\}$, the attention output is computed as:

$$\text{Attention}(\mathbf{q}, \mathbf{K}, \mathbf{V}) = \sum_i \frac{\exp(\mathbf{q} \cdot \mathbf{k}_i / \sqrt{d_k})}{\sum_j \exp(\mathbf{q} \cdot \mathbf{k}_j / \sqrt{d_k})} \mathbf{v}_i \tag{2.1}$$

where d_k is the dimension of the key vectors. Multi-head attention extends this by projecting queries, keys, and values into multiple representation subspaces and applying attention in parallel, allowing the model to jointly attend to information from different perspectives.

While attention is powerful, it presents significant computational challenges, particularly for LLM inference. The computational complexity of naive attention implementation scales quadratically with sequence length, making it prohibitively expensive for long contexts. Recent innovations have addressed these challenges through algorithms like FlashAttention [39, 40, 141], which reduces memory bandwidth requirements by avoiding materialization of the full attention matrix. Nevertheless, LLM inference introduces several specific challenges:

- **KV-Cache Management**: During autoregressive generation, previously computed key and value vectors must be cached and reused, necessitating efficient memory management strategies such as paged attention and radix trees.

- **Input Dynamism**: LLM serving encompasses diverse computation patterns, ranging from full attention computation for context processing (prefill) to incremental attention during token generation (decoding). Recent innovations such as speculative decoding and prefix caching introduce a new phase called append, which computes cross-attention between new tokens and the existing context in the KV-Cache. Each phase presents distinct computational characteristics requiring tailored optimization strategies. Fur-

thermore, sequence lengths typically vary across requests in a batch, necessitating dynamic scheduling to achieve optimal hardware utilization.

- **Attention Variants and Binary Size Explosion**: The proliferation of attention variants has led to an explosion in CUDA kernel implementations and binary size. Organizations seeking to accelerate proprietary models with custom attention mechanisms often find no suitable library support, as their specific variants remain unimplemented. Current tensor compilers fail to bridge this gap, unable to meet the performance requirements of these specialized attention mechanisms.

FlashInfer [182], introduced in this dissertation, tackles these challenges through a principled abstraction approach similar to how BLAS provides for GEMM operations. It offers a unified template with specialized implementations that addresses the complex landscape of attention computation. Specifically, FlashInfer:

- **Flexible KV-Cache Representation**: Introduces a block-sparse KV abstraction that efficiently handles various storage formats and sparsity patterns.

- **Just-in-Time Optimization**: Dynamically generates optimized CUDA kernels at runtime to adapt to specific workload characteristics.

- **Intelligent Workload Distribution**: Schedules computation tiles across streaming multiprocessors to minimize tail latency and maximize hardware utilization.

This approach enables FlashInfer to seamlessly support diverse attention variants, dynamic sequence lengths, and evolving KV-cache layouts while maintaining high performance. By bridging the gap between algorithmic innovations and hardware capabilities, FlashInfer significantly improves the efficiency of attention computation in LLM inference systems.

Chapters 3 and 4 present the design and evaluation of SparseTIR and FlashInfer. The material in these chapters builds upon my prior work: SparseTIR [183], published at ASPLOS 2023, and FlashInfer [182], published at MLSys 2025.

Chapter 3

SPARSETIR

Sparsity is becoming ubiquitous in deep learning due to the application of deep learning to graphs and the need for more efficient backbone models. Graph neural networks (GNNs) [59, 78, 163] have made substantial progress in modeling relations in social networks, proteins, point clouds, etc., using highly sparse matrices. Sparse transformers [10, 23, 32] reduce both the time and space complexity of transformers [162] by making the attention mask sparse using manually designed and moderately sparse matrices. Network Pruning [60, 83, 135] prunes the network weight to sparse matrix to reduce model size, the pruned weights are moderately sparse and stored in various formats depending on the pruning algorithm.

Existing vendor libraries, such as cuSPARSE [37], dgSPARSE [44], Sputnik [54] and Intel MKL [166], support only a few sparse operators. As such, they fail to accelerate rapidly evolving emerging workloads such as GNNs on heterogeneous graphs [70, 137, 168] and hypergraphs [52]. Manually optimizing sparse operators can be difficult and tedious. Sparse matrices are stored in compressed formats, and programmers must write manual code to compress or decompress coordinates to access non-zero elements. Furthermore, the compressed sparse formats vary, and operators designed for one format cannot generalize to others. Therefore, *we need a more scalable and efficient approach to developing optimized sparse operators.*

Sparse tensor compilers, such as MT1 [13] and TACO [80], greatly simplify the development of sparse operators by decoupling format specification and format-agnostic computation descriptions. However, applying sparse compilation to deep learning must overcome two major challenges. First, *modern deep learning workloads are quite diverse,* making them hard to fit into a single sparse format pattern provided by existing solutions. Second, *hardware*

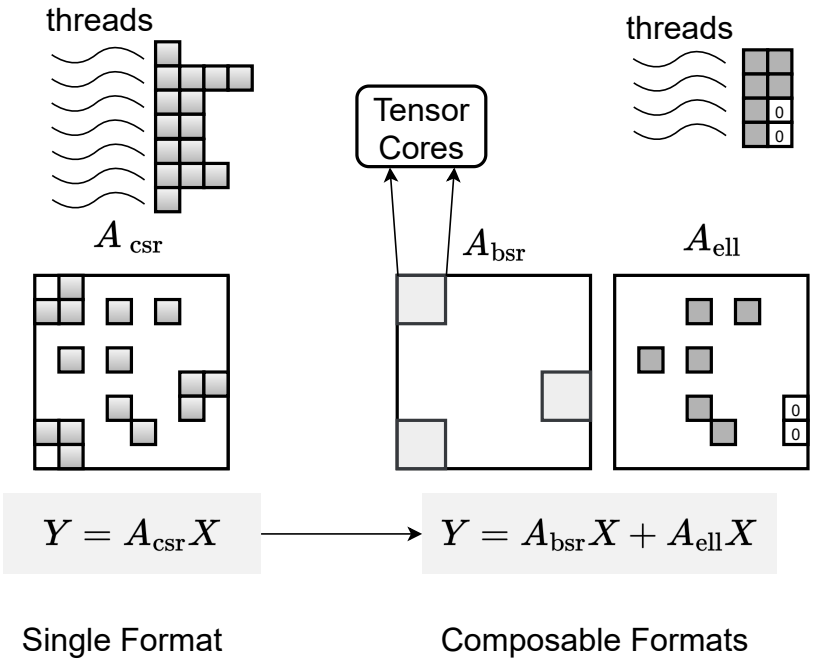

Figure 3.1: Format composability enables us to leverage multiple formats for different parts in sparse pattern we face in deep learning, and maximize the use of underlying hardware resources.

backends are evolving and becoming heterogeneous, making it hard for single-shot compilers to keep up with the latest hardware and system advances.

Our key observation is that we can resolve all challenges by introducing two forms of composability:

Format composability. We propose to go beyond the single format option provided by most existing solutions to composable formats (Figure 3.1) that store different parts of a sparse matrix in the different formats that best fit their local patterns. The compilation process decomposes the original computations into sub-computation routines to enable efficient executions on each local pattern that better match the characteristics of the corresponding deep learning workloads.

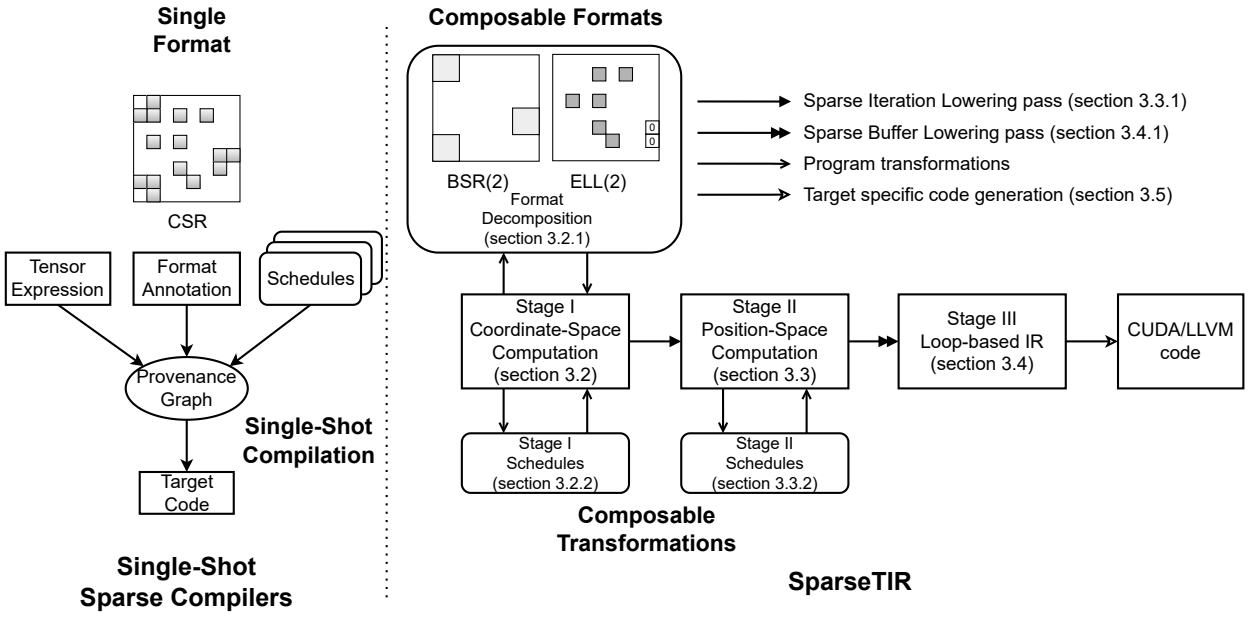

Figure 3.2: Single-shot sparse compilers vs SparseTIR. The composable formats and composable transformations enable us to create optimizations that fit into broader range of deep learning workloads and leverage more advances in hardware backends.

Transformation composability. We reconfigure the single-shot sparse tensor program compilation process into a composable set of program transformations. Additionally, we enable a design that incorporates existing loop-level abstractions in dense tensor compilers. This design lets us define our own transformations for sparse data while reusing hardware-specific optimizations (such as tensorization and GPU mapping) from existing solutions, increasing our overall efficiency to incorporate advances in hardware backends.

Combining both forms of composability, we propose SparseTIR, an abstraction that generates efficient sparse operators for deep learning. Our contributions include the following.

- We propose an intermediate representation (IR) with *composable formats* and *composable transformations* to accelerate sparse operators by decomposing formats and specifying schedules.

- We build a performance-tuning system that searches over the parameter space of possible composable formats and composable transformations.

- We evaluate SparseTIR generated kernels on several important sparse deep learning workloads.

SparseTIR offers consistent speedup for single operators relative to vendor libraries on GPUs: 1.20-2.34x for GNN operators and 1.05-2.98x for sparse transformer operators. SparseTIR also accelerates end-to-end GNNs by 1.08-1.52x for GraphSAGE [59] training and by 4.20-40.18x for RGCN [137] inference, 0.56-7.45x for Sparse Convolution [35] operators.

3.1 System Overview

This section provides an overview of SparseTIR. Figure 3.2 summarizes our overall design and compares it with existing approaches. The figure's left side shows the design of most existing sparse tensor compilers [139]. Their inputs are (1) tensor expressions, (2) format annotations/specifications that allow only a single format for each matrix, and (3) user-defined schedules. Schedules are applied to high-level IRs such as provenance graph, and then lowered to target device code; we refer to such compilation flow as *single-shot compilation*. These high-level IRs do not reflect low-level information such as loop structures, memory access regions, and branches. However, optimizations such as tensorization[1] requires loop-level AST matching and replacement, which is not exposed in high-level IR. Though tensor compilers such as Halide [125] and TVM [25] implement schedule primitives and code generation on multiple backends, it is difficult to re-use these infrastructures in previous sparse compilers because of the discrepancy of provenance graph and loop-level IR of existing tensor compilers.

SparseTIR builds on top of these previous approaches and introduces a design that enables composable formats and composable transformations. It contains three IR stages. The first stage presents computation in coordinate space, where we describe sparse tensor computations;

[1]We use this term to describe rewriting the program to use Matrix-Multiply Units such as Tensor Cores in GPU and MXU in TPU.

like in previous work, we decouple format specification and computations. Unlike a single-shot sparse compiler that accepts a single format for each sparse tensor, SparseTIR lets users specify composable formats. The second stage characterizes computation in position space, where the position refers to the index of non-zero elements in the compressed sparse data structure. The concepts of "coordinates" and "positions" were first proposed in Vivienne et al. [151] and then used in Senanayake et al. [139]. The last stage of SparseTIR is a loop-level IR in existing tensor compilers, such as TVM [25], AKG [188] and the affine dialect in MLIR [160]. We design two passes on the IR, namely, sparse iteration lowering and sparse buffer lowering, to transform code from stage I to stage II and stage II to stage III, respectively.

Instead of single-shot compilation, all schedules in SparseTIR are performed as composable program transformations (which do not change the stage of the IR) on the IR instantly. The composable design lets user transform the IR step-by-step and stage-by-stage. To manipulate the coordinate space computation in stage I IR, we can define new schedules as composable transformations applied to the stage I (i.e., stage I schedules). For stages compatible with target loop-level IR, we can apply schedules defined for backend tensor compilers (i.e., stage II/III schedules). Notably, format decomposition can also be formulated as a program transformation at stage I (see §3.2.2).

SparseTIR constructs a joint search space of composable formats and composable transformations for performance tuning of sparse operators. Users can customize the parameterized search space by specifying format and schedule templates based on their domain-specific knowledge about the operator and sparse tensor characteristics. When the sparse structure is present at compile-time, we can search for the best formats and schedules that achieve optimal runtime performance in advance. Though the compilation might take some time due to the large search space, the overhead can be amortized because the compiled operator will be re-used many times during training or inference for a fixed sparse structure (as is typical in deep learning).

The rest of the paper is organized as follows. We introduce the SparseTIR design of each stage and compiler passes in Section 3.2. In Section 3.3 we evaluate our system in real world

sparse deep learning workloads. Section 3.4 positions SparseTIR relative to related work. Finally, we discuss future work in Section 3.5 and conclude our work in Section 3.6.

3.2 Our Approach

In this section, we introduce the language constructs in SparseTIR, then describe each compilation stage and transformations in the order they appeared in the flow.

3.2.1 Language Constructs

Axis declarations

```
I = dense_fixed(m, "int32")
J = sparse_variable(I, (n, nnz),\
      (j_indptr, j_indices), "int32")
J_ = dense_fixed(n, "int32")
K = dense_fixed(feat_size, "int32")
```

Sparse buffer declarations

```
A = match_sparse_buffer(a, (I, J), "float32")
B = match_sparse_buffer(b, (J_, K), "float32")
C = match_sparse_buffer(c, (I, K), "float32")
```

Sparse iteration declarations

```
with sp_iter([I, J, K], "SRS", "spmm") as [i, j, k]:
    with init():
        C[i, k] = 0.0
    C[i, k] = C[i, k] + A[i, j] * B[j, k]
```

Figure 3.3: Language constructs in the SpMM operator. Users specify axis dependencies and metadata to create axes. The match_sparse_buffer defines sparse buffers and binds them to pointers to their value, and sp_iter creates a sparse iteration structure, where "S" and "R" indicate whether the iterator is for spatial or reduction purposes, "spmm" is the name of the sparse iteration as a reference for scheduling.

The SparseTIR language has three major components: axes, sparse buffers and sparse iterations.

Axes. An *axis* is a data structure that defines sparse iteration spaces, which generalize the idea of abstraction levels in previous work [33]. Each axis in SparseTIR has two orthogonal attributes, dense/sparse and fixed/variable, denoting whether the index of non-zero elements in the axis is contiguous or not and whether the number of non-zero elements in the axis is fixed or not. Variable axes are associated with a indptr (short for "index pointer") field that points to the address of the indices pointer array; sparse axes are associated with an indices field that points to the address of the indices array. Each axis has a parent field that directs to the axis it depends on; a dense-fixed axis has no dependency, and its parent field is always set to none. Axis metadata includes its indices' data type, maximum length, number of accumulated non-zeros in this dimension (if variable), and number of non-zeros per row in this dimension (if fixed).

Sparse buffers. A sparse buffer is SparseTIR's data structure for a sparse matrix. We use defined axes to compose the format specification of sparse matrices. We split sparse structure-related auxiliary data and values: axes store auxiliary data, and sparse buffers store only values. Such design lets Two sparse buffers can re-use auxiliary data if they share the sparse layout. Figure 3.4 shows the decoupled storage of sparse buffers/axes in the SpMM (Sparse-Dense Matrix Multiplication) operator. The composition of axes is expressive to describe various sparse formats, including Compressed Sparse Row/Column (CSR/CSC) format [48], Block Compressed Sparse Row (BSR) format [133], Diagonal Format (DIA) [132], ELLPACK (ELL) format [47, 66, 112], Ragged Tensor [155], Compressed Sparse Fiber (CSF) [145] etc, please refer to Duff et al. [48] for an overview of sparse formats.

Sparse iterations. Sparse iterations generates iterators over the space composed of a sparse axes array and a body containing statements describing tensor computations and orchestrating data movements. Notably, unlike TACO [80] which only allows the iterator variable to be used as an index to access sparse data structures (e.g. A[i, j] where i and j are iterator variables), SparseTIR supports any expressions, including affine indices (e.g.

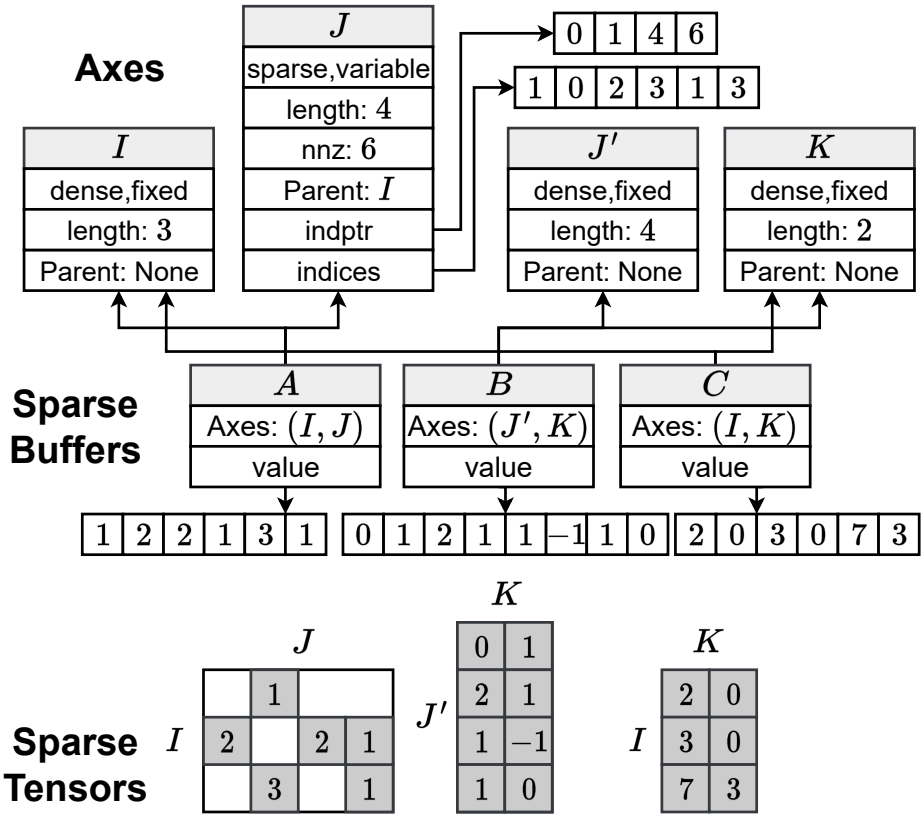

Figure 3.4: Internal storage of axes and sparse buffers in SpMM: $C_{ik} = A_{ij}B_{jk}$. Sparse buffers store their axes' composition and pointers to their value; axes store dense/sparse and fixed/variable attributes, metadata, their dependent axes, and pointers to indices and indptr arrays.

A[i $*$ m + j, k]) and integer values loaded from another buffer (e.g. B[eid[i], j $*$ n + k]). This enhances the capabilities of the SparseTIR, allowing for more complex operations such as convolution. SparseTIR enables multiple sparse iterations within a single program and even allows for nested sparse iterations within the body of another iteration, enabling branching and decomposing computation.

Figure 3.3 shows how to define these constructs in SparseTIR for the SpMM operator.[2] In

[2]The SparseTIR has round-trip compatibility with Python, and this paper presents only its Python form.

SparseTIR, axes are used to construct both sparse buffers and sparse iterations. This design lets us iterate over a sparse iteration space that is not bound to any sparse buffers.

3.2.2 Stage I: Coordinate Space Computation

In stage I SparseTIR defines sparse computations inside sparse iterations, where we iterate over non-zero elements and access sparse buffers in the coordinate space. At this stage, we can define program transformations, such as format decomposition and sparse iteration fusion, that manipulate only the three constructs of the SparseTIR .

Format Decomposition

Format decomposition is a transformation that decomposes computations for composable formats (introduced in Section 3). The transformation accepts a list of format descriptions and rewrites the IR according to these formats. Figure 3.5 shows the generated IR for the Sparse Matrix-Matrix multiplication (SpMM) operation after decomposing the computation in the CSR format to a computation in the BSR format, with block size 2 and an ELL format with 2 non-zero columns per row. In addition to SpMM computations on the new formats, another two sparse iterations that copy data from original to new formats are generated, as well. When the sparse matrix to decompose is stationary, we can perform data copying at pre-processing step to avoid the overhead of run-time format conversion.

The information used to create new sparse buffers: indptr_bsr, indices_bsr and indices_ell need to be pre-computed and specified by user as input arguments. Each format decomposition rule in SparseTIR needs to be registered as a function $F : (x, i) \to (x', i')$, where x, i refers to original SparseTIR program and indices/index pointer information, and x', i' are transformed ones. Figure 3.5 describes the IR transformation from x to x', and the conversion between i to i' need to be implemented by user manually. We have wrapped all format decomposition rules used in this paper as standard APIs, for new composable formats, user can use existing sparse libraries such as Scipy [164] to ease the implementation of indices

inference. SparseTIR leaves the flexibility of integrating with existing systems such as Chou et al. [34] for automatic indices inference.

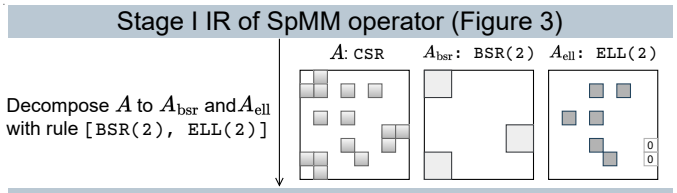

Figure 3.5: Format decomposition for SpMM Stage I IR in Figure 3.3. New axes and sparse buffers are created for decomposed formats BSR and ELL. New sparse iterations are generated to copy data from original to new formats and for computations on these new formats.

Stage I Schedules

We define two schedule primitives at stage I, sparse_reorder and sparse_fuse.

Sparse reorder. The order of sparse axes in the sparse iteration influences the order of generated loops in stage II. This primitive enables manipulation of the order of sparse axes.

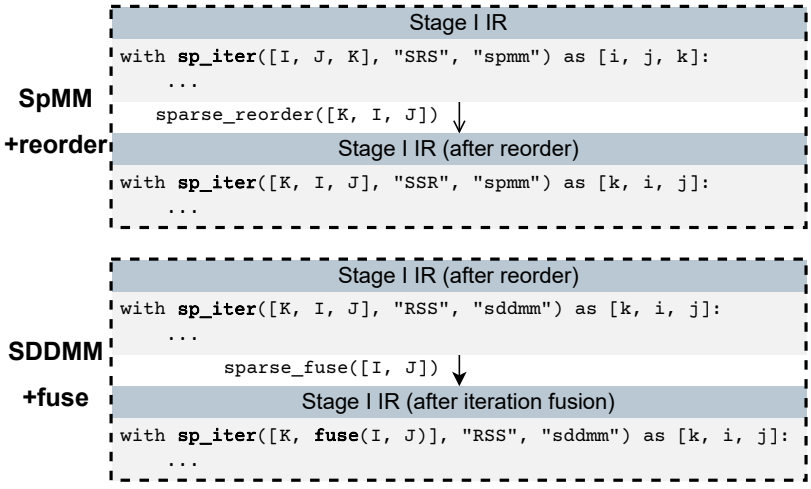

Figure 3.6: Stage I schedules sequentially applied to stage I IR.

Sparse fuse. This schedule primitive fuses several iterators in a given sparse iteration into one. It is helpful when we want a single loop rather two nested loops that iterate over all non-zero elements, such as in the SDDMM [105].

Figure 3.6 shows how stage I schedules transform the IR.

3.2.3 Stage II: Position Space Computation

The State II IR in SparseTIR introduces loop structures and removes the sparse iteration constructs and restructuring them as nested loops. Unlike in stage I where we access sparse buffers in *coordinate space*, in stage II access sparse buffers in *position space*, with the "position" referring to an element's non-zero index. The difference between coordinate and position applies to "sparse" dimensions: if the coordinate of the first 4 non-zero elements in a sparse row A is $\{1, 3, 9, 10\}$, the position of 9 is 2 (assuming the index is 0-based), and we use $A[9]$ to access the element in coordinate space and $A[2]$ to access the element in position space. Our stage II IR extends TensorIR [51] and treats sparse buffer as first-class citizens.

