Design Automation with Efficient Compilation on Hardware Accelerators

BAI, Yang

A Thesis Submitted in Partial Fulfillment

of the Requirements for the Degree of

Doctor of Philosophy

in

Computer Science and Engineering

The Chinese University of Hong Kong

July 2024

Thesis Assessment Committee

Professor XU Hong (Chair)

Professor YU Bei (Thesis Supervisor)

Professor SHAO Zili (Thesis Co-Supervisor)

Professor LO Chi Lik Eric (Committee Member)

Professor WANG Wei (External Examiner)

Abstract

Acquiring high-performance tensor programs for machine learning applications on diverse hardware platforms poses substantial challenges. Current methods often rely on vendor-provided libraries or manual search policies with limited optimization capabilities, which either necessitate extensive engineering efforts or have a restricted search space. Moreover, the limitations of transistor scaling demand the development of domain-specific hardware acceleration. However, constructing specialized accelerators and compilation stacks for application acceleration requires significant engineering expertise. The aim of this dissertation is to tackle these challenges by elevating the compiler to a first-class citizen and integrating tensor program generation, hardware acceleration, and design automation. Firstly, it explores the compiler's potential in automatically generating computation graphs and tensor programs. Furthermore, it investigates the compiler's ability to efficiently facilitate knowledge transfer between hardware platforms through transfer learning. Moreover, it harnesses the power of pipelining between data movement and computation through automatic compilation pass optimization. Ultimately, a unified compilation interface is devised to achieve collaborative optimization between the hardware and software design spaces, enabling end-to-end optimization for high-performance tensor programs and hardware acceleration.

摘要

為機器學習應用程式在不同硬體平臺上獲得高效能的張量程序,提出了很大的挑戰. 目前的方法經常依賴廠商提供的庫或手動搜索政策, 但其優化能力有限, 這要麼需要龐大的工程工作, 或搜索空間受限制. 此外, 電晶體規模的限制需要開發特定領域的硬體加速. 然而, 建構專用加速器與編譯堆疊以加速應用, 需要大量的工程專長. 本論文的目標是通過提升編譯器為第一優先級, 並將張量程式產生, 硬體加速與設計, 自動化整合來解決這些挑戰. 首先, 它探討編譯器在自動產生計算圖與張量程式的潛力. 此外, 它研究編譯器在不同硬體平臺之間有效地知識傳遞的能力. 而且, 它利用管線處理資料移動與計算來最大化效能, 通過自動編譯通過優化. 最終, 一個統一的編譯接口被設計出來, 實現硬體與軟體設計空間之間的協同優化, 從而實現張量程式和硬體加速的端對端優化.

Contents

List of Tables

List of Figures

Acknowledgments

My dissertation owes its realization to the invaluable guidance of numerous individuals.

Foremost, I extend my deep appreciation to my supervisor, Prof. Yu Bei, whose unwavering support for my Ph.D. study and associated research has been instrumental. I am sincerely grateful for his unwavering patience, motivational influence, and profound expertise. Prof. Yu's mentorship has been indispensable to the progression of my research. His teachings have instilled in me the key principle of astutely selecting research topics. He consistently encourages me to pose insightful inquiries, diligently nurtures my research skills, and imparts upon me an unwavering commitment to perfection. Such guidance has propelled me to refine my work and strive for superior research quality.

During the initial year of my Ph.D. program, I developed a keen interest in constructing deep learning compilers to enhance the execution speed of model operations on NVIDIA GPUs. With my strong engineering background, I initially believed that I possessed the capability to tackle the associated challenges. However, I soon observed a recurring pattern: whenever I identified new optimization opportunities for widely used models and invested considerable time in crafting CUDA code, TensorRT consistently incorporated corresponding enhancements in subsequent versions. This disheartening situation left me perplexed and disillusioned, as I struggled to achieve state-of-the-art performance.

In response to my predicament, Prof. Yu promptly advised me against engaging in a competitive pursuit with hardware vendors and discouraged me from undertaking tasks within their domain. He emphasized that NVIDIA already boasts dedicated teams exclusively focused on addressing such matters, making it ex-

ceedingly arduous for a Ph.D. candidate like myself to surpass their expertise and attain superior performance. Instead, Prof. Yu urged me to broaden my research scope within the compilation framework. He suggested exploring alternative avenues, such as integrating the open-source RISC-V instruction set with underlying hardware accelerators, to unlock the potentials for compilation optimization.

Significantly, Prof. Yu imparted a crucial lesson to me regarding the essence of impactful system research. He emphasized that the foundation of such research lies in its alignment with real-world applications, and that its essence lies in gaining profound insights into existing systems rather than merely constructing new ones. This revelation prompted me to reconsider the intricate relationship between engineering and research. Subsequently, I acquired the skill to strategically identify appropriate and pragmatic entry points for my research endeavors, which greatly facilitated the smooth progression of my subsequent investigations.

In addition to his guidance in research, Prof. Yu also provided invaluable advice on maintaining personal competitiveness within the academic job market. Drawing from his own experiences of navigating the highs and lows of various research directions, he possesses a keen sense of urgency and consistently encourages me to continually update my knowledge and understanding of past and future works. This emphasis on self-improvement serves as a constant stimulus, ensuring that I remain at the forefront of academic excellence.