Sparse Iteration Lowering.

This pass transforms stage I IR to stage II IR. It consists of the following 4 steps.

Step 1: Auxiliary buffer materialization. Pointers to the indices pointer array and indices array are specified as arguments when creating axes. In stage II we need to declare these auxiliary buffers explicitly to access their value when determining loop range and translating coordinates. Figure 3.7 shows how the materialization works. In addition to auxiliary buffers, we also create hints that indicate the domain of buffer values; these are used for integer set analysis in stage II when performing schedules.

Stage I IR

```
I = dense_fixed(m, "int32")
J = sparse_variable(I, (n, nnz), (j_indptr, j_indices), "int32")
```

↓

Stage II IR

```
I = dense_fixed(m, "int32")
J = sparse_variable(I, (n, nnz), (j_indptr, j_indices), "int32")
J_dense = dense_variable(I, (n, nnz), j_indptr, "int32")
J_indptr = match_sparse_buffer(j_indptr, (I,), "int32")
J_indices = match_sparse_buffer(j_indices, (I, J_dense), "int32")
assume_buffer_domain(J_indptr, [0, nnz])
assume_buffer_domain(J_indices, [0, n1])
```

Figure 3.7: Example of auxiliary buffer materialization. Sparse buffers storing auxiliary information are created.

Step 2: Nested loop generation. This step restructures sparse iterations in stage I as nested loops in stage II: we emit one loop per axis in the sparse iteration. The generated loops start from 0, and the extent is determined by whether the axis is fixed or variable. They are separated by TensorIR's block constructs, which establish boundaries to prevent cross-block loop reordering. Additionally, We add a block inside the innermost generated loop and place the body of original sparse iterations inside of it. Figure 3.8 shows the emitted nested loop structures of different sparse iterations. In the first case, the loops I and J cannot

not be reordered in stage II because they are separated by a block; in the second case, we fuse I, J and emit only one loop (ij).

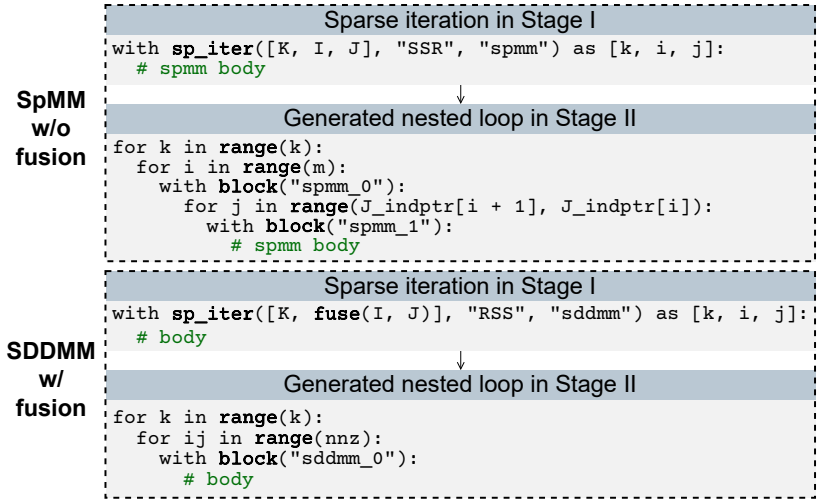

Figure 3.8: Nested loop generation in sparse iteration lowering. Without fusion, we emit one loop per axis in the sparse iteration; With fusion of i and j, we only emit one loop ij over the fused iteration space.

Step 3: Coordinate translation. This step rewrites the indices used to access sparse buffers from coordinate space to non-zero position space to bridge the semantic gap between stages I and II. See Figure 3.9 for an example. Suppose $\{\mathbf{A}_i^{(\text{iter})}\}_{i=1}^{M}$ is the array of axes used in sparse iterations, $\{\mathbf{v}_i^{(c)}\}_{i=1}^{M}$ is the array of iterator variables in coordinate space (before translation) and $\{\mathbf{v}_i^{(p)}\}_{i=1}^{M}$ is the array of loop variables in position space (after translation). For a sparse buffer access to be translated, suppose the buffer is composed of axes $\{\mathbf{A}_j^{(\text{buffer})}\}_{j=1}^{N}$, and the indices can be viewed as an array of functions $\{\mathbf{I}_j^{(\text{coord})}\}_{j=1}^{N}$ that maps iterator varibles to indices (for buffer access $B[x+y, z]$ within the sparse iteration where $\mathbf{v}^{(c)} = \{x, y, z\}$, its indices functions $\mathbf{I}^{(\text{coord})}$ should be $\{(x, y, z) \mapsto x + y, (x, y, z) \mapsto z\}$). The coordinate translation can be formulated as an iterative algorithm:

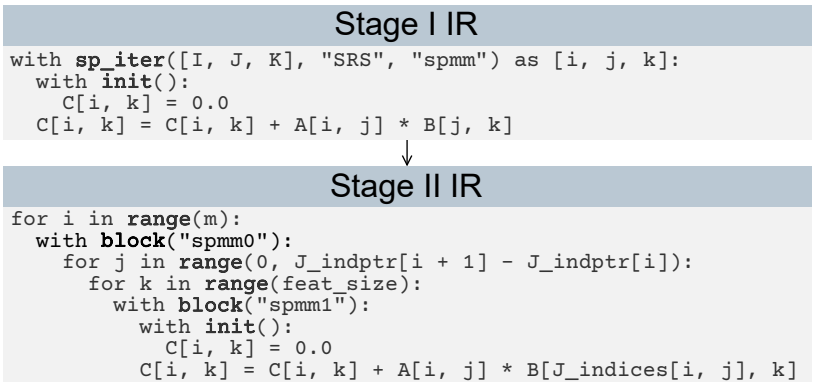

Figure 3.9: Translation from coordinate space to position space for SpMM operator.

$$\mathbf{p}_j \triangleq f^{-1}(\mathbf{A}^{(\text{buffer})}, j, \{\mathbf{p}\}_1^{j-1}, \mathbf{I}^{(\text{coord})}(\mathbf{c}_1, \ldots, \mathbf{c}_M)) \tag{3.1}$$

where $\mathbf{c}$ refers the coordinate array corresponding to $\mathbf{v}^{(p)}$ after translation from position space, f and $f^{(-1)}$ are decompress (position to coordinate) and compress (coordinate to position) functions:

$$\mathbf{c}_i \triangleq f(\mathbf{A}^{(\text{iter})}, \{\mathbf{c}\}_1^{i-1}, \mathbf{v}_i^{(p)}) \tag{3.2}$$

$$f(\mathbf{A}, i, \mathbf{c}, x) \triangleq \begin{cases} x & \mathbf{A}_i\text{: D(ense)} \\ \mathbf{A}_i\_\text{indices}[\mathbf{c}[\text{anc}(\mathbf{A}, i)], x] & \mathbf{A}_i\text{: S(parse)} \end{cases} \tag{3.3}$$

$$f^{(-1)}(\mathbf{A}, j, \mathbf{p}, x) \triangleq \begin{cases} x & \mathbf{A}_j\text{: D} \\ \text{find}(\mathbf{A}_j\_\text{indices}[\mathbf{p}[\text{anc}(\mathbf{A}, j)], :], x) & \mathbf{A}_j\text{: S} \end{cases} \tag{3.4}$$

The "find" function in the later case of equation 3.4 refers to searching a given value in sorted array, SparseTIR emits a binary search block to search for the index of x in sorted indices array. The "anc" function collects the indices of ancestor(including self) axes of $\mathbf{A}_i$ from its root in axes dependency tree, and $\mathbf{p}[\text{anc}(\mathbf{A}, j)]$ gathers values from $\mathbf{p}$ by ancestors' indices:

$$\text{anc}(\mathbf{A}, i) \triangleq \begin{cases} [i] & \mathbf{A}_i \text{ is root} \\ [\text{anc}(\mathbf{A}, j) : i] & \mathbf{A}_j = \text{parent}(\mathbf{A}_i) \end{cases} \tag{3.5}$$

Read/Write Region Analysis The buffer read/write region information is necessary for TensorIR's block construct. We perform a buffer region analysis pass to collect buffer access information and takes the union of all read/write regions accessed inside each block and annotate them as block attributes.

Stage II Schedules

The stage II schedules are responsible for manipulating loops (fuse/reorder/split), moving data across the memory hierarchy (cache_read/cache_write), binding loops to physical/logical threads to parallelize them, and using vector/tensor instructions in hardware (vectorize/tensorize). As a dialect of TensorIR, we fully support TVM schedule primitives [3] at stage II.

3.2.4 Stage III: Loop-Level IR

Stage III removes all SparseTIR constructs. It keeps only the nested loop structures whose body includes statements that operate on flattened buffers. This stage should be compatible with loop-level IR in existing tensor compilers. We select TensorIR [51] in Apache TVM [25] as stage III IR to make efficient use of NVIDIA's Tensor Cores, as it fully supports tensorization.

Sparse Buffer Lowering

Sparse buffer lowering removes all axes, flattens all multi-dimensional sparse buffers to 1-dimension, and rewrites memory access to these buffers. Suppose the original sparse buffer A

[3] `https://tvm.apache.org/docs/reference/api/python/tir.html#tvm.tir.Schedule`

is composed of axes $\{\mathbf{A}_i\}_{i=1}^{n}$. For memory access $A[x_1, ..., x_n]$, the overall offset after flattening is computed by:

$$\sum_{i=1}^{n} \text{is\_leaf}(\mathbf{A}_i) \times \text{offset}(i) \times \text{stride}(i+1), \tag{3.6}$$

where $\text{is\_leaf}(\mathbf{A}_i)$ means that if axis $\mathbf{A}_i$ has no dependence in $\{\mathbf{A}_j\}_{j=i+1}^{n}$, offset and stride are defined as:

$$\text{offset}(i) \triangleq \begin{cases} x_i & \text{is\_root}(\mathbf{A}_i) \\ \mathbf{A}_i\text{\_indptr}[\text{offset}(j)] + x_i & \mathbf{A}_j = \text{parent}(\mathbf{A}_i) \end{cases} \tag{3.7}$$

$$\text{stride}(i) \triangleq \begin{cases} 1 & i > n \\ \text{nnz}(\text{Tree}(\mathbf{A}_i)) \times \text{stride}(i+1) & \text{is\_root}(\mathbf{A}_i) \\ \text{stride}(i+1) & \text{otherwise,} \end{cases} \tag{3.8}$$

where $\text{nnz}(\text{Tree}(\mathbf{A}_i))$ refers to the number of non-zero elements of the sparse iteration space composed by the tree with $\mathbf{A}_i$ as its root. Figure 3.10 shows an example of sparse buffer lowering: sparse buffers A, B, C are flattened. The buffer access $A[i, j]$ is translated to $A[\text{J\_indptr}[i] + j]$ by equation 3.6.

Stage II IR

```
# loop structures
with block("spmm1"):
  with init():
    C[i, k] = 0.0
  C[i, k] = C[i, k] + A[i, j] * B[J_indices[i, j], k]
```

↓

Stage III IR

```
# loop structures
with block("spmm1"):
  with init():
    C[i * feat_size + k] = 0.0
  C[i * feat_size + k] = C[i * feat_size + k]
    A[J_indptr[i] + j] * B[J_indices[J_indptr[i] + j] * feat_size + k]
```

Figure 3.10: Sparse buffer lowering: sparse constructs are totally removed, and memory accesses are flattened to 1-dimension.

3.2.5 Target-Specific Code Generation

SparseTIR re-uses the backend provided by existing tensor compilers for target-specific code generation. SparseTIR emits multiple CUDA kernels for composable formats, which incur extra kernel-launching overhead on the GPU. We insert a horizontal fusion [50, 86] pass to the TVM backend to reduce this overhead.

3.2.6 Programming Interface for Composable Formats

This section further explains the programming interface for composable formats and the format decomposition pass introduced in §3.2.2, SparseTIR provide two APIs for composable formats:

FormatRewriteRule is a class for a sparse format rewriting rule description, its input include: the name of format rewrite rule, the sparse buffer to rewrite, a SparseTIR description of new format, the mapping from original axes to new axes, and the index mapping f and inverse index mapping f^{-1} between original sparse buffer A and the transformed sparse buffer A': $A[\mathbf{I}] = A'[f(\mathbf{I})], A[f^{-1}(\mathbf{I'})] = A'[\mathbf{I'}]$, both f and f^{-1} need to be affine maps written in Python's lambda functions.

decompose_format is a function that accepts a list of format rewrite rules and an SparseTIR program as input and performs the format decomposition pass on the given SparseTIR program.

Below is an example illustrating how to use the two APIs to compose ELL(2) and BSR(2) rewrite rules and perform format decomposition in Figure 3.5:

```
@T.prim_func
def spmm(
    a: T.handle, b: T.handle, c: T.handle,
    indptr: T.handle, indices: T.handle,
    m: T.int32, n: T.int32, nnz: T.int32, feat_size: T.int32
) -> None:
    I = T.dense_fixed(m, idtype="int32")
```

```
    J = T.sparse_variable(
        I, (n, nnz), (indptr, indices), idtype="int32")
    J_ = T.dense_fixed(n, idtype="int32")
    K = T.dense_fixed(feat_size, idtype="int32")
    A = T.match_sparse_buffer(a, (I, J), "float32")
    B = T.match_sparse_buffer(b, (J_, K), "float32")
    C = T.match_sparse_buffer(c, (I, K), "float32")
    with T.sp_iter([I, J, K], "SRS", "csrmm") as [i, j, k]:
        with T.init():
            C[i, k] = 0.0
        C[i, k] = C[i, k] + A[i, j] * B[j, k]


def BSR(block_size: int):
    # block_size: the block size in BSR format.
    @T.prim_func
    def bsr_desc(
        a: T.handle,
        indptr: T.handle, indices: T.handle,
        m: T.int32, n: T.int32, nnz: T.int32
    ) -> None:
        IO = T.dense_fixed(m, idtype="int32")
        JO = T.sparse_variable(
            IO, (n, nnz), (indptr, indices), idtype="int32")
        II = T.dense_fixed(block_size, idtype="int32")
        JI = T.dense_fixed(block_size, idtype="int32")
        A = T.match_sparse_buffer(a, (IO, JO, II, JI), "float32")
        pass

    return FormatRewriteRule(
        "bsr_{}".format(str(block_size)),
        bsr_desc,
        ["A"], ["I", "J"], ["IO", "JO", "II", "JI"],
        {"I": ["IO", "II"], "J": ["JO", "JI"]},
        lambda i, j:
            return (i // block_size, j // block_size,
                    i % block_size, j % block_size)
        lambda io, jo, ii, ji:
            return io * block_size + ii, jo * block_size + ji
    )

def ELL(nnz_cols: int):
```

```
    # nnz_cols: number of non-zero columns per row in ELL format.
    @T.prim_func
    def ell(
        a: T.handle,
        indices: T.handle,
        m: T.int32, n: T.int32,
    ) -> None:
        I2 = T.dense_fixed(m, idtype="int32")
        J2 = T.sparse_fixed(
            I2, (n, nnz_cols), indices, idtype="int32")
        A = T.match_sparse_buffer(a, (I2, J2), "float32")
        pass

    return FormatRewriteRule(
        "ell_".format(str(nnz_cols)),
        ell_desc,
        ["A"], ["I", "J"], ["I2", "J2"],
        {"I": ["I2"], "J": ["J2"]},
        lambda i, j: return i, j
        lambda i2, j2: return i2, j2
    )

composable_format = [BSR(2), ELL(2)]
spmm_hybrid = decompose_format(spmm, composable_format)
```

Listing 3.1: Format decomposition example

where the prefix T is used to prevent name conflicts with keywords in Python. Note that format conversion is a special case of format decomposition where we only put one FormatRewriteRule in the list of composable formats.

3.3 Evaluation

We now study how composable formats and composable transformations help optimize sparse deep learning workloads in both single-operator and end-to-end settings. In summary, compared to vendor libraries, SparseTIR obtains a 1.20-2.34x speedup on GNN operators and a 1.05-2.98x speedup on sparse attention operators. When used in an end-to-end setting, SparseTIR obtains a 1.08-1.52x speedup on end-to-end GraphSAGE training and a 4.20-40.18x speedup on end-to-end RGCN inference, 0.56-7.44x on Sparse Convolution operators.

3.3.1 Experiment Setup

Environment. We evaluate all experiments under two different GPU environments: NVIDIA RTX 3070 and NVIDIA Tesla V100.

Baselines. cuSPARSE [37] is NVIDIA's official library for sparse tensor algebra, which includes high-performance implementation of common sparse operators. dgSPARSE [44] is a collection of state-of-the-art sparse kernel implementations for GNNs, which includes GE-SpMM [71], DA-SpMM [38] and PRedS [185]. PyG [53] and DGL [167] are two open-source frameworks that support GNN training and inference. Sputnik [54] is a library for sparsity in Deep Learning. Neither dgSPARSE nor Sputnik uses Tensor Cores. TACO [80] is an open-source sparse tensor compiler. Triton [158] is a tiling-based IR for programming neural networks, and we use its block sparse operator implementation. TorchSparse [152] is a library for point cloud processing, with state-of-the-art sparse convolution implementation.

For SpMM, we select the TACO-generated operator, cuSPARSE 11.7, and dgSPARSE 0.1 as baselines. For SDDMM, we select the TACO-generated operator, cuSPARSE, dgSPARSE and DGL 0.9.1's implementation as baselines. The DGL's SDDMM implementation uses the optimizations proposed in FeatGraph [69]. For end-to-end GNN training, we compare a GraphSAGE model written in PyTorch 1.12 [117] that integrates a SparseTIR-tuned kernel with DGL. For RGCN, we select the Graphiler [177], DGL 0.9.1 and PyG 2.2.0 implementations as our baseline.[4] For sparse transformers, we select Triton[5]'s block-sparse kernel as our baseline. For sparse convolution, we select TorchSparse [6] for comparison. The computation results of all SparseTIR generated kernels have been compared with existing frameworks/libraries to confirm numerical accuracy.

[4]Both DGL and PyG provide several different official implementations of RGCN; we select the best performing among them.

[5]Main branch until commit 0e8590

[6]Main branch until commit 2caf084

Graph	#nodes	#edges	%padding
cora [138]	2,708	10,556	15.9
citeseer [138]	3,327	9,228	13.0
pubmed [138]	19,717	88,651	23.1
ppi [59]	44,906	1,271,274	22.9
ogbn-arxiv [67]	169,343	1,166,243	17.5
ogbn-proteins [67]	132,534	39,561,252	21.6
reddit [59]	232,965	114,615,892	28.6

Table 3.1: Statistics of Graphs used in GNN experiments.

3.3.2 Graph Neural Networks

In this section, we evaluate the performance of SparseTIR on GNN workloads. SpMM and SDDMM [105] are two of the most generic operators in GNNs. Table 3.1 describes the characteristics of graphs used in our evaluation; on the table, %padding refers to the ratio of padded zero elements after we transform the original sparse matrix to composable formats.

SpMM

SpMM is the most generic sparse operator in deep learning, which can be formulated as:

$$Y_{i,k} = \sum_{j=1}^{n} A_{i,j} X_{j,k},$$

where A is a sparse matrix and X, Y are dense matrices. A high-performing SpMM kernel on a GPU requires efficient memory access patterns and load balancing [179]. Runtime load balancing, well studied in SpMM acceleration literature, always incurs runtime overhead. The composable format and composable transformation can help generate kernels that achieve compile-time load balancing and better cache utilization.

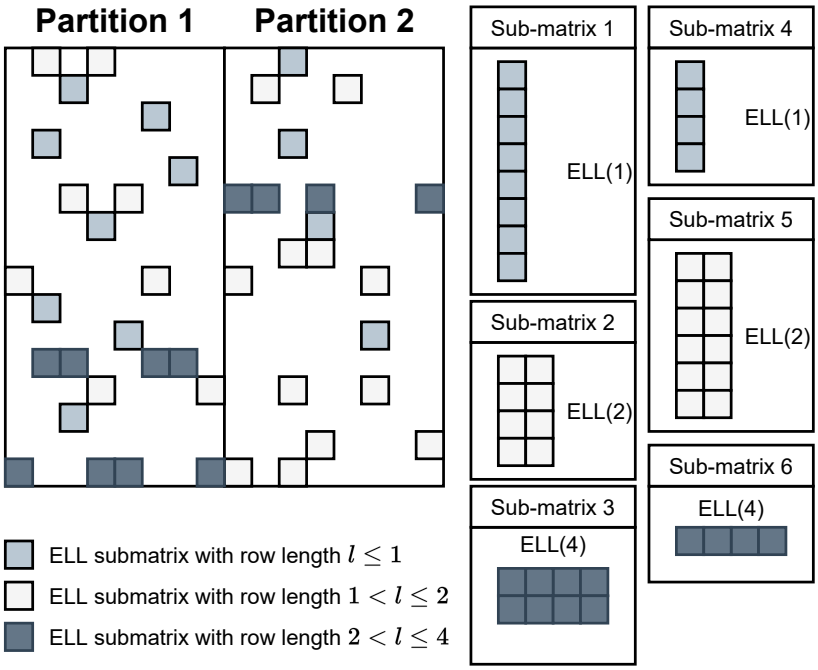

Figure 3.11: Example of $hyb(2, 2)$: the original matrix is decomposed to 6 ELLPACK sub-matrices; elements in partition 1 are stored in sub-matrix 1-3, and elements in partition 2 are stored in sub-matrices 4-6.

We design a parameterized composable format $hyb(c, k)$ for sparse matrix A with two parameters c and k. We partition columns of the sparse matrix by the given factor c, so that each column partition has width w. For each column partition, we collect the rows with length l that satisfy $2^{i-1} < l \leq 2^i$ to bucket i, and we pad the length of these rows to 2^i; each bucket then forms a sub-matrix with the ELL format. Figure 3.11 shows a special case, hyb$(2, 2)$.

For bucket i of each column partition, we group each 2^{k-i} rows and map them to a unique thread block in GPUs. The number of non-zero elements in A that are processed by each thread block is 2^k, which is implemented with TVM's split and bind primitives. We use the schedule proposed in GE-SpMM [71] for each sub-matrix for the remaining dimensions. The

column partition in our design is intended to improve cache locality; when processing column partition j, only $B[jw : (j+1)w]$ would be accessed for B. Featgraph [69] proposes to apply column partitions for SpMM on CPUs; however, it does not extend the idea to GPUs. Our bucketing technique was designed to achieve compile-time load balancing. In practice, we searches for the best c over $\{1, 2, 4, 8, 16\}$ and let $k = \lceil \log_2 \frac{nnz}{n} \rceil$, which generally works well.

We evaluate the SpMM written in SparseTIR with and without the proposed *hyb* format on real-world GNN datasets for both V100 and RTX3070. We measure the geometric mean speedup of different SpMM implementations against cuSPARSE for feature size $d \in \{32, 64, 128, 256, 512\}$. Figure 3.13 shows our results. The SparseTIR kernel on *hyb* format obtains a 1.22-2.34x speedup on V100 and a 1.20-1.91x speedup on RTX 3070 compared to cuSPARSE. We also achieve consistently better performance than state-of-the-art open source sparse libraries dgSPARSE and Sputnik, and TACO with auto-scheduling enabled [139]. Though TACO also explores compile-time load balancing, it does not support caching the partially aggregated result in registers, which is critical to kernel performance on GPUs, and the irregularity of the CSR format limits the application of loop unrolling. SparseTIR perform these optimizations in stage II schedules.

Importance of composable formats. We evaluate the SparseTIR kernel without format decomposition (see SparseTIR(no-hyb) in the figure). Results suggest that the SparseTIR kernel without format decomposition and per-format scheduling performs generally worse: ogbn-arxiv is a citation network graph whose degrees obey power-law distribution, and our designed format can perform significantly better because of more efficient load balancing. Notably, though padded zeros in our proposed composable format slightly increase FLOPs as shown in Table 3.1, the runtime of SparseTIR generated kernels on composable format is still faster because of better scheduling. The degree distribution of the ogbn-proteins graph is centralized, and the benefit of using a hybrid format is compensated for the extra overhead introduced by padding. To evaluate the effect of column partitioning, we fix the feature size to 128 and measure several kernel metrics generated by SparseTIR on a Reddit dataset under

a different column partition setting. Figure 3.12 shows the results; L1 and L2's cache hit rates improve as we increase the number of column partitions. However, more partitions will increase the required memory transactions of the kernel because we will need to update the results matrix c times if the number of partitions is c. As a result, the benefit of column partitioning saturates as we increase the number of partitions.

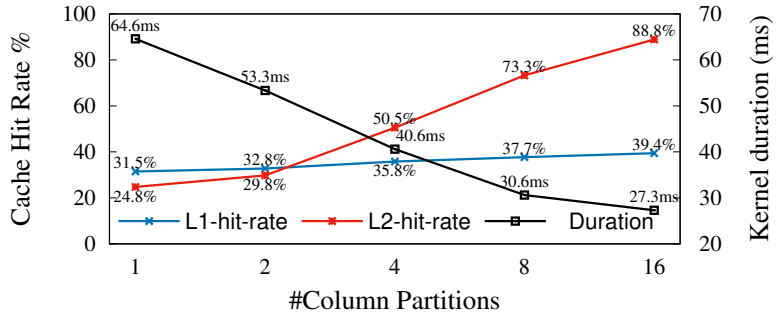

Figure 3.12: The kernel duration and L1/L2 hit-rate of SparseTIR SpMM kernels under different column partitions.