Furthermore, I would like to underscore the significance of leading a well-rounded life and attending to familial responsibilities, which I have come to recognize as being of utmost importance and considerably more demanding than conducting research. Hence, I extend my heartfelt gratitude to my mothers and my wife, who selflessly dedicate themselves to the care and well-being of my son and our family. Their unwavering support enables me to fully immerse myself in the

demanding realm of research. I am also deeply grateful to my son, whose presence has been a source of invaluable lessons in patience and resilience. His unwavering encouragement motivates me to persist in the face of adversity and not falter in my pursuit of knowledge.

Lastly, I express my profound gratitude to all my collaborators and sponsors who have played a crucial role in the realization of my research endeavors. I extend my heartfelt appreciation to Guyue Huang, Wenqian Zhao, Zixiao Wang, Mingjun Li, Shuo Yin, Yuhao Ji, Wendong Xu, Xufeng Yao, Xinyun Zhang, and the CUHK Graduate Division for their invaluable support and contributions. Their unwavering support and assistance have been instrumental in the successful completion of my research.

Chapter 1

Introduction

The optimization of high-performance tensor programs is of paramount importance to ensure efficient runtime for specific applications. Nonetheless, achieving optimal tensor programs for diverse applications across various hardware platforms remains a formidable challenge. Presently, machine learning systems heavily rely on vendor-provided libraries or employ various search policies to attain performant tensor programs. However, both approaches have their limitations: the former necessitates substantial engineering efforts to develop platform-specific optimization code, while the latter often falls short in identifying high-performance programs due to constraints on search space and ineffective exploration strategies.

Simultaneously, the era of exponential transistor scaling, which has propelled the speed-up of general-purpose processors for several decades, is nearing its conclusion. As a result, there has been a notable surge in interest, both within academia and industry, in domain-specific hardware acceleration. Such specialized accelerators offer promising advantages in terms of performance and energy efficiency compared to their general-purpose counterparts. However, the development of a dedicated accelerator, along with its associated compilation stack, entails considerable en-

gineering efforts in terms of design and implementation. Furthermore, previous endeavors to establish a comprehensive full-stack implementation are challenging to repurpose when transitioning to a new hardware accelerator design. To address these challenges, this dissertation proposes a novel approach that combines tensor program generation and hardware acceleration with design automation. Crucially, the compiler assumes a central role as a first-class citizen in our hardware and software co-design framework.

Firstly, this dissertation explores the potential of the compiler by integrating computation graphs and enabling automatic generation of tensor programs. By leveraging this capability, we aim to streamline the process of program generation and enhance the efficiency of tensor computations.

Secondly, this dissertation delves into the effective utilization of the compiler to facilitate knowledge transfer across different hardware platforms through the application of transfer learning techniques. This enables the seamless adaptation of compiled programs to diverse hardware architectures, thereby optimizing performance and efficiency.

Furthermore, this dissertation investigates the optimization potential of pipelining techniques that synchronize data movement and computation. Leveraging the compiler's automatic optimization capabilities, we aim to maximize the efficiency of these operations.

Ultimately, by harmonizing the distinct characteristics of hardware and software design spaces, this dissertation presents a unified compilation interface. This interface facilitates collaborative optimization, enabling the end-to-end generation of high-performance tensor programs and hardware specifications through a cohesive and streamlined process.

1.1 Goal: Automatic Generation of Programs and Hardware Acceleration

To achieve this objective, various design requirements are imposed on all components of the complete compilation flow.

- **Compiler Optimization:** The programming interface must possess a sufficient level of expressiveness to facilitate the development of versatile applications, while ensuring that the compiler can handle diverse combinations of these features with robustness. This is crucial because tensor programs inherently encapsulate the design requirements for hardware acceleration, and the compiler plays a vital role in understanding the correspondence between the software and hardware through the instruction set architecture. The implementation of all optimizations relies on the compiler pass, which acts as a bridge between the software and hardware components.

- **Design Space:** The design space should embody a modular and comprehensive approach. By adopting a modular design, we can achieve independent integration of software and hardware co-designed features into specialized accelerators. This approach also enables us to explore various optimization opportunities from different perspectives. Additionally, through a general design, the resulting tensor program and domain-specific accelerators can effectively cater to the target domain. Therefore, the design space should possess adequate generality to encompass a broad range of potential applications and provide abundant optimization possibilities for software and hardware co-design.

- **Performance Auto-tuning:** To effectively leverage the compiler's awareness of software and hardware co-optimization, it is crucial to develop an auto-tuner that is both effective and efficient. This requires the auto-tuner to have the capability to select the optimal performance from the tensor program and hardware design spaces, thus minimizing the need for time-consuming on-device measurements and hardware synthesis. The entire tuning process should be automated, allowing for seamless integration of the auto-tuner into the compilation stack.

- **Unified Compilation Interface:** By identifying the tensor program and hardware features within the design space, it becomes possible to adopt a high-level mainstream programming paradigm with moderate extensions as the unified compilation interface. As a result, the optimization of hardware parameters and algorithm-to-hardware mappings can be achieved in a unified manner. This unified approach allows for the integration of domain-specific knowledge related to hardware accelerators into the interface, enabling compilers to effectively facilitate software and hardware co-design.

1.2 Main Contributions

The contributions of this dissertation can be summarized as follows:

- **GTCO.** Deep learning frameworks or compilers often optimize operators in computation graphs using fixed templates, which can sometimes overlook potential optimizations such as operator fusion. Therefore, it is crucial to automatically implement and optimize new combinations of operators on specific hardware accelerators. In this dissertation, we present GTCO, a tensor compilation system designed to accelerate inference of transformer-based vision models on GPUs.

GTCO addresses operator fusion techniques in transformer-based models through a novel dynamic programming algorithm and proposes a search policy with new sketch generation rules for fused batch matrix multiplication and softmax operators. Tensor programs are sampled from an effective search space, and a hardware abstraction with hierarchical mapping from tensor computation to domain-specific accelerators (Tensor Cores) is formally defined. Finally, our framework can efficiently map and transform tensor expressions into CUDA kernels with hardware intrinsics on GPUs. Experimental results demonstrate that GTCO significantly improves end-to-end execution performance, achieving up to 1.73x relative improvement compared to the state-of-the-art deep learning library TensorRT on NVIDIA GPUs with Tensor Cores.

- **ATFormer.** The efficiency of training and inference for increasingly large deep neural networks heavily relies on the performance of tensor operators on specific hardware accelerators. Therefore, it is crucial to have a performance tuning framework with tensorized instruction compilation for automatic tensor generation to achieve efficient deployment. However, these novel tensorized instructions, along with emerging machine learning models, pose significant engineering challenges in compilation-based methods. They entail exploring a large design space with limited measurement accuracy and poor transferability among specialized instructions that have specific hardware constraints. This dissertation introduces a novel performance model for automatic code optimization with tensorized instructions. The performance model is centered around the assignment feature, which not only clearly specifies the behavior of instructions with memory and computation abstraction but also formally defines the matching problem from the algorithm to tensorized instructions. Additionally, a simple yet efficient design with attention-inspired modules is employed to accurately predict the

performance of optimized tensor programs by capturing global and long-range dependencies within a complete scheduling space. In comparison to state-of-the-art methods, our performance model can predict the optimal implementation of code configurations with tensorized instructions, resulting in reduced inference time and search time by up to 1.22x and 6.97x, respectively, on modern deep neural network benchmarks. Furthermore, with pre-trained parameters, our performance model can quickly adapt to different workloads and platforms using tensorized instructions through transfer learning.

- **ALCOP.** Pipelining between data loading and computation is a crucial optimization technique for tensor programs on GPUs. For the latest NVIDIA Ampere GPUs, multi-stage pipelining across the multi-level buffer hierarchy is particularly essential to minimize resource idleness and ensure efficient kernel performance. Currently, people rely on expert-written libraries like cuBLAS to access pipelining optimizations, rather than through tensor program transformations. However, this approach lacks extensibility to new operators and compatibility with prior tensor compiler optimizations. We introduce ALCOP, an automatic pipelining framework based on the TVM infrastructure, which overcomes three critical challenges in code generation for pipelining. These challenges include detecting pipelining-applicable buffers, transforming programs for multi-level multi-stage pipelining, and efficient schedule parameter search using static analysis. Experimental results demonstrate that ALCOP can generate programs with an average speedup of 1.23× (up to 1.73×) compared to vanilla TVM. Moreover, in end-to-end models, ALCOP outperforms TVM by up to 1.18× and XLA by up to 1.64×. Additionally, our performance model significantly enhances the efficiency of the schedule tuning process, finding schedules with 99% of the performance achieved by exhaustive search while requiring 40× fewer trials.

- **BACO.** Domain-specific hardware accelerators hold great promise for achieving significant enhancements in both speed and energy efficiency compared to general-purpose processors. Nevertheless, the design and development process of these accelerators often demands a substantial amount of manual effort, particularly in the realms of hardware design and the development of high-performance tensor programs. Extensive research has been dedicated to optimizing these aspects individually, which has resulted in a proliferation of possibilities, posing considerable challenges. To tackle this challenge, we propose BACO, an end-to-end one-loop search framework. The aim of BACO is to provide a unified compilation interface that facilitates the co-exploration of hardware and software. The key element of BACO is its multi-level abstraction, which not only precisely defines the behavior of custom hardware instructions and configuration state, but also enables fine-grained optimization of computation and data movement through the concept of a block. Building upon this abstraction, we develop a programming language and compiler that incorporates differentiable performance models to automatically explore various high-performance design points. Additionally, BACO employs an efficient strategy for estimating importance, allowing for simultaneous optimization of multiple layers to enhance end-to-end performance. This approach facilitates rapid co-exploration while reducing development efforts. Experimental results demonstrate that BACO surpasses existing state-of-the-art search solutions and manually tuned versions by a substantial margin.

1.3 Produced Publications

The research work presented in this dissertation has resulted in several direct and indirect publications, which are listed below:

- **Yang Bai**, Xufeng Yao, Qi Sun, Bei Yu, "AutoGTCO: Graph and Tensor Co-Optimize for Image Recognition with Transformers on GPU", IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Nov. 1–4, 2021.