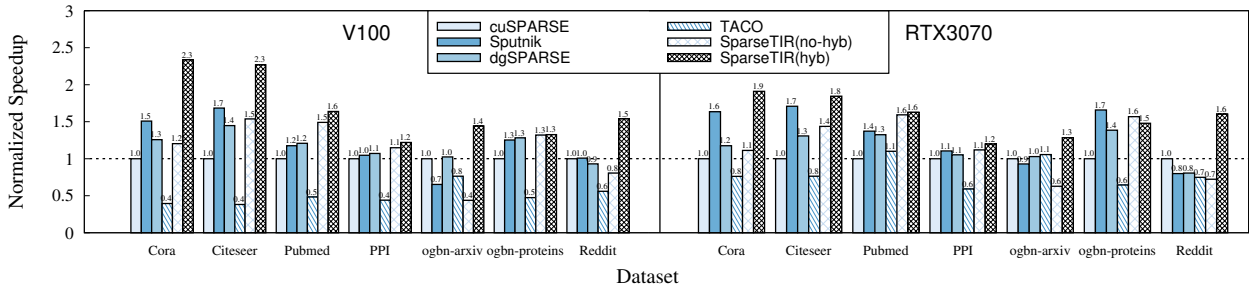

Figure 3.13: Normalized speedup against cuSPARSE for SpMM. SparseTIR consistently outperforms vendor libraries and TACO. Comparing SparseTIR(no-hyb) and SparseTIR(hyb) demonstrates the importance of format composability.

SDDMM

SDDMM can be formulated as the following:

$$B_{i,j} = \sum_{k=1}^{d} A_{i,j} X_{i,k} Y_{k,j},$$

where A and B are two sparse matrices that share a sparse structure, X, Y are dense matrices, and d is the feature size. In SDDMM, the computation per (i, j) is independent, and the workload per position is the same, so we need not worry about load balancing issues if we parallelize the computation by each non-zero (i, j). The sparse_fuse schedule primitive in stage I introduced in Section 3.2.2 helps us iterate over non-zero (i, j) directly instead of first iterating over i and then iterating over non-zero j for each i.

PRedS [185] is the state-of-the-art open-source SDDMM implementation, which optimizes SDDMM in two ways. First, it uses vectorized load/store intrinsics in CUDA, such as float4 / float2, which improves memory throughput. Second, it performs the reduction in two stages: (1) *intra-group reduction*, which computes the reduction inside each group independently, and (2) *inter-group reduction*, which summarizes the reduction result per group. We formulate the optimization in PRedS as composable transformations in SparseTIR with vectorize and rfactor [150] schedule primitives at stage II, and we generalize the parameters, such as group size, vector length and number of workloads per CTA, as tunable parameters.

Figure 3.14 shows the geometric mean speedup of different SDDMM implementations vs our baseline for feature size $d \in \{32, 64, 128, 256, 512\}$. We do not use composable formats in SDDMM. The baseline we select is DGL's SDDMM implementation, which uses the optimization proposed in Featgraph [69]. cuSPARSE and Sputnik's SDDMM implementations are not optimized for highly sparse matrices such as graphs and thus achieve very low performance. We obtain generally better performance than dgSPARSE [44], which implements the PRedS [185] algorithm, because of the parameterized scheduling space. SparseTIR significantly outperforms the DGL baseline and the TACO scheduled kernel

because these implementations do not include two-stage reduction and vectorized load/store.

Importance of composable transformations. The provenance graph data structure in TACO does not support multiple branches, thus we cannot perform schedules such as rfactor at this level. The composable transformation design of SparseTIR enables us to apply such schedules at lower stages.

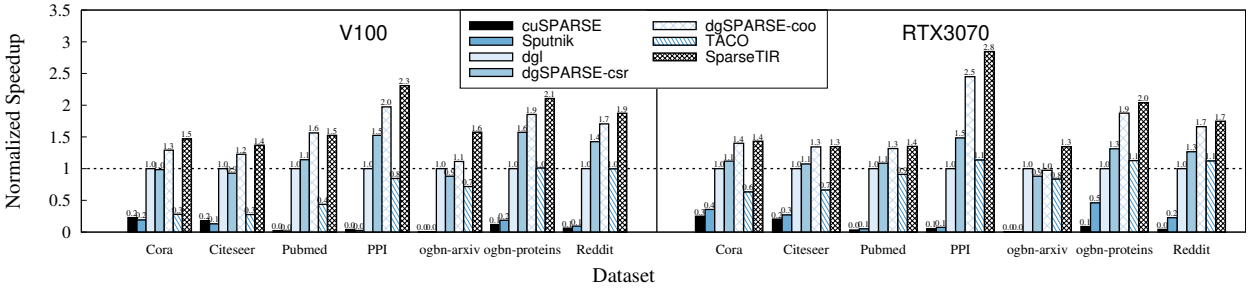

Figure 3.14: Normalized speedup against Featgraph for SDDMM. SparseTIR beats the state-of-the-art vendor library dgSPARSE on average by parametrizing scheduling space.

End-to-end GraphSAGE training.

We also integrate SparseTIR-generated SpMM operators in the GraphSAGE [59] model written in PyTorch and compare the end-to-end speedup to DGL. Figure 3.15 shows that we obtain a 1.18-1.52x speedup on V100 and a 1.08-1.47x speedup on RTX 3070 [7].

3.3.3 Sparsity in Transformers

Sparsity in Transformers comes from (1) sparse attentions [10, 23, 32], and (2) sparsity in network weights after pruning [83, 135]. We evaluate SparseTIR generated kernel in both cases[8].

[7] Reddit result is not reported on RTX 3070 because of Out-Of-Memory issue.

[8] In this section, we use half-precision data type for all operators to use Tensor Cores.

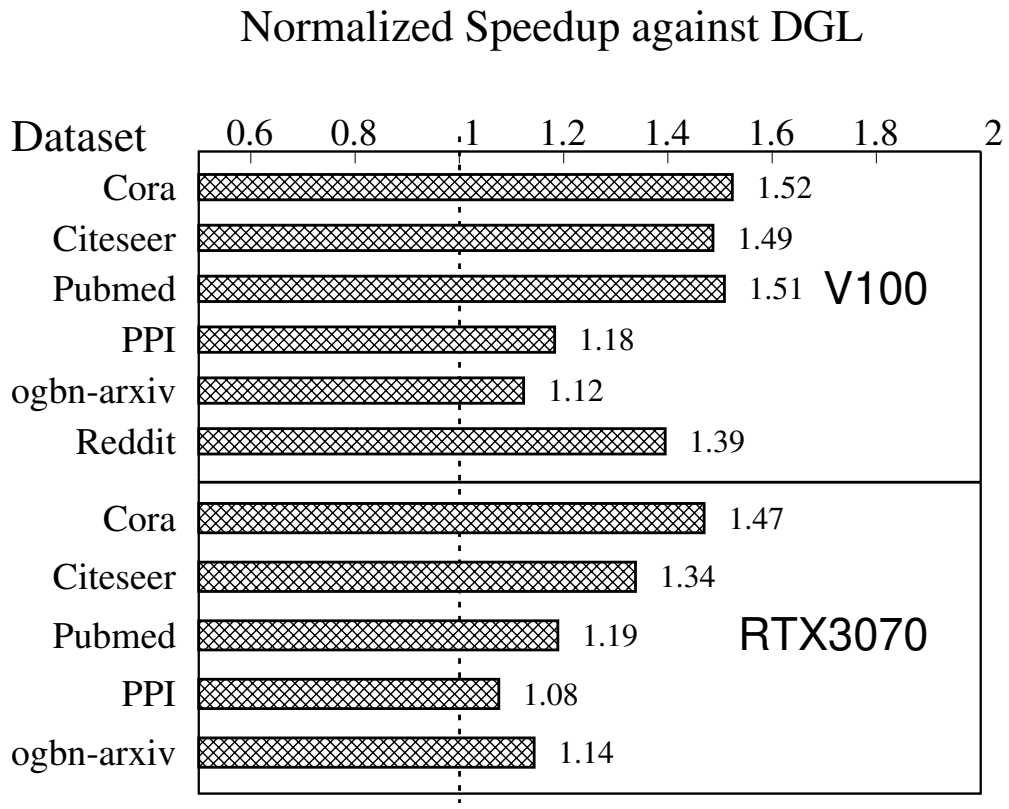

Figure 3.15: Normalized speedup of PyTorch+SparseTIR against DGL on end-to-end GraphSAGE training.

Sparse Attention.

Sparse transformers reduce the complexity of Transformers by making the attention matrix sparse. The key operator in Sparse Transformers is still SpMM and SDDMM, but unlike GNNs whose sparse matrices are provided by graph structures, the sparse matrices used in sparse attentions are mostly manually designed and have a block-sparse pattern to better utilize tensor cores in modern GPUs. We select two examples: Longformer [10] and Pixelated Butterfly Transformer [23], whose sparse structures are band matrix and butterfly matrix [116], respectively. We implement the batched-SpMM and batched-SDDMM operators for both CSR and BSR formats. For BSR operators, we use the tensorize primitive during

stage II IR schedules to use tensorized instructions in CUDA. Figure 3.16 shows different implementations' speedup against Triton's [158] block-sparse operator. We fix the matrix size to 4096×4096, batch(head) size to 12, band size to 256, and feature size per head to 64. Results show that SparseTIR-BSR obtains a 1.05-1.59x speedup on multi-head SpMM and a 1.50-2.98x speedup on multi-head SDDMM.

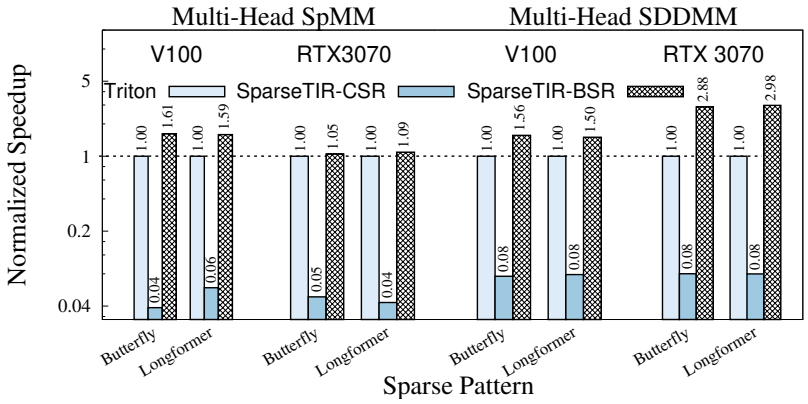

Figure 3.16: Normalized speedup against Triton on sparse transformer operators.

Sparse Weight (Network Pruning)

Network pruning [60] is another source of sparsity in Transformers. Pruning can significantly reduce the number of model parameters at the cost of negligible performance (accuracy) loss by making the weights sparse. PruneBERT [83, 135] applies pruning to Transformers, and we evaluate SparseTIR's performance on PruneBERT in both structured pruning and unstructured pruning settings.

Structured Pruning. Structured Pruning prunes groups of weights together at the channel or block level to speed up execution. Block pruning [83] is an example of structured pruning on Transformers where network weights are pruned to block-sparse format, the operator used in block-pruned Transformer is SpMM. We extract all SpMM operators in a block-pruned

model[9] with block size 32 and average weight sparsity 93% for the benchmark. We fix the batch size to 1 and the sequence length to 512. Figure 3.17 shows the performance of SparseTIR kernels, Triton's BSRMM, and cuBLAS on these operators. The block sparse weights in the block-pruned model have many all-zero rows, and we propose to use doubly-compressed BSR (DBSR, inspired by doubly compressed sparse row (DCSR) format [19]) format to skip zero rows. The results show that SparseTIR kernel on DBSR format can consistently outperform SparseTIR kernel on BSR format, and achieve better with Triton's BSRMM implementation.

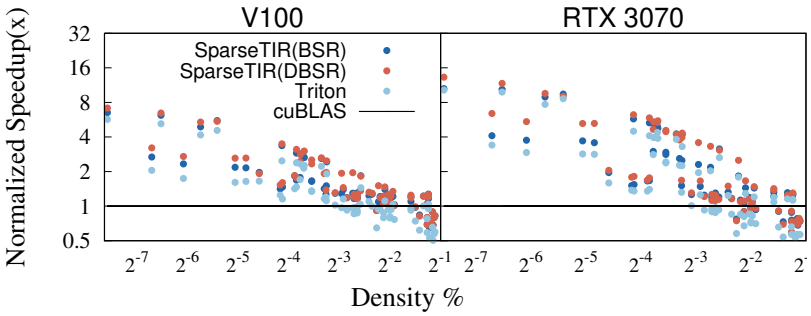

Figure 3.17: Normalized speedup against cuBLAS for operators extracted from block-pruned transformers. The X-axis refers to the weight density in the SpMM operator, and Y-axis refers to the normalized Speedup against cuBLAS implementation which uses a dense matrix for sparse weight.

Unstructured Pruning. Unstructured pruning does not pose any constraints on the format of pruned weights, and the pruned weight matrices are typically stored in CSR format. Unstructured pruned model is known to be hard to optimize because of irregular computation, and directly converting them to BSR format would introduce too much fragmentation inside blocks. We use the SR-BCRS format proposed in Magicube [88] to alleviate the issue. Figure 3.18 explains how to represent SR-BCRS(t, g) and corresponding SpMM schedules in

[9] `https://huggingface.co/madlag/bert-base-uncased-squad1.1-block-sparse-0.07-v1`

SparseTIR: the matrix is firstly divided into many $t \times 1$ tiles, and we omit tiles whose elements are all zero. The non-zero tiles inside the same rows are grouped by a factor of g, and we pad the tailing groups with zero tiles. Sparse matrices in SR-BCRS format can be composed by 4 axes in SparseTIR. When performing SpMM on SR-BCRS, we can load a group of tiles in A and corresponding rows in X to local registers and use Tensor Cores in GPU (or Matrix-Multiply Units(MXU) in TPU [75], equivalently) to compute their multiplication results, these schedules can be described as cache—read/write and tensorize primitives at stage-II in SparseTIR. Compared to BSR, the SR-BCRS format greatly reduces intra-block fragmentation: the non-zero ratio lower bound in SR-BCRS(t, g) is $1/t$, while BSR with block size b has a lower bound of $1/b^2$.

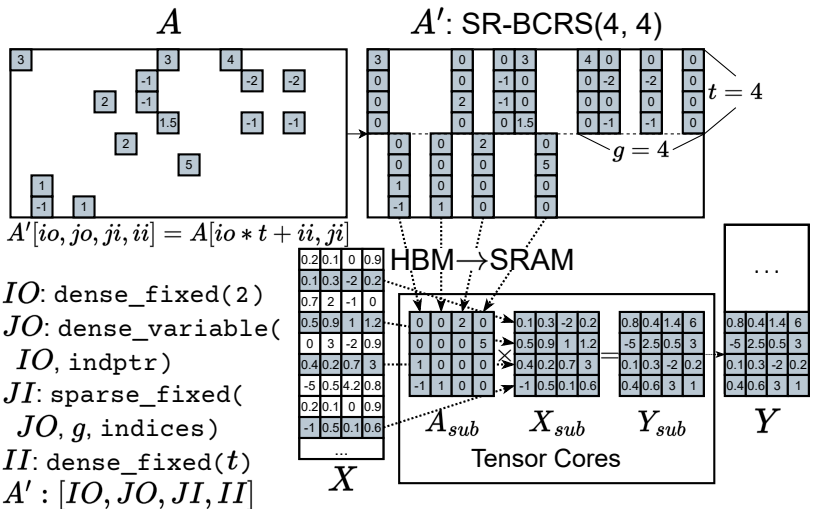

Figure 3.18: Conversion from unstructured sparse matrix to SR-BCRS(t, g), and SpMM schedule on the it.

We extract all SpMM operators in a movement-pruned model[10] with average weight sparsity of 94% for benchmark. Figure 3.19 shows the performance of SparseTIR on SR-

[10] `https://huggingface.co/huggingface/prunebert-base-uncased-6-finepruned-w-distil-squad`

BCRS$(8, 32)$[11], BSR format with block size 32, and vendor libraries cuBLAS and cuSPARSE's CSRMM. We set the batch size to 1 and the sequence length to 512. We do not compare with Triton because it has no native implementation of SpMM on SR-BCRS. SparseTIR on SR-BCRS beats SparseTIR on BSR in most of the settings except for density $\geq 2^{-3}$, in which case both transformed sparse matrices have a density close to 1. cuSPARSE's CSRMM can only beat cuBLAS' GeMM when weight density is extremely low (e.g., $\leq 2^{-6}$).

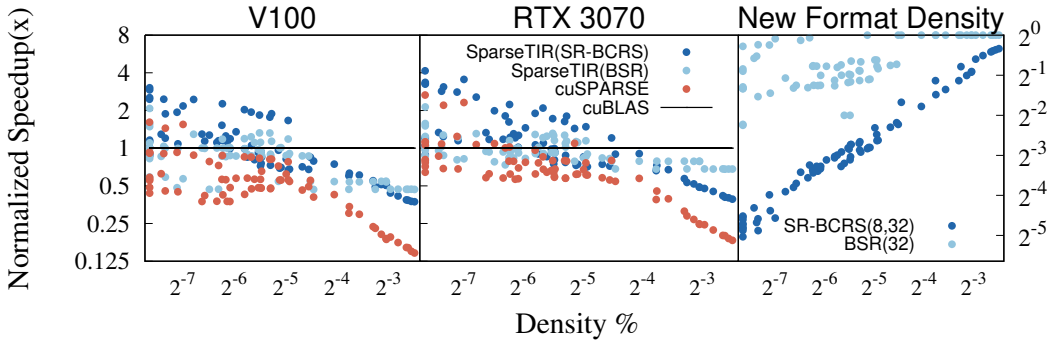

Figure 3.19: Normalized speedup aginst cuBLAS for operators extracted from unstructured pruned transformers, and the weight density in new format vs original weight.

3.3.4 Relational Gather-Matmul-Scatter

Relational Gather-Matmul-Scatter (RGMS for short) is an emerging sparse operator which can be expressed as follows:

$$Y_{i,l} = \sum_{r=1}^{R} \sum_{j=1}^{n} \sum_{k=1}^{d_{in}} A_{r,i,j} X_{j,k} W_{r,k,l},$$

where A is a 3D sparse matrix, whose leading dimension size is R, denoting number of relations. For each relation, the last two dimensions of A form a unique 2D sparse matrix. X is a 2D feature matrix and W is a 3D weight matrix whose leading dimension size is also

[11]To use m8n32k16 MMA instructions in GPU.

R. For each relation, the last two dimensions of W form a unique 2D dense weight matrix. The scheduling for the RGMS operator is complicated because we need to consider (1) load balancing and (2) the utilization of Tensor Cores. Until now, no sparse library implements this kernel.

Relational Graph Convolution Network.

RGCN [137] is a generalization of GCN model to heterogeneous graphs (graphs with multiple relations/edge types). The operator used in RGCN is RGMS, where A_r refers to the adjacency matrix corresponding to sub-graph whose edge type is r, and W_i refers to the weight matrix corresponding to edge type r. Table 3.2 introduces the characteristics of heterogeneous graphs used in RGCN evaluation; in the table, #etypes refers to the number of edge types (also known as "relations") in the heterogeneous graph, %padding refers to the ratio of padded zero elements after we transform the original sparse matrix with composable formats. Existing GNN libraries implement RGMS operator in a two-stage approach:

$$T_{r,j,l} = \sum_{k=1}^{d_{in}} X_{j,k} W_{r,k,l}, \tag{3.9}$$

$$Y_{i,l} = \sum_{r=1}^{R} \sum_{j=1}^{n} A_{r,i,j} T_{r,j,l}, \tag{3.10}$$

where the first stage fuses gathering and matrix multiplication, and the second stage performs scattering. Such implementation materializes the intermediate result T on HBM, which incurs a lot of GPU memory consumption. In SparseTIR we fuses the two stage into a single operator: we generalize the *hyb* format proposed in Figure 3.11 to 3-dimensional so that 2D sparse matrix corresponding to each relation is decomposed to $hyb(1, 5)$ formats. Figure 3.21 explain the scheduling of RGMS operator on 3D *hyb* in SparseTIR: for each ELL matrix A^{rk} (r refers to edge type and k refers to bucket index), we pin its corresponding weight matrix W^r in SRAM and gather related rows of X from HBM to SRAM, then perform partial matrix multiplication with Tensor Cores and scatter results to Y. Note that the matrix multiplication

Graph	#nodes	#edges	#etypes	%padding
AIFB [129]	7,262	48,810	45	17.9
MUTAG [129]	27,163	148,100	46	8.0
BGS [129]	94,806	672,884	96	4.3
ogbl-biokg [67]	93,773	4,762,678	51	4.2
AM [129]	1,885,136	5,668,682	96	10.8

Table 3.2: Statistics of Heterogeneous Graphs used in RGCN.

and intra-group scatter are all performed inside SRAM. Such design reduces the overhead of data copy between SRAM and HBM for intermediate matrix T. We evaluate end-to-end RGCN inference (feature size: 32) and Figure 3.20 shows results: SparseTIR(*hyb*+TC) can significantly improve previous state-of-the-art GNN compiler Graphiler [177] by 4.2-40.2x in different settings. By comparing SparseTIR(naive), SparseTIR(*hyb*) and SparseTIR(*hyb*+TC) we show that both composable formats and composable transformations (which enables Tensorization) matter: even though *hyb* increases FLOPs by padding zeros (as shown in Table 3.2), it still makes the kernel faster by 2-4.4x because of better load-balancing. SparseTIR's generated fused kernel can also greatly reduce GPU memory footprint because we do not explicitly stores T in HBM, with fragments of T consumed immediately after produced in SRAM. SparseTIR(*hyb*+TC) consumes more GPU memory than SparseTIR(naive) and SparseTIR(*hyb*) because of the half-precision/single-precision data type conversion.

Sparse Convolution

Sparse Convolution [35] is widely used in 3D cloud point data. We found that the Sparse Convolution operator is a special form of RGMS, and Figure 3.22 illustrates the equivalence: each relative offset inside the convolution kernel can be viewed as a relation in RGMS. For each relation, the mapping between non-zero elements in feature map of previous layer to

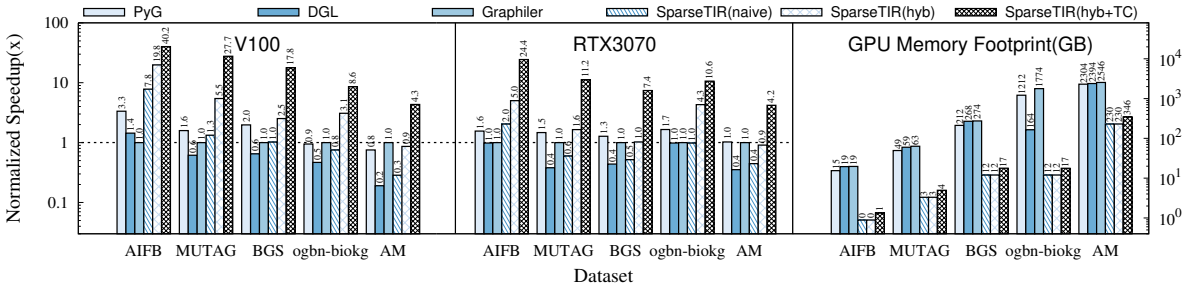

Figure 3.20: Normalized RGCN inference speedup against Graphiler and GPU Memory Footprint. SparseTIR(*hyb*+TC) uses schedule proposed in Figure 3.21, SparseTIR(*hyb*) uses composable format but use CUDA Cores instead of Tensor Cores for on-chip Matrix Multiplication, SparseTIR(naive) uses neither composable formats nor Tensor Cores.

non-zero elements in feature of next layer forms a bipartite graph which can be viewed as a 2D sparse matrix whose number of non-zero elements per row is no greater than 1.

We extract all of the Sparse Convolution operators in MinkowskiNet [35] on SemanticKitti dataset [9] for benchmark, and evaluate SparseTIR's RGMS kernel[12]. Figure 3.23 shows our normalized speedup against state-of-the-art TorchSparse [152] library. Unlike the SparseTIR's schedule in Figure 3.21, TorchSparse does not fuse Gather-Matmul-Scatter on chip. Instead, it explicit materializes T and uses coarse-grained cuBLAS operators rather than Tensor-Core level instructions for matrix multiplication[13]. SparseTIR's RGMS can outperform TorchSparse for most of the operators because of less HBM/SRAM data exchange as mentioned before. However, for large channel size (> 128), SparseTIR's RGMS cannot beat TorchSparse because matrix multiplication overhead become dominant (The FLOPs of Matmul is quadratic to channel size while the FLOPs of Gather and Scatter is linear to channel size) and cuBLAS is better optimized than SparseTIR's RGMS for large channel.

[12]We don't need to use composable formats for Sparse Convolution because the sparse matrix for each relation is already an $ELL(1)$.

[13]It's not necessary to use adaptive matrix multiplication grouping when using fine-grained Tensor-Core instructions.