- **Yang Bai**, Xufeng Yao, Qi Sun, Wenqian Zhao, Shixin Chen, Zixiao Wang, Bei Yu, "GTCO: Graph and Tensor Co-Design for Transformer-based Image Recognition on Tensor Cores", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), vol. 43, no. 02, pp. 586–599, 2023.

- **Yang Bai**, Wenqian Zhao, Shuo Yin, Zixiao Wang, Bei Yu, "ATFormer: A Learned Performance Model with Transfer Learning Across Devices for Deep Learning Tensor Programs", Empirical Methods in Natural Language Processing (EMNLP-main Long Paper), Singapore, Dec. 06–10, 2023.

- **Yang Bai**, Wenqian Zhao, Shuo Yin, Zixiao Wang, Bei Yu, "A Learned Performance Model with Transfer Learning Across GPUs on Tensorized Instruction", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), under review

- **Yang Bai**, Mingjun Li, Yuhao Ji, Bei Yu, "BACO: Co-exploration of Hardware Acceleration and Tensor Program with Unified Compilation Interface", ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2025, Summer), under review

- Guyue Huang, **Yang Bai**, Liu Liu, Yuke Wang, Bei Yu, Yufei Ding, Yuan Xie, "ALCOP: Automatic Load-Compute Pipelining in Deep Learning Compiler for AI-GPUs", Conference on Machine Learning and Systems (MLSys), Jun. 04–08, 2023.

- Wenqian Zhao, **Yang Bai**, Qi Sun, Wenbo Li, Haisheng Zheng, Nianjuan Jiang, Jiangbo Lu, Bei Yu, Martin D.F. Wong, "A High-Performance Accelerator for Super-Resolution Processing on Embedded GPU", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), vol. 42, no. 10, pp. 3210–3223, 2023.

- Xufeng Yao, **Yang Bai**, Xinyun Zhang, Yuechen Zhang, Qi Sun, Ran Chen, Ruiyu Li, Bei Yu, "PCL: Proxy-based Contrastive Learning for Domain Generalization", IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), New Orleans, Jun. 19–24, 2022.

- Wenqian Zhao, Xufeng Yao, Shuo Yin, **Yang Bai**, Ziyang Yu, Yuzhe Ma, Bei Yu, Martin D.F. Wong, "AdaOPC 2.0: Enhanced Adaptive Mask Optimization Framework for Via Layers", accepted by IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD).

- Zixiao Wang, Yunheng Shen, Xufeng Yao, Wenqian Zhao, **Yang Bai**, Farzan Farnia, Bei Yu, "ChatPattern: Layout Pattern Customization via Natural Language", ACM/IEEE Design Automation Conference (DAC), San Francisco, Jun. 23–27, 2024.

- Shixin Chen, Su Zheng, Chen Bai, Wenqian Zhao, Shuo Yin, **Yang Bai**, Bei Yu, "SoC-Tuner: An Importance-guided Exploration Framework for DNN-targeting

SoC Design", IEEE/ACM Asian and South Pacific Design Automation Conference (ASPDAC), South Korea, Jan. 22–25, 2024.

- Weizheng Lu, Yaobo Jia, Feng Zhang, **Yang Bai**, Yueguo Chen, Bei Yu, Xiaoyong Du, "GADUS: GPU-based Automatically Dynamic Unrolling Neural Differential Equation Solver", Transactions on Parallel and Distributed Systems (TPDS), under review

- Zixiao Wang, Yunheng Shen, Wenqian Zhao, **Yang Bai**, Guojin Chen, Farzan Farnia, Bei Yu, "DiffPattern: Layout Pattern Generation via Discrete Diffusion", ACM/IEEE Design Automation Conference (DAC), San Francisco, Jul. 09–13, 2023.

- Yuxuan Zhao, Qi Sun, Zhuolun He, **Yang Bai**, Bei Yu, "AutoGraph: Optimizing DNN Computation Graph for Parallel GPU Kernel Execution", AAAI Conference on Artificial Intelligence (AAAI), Feb. 7–14, 2023.

- Qi Sun, Xinyun Zhang, Hao Geng, Yuxuan Zhao, **Yang Bai**, Haisheng Zheng, Bei Yu, "GTuner: Tuning DNN Computations on GPU via Graph Attention Network", ACM/IEEE Design Automation Conference (DAC), San Francisco, Jul. 10–14, 2022.

- Wenqian Zhao, Qi Sun, **Yang Bai**, Haisheng Zheng, Wenbo Li, Bei Yu, Martin D.F. Wong, "A High-Performance Accelerator for Super-Resolution Processing on Embedded GPU", IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Nov. 1–4, 2021.

- Qi Sun, Chen Bai, Tinghuan Chen, Hao Geng, Xinyun Zhang, **Yang Bai**, Bei Yu, "Fast and Efficient DNN Deployment via Deep Gaussian Transfer Learning", IEEE International Conference on Computer Vision (ICCV), Oct. 11–17, 2021.

1.4 Dissertation Outline

This dissertation endeavors to tackle the automation of tensor program generation and domain-specific accelerator design from a compiler-centric standpoint. The focal challenges encompass various layers of the system, encompassing software, programming interface, and hardware.

Chapter 2 of this dissertation presents an extensive and meticulous exposition of the background pertaining to several pivotal subjects. These encompass the transformer model architecture, the architecture of GPU memory and computation, the programming model, the machine learning compiler, the domain-specific language, and the spatial accelerator. Each of these topics is exhaustively examined, providing a robust groundwork for the subsequent chapters and the overall research expounded in this dissertation.

Chapter 3 of this dissertation presents a comprehensive and detailed introduction to a dynamic programming algorithm that focuses on exploring operator fusion patterns in transformer-based models. This algorithm facilitates the efficient exploration of various fusion possibilities, thereby enhancing the overall performance of these models. Additionally, a search policy is proposed, incorporating novel sketch generation rules and a unique hardware abstraction with register-level optimization. These advancements enable a more flexible mapping for tensor computation, resulting in improved performance. The optimization of fused matrix multiplication and software operators is achieved through the utilization of tensorized instructions on GPUs, further enhancing system efficiency. Lastly, a regression-based learned performance model is employed to fine-tune the performance of each kernel. This approach facilitates the optimization of individual kernel performance, ultimately leading to improved overall system performance.

Chapter 4 of this dissertation introduces a pioneering performance model specifically designed for the automatic optimization of tensor programs employing both general and tensorized instructions. We propose a collection of innovative assignment features that encode the tensor program with computation and memory abstraction, enabling the representation of intrinsic hardware accelerator characteristics. This approach facilitates systematic exploration of the scheduling space and efficient matching, resulting in improved performance for a wide range of tensor operators. In our experiments, the learned performance model demonstrates substantial speedup in model deployment on GPU Tensor Cores. Through transfer learning, the performance model achieves faster converged latency and exhibits superior transferability across various hardware platforms, surpassing previous state-of-the-art benchmarks.

Chapter 5 of this dissertation addresses the crucial need for automatic pipelining optimization in the context of deep learning compilers. Given the substantial tiling size required to mitigate bandwidth constraints, inter-tiling parallelism alone is insufficient for achieving optimal utilization. Consequently, intra-tiling pipelining emerges as a vital requirement. In response, we propose the first compiler solution that supports multi-stage, multi-level pipelining. By introducing automatic pipelining, our compiler is capable of generating GPU programs that exhibit an average speedup of 1.23$\times$ compared to vanilla TVM, with a maximum speedup of 1.73$\times$. Furthermore, we develop an analytical performance model that significantly enhances the search efficiency of the schedule tuning process.

Chapter 6 of this dissertation introduces the BACO framework, which offers a comprehensive solution for the co-exploration of hardware and software using the one-loop search approach. To facilitate the co-exploration process, the framework is equipped with a programming language and compiler. The fundamental principle

of BACO lies in its utilization of a unified interface that treats blocks as first-class citizens. This approach provides a straightforward yet highly effective granularity for co-exploration. A noteworthy aspect of BACO is its internalization of a substantial portion of the hardware-specific code generation logic and optimization policy within the compilation flow. This design choice significantly reduces the complexity exposed to users. The primary contributions can be summarized as follows:

- **Productivity:** BACO offers a Python-based DSL and compiler, enabling flexible descriptions of desired target programs. The design principle revolves around the concept of blocks, which represent statically shaped multi-dimensional sub-arrays. At the block level, the DSL controls tensor splitting and accumulation. Within each block, the compiler takes care of complex memory management, accelerator-specific instructions, and configuration states. As a result, the development and co-exploration process is made more streamlined, as users are shielded from the tedious aspects of these tasks.

- **Automation:** BACO incorporates differentiable analytical performance models that assess the number of arithmetic operations and memory accesses to different buffers during program transformations. This feature allows for the inference of minimal hardware requirements and the automatic updating of parameters in an end-to-end manner. Consequently, it facilitates hardware and software co-exploration within a unified space. Additionally, BACO employs an importance estimation strategy based on gradient descent to automatically prioritize layers that have a significant impact on overall performance.

- **Performance:** To the best of our knowledge, BACO is the first work to introduce a unified compilation interface with programming language and compiler support for hardware and software co-exploration. We conducted

evaluations of BACO using comprehensive benchmarks with open-source DNN accelerators. The experimental results demonstrate significant performance improvements. BACO achieves a 3.54× reduction in energy-delay product (EDP) compared to manually-tuned libraries, and a 1.32× reduction in EDP compared to the state-of-the-art one-loop search approach.

Chapter 2

Background and Related Work

2.1 Transformer-based Model Architecture

The transformer model, originally developed for a new attention-based building block for machine translation. Transformers are built on a long sequence within natural language processing and the key element in transformer architecture is the attention mechanism, which is composed of neural network layers that aggregate information from the entire input sequence and make the model learn to focus on particular parts of a sequence. An important advantage of the attention-based models is their global computations and large-scale memory scheduling, which leads them more eligible than RNNs on long sequences though this would lead to heavy computation and communication workloads during inference. Transformer models make two main contributions. First, it popularizes attention mechanisms to a particular module named multi-head attention (MHA) [**ATTN-CVPR2021-SETR**]. Second, it does not rely on recurrent or convolutional algorithms. Besides, an important characteristic is that the transformer model contains many similar sub-graph structures which can be executed in parallel with the same configuration.