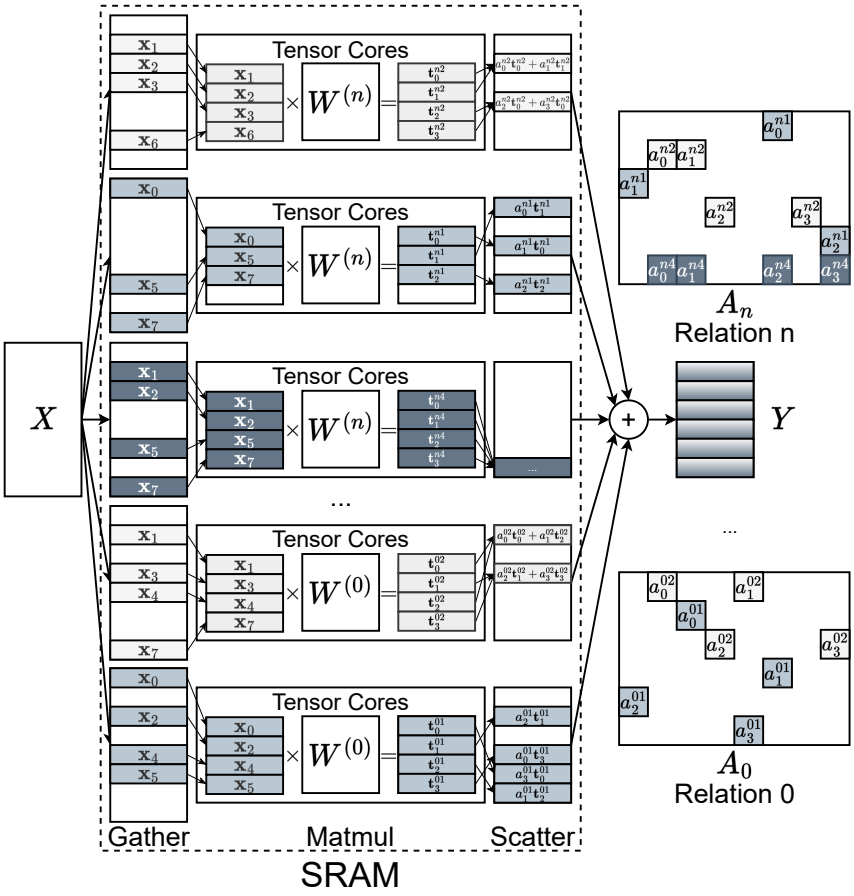

Figure 3.21: Schedule of RGMS operator in SparseTIR. Composable formats *hyb* are used for load balancing.

3.4 Related Work

Tensor and deep learning compilers. Halide [125] and TVM [25, 26] are tensor compilers that decouple kernel description and schedules for dense computation. XLA [43] and Relay [131] proposed computational-graph-level abstractions for deep learning, where we can apply optimizations such as *kernel fusion* and *graph substitution* [74]. However, these compilers have limited support for representing and optimizing sparse operators, impeding the wider deployment of sparse deep learning workloads such as GNNs. TensorIR [51] is

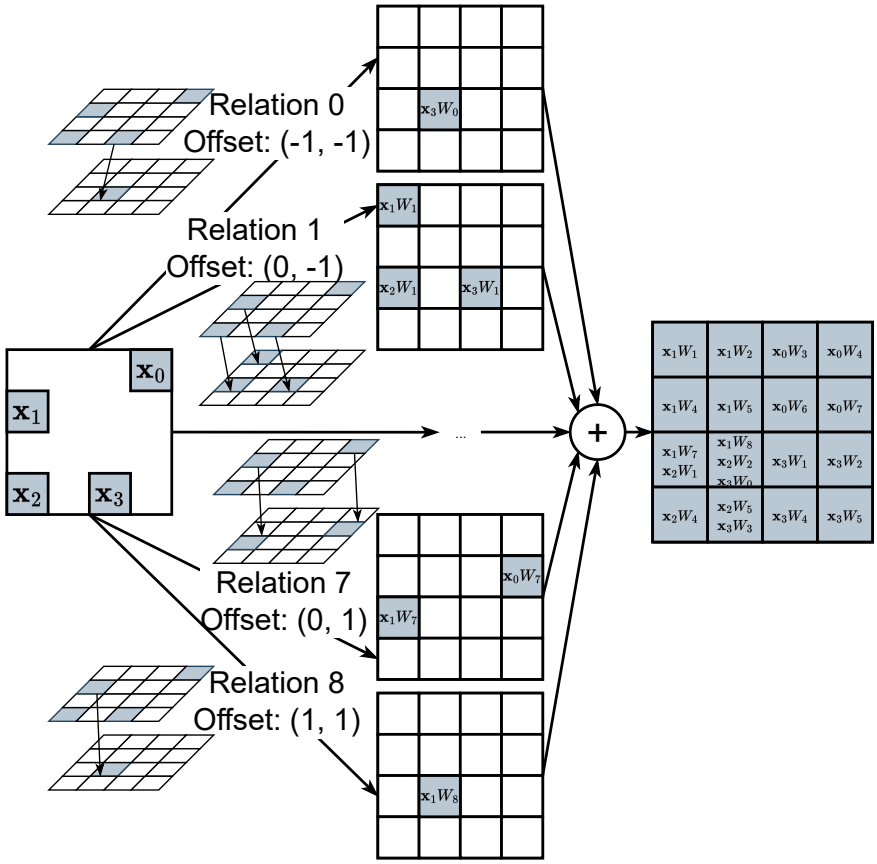

Figure 3.22: Equivalence of RGMS and Sparse Convolution, each relative offset inside the convolution kernel forms a relation in RGMS. The equivalence also holds in 3D setting.

TVM's new tensor-level programming abstraction for automatic tensorization. Triton [158] is an intermediate language that offers tile-level operations and optimizations, FreeTensor [154] is a compiler for irregular tensor programs with loop-based programming model. These IRs could serve as stage-III IR for SparseTIR.

Sparse compilers. MT1 [13, 14, 15, 16, 17], SIPR [123], Ironman [94] and Ahmed et al. [4] introduces the idea of compiling kernels for a given sparse data structure and a kernel description. TACO [33, 79, 80] proposes sparse format abstractions and a merge lattices-based code generation routine. Senanayake et al. [139] propose a sparse-iteration

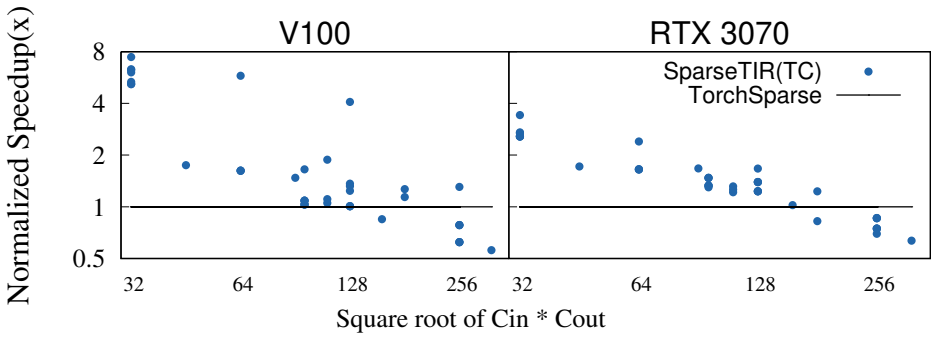

Figure 3.23: Normalized speedup against TorchSparse for Sparse Convolution. The X-axis refers to square root of input channel and output channel: $\sqrt{C_{in}C_{out}}$, and the Y-axis refers to speedup against TorchSparse.

space transformation framework for scheduling sparse operators. Chou et al. [34] introduce an approach for generating efficient kernels for sparse format conversion. Henry et al. [62] generalize TACO to sparse array programming. These works have huge impact on the design of SparseTIR. Sympiler [29] builds a symbolic inspector to analyze sparse structure at compile-time and generates efficient code. Parsy [30] generalize the idea to support parallelization. SPF [148] proposes a inspector-executor framework compatible with polyhedral transformations. Mohammadi et al. [101] proposes data dependence simplication algorithm for compiler generated inspectors. These compilers have huge potential for utilizing sparse structures, and we're exploring the possibility of combining them with composable formats. Taichi [68] decouple data structure and kernel description for physics simulation programming; its compiler optimization focuses on spatial sparse data, unsuitable for DL. Tiramisu [8] supports automatic selection of dense/sparse kernels at computational graph-level. However, it lacks tensor-level sparse code generation. COMET [157] and MLIR Sparse Dialect [12] are two MLIR dialects that explore composable IR design for sparse tensor algebra. Both treat sparse tensors with format annotation as first-class members in the IR; however, neither considers decomposable formats. CoRA [50] proposes a compiler infrastructure for ragged tensors [155]: a special form of sparse tensors. The operation splitting in CoRA is also a special case of

format decomposition in SparseTIR. SparTA [193] proposes sparse annotations for network pruning; its annotation is still dense and thus not applicable to highly sparse matrices used in GNNs. SparseLNR [45] proposes *branched iteration graph* to support factoring reductions and loop-fusion for sparse tensor algebra, these schedules can be formulated as stage-I schedules in SparseTIR as we support branches in the IR.

GNN systems and compilers. PyG [53] and DGL [167] propose programming interfaces for the programming message-passing [55] modules in GNN models. Both frameworks use vendor libraries and handwritten operators to accelerate specific message-passing patterns. Featgraph [69] optimizes generic GNN operators with TVM. However, it fails to support more operators because TVM lacks sparsity support. FusedMM [126] fuses SDDMM and SpMM operators, thus accelerating GNN training and saving GPU memory footprint. FusedMM can be described and optimized in SparseTIR. Seastar [174] and Graphiler [177] compile user-defined message-passing functions to their intermediate representations (IR) and then optimize the IR and emit template-based, target-specific code: these templates still have limited expressiveness and cannot consider a wide range of the optimization space. SparseTIR could serve as a backend for these GNN compilers. GNNAdvisor [171] proposes a CUDA template for GNN computations and uses graph characteristics to guide the performance tuning of GNN training. QGTC [170] and TC-GNN [169] explore accelerating GNNs with TensorCores. Notably, the "condensing" technique proposed in TC-GNN is equivalent to SpMM on SR-BCRS format as shown in Section 3.3.3. The contribution of these papers is orthogonal to SparseTIR.

Sparse kernel optimizations. Merge-SpMM [179], ASpT [64], GE-SpMM [71], Sputnik [54] and DA-SpMM [38] explore different schedule spaces for SpMM optimization on GPUs. We carefully examined the optimizations suggested in theses papers and propose a composable abstraction to unify them. OSKI [165] is a library for auto-tuning sparse operators, with a focus on optimizing operators on cache-based, super-scalar architectures such as CPUs.

However, OSKI do not support customizing sparse operators.

Sparse format optimizations. Pichon et al. [118] propose to reorder rows and columns in 2D sparse matrices to increase the block granularity of sparse matrices. Li et al. [87] study the problem of reordering sparse matrices to improve cache locality of operators on them. Mehrabi et al. [95] and Wang et al. [171] propose to reorder rows and columns of sparse matrices to accelerate SpMM on GPUs. These algorithms can act as pre-processing steps in SparseTIR to discover efficient composable formats.

Hardware-efficient algorithms. There have been a growing trend of sparsity in Deep Learning [63]. To make better use of underlying hardware, researchers propose pruning algorithms with block-sparsity [83] and bank-sparsity [21, 195] to utilize acceleration units in GPUs, and ES-SpMM [91] for load balancing. SparseTIR's composable abstractions can help researchers explore more complex sparse patterns with ideal performance on modern hardware.

3.5 Future Work

Automatic scheduling SparseTIR still requires users to specify schedule templates like they do for the first-generation of Halide and TVM. The Halide auto-scheduler [2], FlexTensor [194], Ansor [191] and Meta-scheduler [142] have been proposed to automatically generate schedule templates for dense tensor compilers. We expect these techniques would also prove helpful for sparse compilation. Searching for the optimal schedule is time consuming, Ahrens et al. [5] propose an asymptotic cost model for sparse tensor algebra to narrow the schedule space of sparse kernels, which could also benefit our work.

Automatic format decomposition In this paper we explore only manually designed format decomposition rules. We leave automatic format selection [15, 16] and decomposition for future work.

Dynamic Sparsity Some models [49, 120, 144] exhibit dynamic sparsity, where the position of non-zero elements changes overtime thus searching for best schedule for each matrix become impractical. DietCode [189] proposes *shape-generic* search space, micro-kernel based cost model and a lightweight dispatcher to dispatch kernel at runtime, the idea is also applicable to sparse tensor programs.

Integration with graph-level IR SparseTIR models only tensor-level sparsity, we plan to extend the sparse attributes in SparseTIR to graph-level IRs like XLA [43] and Relay [131].

3.6 Conclusion

We introduce SparseTIR, a composable abstraction for sparse operators in deep learning. Its key innovation is the use of composable formats and composable transformations, and together they form the parameter search space for performance tuning. Evaluations on generic sparse deep learning show that SparseTIR achieves significant performance improvements over existing vendor libraries and frameworks.

Chapter 4

FLASHINFER

The Transformer architecture has become the primary backbone for large language models (LLMs), prominently featuring attention mechanism [161] as its most salient component. As LLMs rapidly evolve and find applications in diverse fields, the demand for efficient GPU attention kernels grows, with the goal of enabling scalable and responsive model inference. At the heart of LLM inference lies the attention computation, which plays a crucial role in processing historical context and generating outputs based on query vectors. In LLM serving, the attention mechanism reads from the KV cache, which stores historical context, and computes outputs based on the current query. The efficiency of this attention operator is paramount to the overall performance of an LLM inference systems. However, creating high-performance attention kernels tailored for LLM serving introduces challenges not typically encountered in traditional training environments.

Two major challenges arise when building efficient attention support for LLM systems:

LLM applications exhibit diverse workload patterns and input dynamics. LLM serving involves various attention computation patterns, from prefill computation for context processing to batched decoding during serving [184]. As multiple requests are processed, opportunities for *prefix-reuse* emerge, and the introduction of tree decoding in speculative scenarios creates additional attention patterns [20, 28, 96]. Moreover, query lengths and KV caches vary within batches and over time, naive implementation might suffer load-imbalance issue, optimal scheduling requiring kernel to adapt dynamically for optimal performance.

Modern hardware implementations necessitate the customization of attention operators. On the memory side, efficient storage formats, such as paged attention [82] and radix trees [192], are critical for managing the growing KV cache sizes and diverse

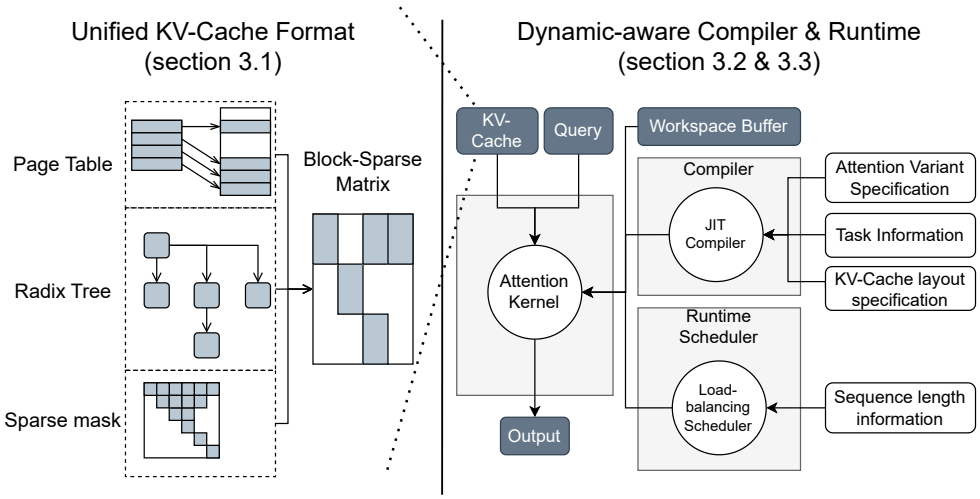

Figure 4.1: Overview of the FlashInfer system design: Attention variant specifications, task information and KV-cache layout specifics are provided at compile time for JIT compilation, while sequence length information is input at runtime for dynamic scheduling.

storage patterns. On the compute side, crafting hardware-specific pipelines and templates is indispensable to fully exploit the performance potential of each GPU architecture [39, 141]. Furthermore, the design must accommodate the increasing variety of attention mechanisms in modern LLMs, such as grouped attention heads [6, 143], specialized masks [11], and customized attention score computations [128, 130, 175], necessitating flexible and scalable implementation strategies.

The combined complexity of workload diversity and hardware heterogeneity complicates the development of a comprehensive attention solution. Currently, each system implements a specialized attention solution based on a subset of these characteristics, leading to high maintenance overhead and potential inefficiencies. To address these challenges, we introduce FlashInfer, a code-generation based attention engine designed to accelerate attention computation in LLMs. Our approach incorporates several key designs:

FlashInfer utilizes a block-sparse format to tackle KV-Cache storage hetero-

geneity. This format serves as a unified data structure for various KV-Cache configurations, with adjustable block sizes allowing fine-grained sparsity, such as vector-level sparsity [27, 89]. This approach unifies diverse KV-Cache patterns and enhances memory access efficiency.

A customizable attention template supports different attention variants in FlashInfer. FlashInfer provides a customizable programming interface for users to implement their attention variants. FlashInfer uses Just-In-Time (JIT) compilation to translate these variants into highly optimized block-sparse implementations, ensuring rapid adaptation to varying attention configurations.

FlashInfer employs a dynamic load-balanced scheduling framework to handle input dynamism effectively. It separates compile-time tile size selection from runtime scheduling, offering lightweight APIs that adaptively manage scheduling with changing KV-Cache lengths during inference, while maintaining compatibility with CUDAGraph's requirement for constant configurations [56, 104].

Figure 4.1 depicts our system design. We evaluated FlashInfer's performance across standard LLM serving environments and innovative scenarios, including prefix sharing and speculative decoding. FlashInfer have been integrated with mainstream LLM serving engines, including vLLM [82], MLC-Engine [84, 100], and SGLang [192], we assessed its impact on end-to-end latency and throughput improvements, showing significant enhancements on standard LLM serving benchmarks and novel applications such as long-context inference and parallel generation.

Our contributions include:

- Introduction of flexible block-sparse and composable formats addressing KV-Cache storage heterogeneity for efficient memory management and access.
- Development of a customizable attention template accommodating diverse attention variants, ensuring high-performance execution via JIT compilation.
- Design of a dynamic scheduling framework managing input dynamism while remaining compatible with CUDAGraph, maximizing hardware utilization.
- Comprehensive evaluation demonstrating substantial improvements in kernel and end-

to-end performance.

4.1 Background

4.1.1 FlashAttention

FlashAttention [40] is an efficient algorithm for computing exact attention with reduced memory usage. During the forward pass, it employs the online-softmax trick [98], updating attention outputs on-the-fly using a constant amount of on-chip memory, thus avoiding materializing the attention matrix in GPU global memory. FlashAttention2&3 [39, 141] improve performance by optimizing loop ordering and pipeline design for Ampere and Hopper GPUs. FlashInfer builds upon these advancements.

The operational intensity of FlashAttention is given by $O\left(\frac{1}{1/l_{qo}+1/l_{kv}}\right)$, where l_{qo} and l_{kv} are the query and key-value cache lengths, respectively. In LLM serving, the query length is either equal to (prefill) or smaller than (decode/incremental prefill) the key-value cache length, simplifying the operational intensity to $O(l_{qo})$. Techniques like batching [184] do not alter this operational intensity. Multi-Query Attention (MQA) [143] and Grouped Query Attention (GQA) [6] optimize the KV-Cache size by grouping queries and sharing the same KV-Cache entries. The ratio of the number of queries to the number of KV-Cache entries is denoted as the group size $g = \frac{H_{qo}}{H_{kv}}$, enhancing operational intensity to $O(g \cdot l_{qo})$.

4.1.2 Attention Composition

Block-Parallel Transformer (BPT) [92] demonstrates that attention outputs for the same query and different keys/values can be composed by preserving both the attention outputs and their scales. Let $\mathbf{q}$ be a query, and let $\mathcal{I}$ be an index set. We define the *attention scale* over $\mathcal{I}$ via the log-sum-exp operation on the attention scores:

$$\mathbf{LSE}(\mathcal{I}) = \log \sum_{i \in \mathcal{I}} \exp(\mathbf{q} \cdot \mathbf{k}_i) \tag{4.1}$$

where $\mathbf{k}_i$ is the i-th key vector. The corresponding *attention output* $\mathbf{O}(\mathcal{I})$ is then

$$\mathbf{O}(\mathcal{I}) = \sum_{i \in \mathcal{I}} \frac{\exp(\mathbf{q} \cdot \mathbf{k}_i)}{\exp(\mathbf{LSE}(\mathcal{I}))} \cdot \mathbf{v}_i \tag{4.2}$$

We define the *Attention State* for $\mathcal{I}$ as the tuple of *attention output* and *attention scale*: $\begin{bmatrix} \mathbf{O}(\mathcal{I}) \\ \mathbf{LSE}(\mathcal{I}) \end{bmatrix}$. Crucially, the Attention State of $\mathcal{I} \cup \mathcal{J}$ can be derived by composing the states of $\mathcal{I}$ and $\mathcal{J}$. Specifically, introducing an operator $\oplus$:

$$\begin{aligned} \begin{bmatrix} \mathbf{O}(\mathcal{I} \cup \mathcal{J}) \\ \mathbf{LSE}(\mathcal{I} \cup \mathcal{J}) \end{bmatrix} &= \begin{bmatrix} \mathbf{O}(\mathcal{I}) \\ \mathbf{LSE}(\mathcal{I}) \end{bmatrix} \oplus \begin{bmatrix} \mathbf{O}(\mathcal{J}) \\ \mathbf{LSE}(\mathcal{J}) \end{bmatrix} \\ &= \begin{bmatrix} \frac{\exp(\mathbf{LSE}(\mathcal{I}))\mathbf{O}(\mathcal{I}) + \exp(\mathbf{LSE}(\mathcal{J}))\mathbf{O}(\mathcal{J})}{\exp(\mathbf{LSE}(\mathcal{I})) + \exp(\mathbf{LSE}(\mathcal{J}))} \\ \log(\exp(\mathbf{LSE}(\mathcal{I})) + \exp(\mathbf{LSE}(\mathcal{J}))) \end{bmatrix} \end{aligned}$$

Since $\oplus$ is associative and commutative, multiple sets of attention states can be composed in any order. Ring-Attention [93] and Flash-Decoding [41] utilize this property to offload partial-attention computations, thereby reducing memory usage and improving hardware efficiency. In FlashInfer, the *Attention State* is adopted as the canonical output of an attention operation, and $\oplus$ serves as the standard reduction operator (analogous to summation in GEMM) on these states.

4.1.3 Block/Vector Sparsity

Block Compressed Sparse Row (BSR) is a hardware-efficient sparse format that groups non-zero elements into contiguous matrices of size (b_r, b_c), as opposed to the random scattering found in unstructured sparsity. This format offers several advantages over the standard Compressed Sparse Row (CSR) format. BSR improves register reuse efficiency [18, 72] and demonstrates better compatibility with hardware matrix multiplication units on GPUs and NPUs [57, 103]. In addition, it provides the ability to skip empty blocks, reducing computational overhead. BSR's efficiency is particularly evident when subcomputations are aligned with hardware matrix multiplication instructions, such as NVIDIA's `mma` instructions. Traditionally, tensor core instructions operate on minimal dimensions of 16 (or larger for

newer GPUs), leading most block-sparse kernels to use block sizes that are multiples of $(16, 16)$. However, this approach is not always optimal for applications with fine-grained sparsity patterns [172]. Many attention libraries restrict their block sizes to multiples of $(128, 128)$ for block-sparse attention kernels.

Recent research [27, 89] has demonstrated that efficient utilization of the tensor core can be achieved with smaller block sizes, such as $(16, 1)$ for matrix B in GEMM, or $(1, 16)$ for matrix A (also known as vector-sparse). This is accomplished by first gathering rows/columns into contiguous shared memory and then applying dense tensor cores to these contiguous shared-memory data. This approach is particularly beneficial for applications with fine-grained sparsity patterns. FlashInfer builds upon these techniques to support blocks with arbitrary column sizes B_c, offering greater flexibility and efficiency in handling diverse sparsity patterns.

4.2 Design

In this section, we introduce the system design of FlashInfer. We begin by presenting the data structure employed in FlashInfer and demonstrate how Block-Sparse Row (BSR) acts as a versatile abstraction for KV cache storage in attention kernels. Next, we discuss the FlashInfer compiler, which supports various attention variants, alongside a dynamic-aware runtime scheduler that facilitates load-balanced scheduling of attention kernels. Finally, we describe the user-level API designed for integrating FlashInfer with existing LLM serving systems.

4.2.1 KV-Cache Storage

Block-Sparse Matrix as Unified Format

Recent advancements in KV-Cache storage, such as PageAttention [82] and RadixAttention [192], employ non-contiguous memory storage with a minimum granularity of a block (or token) of (H, D) tensors, where H represents the number of heads and D the hidden dimension. These structures are optimized to minimize memory fragmentation while enhancing

memory reuse and cache hit rates. We demonstrate that these diverse data structures can be unified under a block sparse format, as illustrated in Figure 4.2.