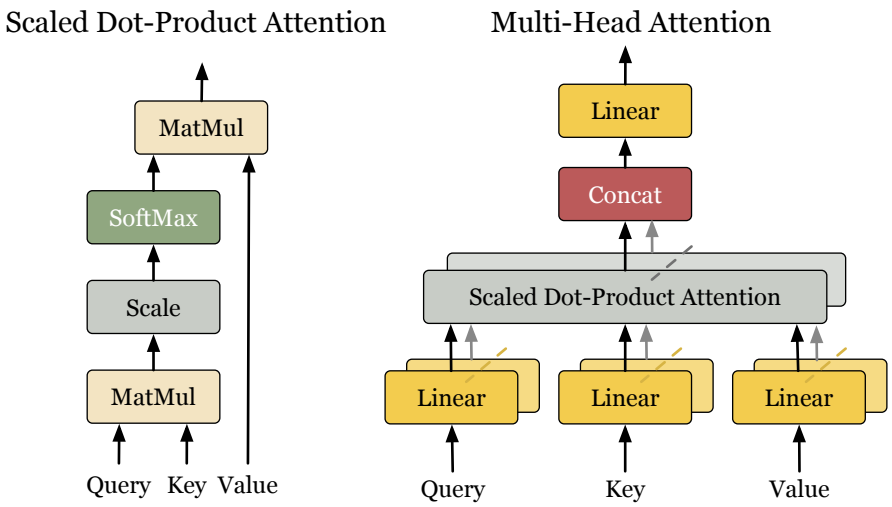


Figure 2.1: *Scaled Dot-Product and Multi-Head Attention (MHA).*

Besides, transformer models contain many similar subgraph structures which can be executed in parallel with the same configuration.

Encoder. The encoder module is composed of a stack of attention-based layers with identical structures. There are two sub-layers in each layer. The first sub-layer is a multi-head attention mechanism, and the second sub-layer is a simple, position-wise fully dense layer. A residual connection layer is used between each of the two sub-layer with a layer normalization. In other words, the output of each sub-layer is the sum of the input and the attention of input under the layer normalization. And the output of dimension is defined as d_{model}.

Decoder. The structure of the decoder is akin to the encoder. In addition to the two sub-layers mentioned in the encoder module, a third sub-layer is inserted into the decoder module. The function of it is to perform multi-head attention over the output of the encoder part. The function of the third sub-layer is to perform multi-head attention over the output of the encoder part. Besides, some modifications about masking mechanisms in the self-attention sub-layer are to prevent positions from attending to subsequent positions.

Multi-Head Attention. Multi-head Attention generalizes attention mechanisms and employs h attention heads parallelly to get different learned projections of a given sequence. Each attention head is an instance of scaled dot-product attention, and takes queries (q), keys (k), values (v) as its input. The function of attention is to find values corresponding to the keys closest to the input queries. The functions of heads are also augmented with linear layers that project their inputs into a lower-dimensional space. The three inputs are first multiplied by weight tensors *wq*, *wk*, *wv*, respectively, as a learned input projection. The query and key tensors are subsequently multiplied together and scaled, followed by applying the softmax operation to weight and select the most relevant results. This is then multiplied with *vv* to produce the per-head output. The outputs of all the heads are finally concatenated and linearly projected back to the input dimensional size, as depicted in Figure 2.1.

$$\mathrm{MHA}(\boldsymbol{x}) = \sum_{i=1}^{H} \mathrm{Att}_i(\boldsymbol{x}), \quad \boldsymbol{x}_{\mathrm{MHA}} = \mathrm{LN}\big(\boldsymbol{x} + \mathrm{MHA}(\boldsymbol{x})\big),$$

where Att is a dot product attention head, LN is layer normalization, and $\boldsymbol{x}$ is the input vector. The output of the MHA layer is then entered into the FFN layer, which consists of N filters:

$$\mathrm{FFN}(\boldsymbol{x}) = \big(\sum_{i=1}^{N} \mathcal{W}^{(2)}_{:,i}\sigma(\mathcal{W}^{(1)}_{i,:}\boldsymbol{x} + \boldsymbol{b}^{(1)}_i)\big) + \boldsymbol{b}^{(2)},$$

$$\boldsymbol{x}_{\mathrm{out}} = \mathrm{LN}\big(\boldsymbol{x}_{\mathrm{MHA}} + \mathrm{FFN}(\boldsymbol{x}_{\mathrm{MHA}})\big),$$

where $\mathcal{W}^{(1)}, \mathcal{W}^{(2)}, \boldsymbol{b}^{(1)}$ and $\boldsymbol{b}^{(2)}$ are the FFN parameters, and σ is the activation function, typically GELU [**hendrycks2016gaussian**].