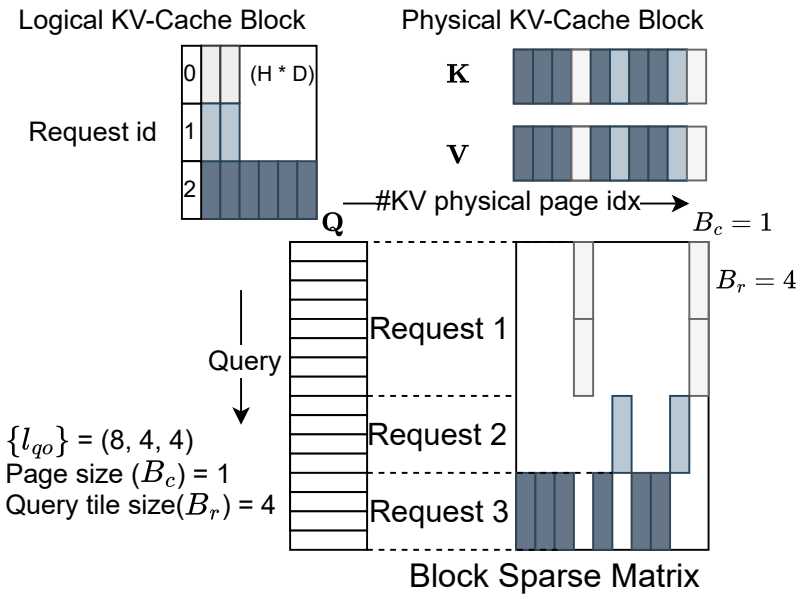

Figure 4.2: Representation of Page Table in BSR ($B_r = 4, B_c = 1$) format. The number of column blocks in the block sparse matrix corresponds to the total number of blocks allocated for the Page Table. Non-zero blocks represent KV-Cache pages accessed by queries.

The conceptual equivalence between page tables and sparse matrices has been previously explored in SPGrid [140], which leverages Translation Lookaside Buffer (TLB) hardware for efficient sparse structure indexing. Beyond page tables and radix trees, sparse matrices can also effectively represent various attention mechanisms, such as Tree Attentions used in speculative decoding [20, 28, 96] and importance masks applied to KV-Cache [153].

In FlashInfer, we implement a unified strategy for data representation. Query and output matrices are efficiently stored as ragged tensors (also known as jagged arrays) [155] without padding, which facilitates the compact packing of queries and outputs from diverse requests

into a single tensor. Initially, keys and values are maintained in ragged tensors using the same index pointers as queries, as they originate from the projection matrices W_q, W_k, W_v applied to the same input. These keys and values are subsequently incorporated into the KV-Cache with newly updated entries. The KV-Cache employs a block-sparse row (BSR) format, where block sizes are defined by application requirements: B_r corresponds to the query tile size, details of which will be discussed in later sections, and B_c is specified by KV-Cache management algorithms. FlashInfer kernel implementations supports arbitrary (B_r, B_c) values.

Composable Formats for Memory Efficiency

Inspired by SparseTIR [183], we enhance attention computation efficiency through composable formats. This approach leverages multiple block sparse formats instead of a single format to store the sparse matrix, offering greater flexibility and memory efficiency. Single block-sparse formats are constrained by a fixed block size, limiting memory efficiency based on the number of rows in the block (B_r). While larger B_r values improve shared memory and register reuse for requests within the same block, they also increase fragmentation. Conversely, requests in different blocks cannot access each other's shared memory.

Our composable format design allows for the decomposition of the KV cache sparse matrix based on prior knowledge. For instance, if certain requests share a prefix, the corresponding rows and columns in the KV cache form a dense submatrix. We can then use a block sparse matrix with a larger B_r to store these submatrices efficiently. Figure 4.3 illustrates this concept, showing how shared prefixes can be optimized using composable formats. This approach doesn't require data movement in the KV cache; instead, we compute the indices and index pointer arrays for the sparse submatrices. Attention computations on larger block sizes can access shared KV cache entries using fast shared memory and registers, significantly improving memory efficiency.

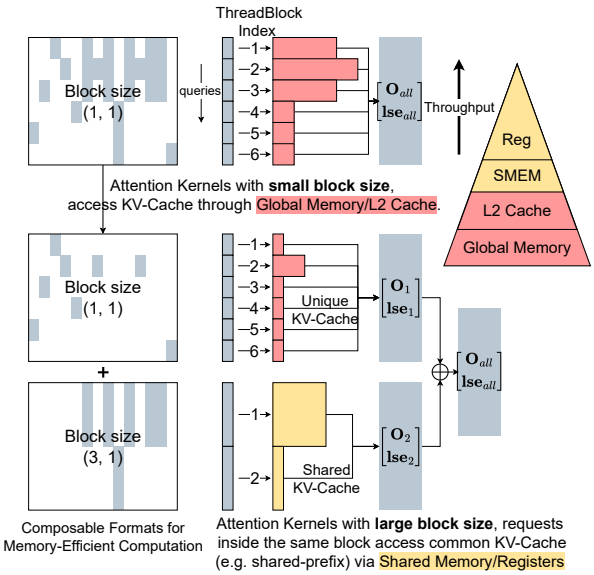

Figure 4.3: Composable formats for shared-prefix decomposition in attention computation. The queries corresponding to the first 6 rows have a shared prefix, as do the queries in the last 6 rows. We store the KV cache corresponding to the shared prefix in a block sparse matrix with a block size of $(3, 1)$, while storing the remaining unique KV cache in another block sparse matrix with a block size of $(1, 1)$. For block size $(3, 1)$, 3 queries can share the same KV cache in high-bandwidth shared memory, while for block size $(1, 1)$, each query access KV-Cache within its own threadblock, which can only go through low-bandwidth global memory or L2 cache.

4.2.2 Compute Abstraction

We developed CUDA/CUTLASS [156] templates for FlashAttention, designed specifically for both dense and block-sparse matrices and compatible with NVIDIA GPU architectures from Turing to Hopper (sm75 to sm90a). Our implementations utilize the FlashAttention2 (FA2 for short) algorithm [39] for architectures up to Ada(sm89), and the FlashAttention3 (FA3 for short) algorithm [141] for Hopper. Key improvements include enhanced loading of sparse tiles into shared memory, expanded tile-size configurations, optimized memory access

patterns for grouped query attention, and customizable attention variants.

Global to Shared Memory Data Movement

The FlashInfer attention template supports any block size, requiring a specialized data loading approach since blocks may not align with tensor core shapes. As discussed in Section 4.1.3, we address these challenges by transferring tiles from scattered global memory to contiguous shared memory for dense tensor core operations. Tensor core inputs for a single MMA instruction can originate from different blocks within a block-sparse matrix. Figure 4.4 illustrates how FlashInfer loads tiles from sparse/dense KV-Cache into shared memory; sparse KV-Cache addresses are computed using the `indices` arrays of the BSR matrix, while dense ones use row index affine transformations.

The last dimension of the KV-Cache remains contiguous (with size of head dimension d, commonly 128 or 256), maintaining coalesced memory access that fits GPU cache line sizes. We use asynchronous copy instructions `LDGSTS` with a 128B width to maximize memory bandwidth. Although the Tensor Memory Accelerator (TMA) in Hopper architecture can further accelerate data movement, it doesn't support non-affine memory access patterns. Consequently, we only use TMA for contiguous KV-Cache on Hopper GPUs and fall back to Ampere-style asynchronous copies for other settings where TMA isn't suitable.

Post-transfer to shared memory, the sparse and dense FlashAttention implementations converge, allowing consistent kernel usage with variations only in data loading modules.

Microkernel with Different Tile Sizes

To adapt to the varying operational intensities of LLM applications, FlashInfer implements the FA2 algorithm across multiple sizes. Traditional FA2 uses limited number of tile sizes (e.g., $(128, 64)$), optimal for prefill on A100 but inefficient for shorter-query-length decoding. One architecture's ideal tile size may not suit others; for instance, Ada(sm89) has limited shared memory, affecting SM occupancy with large tiles.

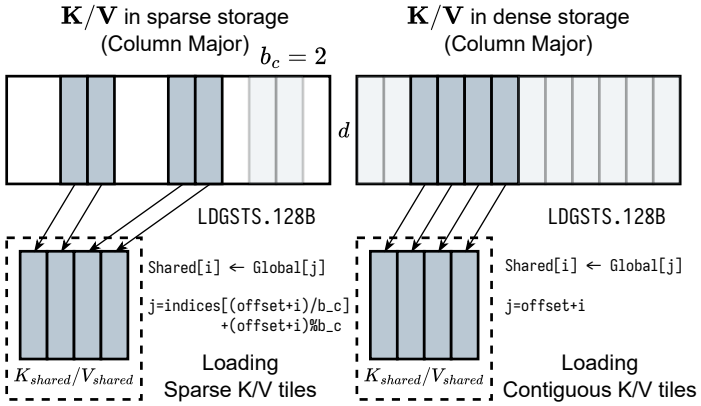

Figure 4.4: Data transfer from global to shared memory for sparse/dense KV-Cache in FlashInfer. Left: Sparse KV-Cache with $b_c = 2$; Right: Dense KV-Cache. Head dimension d.

FlashInfer offers FA2 kernels with tile sizes $(1, 16, 32, 64, 128) \times (32, 64, 128)$ and selects using heuristics based on hardware resources and workload intensity:

1. Determine average query length (for Grouped-Query Attention, the query length are fused with head group dimension, see Appendix 4.2.2) per batch, choosing the minimal query tile size meeting or exceeding it.
2. Formulate register and shared memory constraints as functions of K/V tile size, maximizing SM resource occupancy.

JIT Compiler for Attention Variants

For query tile size 1, we use CUDA Cores template since tensor core instruction m (minimum rows) is 16, and use Tensor Cores for other query tile sizes. For FA3 we provide row tile sizes that are multiples of 64, aligning with Hopper's WGMMA requirements. Tile sizes resolve at compile-time considering task specifics (decoding, prefill, etc.) and hardware capabilities. The block row size B_r is block-sparse matrix is aligned with the query tile size T_q.

Recent LLM models use variants to standard attention algorithms. Supporting various

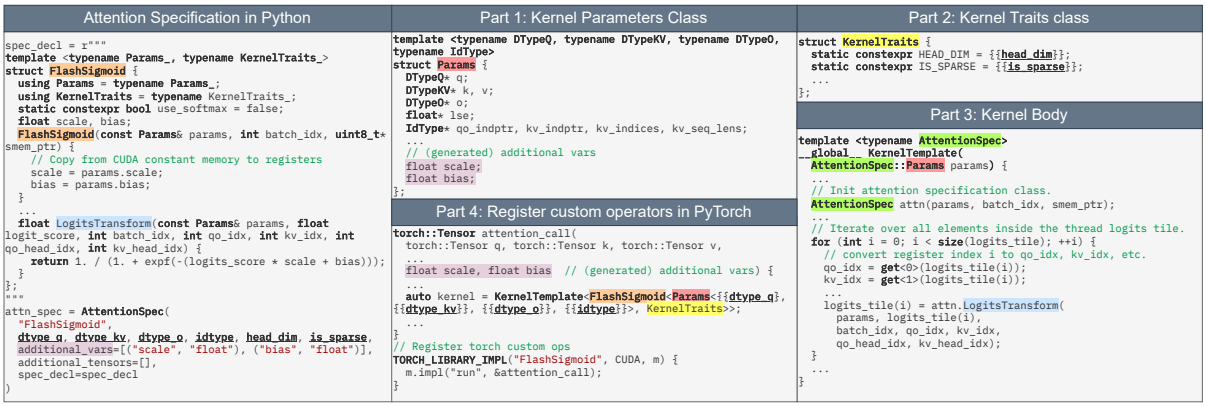

Figure 4.5: JIT compiler for attention variants in FlashInfer, featuring CUDA code strings defining variant functors, additional variables/tensors, and data types, used to populate kernel templates. Corresponding codes share highlighting.

attention variants in CUDA library is not sustainable because the we specialize the kernel for each variant for maximum performance, and the number of variants is growing rapidly. However, most attention variants have similar structure to the vanilla attention so we can use the same skeleton of FlashAttention kernels with small modifications. Inspired by FlexAttention [61], we designed a customizable CUDA template and a JIT compiler that takes the attention variant specification as input and generates the optimized kernel code. The variant specification includes the following functors:

- `QueryTransform, KeyTransform, ValueTransform`: The transformation applied to the query/key/value tensor before the attention computation.
- `OutputTransform`: The transformation applied to the attention output tensor before returning.
- `LogitsTransform, LogitsMask`: The transformation applied to the logits tensor before the softmax computation, and the mask applied to the logits tensor.

Each functor has a fixed signature that takes the kernel parameters, input and current query/key/head index as input, and returns the output. Those variant functions are specified

as member of a user-defined variant class which creates a closure for the variant functors. Functors such as `LogitsTransform` and `LogitsMask` are inspired by FlexAttention [61] and can be used to implement the attention variants with customized logits postprocessing such as custom mask, logits soft-cap [130, 175] and sliding window attention [11]. FlashInfer has an option of using softmax or not in the attention variant specification, which makes it capable of supporting attention variants that don't use softmax, such as FlashSigmoid [128]. FlashInfer's query and key transformation functors making it possible to fuse normalization, RoPE [149] and projection [42] into the attention kernel.

Figure 4.5 shows how FlashInfer maps FlashSigmoid's [128] attention specification to different parts of FlashInfer's CUDA templates. Attention specificiation accepts a piece of CUDA code to define the variant functors, such design also enables user to use advanced PTX instructions [1] or even their own libraries. The JIT compiler generates the CUDA code by inserting the variant class and other information into the template, and the generated CUDA code is compiled with PyTorch's JIT compiler [2] and registered as a custom operator [3]. We also support compiling to other runtime systems through a framework agnostic DLPack [4] interface.

Head Group Fusion for Grouped-Query Attention

Grouped-Query Attention (GQA) [6] allows multiple query heads to share the same key-value (KV) heads. A straightforward implementation that assigns distinct GPU threadblocks to each query head leaves much of the potential KV-Cache reuse underutilized when the query length is short. To address this limitation, FlashInfer offers a *head-group fusion* strategy: different KV heads are mapped to individual threadblocks, while query heads are fused with the query length dimension. This fusion scheme is illustrated in Figure 4.6, which shows

[1] `https://docs.nvidia.com/cuda/parallel-thread-execution/`

[2] `https://pytorch.org/tutorials/advanced/cpp_extension.html#jit-compiling-extensions`

[3] `https://pytorch.org/tutorials/advanced/custom_ops_landing_page.html`

[4] `https://github.com/dmlc/dlpack`

how the fused row index relates to the original row index and the head indices. By merging the query-head dimension with the row dimension in the threadblock mapping, a single shared-memory load of the KV-Cache suffices for all query heads in the group, leading to better memory reuse and improved throughput for GQA operations.

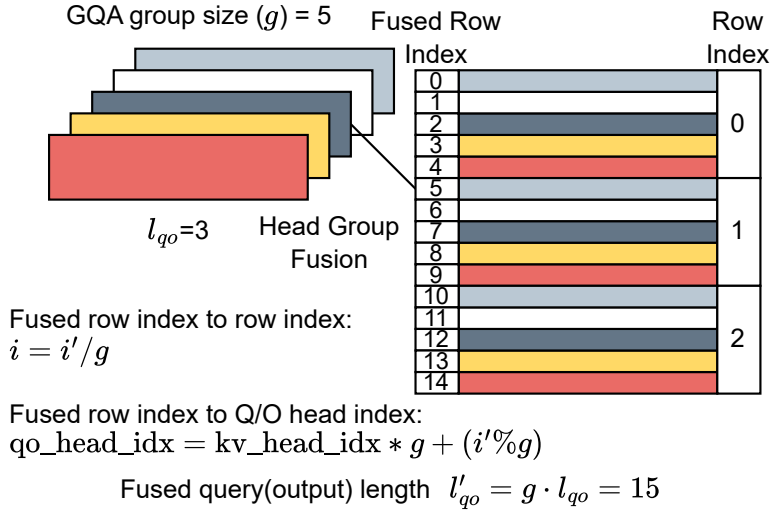

Figure 4.6: FlashInfer's head-group fusion of query heads with the query length dimension in GQA.

We prefer head-group fusion primarily for short query lengths. When the query length is sufficiently large, the query dimension itself yields enough workload to effectively utilize the KV-Cache, making head-group fusion less critical. Similar ideas have also been explored in other frameworks, such as XQA [110] in TensorRT-LLM [109].

FP8–FP16 Mixed-Precision Attention

Recent LLMs frequently adopt `fp8` KV-Cache to reduce memory bandwidth and storage costs [97]. In FlashInfer, we implement *mixed-precision* attention kernels wherein the query

and output remain in `fp16`, while the KV-Cache is stored in `fp8`. We leverage the fast numerical array converter and fragment shuffler proposed by CUTLASS [58] to accelerate dequantization and handle bitwidth mismatches efficiently. This design allows for reduced memory footprints and higher bandwidth utilization without significantly compromising numerical accuracy.

4.2.3 Dynamism-Aware Runtime

In this section we introduce the runtime design of FlashInfer, including the dynamic scheduling framework, and the composable formats for memory efficient attention.

Load-balanced Scheduling

In FlashInfer, the load-balanced scheduling algorithm aims to minimize SM idle time by distributing the workload evenly across all SMs. It takes the sequence length information of the query/output and key/value dimensions as input, and produces both the mapping between the workload and Cooperative Thread Arrays (CTAs) and the index mapping for partial and final outputs. The algorithm is presented in Algorithm 1 (the head dimension is omitted for simplicity). Our approach is inspired by Stream-K [113]; however, because LLM serving requires deterministic outputs, we did not incorporate atomic aggregation in Stream-K implementation to avoid non-deterministic behavior. The scheduling algorithm generates deterministic aggregation order when provided with identical sequence length information.

Figure 4.7 shows the workflow of FlashInfer's runtime scheduler. The attention kernel do not produce the final output directly because some long KV are split into multiple chunks, and the final output is the contraction (using the attention composition operator mentioned in section 4.1.2) of all chunks' partial outputs. The partial outputs are stored in a workspace buffer provided by the user (see section 4.2.5). FlashInfer implements efficient attention composition operator that can deal with variable length aggregation. The work queue of each CTA, and the mapping between partial and final outputs need to be planned by the

Algorithm 1 FlashInfer's balanced scheduling algorithm

Input: $\{l_{qo}(i), l_{kv}(i)\}_i$, query tile size T_q.

Define the cost of a tile l_q, l_{kv} as (α, β are hyperparameters):

$$\text{cost}(l_q, l_{kv}) = \alpha l_q + \beta l_{kv}$$

Compute the maximum KV chunk size L_{kv} by

$$L_{kv} \leftarrow \frac{\sum_i \lceil \frac{l_{qo}(i)}{T_q} \rceil \cdot l_{kv}(i)}{\#\text{CTA}}$$

Split each query tile's KV into chunks, with maximum size L_{kv}, we assign each chunk a work index w, and the length of the chunk is $l_{kv}(w)$.

Let $W = \{(w, l_{kv}(w))\}$ and sort the entries in descending order of length.

$Q \leftarrow \text{PriorityQueue}(\{(c, 0)\})$ where c is the CTA index.

while $W \neq \emptyset$ **do**

    $c, \text{current\_cost} \leftarrow Q.\text{pop}_{min}()$

    $w, l_{kv}(w) \leftarrow W.\text{pop}()$

    $\text{new\_cost} \leftarrow \text{current\_cost} + \text{cost}(T_q, l_{kv}(w))$

    Assign chunk w to CTA c

    $Q.\text{push}((c, \text{new\_cost}))$

end while

scheduler. Once plan information is computed on CPU [5] FlashInfer asynchronously copy the plan information to a specific region of the workspace buffer on GPU, and the plan information is used as inputs for persistent attention/contraction kernels. The scheduler runs per generation step to produce plan information as the sequence length changes for each generation step on CPU, and overhead can be amortized over multiple layers because the same plan information can be reused for all layers.

FlashInfer guarantees both attention and contraction stage are compatible with CUDA-Graphs [56, 104]. Both attention and contraction stage use persistent kernel and the grid size is fixed once compiled, which means the kernel is launched with the same grid size for

[5] In some LLM inference workloads, each layer may have different sequence length information, making it impractical to cache plan information for all layers. In such cases, we provide a GPU-based scheduler implementation to avoid CPU-GPU communication overhead. The scheduling algorithm can also be executed on-the-fly when minimizing latency is critical.

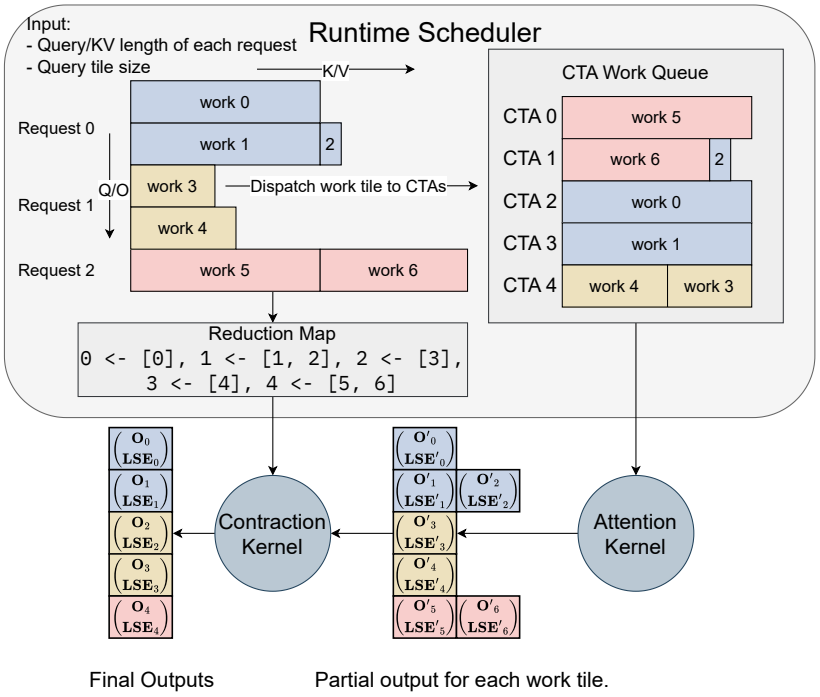

Figure 4.7: FlashInfer's load-balanced runtime scheduler, sequence length information (both on query/output and key/value dimension) are provided to the scheduler to compute the plan information: (1) Work queue of each CTA (2) Index mapping between partial and final outputs. These plan information are cached at GPU-side and used as inputs for persistent attention/contraction kernels.

each generation step. We set the fixed offset for each section of the workspace buffer to store partial outputs and plan information to make sure the pointers passed to the kernel are the same for each generation step, meeting the requirement of CUDAGraphs (see Appendix 4.2.4 for details). We merge the two stages into one persistent kernel, separated by a global barrier `grid.sync()`, eliminating intra-kernel overhead. [6].

[6]The overhead could be further reduced by fine-grained CTA-level semaphores stored in global memory, see `https://github.com/flashinfer-ai/flashinfer/pull/1200` for more details

4.2.4 Memory Management

FlashInfer manages a page-locked (pinned) host buffer and a device workspace buffer to store scheduler metadata and split-k partial outputs. We divide the device workspace buffer into *sections*, each corresponding to an array of either scheduler metadata or partial split-k outputs. For each plan call in the scheduler, we compute the scheduler metadata on the pinned host buffer and then issue a cudaMemcpyAsync to transfer this data into the corresponding sections of the device workspace buffer.

CUDAGraph-Compatible Workspace Layout

Once a kernel is captured by CUDA Graph, its arguments (pointers and scalars) become fixed, implying that each section of the device workspace buffer must maintain a consistent address for the entire captured graph's lifetime. Therefore, we allocate the workspace buffer to its maximum required capacity for each section, based on upper-bound estimations of scheduler metadata and partial outputs.

Split-K Writethrough Optimizations

In FlashInfer's load-balancing scheduler (Section 4.2.3), KV-splitting is only applied to requests that have large KV lengths. Requests with short KV lengths do not require splitting and hence have no reduction step from partial output. To save both computation and workspace memory, these small requests can write their partial outputs directly to the final output buffer (bypassing the device workspace buffer). This approach reduces both the required workspace size and the computational load within the contraction kernel.

Workspace Buffer Size Estimation

The workspace buffer size depends on two main factors: (1) the required space for scheduler metadata, and (2) the required space for storing partial split-k outputs.

Scheduler Metadata. The maximum size of each metadata section is derived from the largest possible number of concurrent requests and the maximum accumulated request length. Users must provide these upper bounds during the scheduler's first planning stage.

Partial Outputs. The size of partial outputs depends on both the problem dimensions (i.e., the number of heads and the head dimension) and the number of CTAs per kernel launch. In our load-balancing algorithm 4.2.3, only requests deemed "long" – those whose KV length exceeds the total KV length divided by the number of CTAs – are split. According to the Writethrough Optimizations in Section 4.2.4, only these split requests produce outputs in the workspace buffer. Because the number of splits cannot exceed the total number of CTAs, and each split yields at most two tiles that must be merged, there are at most $2 \times \#\text{CTA}$ partial outputs. Each tile produces a partial output of size $T_q \cdot H_{qo} \cdot (D+1)$, where T_q is the query tile size, H_{qo} is the number of heads, and $D+1$ is the head dimension and LSE dimension. Therefore, the upper bound for the total partial output size is:

$$2\,\#\text{CTA} \times T_q \times H_{qo} \times (D+1).$$

By default, the total number of CTAs is set to $k \times \#\text{SM}$, where #SM denotes the number of streaming multiprocessors on the GPU and k is chosen to maximize CTA-level occupancy. For tensor-core based microkernels with high register usage, k typically does not exceed 2 on Ampere, and it is often 1 on Hopper (one CTA per SM, also referred to as a persistent kernel).