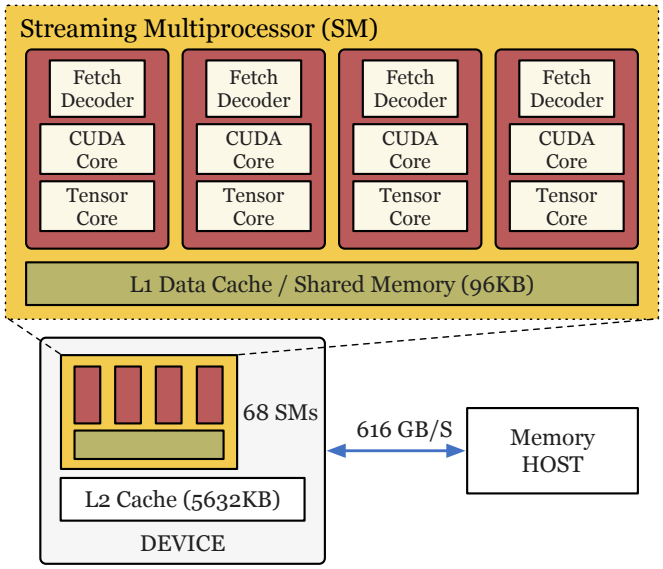


Figure 2.2: *Hardware details of the streaming multiprocessor (SM) and the memory hierarchy of NVIDIA RTX 2080 Ti GPU.*

2.2 GPU Architecture and Programming Model

Hardware Details of 2080Ti GPU. Compute Unified Device Architecture (CUDA) is a high-performance programming language developed by NVIDIA to expose software programmers to concepts such as computation parallelism and memory hierarchy. However, effectively leveraging the memory and computation units of GPUs to accelerate the execution time of deep neural networks (DNNs) requires significant engineering efforts. As depicted in Figure 2.2, the GPU device has many programmable units at different levels. For our experiments, we utilize the NVIDIA RTX 2080Ti GPU, which follows the Turing microarchitecture and has 68 parallel streaming multiprocessors (SMs) within the processing elements. Each SM has its local shared memory that can be accessed by threads within the same SM. An SM is further divided into four processing blocks, with each block possessing a 64 KB register file, and every four blocks sharing a combined 96 KB L1 data

cache/shared memory. Thread blocks are scheduled on each processing block, and each thread block contains a group of threads that can execute the same code on different data using the single instruction multiple threads (SIMT) technique. Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model for GPUs, which exposes high-performance computing programmers to the concepts of the memory hierarchy and threads hierarchy. Under the hood, accelerating deep learning models on complex memory hierarchy needs to make good use of memory units and computation units. As shown in Figure 2.2, there are many programmable units at different levels of GPU devices. RTX 2080 Ti GPU follows the Turing micro-architecture and contains 68 parallel streaming multiprocessors (SMs). Each SM has its local shared memory, which can be accessed by threads in the same SM. An SM is partitioned into four processing blocks. Each processing block possesses a 64 KB register file, and the four processing blocks share a combined 96 KB L1 data cache/shared memory. Some thread blocks are scheduled on each processing block. And each thread block contains a group of threads that can execute the same code on different data, following the single instruction multiple thread (SIMT) mechanism. Typically, prior to launching the computation kernel, data is first copied from the host memory to the device memory. On-chip memory, which is close to the computing elements, has a faster access speed than off-chip memory, which is slower and further away from the computing elements. Each type of memory has its own unique access patterns. Registers and local memory are private to the threads within a block and are located on-chip with low latency. Shared memory is composed of a set of full-sized banks, and multiple threads accessing the same bank simultaneously may lead to conflicts and increased latency.

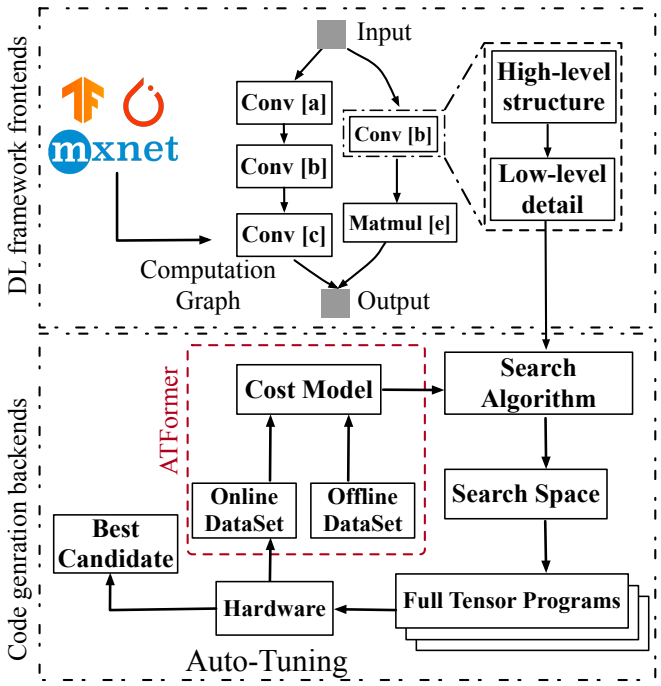


Figure 2.3: *The overview of a search-based framework with computation graph, cost model, search space and tuning algorithm during the compilation.*