4.2.5 Programming Interface

FlashInfer offers a programming interface designed for seamless integration with existing LLM serving frameworks such as vLLM [82], MLC-Engine [100], and SGLang [192].

```
# Create workspace buffer
workspace = torch.empty(...)
seqlen_info.init()

# Compile: create CUDAGraphs
```

```
graphs = []
for task_info in task_infos:
  # Init: compile kernels according to spec
  attn = AttentionWrapper(attn_spec, task_info, workspace)
  g = torch.cuda.CUDAGraph()
  # Dummy plan
  attn.plan(seqlen_info)
  # Capture CUDA graphs
  with torch.cuda.graph(g):
    for i, layer in enumerate(layers):
      ...
      attn.run(...)
      ...
  graphs.append(g)

# Runtime: select the best CUDAGraph
g = select_graph(graphs)
finished = False
# Text generation loop
while not finished:
  seqlen_info.update()
  # Plan per generation step
  attn.plan(seqlen_info)
  # Replay CUDA-Graph
  g.replay()
```

Listing 4.1: FlashInfer PyTorch Programming Interface

Listing 4.1 shows the PyTorch programming interface of FlashInfer. The user initializes the wrapper by providing the attention variant specification, task information, and a user-allocated workspace buffer (see Appendix 4.2.4 for details) to store partial output and plan information for FlashInfer dynamic scheduling. Kernel are JIT-compiled at init time and cached for reuse. For composable formats (section 4.2.1), FlashInfer creates multiple attention wrappers, each with distinct block sizes. Kernels with different average query length and composable format configurations are compiled and captured in different CUDAGraphs. At runtime, the serving framework selects the most appropriate CUDAGraph based on the current KV-Cache configuration, ensuring optimal performance for varying workload characteristics.

The `plan` function activates the dynamic scheduler by processing sequence length data to generate load-balanced scheduling plans. These plans are cacheable, allowing reuse across operators with matching sequence length specs, such as all decode attentions in a generation step. The `run` function executes the attention computation using inputs of query, key, value, and cached plan data, outputting the attention results. CUDAGraph can capture calls to `run` functions and compile the entire attention generation step into a single graph. However, `plan` function is not captured by CUDAGraph because it's on CPU. The `plan` and `run` division is inspired by the Inspector-Executor (IE) model [99, 119, 134], which is widely used for parallelizing irregular workloads.

4.3 Evaluation

In this section, we evaluate FlashInfer v0.2 on kernel-level and end-to-end performance showing how FlashInfer's design address the challenges of LLM serving. We achieve 29-69% inter-token-latency reduction compared to Triton backend for LLM serving benchmark, 28-30% latency reduction for long-context inference, and 13-17% speedup for LLM serving with parallel generation. We conduct experiments on NVIDIA A100 40GB SXM and H100 80GB SXM GPUs, using CUDA 12.4 and PyTorch 2.4.0 and f16 precision for storage and computation.

4.3.1 End-to-end LLM serving performance

We evaluate FlashInfer with SGLang v0.3.4 [192] and compare its performance against two settings: SGLang with Triton v3.0 [159]. The latter is a leading LLM serving engine optimized for NVIDIA GPUs; however, its attention kernels are closed-source, which limits transparency and potential for community-driven improvements. To ensure a comprehensive evaluation, we employ two datasets: the widely-used ShareGPT dataset [7] and a synthetic workload (Variable) with sequence lengths uniformly distributed between 512 and 2048 tokens. We

[7] `https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered/resolve/main/ShareGPT_V3_unfiltered_cleaned_split.json`

measure the TTFT(time-to-first-token) and ITL(inter-token-latency) under latency-sensitive online serving settings, the request rate is adjusted to maintain P99 TTFT below 200ms.

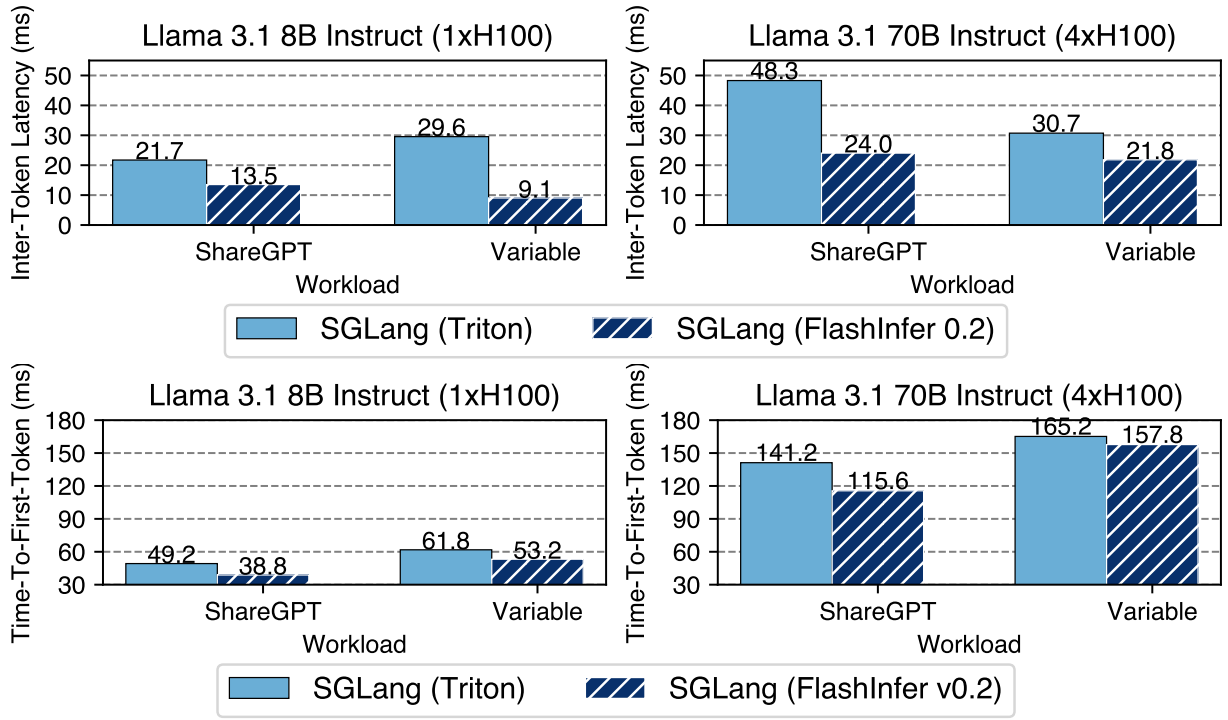

Figure 4.8: Medium Inter-Token-Latency (ITL) and medium Time-To-First-Token (TTFT) of SGLang integrated with FlashInfer and Triton.

Figure 4.8 shows the ITL and TTFT measured on both Llama 3.1 [46] 8B (on 1xH100) and Llama 3.1 70B (on 4xH100) models. Compared to SGLang with Triton backend, FlashInfer backend shows consistent speedup in all settings.

4.3.2 Kernel Performance for Input Dynamism

In this section we measure FlashInfer's generated kernel performance against state-of-the-art open-source FlashAttention library under different sequence length distributions, we use the latest main branch [8] which includes both FlashAttention2 and FlashAttention3 kernels. We

[8]Commit: c1d146c

fix the batch size to 16 and select three different sequence length distributions: constant (1024), uniform (512 to 1024) and skewed (Zipf distribution with average length 1024). For prefill kernels, we enabled causal masking because it's a common setting in LLM serving.

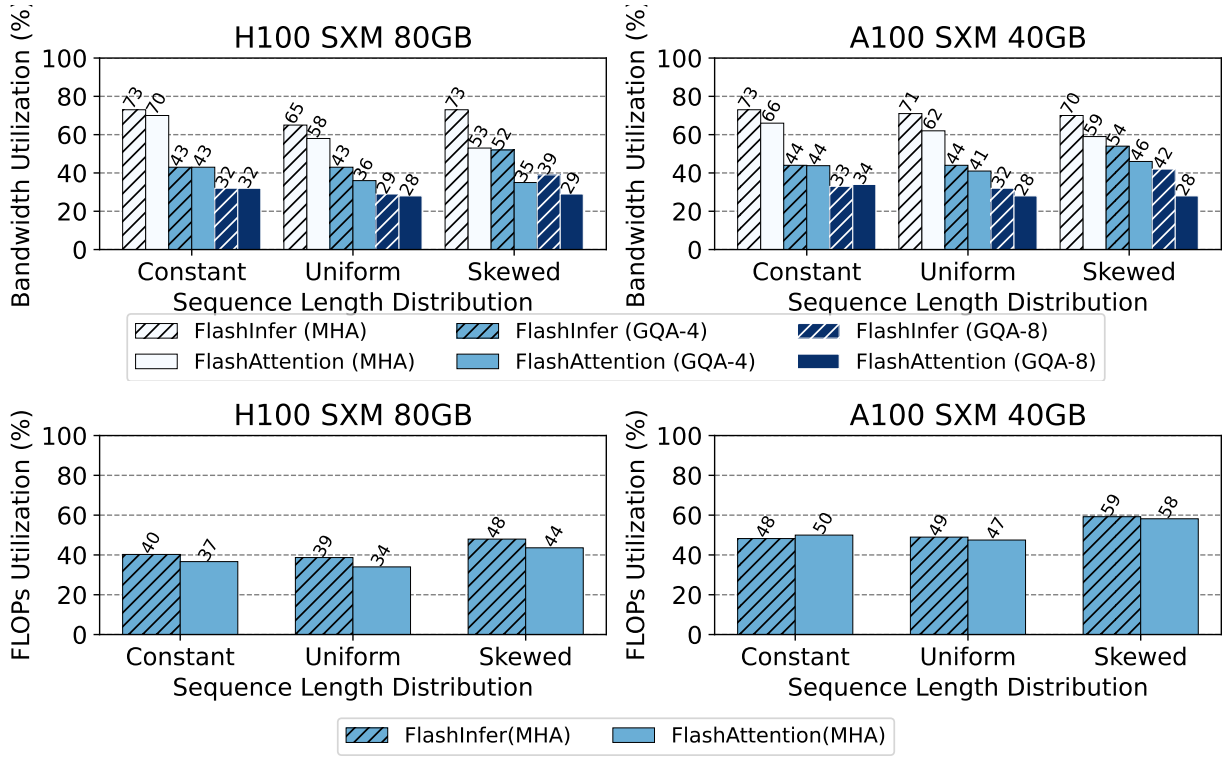

Figure 4.9: Achieved bandwidth and FLOPs utilizations (the higher the better) for decode (top) and prefill (down) kernels.

Figure 4.9 shows the achieved bandwidth and FLOPs utilization for decode and prefill kernels. FlashInfer's kernel significantly outperforms FlashAttention kernels in uniform and skewed sequence length distributions because of our load-balanced dynamic scheduler (section 4.2.3). FlashInfer's decode attention outperforms FlashAttention kernels because our versatile tile size selection (section 4.2.2) and FlashAttention use suboptimal tile size for decoding.

4.4 Overhead of Sparse Gathering

In Section 4.2.2, we detailed the design of FlashInfer's sparse loading module, which transfers sparse rows from global memory into contiguous shared memory. Here, we measure the performance overhead associated with sparse gathering in FlashInfer for both *decode* and *prefill* kernels.

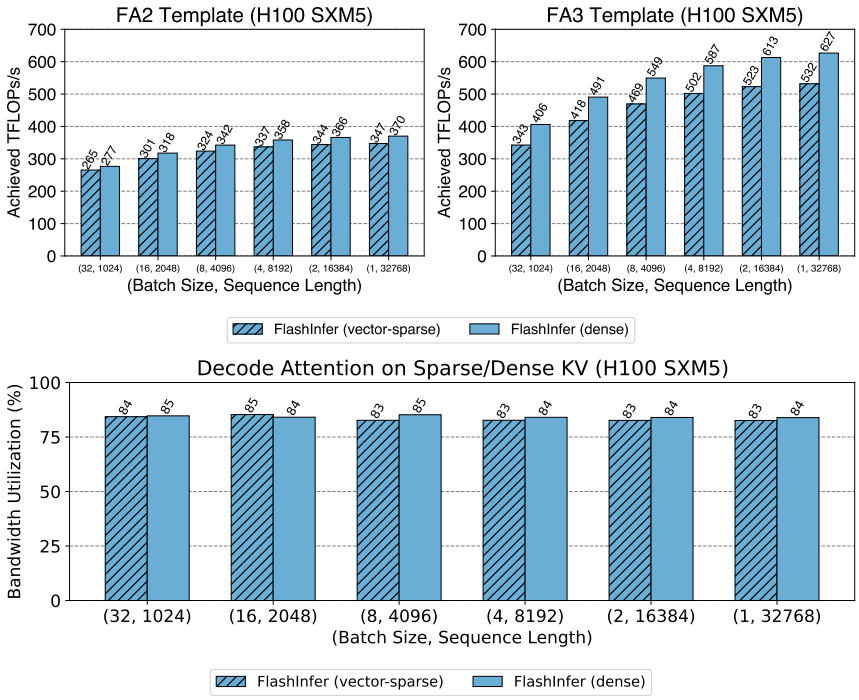

Figure 4.10: **Top:** Achieved TFLOPs/s for (causal) prefill attention kernels on FA2/FA3 templates with both dense/sparse KV-Cache. **Bottom:** Achieved bandwidth utilization for decode attention kernels for both dense/sparse KV-Cache. We use PageAttention with page size 1 (vector-sparse) for sparse KV-Cache. The x-axis shows various batch sizes and sequence lengths.

Figure 4.10 compares achieved throughput in both prefill and decode kernels for sparse and dense (contiguous) KV-Cache. For the prefill kernels, we measure the *causal* attention scenario, which is common in LLM serving. For contiguous KV-Cache, We use the variable-

length RaggedTensor prefill attention API[9]. For sparse KV-Cache, we use the PagedKVCache prefill attention API[10].

The number of query heads and KV heads are both fixed at 32, head dimension is set to 128. We vary batch size and sequence length to measure the achieved throughput. For decode kernels, the performance gap between sparse and dense KV-Cache is negligible (within 1%). For prefill kernels, there is approximately a 10% performance gap.

Note that dense attention in the FA3 template uses TMA instructions (Tensor Memory Access) for key/value loading, which is unavailable for sparse gathering because Hopper Architecture's TMA only supports fixed-stride accesses, whereas sparse gathering requires arbitrary row indices. Consequently, sparse gathering on FA3 relies on Ampere-style asynchronous copy instructions and manual pointer arithmetic. This approach consumes more registers and necessitates smaller KV-tile size to avoid register spilling, leading to a slightly larger performance gap. By contrast, in the FA2 template (where both sparse and dense use Ampere's async-copy), the gap is smaller because the same tile size is used.

When the block column size in a block-sparse matrix is large (e.g., 128 or greater), TMA can be used for sparse gathering since each TMA instruction operates within a single block with fixed stride. We leave this optimization for future work. However, increasing the block column size reduces the flexibility of the block-sparse format, which might not be suitable for all use cases.

4.4.1 Customizability for Long-Context Inference

In this section, we demonstrate how FlashInfer's customized attention kernels significantly accelerate LLM inference. We focus on Streaming-LLM [176], a recent algorithm capable of million-token inference with constant GPU memory usage. While Streaming-LLM requires

[9] `https://docs.flashinfer.ai/api/prefill.html#flashinfer.prefill.BatchPrefillWithRaggedKVCacheWrapper`

[10] `https://docs.flashinfer.ai/api/prefill.html#flashinfer.prefill.BatchPrefillWithPagedKVCacheWrapper`

specialized attention kernels for optimal performance, particularly a fused kernel combining RoPE [149] with attention, FlashInfer can generate such fused kernels with merely 20 additional lines of code for query/key transformations. We compare the performance of FlashInfer-generated fused kernels against un-fused kernels (both FlashInfer's and FlashAttention's) and quantify the end-to-end latency reduction achieved by integrating FlashInfer kernels into StreamingLLM.

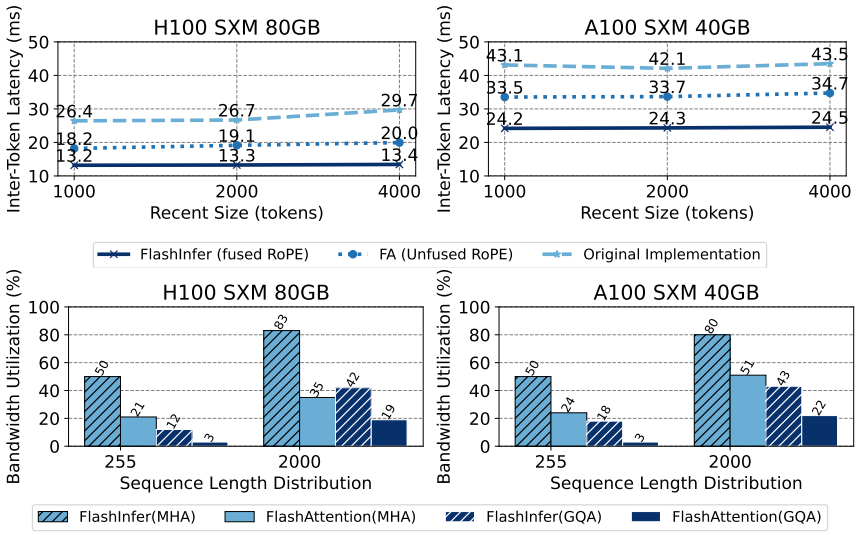

Figure 4.11: Top: End-to-end latency of Streaming-LLM with FlashInfer fused and FlashAttention's unfused kernels, original implementation is included. Down: bandwidth utilization of FlashInfer fused RoPE kernel compared to FlashAttention's unfused kernel.

For end-to-end performance, we run Vicuna-13B [31] inference on MT-Bench [190] dataset and measure the inter-token-latency (ITL) of Streaming-LLM with and without FlashInfer kernels. Figure 4.11 show the ITL of Streaming-LLM with and without FlashInfer fused kernels on our optimized implementation of Streaming-LLM (we noticed that the original implementation is sub-optimal and have unnecessary overheads). FlashInfer's fused kernel can yield $28-30\%$ latency reduction under different settings (by changing the recent window size of Streaming-LLM). Original implementation is included as a baseline reference. We

also show the kernel-level performance comparison between FlashInfer's fused RoPE kernel and the combination of FlashAttention's RoPE kernel and FlashAttention's attention kernel. FlashInfer's fused RoPE kernel achieves 1.6-3.7x higher bandwidth utilization compared to not fusing attention with RoPE, which highlights the importance of customizability of attention kernels.

4.4.2 Parallel-Generation Performance

In this section, we illustrate how the composable formats of FlashInfer can enhance parallel decoding. With parallel generation emerging as a significant task in LLM serving, it offers great utility in LLM agents. The OpenAI API provides an `"n"` parameter[11] to facilitate the generation of multiple tokens simultaneously. As shared prefixes often exist, prefix-caching can significantly boost the efficiency of parallel generation. The composable formats found in FlashInfer (see Section 4.2.1) allow for the decoupling of attention computation between the shared prefix and the subsequent suffix, which can be leveraged to expedite parallel decoding.

We implemented composable formats within MLC-Engine [100] under a prefix-caching configuration and assessed the performance during parallel generation. Evaluations were conducted on the Llama 3.1 models with 8B and 70B parameters [46] using the ShareGPT dataset. With a fixed request rate of 16, we varied the number of parallel tokens over the set $1, 2, 4, 8, 16, 32, 64$, comparing these results against MLC-Engine configurations where composable formats were disabled. Figure 4.12 presents the ITL (Inference Time Latency) and TTFT (Time To First Token) results for MLC-Engine both with and without composable formats.

For moderate levels of parallel generation ($4 \leq n \leq 32$), FlashInfer's composable formats yield consistent speedups for both ITL and TTFT. Peak speedups occur at $n = 4$, where ITL decreases by 13.73% for the 8B model and 17.42% for the 70B model, while TTFT is

[11] `https://platform.openai.com/docs/api-reference/chat/create`

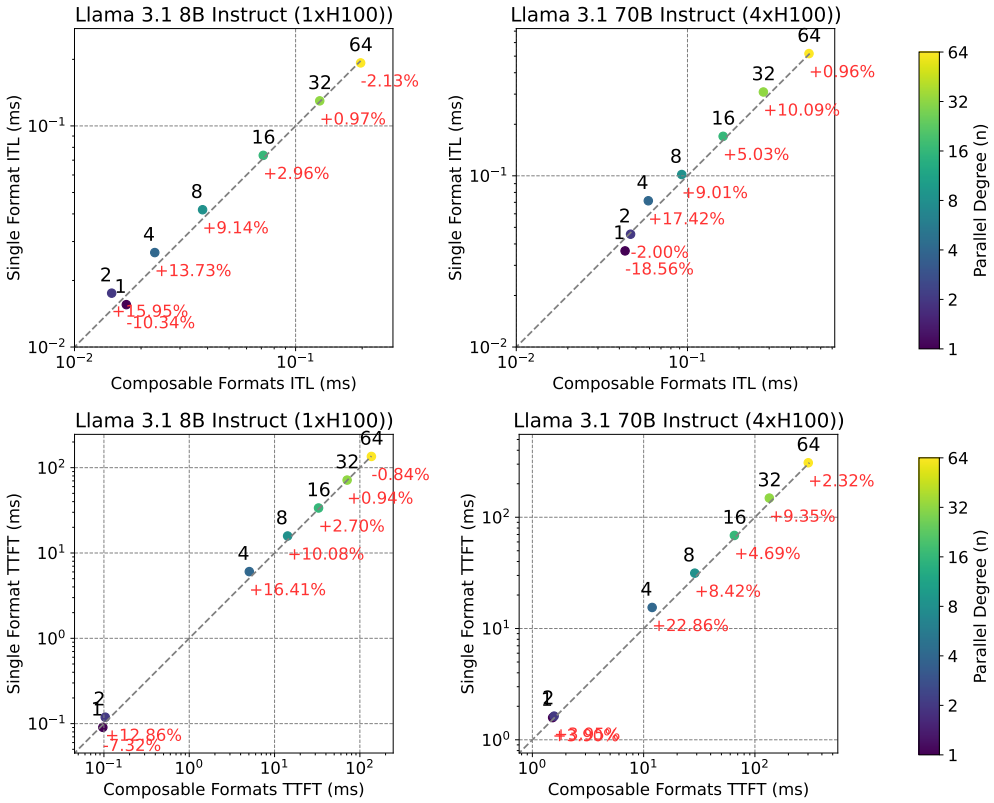

Figure 4.12: ITL and TTFT of MLC-Engine with and without composable formats during parallel generation, the x-axis refers to composable formats performance, and the y-axis refers to single format performance, if a point is above the diagonal line, it means composable formats outperform single format. Different parallel generation n are shown in different colors.

reduced by 16.41% for the 8B model and 22.86% for the 70B model. Smaller values of n do not benefit from composable formats due to insufficient increase in block size. For larger n, the computation ceases to be dominated by attention processes (especially in the case of ShareGPT with its short sequence length), causing the advantage of composable formats to plateau.

4.4.3 Comparison with FlexAttention

We compare FlashInfer and FlexAttention [61] on different attention variants using the AttentionGym [124] benchmark on NVIDIA H100 80GB SXM. We evaluated with batch size 16, number of heads 16 and head dim 128, the CUDA version and the Triton version were fixed to 12.4 and 3.2, respectively. Tables 4.1 to 4.4 show the performance of FlashInfer and FlexAttention in TFLOPS/s, where higher numbers mean better performance. Across all four scenarios and a range of sequence lengths, FlashInfer consistently outperforms FlexAttention, with especially large gains at longer sequence lengths. The better performance is mainly due to the usage of Hopper microarchitecture's advanced features (such as warp specialization and TMA), and CUTLASS's fine-grained resource control (at register-level rather than tile-level) over Triton. Note that these gaps will be alleviated once Triton fully supports these features.

Table 4.1: Causal Attention

Seq Length	FlexAttention	FlashInfer
512	209.11	**250.454**
1024	294.53	**406.554**
2048	376.90	**487.236**
4096	421.00	**548.388**
8192	441.26	**587.903**
16384	453.57	**612.259**

4.4.4 Evaluation of Shared-Prefix Attention Kernels

We measure shared-prefix attention kernels with suffix length 128. Table 4.5 shows the kernel latency under different shared prefix lengths, scenarios and batch sizes, where numbers are in microseconds (us), and "composable" means composable format while "single" means single

Table 4.2: Attention with Logits SoftCap

Seq Length	FlexAttention	FlashInfer
512	241.51	**336.487**
1024	327.50	**409.534**
2048	379.57	**468.769**
4096	403.39	**489.667**
8192	407.82	**515.573**
16384	409.89	**520.935**

format. The composable format benefits long prefixes (e.g., 32k) and large batch sizes (e.g., 64). However, these speedups do not always yield proportional end-to-end gains because real-world shared prefix sizes tend to be smaller.

4.4.5 Ablation Study on Variable Sequence Length and load-balancing scheduler

We conduct ablations on the effect of load-balancing scheduler (Section 4.2.3). Table 4.6 and 4.7 show the results for Llama 3.1-8B-Instruct running on an NVIDIA H100 SXM5 GPU with SGLang [192] + FlashInfer (with and without load-balancing scheduler). We evaluate the inter-token latency (ITL, ms) and time-to-first-token (TTFT, ms) with three datasets: ShareGPT, variable sequence length with input lengths sampled from $U(512, 2048)$ and output fixed at 256, and variable sequence length with input lengths sampled from $U(4096, 16384)$ and output fixed at 256. "RR" in the tables means request rate.

4.4.6 vLLM Integration Evaluation

We integrate FlashInfer to vLLM and compare with the default backend with a fixed request rate of 16, reporting throughput (tokens/s), inter-token latency (ITL, ms), and time-to-first-token (TTFT, ms) in Table 4.8. FlashInfer reduces ITL by aroudn 13% using fp8 KV-cache,

Table 4.3: ALiBi Bias [122]

Seq Length	FlexAttention	FlashInfer
512	253.22	**403.899**
1024	344.70	**500.220**
2048	406.14	**535.498**
4096	426.13	**561.324**
8192	436.35	**573.493**
16384	434.86	**578.005**

but heavy Python overhead in our vLLM integration (e.g. array operations) at host side causes minor regressions with bf16. Our future optimizations will address these in C++ and move the scheduler to device.

4.4.7 Fine-Grained Block-Sparsity Evaluation

FlashInfer supports fine-grained block-sparse matrices, which is useful in many KV-Cache pruning algorithms. We measure the kernel performance on Quest [153], a state-of-the-art long-context modeling algorithm, which uses fine-grained sparsity in KV-Cache. We compared the batch decoding attention kernel in Quest using FlashInfer and compared its performance to PyTorch SDPA and FlexAttention on an NVIDIA H100 SXM5 GPU with the configuration (block size 16, num_qo_heads 32, num_kv_heads 32, head_dim 128). All latency values reported are in microseconds (us).

As shown in Table 4.9 to 4.11, FlashInfer demonstrates a considerable performance advantage, achieving up to a 20x speedup for long sequence lengths. Currently FlexAttention relies on large block templates, while FlashInfer employs a sparse-row gathering strategy to leverage dense tensor cores for small block sizes. This design choice supports fine-grained KV-cache pruning.

Table 4.4: Sliding Window (window size = 1024)

Seq Length	FlexAttention	FlashInfer
512	206.51	**236.363**
1024	292.25	**374.108**
2048	350.91	**381.464**
4096	368.45	**384.998**
8192	373.25	**384.514**
16384	367.91	**380.506**

Table 4.5: Latency of Shared-Prefix Attention Kernels

Shared Prefix Length	Composable (BS=16)	Single (BS=16)	Composable (BS=64)	Single (BS=64)
1024	**45.17**	46.52	87.86	130.49
8192	**88.67**	226.57	125.76	931.75
32768	**217.42**	945.67	254.54	4090

4.5 Related Work

4.5.1 Attention Optimizations

Multi-Head Attention (MHA) [161] faces computational and IO challenges. FasterTransformer [106] reduces global memory footprint via Fused Multi-Head Attention (FMHA), but doesn't scale to long contexts because shared memory usage is linear to sequence length. ByteTransformer [186] optimizes FMHA on variable-length input. FlashAttention [40] uses online-softmax [98] trick to reduce the shared memory footprint to constant size, enabling long

Table 4.6: Load-balancing Scheduler Ablation Study (ITL)

Scenario	w/ Load-Balancing	w/o Load-Balancing	Triton
ShareGPT (RR=16)	**8.96**	9.16	9.36
$U(512, 2048)$ (RR=8)	**8.21**	8.42	8.49
$U(4096, 16384)$ (RR=1)	**8.63**	13.89	11.08

Table 4.7: Load-balancing Scheduler Ablation Study (TTFT)

Scenario	w/ Load-Balancing	w/o Load-Balancing	Triton
ShareGPT (RR=16)	**39.05**	39.42	52.92
$U(512, 2048)$ (RR=8)	**66.78**	67.38	68.48
$U(4096, 16384)$ (RR=1)	**411.02**	421.60	566.30

contexts. FlashAttention2&3 [39, 141] further optimizes FlashAttention by improving loop structure and overlapping softmax and GEMM. FlashDecoding [41] applies Split-K to decode attention kernels. LeanAttention [136] uses StreamK [113] to reduce wave-quantization [108] in attetnion (with fixed sequence length). FlashInfer extends the FlashAttention2&3 template to support sparse attention kernels, while using StreamK-like optimizations on variable length sequences. Nanoflow [196] introduces horizontal fusion of GEMM, attention, and communication operations, while POD-Attention [77] focuses on optimizing chunked-prefill attention. The JIT compilation framework of FlashInfer can be extended to generate kernels supporting these fusion techniques. FlashDecoding++ [65] leverages attention scale statistics

Table 4.8: vLLM Integration Evaluation

Backend	Throughput	Median ITL	Median TTFT
Default (bf16)	6062.89	10.42	35.85
FlashInfer (bf16)	6065.41	10.63	36.60
Default (e4m3)	6015.86	12.56	39.74
FlashInfer (e4m3)	6020.32	10.92	37.93

Table 4.9: FlashInfer Fine-Grained Sparsity Latency (us)

	page_budget			
seq_len	64	128	256	512
4096	20.299	30.361	44.383	44.430
8192	22.273	28.603	44.928	68.194
16384	20.485	28.678	44.677	68.700
32768	22.371	28.700	44.988	68.478

to predefine a unified max value. This process converts attention composition (section 4.1.2) to summation, enabling TMA Store Reduce [36] to asynchronously updating global *attention state*s, it's orthogonal to FlashInfer's contribution and we leave it for future work.

Recent works like RelayAttention [197], Hydragen [76], ChunkAttention [181], and Parrot [90] explore shared prefix decoding attention but require separate KV-Cache management for prefixes and suffixes. In contrast, FlashInfer's composable formats support multi-level, multiple-prefix decoding with unified page table management, enabling seamless integration into LLM serving frameworks without modifying memory management modules.

Table 4.10: PyTorch SDPA Fine-Grained Sparsity Latency (us)

	page_budget			
seq_len	64	128	256	512
4096	287.684	288.904	287.715	287.807
8192	474.631	474.508	474.683	473.070
16384	857.319	857.570	857.094	857.728
32768	1711.955	1711.621	1713.093	1711.709

Table 4.11: FlexAttention Fine-Grained Sparsity Latency (us)

	page_budget			
seq_len	64	128	256	512
4096	1100.349	1097.356	1073.753	1071.797
8192	1092.695	1099.100	1078.081	1074.886
16384	1109.817	1101.535	1077.639	1076.859
32768	1169.109	1187.395	1176.332	1174.502

4.5.2 Sparse Optimizations on GPUs

FusedMM [127] explores Sparse-dense Matrix Multiplication (SpMM) fusion, though it omits softmax computation, limiting direct applicability for accelerating attention. FusedGAT [187] explore Graph Attention Networks (GAT) kernel fusion, SAR [102] serializes Sparse Attention aggregation, akin to FlashAttention, neither work explores using Tensor Cores. Blocksparse library [57] implements BSR GEMM with tensor cores. Octet Tiling [27], TC-GNN [172] and Magicube [89] propose vector sparse formats to leverage Tensor Cores effectively. FlashInfer improves upon these to support any block sizes (b_r, b_c) in FlashAttention.

4.5.3 Attention Compilers

FlexAttention [61] provides a user-friendly interface for programming attention variants, compiling them into block-sparse flashattention implemented in Triton [159]. It uses PyTorch Compiler [7] to automatically generate backward passes. FlashInfer expands the FlexAttention's programming interface to support query/key transformations, and focus on vector-sparsity and load-balancing for LLM serving. FlashInfer generates CUDA code instead of Triton because Triton still underperform CUDA & CUTLASS in many use cases. FlashInfer can act as a backend for FlexAttention in forward pass. Mirage [173] optimizes tiling strategies for GEMM and FlashAttention using a probabilistic equivalence verifier, relying on Triton and CUTLASS for code generation. However, it lacks support for variable length and sparse data structures, and doesn't include safe-softmax, unlike FlashInfer, which is directly applicable to LLM serving.

4.5.4 LLM Serving Systems

Orca [184] introduces continuous batching for enhanced throughput. PagedAttention [82] uses a Page Table for KV-Cache management. Sarathi-serve [3] improves efficiency by piggybacking decode operations with chunked-prefill, while SGLang [192] utilizes RadixTree for better prefix-caching and KV-management. FlashInfer provides a unified solution for these attention mechanisms through block-sparse attention kernels. vAttention [121] shows that GPU virtual memory can manage address translation in PageAttention without special kernels. Yet, challenges like dynamic KV-Cache sparsity persist, as seen in Quest [153]. Here, FlashInfer's block sparse kernel remains effective. Additionally, FlashInfer can be combined with vAttention by generating kernels for contiguous KV-Cache storage.

4.6 Discussions

Currently, FlashInfer only supports the forward pass for attention computation. To extend FlashInfer and apply it to training would require developing customizable backward attention

kernel templates, which we plan to explore in future work. Regarding the generality of FlashInfer, our approach decouples computation from tile scheduling, allowing for diverse tiling strategies such as FlashDecoding [65] and Lean Attention [136] by delegating scheduling to a runtime-scheduler rather than embedding it within the attention template. This design generalizes scheduling algorithms, as detailed in Algorithm 1, for load-balancing and wave-quantization reduction while targeting optimal GPU performance across architectures (e.g., FlashAttention2 [39] for Turing/Ampere/Ada and FlashAttention3 [141] for Hopper). Although frameworks like Triton [159] offer GPU-agnostic interfaces, they often lag in adopting new hardware features; our template design space, expressed as $f_{\text{epilogue}}(\text{scan}(f_{\text{logits}}(f_q(Q) \cdot f_k(K))) \cdot f_v(V))$, covers most attention functions, including recent variants such as Multi-head Latent Attention (MLA) [42] and the intra-attention component of Linear Attention [180].

4.6.1 The Choice of Backend

For NVIDIA GPUs, we build FlashInfer on top of CUDA/CUTLASS [156] instead of Triton [159] for the following reasons:

1. **Advanced NVIDIA GPU Features.** CUTLASS supports specialized GPU capabilities such as warp-specialization [111] and TMA instructions [107], which are experimental or unsupported in Triton at this moment.

2. **Fine-Grained Kernel Optimization.** While Triton provides tile-level abstractions, CUDA/CUTLASS affords finer control over thread-level registers. This flexibility simplifies incorporating low-level optimizations (e.g., PTX intrinsics) directly into our JIT templates, which is more challenging in Triton.

Our load-balancing scheduler design (Section 4.2.3) is largely backend-agnostic, allowing us to potentially integrate Triton in future versions of FlashInfer and to adapt our approach to other hardware platforms.

4.6.2 Overlap of Attention with Other Operations

Nanoflow [196] overlaps GEMM, attention, and inter-device communication in separate CUDA streams, assigning a fixed number of SMs to each operation. In FlashInfer, this SM number can be provided by the user through the plan functions, and the FlashInfer load-balancing scheduler will allocate tiles accordingly.

4.7 Conclusion and Future Work

In this paper, we present FlashInfer, an versatile and efficient attention engine for LLM serving. We propose a unified block-sparse storage and composable formats for memory efficiency, JIT compilation for customization and load-balanced scheduler for input dynamism. We evaluate FlashInfer's performance across diverse inference scenarios, showing strong performance in kernel-level and end-to-end LLM serving metrics. In the future, we plan to explore compiling higher-level DSLs [61, 173] to attention specifications in FlashInfer, as well as code generation to other backends [115, 147, 159]. The FlashInfer project is open source and available at `https://github.com/flashinfer-ai/flashinfer`, and has been deployed at scale in production-level systems.

Chapter 5

CONCLUSIONS

This dissertation has presented a co-design approach to compiler and runtime systems for the efficient deployment of next-generation generative AI workloads. In this chapter, we summarize the main contributions and outline promising directions for future research.

5.1 Thesis Summary

We began by motivating the need for new system designs to support the growing demands of generative AI, particularly the challenges posed by sparse computation and the need for high-performance inference in large models. Existing systems often fall short in these areas, motivating our exploration of compiler-runtime co-design.

The first major contribution, **SparseTIR**, introduced a novel intermediate representation and compiler framework tailored for sparse tensor computations. SparseTIR enables composable sparse formats and flexible scheduling, facilitating the efficient generation of sparse kernels across diverse hardware backends. Through a series of case studies and benchmarks, we demonstrated both the expressiveness and performance advantages of this approach.

Building on this foundation, the **FlashInfer** section detailed the design and implementation of a high-performance inference kernel engine for large language models. We described key algorithmic and system-level optimizations–including JIT compilation, efficient runtime design–that together enable low-latency, high-throughput attention computation. Furthermore, we showed how FlashInfer leverages SparseTIR abstractions to support hierarchical KV-Cache for structured generation.

By integrating these contributions, this dissertation demonstrates how compiler and runtime co-design can unlock new capabilities for generative AI workloads, delivering both

programmability and performance. The results establish a foundation for future research and development in efficient AI systems.

5.2 Future Work

FlashInfer has evolved from an attention library into a comprehensive suite of inference kernels for LLMs and video generation models [22]. The library now encompasses four main categories of kernels: **MoE** (including grouped GEMM with customized epilogues and routing kernels), **Communication** (such as allreduce, all-to-all, and fused compute-communication kernels), **Attention** (the attention engine described in this work), and **Logits Processor** (for sampling and constraint decoding) [178].

A notable trend in efficient LLM inference is the adoption of MegaKernels [73, 146]. Traditional kernel-by-kernel execution introduces global barriers and significant overhead, particularly detrimental to low-latency inference, and often results in low occupancy for individual kernels. MegaKernels address these issues by enabling task-level programming models and fine-grained (SM- or warp-level) task execution. Systems such as ThunderKittens and Mirage implement runtime support for tracking task dependencies and scheduling. Looking ahead, we plan to explore the industrial application of MegaKernels within FlashInfer, including the development of compilers and debuggers to facilitate MegaKernel development, as well as user-friendly programming interfaces for seamless framework integration.

Another promising direction is the creation of self-evolving LLM kernel libraries. By collecting traces, problem specifications, and shape information from LLM inference workloads, we are building the FlashInfer benchmark suite. This resource will support performance diagnosis, kernel selection, and optimization, and will also serve as a foundation for LLM-driven, agentic CUDA kernel development [24, 114].

BIBLIOGRAPHY

[1] Block compressed row format (bsr) | scipy lecture notes. `https://scipy-lectures.org/advanced/scipy_sparse/bsr_matrix.html`.

[2] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. Learning to optimize halide with tree search and random programs. *ACM Trans. Graph.*, 38(4):121:1–121:12, 2019.

[3] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in llm inference with sarathi-serve. *Proceedings of 18th USENIX Symposium on Operating Systems Design and Implementation, 2024, Santa Clara*, 2024.

[4] Nawaaz Ahmed, Nikolay Mateev, Keshav Pingali, and Paul Stodghill. A Framework for Sparse Matrix Code Synthesis from High-level Specifications. In Jed Donnelley, editor, *Proceedings Supercomputing 2000, November 4-10, 2000, Dallas, Texas, USA. IEEE Computer Society, CD-ROM*, page 58. IEEE Computer Society, 2000.

[5] Peter Ahrens, Fredrik Kjolstad, and Saman Amarasinghe. Autoscheduling for sparse tensor algebra with an asymptotic cost model. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 269–285, New York, NY, USA, 2022. Association for Computing Machinery.

[6] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. GQA: training generalized multi-query transformer models from

multi-head checkpoints. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 4895–4901. Association for Computational Linguistics, 2023.

[7] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 929–947, New York, NY, USA, 2024. Association for Computing Machinery.

[8] Riyadh Baghdadi, Abdelkader Nadir Debbagh, Kamel Abdous, Fatima-Zohra Benhamida, Alex Renda, Jonathan Elliott Frankle, Michael Carbin, and Saman P. Amarasinghe. TIRAMISU: A polyhedral compiler for dense and sparse deep learning. *CoRR*, abs/2005.04091, 2020.

[9] J. Behley, M. Garbade, A. Milioto, J. Quenzel, S. Behnke, C. Stachniss, and J. Gall. SemanticKITTI: A Dataset for Semantic Scene Understanding of LiDAR Sequences. In *Proc. of the IEEE/CVF International Conf. on Computer Vision (ICCV)*, 2019.

[10] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *CoRR*, abs/2004.05150, 2020.

[11] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *CoRR*, abs/2004.05150, 2020.

[12] Aart Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. Compiler support for sparse tensor computations in mlir. *ACM Trans. Archit. Code Optim.*, 19(4), sep 2022.

[13] Aart J. C. Bik. *Compiler Support for Sparse Matrix Computations*. PhD thesis, Leiden University, 1996.

[14] Aart J. C. Bik and Harry A. G. Wijshoff. Compilation techniques for sparse matrix computations. In Yoichi Muraoka, editor, *Proceedings of the 7th international conference on Supercomputing, ICS 1993, Tokyo, Japan, July 20-22, 1993*, pages 416–424. ACM, 1993.

[15] Aart J. C. Bik and Harry A. G. Wijshoff. Nonzero structure analysis. In *Proceedings of the 8th International Conference on Supercomputing*, ICS '94, page 226–235, New York, NY, USA, 1994. Association for Computing Machinery.

[16] Aart J. C. Bik and Harry A. G. Wijshoff. Automatic data structure selection and transformation for sparse matrix computations. *IEEE Trans. Parallel Distributed Syst.*, 7(2):109–126, 1996.

[17] A.J.C. Bik and H.A.G. Wijshoff. Advanced compiler optimizations for sparse computations. *Journal of Parallel and Distributed Computing*, 31(1):14–24, 1995.

[18] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In Friedhelm Meyer auf der Heide and Michael A. Bender, editors, *SPAA 2009: Proceedings of the 21st Annual ACM Symposium on Parallelism in Algorithms and Architectures, Calgary, Alberta, Canada, August 11-13, 2009*, pages 233–244. ACM, 2009.

[19] Aydin Buluç and John R. Gilbert. On the representation and multiplication of hypersparse matrices. In *22nd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, Miami, Florida USA, April 14-18, 2008*, pages 1–11. IEEE, 2008.

[20] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. Medusa: Simple LLM inference acceleration framework with multiple decoding heads. *CoRR*, abs/2401.10774, 2024.

[21] Shijie Cao, Chen Zhang, Zhuliang Yao, Wencong Xiao, Lanshun Nie, Dechen Zhan, Yunxin Liu, Ming Wu, and Lintao Zhang. Efficient and effective sparse lstm on fpga with bank-balanced sparsity. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, page 63–72, New York, NY, USA, 2019. Association for Computing Machinery.

[22] Luis Ceze, Zihao Ye, Tianqi Chen, Vinod Grover, Mehdi Amini, and Nick Comly. Run high-performance llm inference kernels from nvidia using flashinfer. `https://developer.nvidia.com/blog/run-high-performance-llm-inference-kernels-from-nvidia-using-flashinfer/`, June 2025. NVIDIA Technical Blog.

[23] Beidi Chen, Tri Dao, Kaizhao Liang, Jiaming Yang, Zhao Song, Atri Rudra, and Christopher Ré. Pixelated butterfly: Simple and efficient sparse training for neural network models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022.

[24] Terry Chen, Bing Xu, and Kirthi Devleker. Automating gpu kernel generation with deepseek-r1 and inference time scaling. NVIDIA Technical Blog, 2025. Accessed: 2025-08-22.

[25] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen,

Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, October 2018. USENIX Association.

[26] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 3393–3404, Red Hook, NY, USA, 2018. Curran Associates Inc.

[27] Zhaodong Chen, Zheng Qu, Liu Liu, Yufei Ding, and Yuan Xie. Efficient tensor core-based GPU kernels for structured sparsity under reduced precision. In Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin, editors, *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, page 78. ACM, 2021.

[28] Zhuoming Chen, Avner May, Ruslan Svirschevski, Yuhsun Huang, Max Ryabinin, Zhihao Jia, and Beidi Chen. Sequoia: Scalable, robust, and hardware-aware speculative decoding. *CoRR*, abs/2402.12374, 2024.

[29] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. Sympiler: transforming sparse matrix codes by decoupling symbolic analysis. In Bernd Mohr and Padma Raghavan, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017, Denver, CO, USA, November 12 - 17, 2017*, page 13. ACM, 2017.

[30] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. ParSy: inspection and transformation of sparse matrix computations for parallelism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*, pages 62:1–62:15. IEEE / ACM, 2018.

[31] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March 2023.

[32] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *CoRR*, abs/1904.10509, 2019.

[33] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Format abstraction for sparse tensor algebra compilers. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018.

[34] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Automatic generation of efficient sparse tensor format conversion routines. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 823–838, New York, NY, USA, 2020. Association for Computing Machinery.

[35] Christopher Choy, JunYoung Gwak, and Silvio Savarese. 4d spatio-temporal convnets: Minkowski convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3075–3084, 2019.

[36] Colfax. Cutlass tutorial: Mastering the nvidia tensor memory accelerator (tma), 2024.

[37] NVIDIA Corporation. cusparse :: Cuda toolkit documentation v11.7.1. `https://docs.nvidia.com/cuda/cusparse/index.html`, 2022.

[38] Guohao Dai, Guyue Huang, Shang Yang, Zhongming Yu, Hengrui Zhang, Yufei Ding, Yuan Xie, Huazhong Yang, and Yu Wang. Heuristic adaptability to input dynamics for spmm on gpus. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, DAC '22, page 595–600, New York, NY, USA, 2022. Association for Computing Machinery.

[39] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *CoRR*, abs/2307.08691, 2023.

[40] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

[41] Tri Dao, Daniel Haziza, Francisco Massa, and Grigory Sizov. Flash-decoding for long-context inference, 2023.

[42] DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, Hao Zhang, Hanwei Xu, Hao Yang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jin Chen, Jingyang Yuan, Junjie Qiu, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruizhe Pan, Runxin Xu, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Size Zheng, Tao Wang, Tian Pei, Tian Yuan, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaosha Chen, Xiaotao Nie, and Xiaowen Sun. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *CoRR*, abs/2405.04434, 2024.

[43] Tensorflow Developers. Xla: Optimizing compiler for machine learning | tensorflow. `https://www.tensorflow.org/xla`, 2018.

[44] dgSPARSE team. dgsparse library. `https://github.com/dgSPARSE/dgSPARSE-Library`, 2021.

[45] Adhitha Dias, Kirshanthan Sundararajah, Charitha Saumya, and Milind Kulkarni. Sparselnr: Accelerating sparse tensor computations using loop nest restructuring. In *Proceedings of the 36th ACM International Conference on Supercomputing*, ICS '22, New York, NY, USA, 2022. Association for Computing Machinery.

[46] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurélien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Rozière, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Graeme Nail, Grégoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel M. Kloumann, Ishan Misra, Ivan Evtimov, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, and et al. The llama 3 herd of models. *CoRR*, abs/2407.21783, 2024.

[47] Iain S. Duff. *The Use of Vector and Parallel Computers in the Solution of Large Sparse Linear Equations*, pages 331–348. Birkhäuser Boston, Boston, MA, 1987.

[48] Iain S Duff, Albert M Erisman, and John K Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, Inc., USA, 1986.

[49] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

[50] Pratik Fegade, Tianqi Chen, Phillip B. Gibbons, and Todd C. Mowry. The cora tensor compiler: Compilation for ragged tensors with minimal padding. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, 2022.

[51] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. Tensorir: An abstraction for automatic tensorized program optimization. In Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift, editors, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 804–817. ACM, 2023.

[52] Yifan Feng, Haoxuan You, Zizhao Zhang, Rongrong Ji, and Yue Gao. Hypergraph neural networks. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 3558–3565. AAAI Press, 2019.

[53] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch

Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[54] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse gpu kernels for deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[55] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 1263–1272. JMLR.org, 2017.

[56] Alan Gray. Getting Started with CUDA Graphs | NVIDIA Technical Blog. `https://developer.nvidia.com/blog/cuda-graphs/`, 2019. [Accessed 19-10-2024].

[57] Scott Gray, Alec Radford, and Diederik P Kingma. Gpu kernels for block-sparse weights. *arXiv preprint arXiv:1711.09224*, 3(2):2, 2017.

[58] Manish Gupta. Mixed-input matrix multiplication performance optimizations. `https://research.google/blog/mixed-input-matrix-multiplication-performance-optimizations/`, January 2024. Google Research Blog, Accessed: 2024-01-26.

[59] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 1025–1035, Red Hook, NY, USA, 2017. Curran Associates Inc.

[60] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[61] Horace He, Driss Guessous, Yanbo Liang, and Joy Dong. Flexattention: The flexibility of pytorch with the performance of flashattention, Aug 2024.

[62] Rawn Henry, Olivia Hsu, Rohan Yadav, Stephen Chou, Kunle Olukotun, Saman Amarasinghe, and Fredrik Kjolstad. Compilation of sparse array programming models. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[63] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *J. Mach. Learn. Res.*, 22:241:1–241:124, 2021.

[64] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 300–314, New York, NY, USA, 2019. Association for Computing Machinery.

[65] Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, kangdi chen, Yuhan Dong, and Yu Wang. Flashdecoding++: Faster large language model inference with asynchronization, flat gemm optimization, and heuristics. In P. Gibbons, G. Pekhimenko, and C. De Sa, editors, *Proceedings of Machine Learning and Systems*, volume 6, pages 148–161, 2024.

[66] E. N. Houstis, J. R. Rice, N. P. Chrisochoides, H. C. Karathanasis, P. N. Papachiou, M. K. Samartzis, E. A. Vavalis, Ko Yang Wang, and S. Weerawarana. //ellpack: A numerical simulation programming environment for parallel mimd machines. *SIGARCH Comput. Archit. News*, 18(3b):96–107, jun 1990.

[67] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing*

Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual, 2020.

[68] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. Taichi: A language for high-performance computation on spatially sparse data structures. *ACM Trans. Graph.*, 38(6), nov 2019.

[69] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. Featgraph: A flexible and efficient backend for graph neural network systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[70] Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. Heterogeneous graph transformer. In Yennun Huang, Irwin King, Tie-Yan Liu, and Maarten van Steen, editors, *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, pages 2704–2710. ACM / IW3C2, 2020.

[71] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[72] Eun-Jin Im, Katherine A. Yelick, and Richard W. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158, 2004.

[73] Zhihao Jia, Xinhao Cheng, Bohan Hou, Yingyi Huang, Jianan Ji, Jinchen Jiang, Hongyi Jin, Ruihang Lai, Shengjie Lin, Xupeng Miao, Gabriele Oliaro, Zihao Ye, Zhihang Zhang, Yilong Zhao, and Tianqi Chen. Compiling llms into a megakernel: A path to low-latency inference. `https://zhihaojia.medium.com/`

compiling-llms-into-a-megakernel-a-path-to-low-latency-inference-cf7840913c17, 2025. Medium Blog Post.

[74] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 47–62, New York, NY, USA, 2019. Association for Computing Machinery.

[75] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*, pages 1–12. ACM, 2017.

[76] Jordan Juravsky, Bradley C. A. Brown, Ryan Saul Ehrlich, Daniel Y. Fu, Christopher Ré, and Azalia Mirhoseini. Hydragen: High-throughput LLM inference with shared prefixes. *CoRR*, abs/2402.05099, 2024.

[77] Aditya K Kamath, Ramya Prabhu, Jayashree Mohan, Simon Peter, Ramachandran Ramjee, and Ashish Panwar. Pod-attention: Unlocking full prefill-decode overlap for faster llm inference. *CoRR*, abs/2410.18038, 2024.

[78] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[79] Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. Tensor algebra compilation with workspaces. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, pages 180–192, Piscataway, NJ, USA, 2019. IEEE Press.

[80] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017.

[81] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman P. Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, 2017.

[82] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 611–626. ACM, 2023.

[83] François Lagunas, Ella Charlaix, Victor Sanh, and Alexander Rush. Block pruning for faster transformers. In *Proceedings of the 2021 Conference on Empirical Methods in*

Natural Language Processing, pages 10619–10629, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[84] Ruihang Lai, Junru Shao, Siyuan Feng, Steven S. Lyubomirsky, Bohan Hou, Wuwei Lin, Zihao Ye, Hongyi Jin, Yuchen Jin, Jiawei Liu, Lesheng Jin, Yaxing Cai, Ziheng Jiang, Yong Wu, Sunghyun Park, Prakalp Srivastava, Jared G. Roesch, Todd C. Mowry, and Tianqi Chen. Relax: Composable abstractions for end-to-end dynamic machine learning. *CoRR*, abs/2311.02103, 2023.

[85] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A compiler infrastructure for the end of moore's law. *CoRR*, abs/2002.11054, 2020.

[86] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. Automatic horizontal fusion for gpu kernels. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '22, page 14–27. IEEE Press, 2022.

[87] Jiajia Li, Bora Uçar, Ümit V. Çatalyürek, Jimeng Sun, Kevin Barker, and Richard Vuduc. Efficient and effective sparse tensor reordering. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, page 227–237, New York, NY, USA, 2019. Association for Computing Machinery.

[88] Shigang Li, Kazuki Osawa, and Torsten Hoefler. Efficient quantized sparse matrix operations on tensor cores. *CoRR*, abs/2209.06979, 2022.

[89] Shigang Li, Kazuki Osawa, and Torsten Hoefler. Efficient quantized sparse matrix operations on tensor cores. In Felix Wolf, Sameer Shende, Candace Culhane, Sadaf R. Alam, and Heike Jagode, editors, *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, November 13-18, 2022*, pages 37:1–37:15. IEEE, 2022.

[90] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. Parrot: Efficient serving of LLM-based applications with semantic variable. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 929–945, Santa Clara, CA, July 2024. USENIX Association.

[91] Chien-Yu Lin, Liang Luo, and Luis Ceze. Accelerating spmm kernel with cache-first edge sampling for graph neural networks. *CoRR*, abs/2104.10716, 2021.

[92] Hao Liu and Pieter Abbeel. Blockwise parallel transformers for large context models. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.

[93] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context. *CoRR*, abs/2310.01889, 2023.

[94] Nikolay Mateev, Keshav Pingali, Paul Stodghill, and Vladimir Kotlyar. Next-generation generic programming and its application to sparse matrix computations. In John Reynders and Alexander V. Veidenbaum, editors, *Proceedings of the 14th international conference on Supercomputing, ICS 2000, Santa Fe, NM, USA, May 8-11, 2000*, pages 88–99. ACM, 2000.

[95] Atefeh Mehrabi, Donghyuk Lee, Niladrish Chatterjee, Daniel J. Sorin, Benjamin C. Lee, and Mike O'Connor. Learning sparse matrix row permutations for efficient spmm on GPU architectures. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2021, Stony Brook, NY, USA, March 28-30, 2021*, pages 48–58. IEEE, 2021.

[96] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming

Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. Specinfer: Accelerating large language model serving with tree-based speculative inference and verification. In Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir, editors, *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, pages 932–949. ACM, 2024.

[97] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart F. Oberman, Mohammad Shoeybi, Michael Y. Siu, and Hao Wu. FP8 formats for deep learning. *CoRR*, abs/2209.05433, 2022.

[98] Maxim Milakov and Natalia Gimelshein. Online normalizer calculation for softmax. *CoRR*, abs/1805.02867, 2018.

[99] Ravi Mirchandaney, Joel H. Saltz, Roger M. Smith, David M. Nicol, and Kay Crowley. Principles of runtime support for parallel processors. In Jacques Lenfant, editor, *Proceedings of the 2nd international conference on Supercomputing, ICS 1988, Saint Malo, France, July 4-8, 1988*, pages 140–152. ACM, 1988.

[100] MLC Community. Optimizing and characterizing high-throughput low-latency LLM inference in MLCEngine, Oct 2024. [Online; accessed August 23, 2025].

[101] Mahdi Soltan Mohammadi, Tomofumi Yuki, Kazem Cheshmi, Eddie C. Davis, Mary W. Hall, Maryam Mehri Dehnavi, Payal Nandy, Catherine Olschanowsky, Anand Venkat, and Michelle Mills Strout. Sparse computation data dependence simplification for efficient compiler-generated inspectors. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 594–609. ACM, 2019.

[102] Hesham Mostafa. Sequential aggregation and rematerialization: Distributed full-batch training of graph neural networks on large graphs. In Diana Marculescu, Yuejie Chi, and Carole-Jean Wu, editors, *Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022*. mlsys.org, 2022.

[103] Sharan Narang, Eric Undersander, and Gregory F. Diamos. Block-sparse recurrent neural networks. *CoRR*, abs/1711.02782, 2017.

[104] Vinh Nguyen, Michael Carilli, Sukru Burc Eryilmaz, Vartika Singh, Michelle Lin, Natalia Gimelshein, Alban Desmaison, and Edward Yang. Accelerating PyTorch with CUDA Graphs. `https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs/`, 2021. [Accessed 19-10-2024].

[105] Israt Nisa, Aravind Sukumaran-Rajam, Sureyya Emre Kurt, Changwan Hong, and P. Sadayappan. Sampled dense matrix multiplication for high-performance machine learning. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, pages 32–41, 2018.

[106] NVIDIA. FasterTransformer. `https://github.com/NVIDIA/FasterTransformer`, 2021.

[107] NVIDIA. Nvidia hopper architecture in-depth, 2022.

[108] NVIDIA. Matrix multiplication background user's guide, 2023.

[109] NVIDIA. NVIDIA TensorRT-LLM, 2023. [Online; accessed August 23, 2025].

[110] NVIDIA. New xqa-kernel provides 2.4x more llama-70b throughput within the same latency budget, 2024.

[111] NVIDIA. Spatial partitioning (also known as warp specialization), 2024.

[112] Thomas C. Oppe and David R. Kincaid. The performance of itpack on vector computers for solving large sparse linear systems arising in sample oil reseervoir simulation problems. *Communications in Applied Numerical Methods*, 3(1):23–29, 1987.

[113] Muhammad Osama, Duane Merrill, Cris Cecka, Michael Garland, and John D. Owens. Stream-k: Work-centric parallel decomposition for dense matrix-matrix multiplication on the GPU. In Maryam Mehri Dehnavi, Milind Kulkarni, and Sriram Krishnamoorthy, editors, *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2023, Montreal, QC, Canada, 25 February 2023 - 1 March 2023*, pages 429–431. ACM, 2023.

[114] Anne Ouyang, Simon Guo, and Azalia Mirhoseini. Kernelbench: Can llms write gpu kernels? Stanford Scaling Intelligence Blog, 2024. Accessed: 2025-08-22.

[115] Guray Ozen. Nvdsl: Simplifying tensor cores with python-driven mlir metaprogramming. In *Efficient Systems for Foundation Models (ES-FoMo) Workshop at ICML 2024*, 2024.

[116] Douglass Stott Parker. *Random butterfly transformations with applications in computational linear algebra*. UCLA Computer Science Department, 1995.

[117] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[118] Gregoire Pichon, Mathieu Faverge, Pierre Ramet, and Jean Roman. Reordering strategy

for blocking optimization in sparse linear solvers. *SIAM Journal on Matrix Analysis and Applications*, 38(1):226–248, 2017.

[119] Ravi Ponnusamy, Joel H. Saltz, and Alok N. Choudhary. Runtime compilation techniques for data partitioning and communication schedule reuse. In Bob Borchers and Dona Crawford, editors, *Proceedings Supercomputing '93, Portland, Oregon, USA, November 15-19, 1993*, pages 361–370. ACM, 1993.

[120] Jeff Pool. Accelerating sparsity in the nvidia ampere architecture. `https://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s22085-accelerating-sparsity-in-the-nvidia-ampere-architecture%E2%80%8B.pdf`, 2020.

[121] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. vattention: Dynamic memory management for serving llms without pagedattention, 2024.

[122] Ofir Press, Noah A. Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022.

[123] William W. Pugh and Tatiana Shpeisman. SIPR: A New Framework for Generating Efficient Code for Sparse Matrix Computations. In Siddhartha Chatterjee, Jan F. Prins, Larry Carter, Jeanne Ferrante, Zhiyuan Li, David C. Sehr, and Pen-Chung Yew, editors, *Languages and Compilers for Parallel Computing, 11th International Workshop, LCPC'98, Chapel Hill, NC, USA, August 7-9, 1998, Proceedings*, volume 1656 of *Lecture Notes in Computer Science*, pages 213–229. Springer, 1998.

[124] PyTorch-Labs. attention-gym. `https://github.com/pytorch-labs/attention-gym`, 2024.

[125] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 519–530, New York, NY, USA, 2013. Association for Computing Machinery.

[126] Md. Khaledur Rahman, Majedul Haque Sujon, and Ariful Azad. Fusedmm: A unified sddmm-spmm kernel for graph embedding and graph neural networks. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 256–266, 2021.

[127] Md. Khaledur Rahman, Majedul Haque Sujon, and Ariful Azad. Fusedmm: A unified sddmm-spmm kernel for graph embedding and graph neural networks. In *35th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2021, Portland, OR, USA, May 17-21, 2021*, pages 256–266. IEEE, 2021.

[128] Jason Ramapuram, Federico Danieli, Eeshan Gunesh Dhekane, Floris Weers, Dan Busbridge, Pierre Ablin, Tatiana Likhomanenko, Jagrit Digani, Zijin Gu, Amitis Shidani, and Russ Webb. Theory, analysis, and best practices for sigmoid self-attention. *CoRR*, abs/2409.04431, 2024.

[129] Petar Ristoski, Gerben Klaas Dirk de Vries, and Heiko Paulheim. A collection of benchmark datasets for systematic evaluations of machine learning on the semantic web. In Paul Groth, Elena Simperl, Alasdair Gray, Marta Sabou, Markus Krötzsch, Freddy Lecue, Fabian Flöck, and Yolanda Gil, editors, *The Semantic Web – ISWC 2016*, pages 186–194, Cham, 2016. Springer International Publishing.

[130] Morgane Rivière, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, Johan

Ferret, Peter Liu, Pouya Tafti, Abe Friesen, Michelle Casbon, Sabela Ramos, Ravin Kumar, Charline Le Lan, Sammy Jerome, Anton Tsitsulin, Nino Vieillard, Piotr Stanczyk, Sertan Girgin, Nikola Momchev, Matt Hoffman, Shantanu Thakoor, Jean-Bastien Grill, Behnam Neyshabur, Olivier Bachem, Alanna Walton, Aliaksei Severyn, Alicia Parrish, Aliya Ahmad, Allen Hutchison, Alvin Abdagic, Amanda Carl, Amy Shen, Andy Brock, Andy Coenen, Anthony Laforge, Antonia Paterson, Ben Bastian, Bilal Piot, Bo Wu, Brandon Royal, Charlie Chen, Chintu Kumar, Chris Perry, Chris Welty, Christopher A. Choquette-Choo, Danila Sinopalnikov, David Weinberger, Dimple Vijaykumar, Dominika Rogozinska, Dustin Herbison, Elisa Bandy, Emma Wang, Eric Noland, Erica Moreira, Evan Senter, Evgenii Eltyshev, Francesco Visin, Gabriel Rasskin, Gary Wei, Glenn Cameron, Gus Martins, Hadi Hashemi, Hanna Klimczak-Plucinska, Harleen Batra, Harsh Dhand, Ivan Nardini, Jacinda Mein, Jack Zhou, James Svensson, Jeff Stanway, Jetha Chan, Jin Peng Zhou, Joana Carrasqueira, Joana Iljazi, Jocelyn Becker, Joe Fernandez, Joost van Amersfoort, Josh Gordon, Josh Lipschultz, Josh Newlan, Ju-yeong Ji, Kareem Mohamed, Kartikeya Badola, Kat Black, Katie Millican, Keelin McDonell, Kelvin Nguyen, Kiranbir Sodhia, Kish Greene, Lars Lowe Sjösund, Lauren Usui, Laurent Sifre, Lena Heuermann, Leticia Lago, and Lilly McNealus. Gemma 2: Improving open language models at a practical size. *CoRR*, abs/2408.00118, 2024.

[131] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. Relay: A new ir for machine learning frameworks. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, page 58–68, New York, NY, USA, 2018. Association for Computing Machinery.

[132] Youcef Saad. Krylov subspace methods on supercomputers. *SIAM Journal on Scientific and Statistical Computing*, 10(6):1200–1232, 1989.

[133] Youcef Saad. Sparskit: A basic tool kit for sparse matrix computations. Technical report, 1990.

[134] Joel H. Saltz and Ravi Mirchandaney. The preprocessed doacross loop. In *Proceedings of the International Conference on Parallel Processing, ICPP '91, Austin, Texas, USA, August 1991. Volume II: Software*, pages 174–179. CRC Press, 1991.

[135] Victor Sanh, Thomas Wolf, and Alexander Rush. Movement pruning: Adaptive sparsity by fine-tuning. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 20378–20389. Curran Associates, Inc., 2020.

[136] Rya Sanovar, Srikant Bharadwaj, Renée St. Amant, Victor Rühle, and Saravan Rajmohan. Lean attention: Hardware-aware scalable attention mechanism for the decode-phase of transformers. *CoRR*, abs/2405.10480, 2024.

[137] Michael Sejr Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In Aldo Gangemi, Roberto Navigli, Maria-Esther Vidal, Pascal Hitzler, Raphaël Troncy, Laura Hollink, Anna Tordai, and Mehwish Alam, editors, *The Semantic Web - 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3-7, 2018, Proceedings*, volume 10843 of *Lecture Notes in Computer Science*, pages 593–607. Springer, 2018.

[138] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. Collective classification in network data. *AI Mag.*, 29(3):93–106, 2008.

[139] Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. A sparse iteration space transformation framework for sparse tensor algebra. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[140] Rajsekhar Setaluri, Mridul Aanjaneya, Sean Bauer, and Eftychios Sifakis. Spgrid: a

sparse paged grid structure applied to adaptive smoke simulation. *ACM Trans. Graph.*, 33(6):205:1–205:12, 2014.

[141] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *CoRR*, abs/2407.08608, 2024.

[142] Junru Shao, Xiyou Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. Tensor program optimization with probabilistic programs, 2022.

[143] Noam Shazeer. Fast transformer decoding: One write-head is all you need. *CoRR*, abs/1911.02150, 2019.

[144] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[145] Shaden Smith and George Karypis. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, IA³ '15, New York, NY, USA, 2015. Association for Computing Machinery.

[146] Benjamin Spector, Jordan Juravsky, Stuart Sul, Owen Dugan, Dylan Lim, Dan Fu, Simran Arora, and Chris Ré. Look ma, no bubbles! designing a low-latency megakernel for llama-1b. `https://hazyresearch.stanford.edu/blog/2025-05-27-no-bubbles`, May 2025. Hazy Research Blog Post.

[147] Benjamin Spector, Aaryan Singhal, Simran Arora, and Chris Re. ThunderKittens: A Simple Embedded DSL for AI kernels, May 2024.

[148] Michelle Mills Strout, Mary W. Hall, and Catherine Olschanowsky. The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code. *Proc. IEEE*, 106(11):1921–1934, 2018.

[149] Jianlin Su, Murtadha H. M. Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

[150] Patricia Suriana, Andrew Adams, and Shoaib Kamil. Parallel associative reductions in halide. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, page 281–291. IEEE Press, 2017.

[151] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. *Efficient Processing of Deep Neural Networks*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2020.

[152] Haotian Tang, Zhijian Liu, Xiuyu Li, Yujun Lin, and Song Han. Torchsparse: Efficient point cloud inference engine. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, page 302–315, 2022.

[153] Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. QUEST: query-aware sparsity for efficient long-context LLM inference. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024.

[154] Shizhi Tang, Jidong Zhai, Haojie Wang, Lin Jiang, Liyan Zheng, Zhenhao Yuan, and Chen Zhang. Freetensor: A free-form dsl with holistic optimizations for irregular tensor programs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 872–887, New York, NY, USA, 2022. Association for Computing Machinery.

[155] Tensorflow Developers. Ragged tensors | tensorflow core. `https://www.tensorflow.org/guide/ragged_tensor`, 2018.

[156] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. CUTLASS, January 2023.

[157] Ruiqin Tian, Luanzheng Guo, Jiajia Li, Bin Ren, and Gokcen Kestor. A high performance sparse tensor algebra compiler in mlir. In *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 27–38, 2021.

[158] Philippe Tillet, H. T. Kung, and David Cox. *Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations*, page 10–19. Association for Computing Machinery, New York, NY, USA, 2019.

[159] Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, page 10–19, New York, NY, USA, 2019. Association for Computing Machinery.

[160] Nicolas Vasilache, Oleksandr Zinenko, Aart J. C. Bik, Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, Matthias Springer, Tobias Gysi, Diego Caballero, Stephan Herhut, Stella Laurenzo, and Albert Cohen. Composable and modular code generation in MLIR: A structured and retargetable approach to tensor compiler construction. *CoRR*, abs/2202.03293, 2022.

[161] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N.

Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[162] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

[163] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018.

[164] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[165] Richard Vuduc, James W Demmel, and Katherine A Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16:521–530, jan 2005.

[166] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*, pages 167–188. Springer, 2014.

[167] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.

[168] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S. Yu. Heterogeneous graph attention network. In Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia, editors, *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, pages 2022–2032. ACM, 2019.

[169] Yuke Wang, Boyuan Feng, and Yufei Ding. TC-GNN: accelerating sparse graph neural network computation via dense tensor core on gpus. *CoRR*, abs/2112.02052, 2021.

[170] Yuke Wang, Boyuan Feng, and Yufei Ding. Qgtc: Accelerating quantized graph neural networks via gpu tensor core. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, page 107–119, New York, NY, USA, 2022. Association for Computing Machinery.

[171] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. GNNAdvisor: An adaptive and efficient runtime system for GNN acceleration on GPUs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 515–531. USENIX Association, July 2021.

[172] Yuke Wang, Boyuan Feng, Zheng Wang, Guyue Huang, and Yufei Ding. TC-GNN: bridging sparse GNN computation and dense tensor cores on gpus. In Julia Lawall and Dan Williams, editors, *2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*, pages 149–164. USENIX Association, 2023.

[173] Mengdi Wu, Xinhao Cheng, Oded Padon, and Zhihao Jia. A multi-level superoptimizer for tensor programs. *CoRR*, abs/2405.05751, 2024.

[174] Yidi Wu, Kaihao Ma, Zhenkun Cai, Tatiana Jin, Boyang Li, Chengguang Zheng, James Cheng, and Fan Yu. Seastar: vertex-centric programming for graph neural networks. In Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar, editors, *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 359–375. ACM, 2021.

[175] xAI. Open Release of Grok-1. `https://x.ai/blog/grok-os`, 2023. [Accessed 24-06-2024].

[176] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. *arXiv*, 2023.

[177] Zhiqiang Xie, Minjie Wang, Zihao Ye, Zheng Zhang, and Rui Fan. Graphiler: Optimizing graph neural networks with message passing data flow graph. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 515–528, 2022.

[178] Shanli Xing, Zihao Ye, Bohan Hou, and Tianqi Chen. Sorting-free gpu kernels for llm sampling. `https://flashinfer.ai/2025/03/10/sampling.html`, March 2025. FlashInfer Blog Post.

[179] Carl Yang, Aydin Buluç, and John D. Owens. Design principles for sparse matrix multiplication on the GPU. In Marco Aldinucci, Luca Padovani, and Massimo Torquati, editors, *Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings*, volume 11014 of *Lecture Notes in Computer Science*, pages 672–687. Springer, 2018.

[180] Songlin Yang, Bailin Wang, Yikang Shen, Rameswar Panda, and Yoon Kim. Gated linear attention transformers with hardware-efficient training, 2024.

[181] Lu Ye, Ze Tao, Yong Huang, and Yang Li. Chunkattention: Efficient self-attention with prefix-aware KV cache and two-phase partition. In Lun-Wei Ku, Andre Martins, and

Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 11608–11620. Association for Computational Linguistics, 2024.

[182] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. Flashinfer: Efficient and customizable attention engine for LLM inference serving. In *Proceedings of the 8th Conference on Machine Learning and Systems (MLSys)*, 2025.

[183] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. Sparsetir: Composable abstractions for sparse compilation in deep learning. In Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift, editors, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 660–678. ACM, 2023.

[184] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for transformer-based generative models. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 521–538. USENIX Association, 2022.

[185] Zhongming Yu, Guohao Dai, Guyue Huang, Yu Wang, and Huazhong Yang. Exploiting online locality and reduction parallelism for sampled dense matrix multiplication on gpus. In *39th IEEE International Conference on Computer Design, ICCD 2021, Storrs, CT, USA, October 24-27, 2021*, pages 567–574. IEEE, 2021.

[186] Yujia Zhai, Chengquan Jiang, Leyuan Wang, Xiaoying Jia, Shang Zhang, Zizhong Chen, Xin Liu, and Yibo Zhu. Bytetransformer: A high-performance transformer boosted for variable-length inputs. In *IEEE International Parallel and Distributed Processing*

Symposium, IPDPS 2023, St. Petersburg, FL, USA, May 15-19, 2023, pages 344–355. IEEE, 2023.

[187] Hengrui Zhang, Zhongming Yu, Guohao Dai, Guyue Huang, Yufei Ding, Yuan Xie, and Yu Wang. Understanding GNN computational graph: A coordinated computation, io, and memory perspective. In Diana Marculescu, Yuejie Chi, and Carole-Jean Wu, editors, *Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022*. mlsys.org, 2022.

[188] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, Peng Di, Kun Zhang, and Xuefeng Jin. Akg: Automatic kernel generation for neural processing units using polyhedral transformations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 1233–1248, New York, NY, USA, 2021. Association for Computing Machinery.

[189] Bojian Zheng, Ziheng Jiang, Cody Hao Yu, Haichen Shen, Joshua Fromm, Yizhi Liu, Yida Wang, Luis Ceze, Tianqi Chen, and Gennady Pekhimenko. DietCode: Automatic Optimization for Dynamic Tensor Programs. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 848–863, 2022.

[190] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.

[191] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion

Stoica. Ansor: Generating High-Performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879. USENIX Association, November 2020.

[192] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark W. Barrett, and Ying Sheng. Efficiently programming large language models using sglang. *CoRR*, abs/2312.07104, 2023.

[193] Ningxin Zheng, Bin Lin, Quanlu Zhang, Lingxiao Ma, Yuqing Yang, Fan Yang, Yang Wang, Mao Yang, and Lidong Zhou. SparTA: Deep-Learning model sparsity via Tensor-with-Sparsity-Attribute. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 213–232, Carlsbad, CA, July 2022. USENIX Association.

[194] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 859–873. ACM, 2020.

[195] Aojun Zhou, Yukun Ma, Junnan Zhu, Jianbo Liu, Zhijie Zhang, Kun Yuan, Wenxiu Sun, and Hongsheng Li. Learning N: M fine-grained structured sparse neural networks from scratch. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[196] Kan Zhu, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Yufei Gao, Qinyu Xu, Tian Tang, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. Nanoflow: Towards optimal large language model serving throughput. *CoRR*, abs/2408.12757, 2024.

[197] Lei Zhu, Xinjiang Wang, Wayne Zhang, and Rynson W. H. Lau. Relayattention for efficient large language model serving with long system prompts. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 4945–4957. Association for Computational Linguistics, 2024.