2.3 Machine Learning Compilation

Recently, the development of compiler-based optimization frameworks, such as Halide [**Lerning-TOG2019-Adams**], TVM [**TVM-OSDI2018-Chen**], XLA [**50530**], and TACO [**kjolstad2017tensor**], has progressed rapidly. These optimization schemes typically consist of two parts: DL framework frontends and code generation backends, as illustrated in Figure 2.3. The frontend converts an input model into a high-level graph-based intermediate representation (IR) and applies target-independent optimizations, such as operator fusion and data layout transformation. In the backend, target-dependent optimization passes, along with hardware features, further optimize the final performance. TVM [**Chen-TVM-OSDI**] is a state-of-the-art search-based tensor compiler that is widely used in academia and industry. Its

auto-tuning aims to achieve performance comparable to hand-tailored libraries and has achieved promising results. TVM has two versions of auto-tuning: AutoTVM [**OptTensor-NIPS2018-Chen**] and Ansor [**Ansor-OSDI2020-Zheng**]. While AutoTVM is a semi-automated framework that requires pre-defined manual templates, Ansor is more advanced and fully automated. However, both frameworks need to collect data on-the-fly during the search, resulting in an extremely long compilation time. This dissertation focuses exclusively on the application of deep learning compilers. Specifically, we take Ansor [**Compiler-OSDI2020-Ansor**] as an example, a widely used deep learning compiler for generating tensor programs across various hardware platforms. The compiler leverages a hierarchical search space to optimize and separate the high-level generation structures from the low-level sampling details. This approach enables Ansor to automatically construct the search space for each operator or subgraph, eliminating the need for experienced engineers to manually develop computing templates, which can be a time-consuming and engineering-heavy process. Subsequently, Ansor incorporates an automatic performance tuner that utilizes a comprehensive search space to obtain complete tensor programs, which are then fine-tuned with a regression-based model [**ENL-KDD2016-XGBoost**].

2.4 Performance Model

Tree-based Performance Model. Decision trees are commonly utilized in classification and regression problems. To enhance their performance, an ensemble learning approach is typically employed to reduce variance. XGBoost [**XGBoost-Chen-KDD**] and LightGBM are powerful feature models in sequence modeling tasks. To achieve accurate prediction, a number of works including [**OptTensor-NIPS2018-Chen**,

Ansor-OSDI2020-Zheng, **ahnchameleon**, **Gaussian-ICCV2021-Sun**, **Opevo-XIAO-AAAI**, **AutoGTCO-ICCAD2021-Bai2021**] use XGBoost as the cost model during the search. AutoTVM uses the statistical model and extracts domain-specific features from a provided low-level abstract syntax tree (AST). These features, which include loop structure information and generic annotations, are explored during optimization. Additionally, TreeGRU [**tai2015improved**] recursively encodes a low-level AST into an embedding vector, which is mapped to a final predicted score within a fully-connected layer to enhance performance. The Halide [**Lerning-TOG2019-Adams**] builds regression models with hardware-specific features for auto-scheduling. TabNet [**Tabnet-AAAI2020-Arik**] uses sequential attention to select the most salient features to reason at each decision via a deep tabular architecture. Decision trees are frequently used in classification and regression problems. To enhance their performance, an ensemble learning approach is typically employed to reduce variance. XGBoost [**XGBoost-Chen-KDD**] and LightGBM are powerful feature models in sequence modeling tasks. To achieve accurate prediction, a number of works, including [**OptTensor-NIPS2018-Chen**, **Ansor-OSDI2020-Zheng**, **ahnchameleon**, **Opevo-XIAO-AAAI**, **AutoGTCO-ICCAD2021-Bai2021**, **bai2023gtco**, **huang2023alcop**, **zhao2023high**], use XGBoost as the performance model during the tuning. AutoTVM extracts domain-specific features from a provided low-level abstract syntax tree (AST). During optimization, these features, which include loop structure information and generic annotations, are explored. Moreover, TreeGRU [**tai2015improved**] recursively encodes a low-level AST into an embedding vector, which is mapped to a final predicted score within a fully-connected layer to enhance performance. Halide [**Lerning-TOG2019-Adams**] builds regression models with hardware-specific features for auto-scheduling. TabNet [**Tabnet-AAAI2020-Arik**] uses sequential attention to select the most salient features to reason at each decision via a deep tabular

architecture.

DNN-based Performance Model. In contrast, some recent approaches aim to reduce the impact of search algorithms on final performance by utilizing more robust and powerful cost models. [**Learned-MLsys2020-kaufman**] and [**sun2022gtuner**] employ graph neural networks to predict the latency of DNNs on TPUs. [**Value-mlsys2021-Steiner**] formulates the tuning process as a deterministic Markov Decision Process [**MDP-ICCV-2015**] and solves it by learning an approximation of the value function. Tiramisu [**Tiramisu-CGO2019-Bagh** manually extracts 2534 features from the structure of AST, and forwards the AST as a computation stream to propagate features during the training. These models are trained effectively on a dataset with only a few thousand schedules using the hardware-dependent features crafted by heavy feature engineering techniques. However, complex feature engineering can become problematic in such cases. As hardware-specific features are difficult to transfer to a new platform, a learned performance model trained on one hardware platform typically performs poorly on another. This leads to an issue we call cross-hardware unavailability. Additionally, this approach cannot keep pace with the rapid development of new hardware, which further exacerbates the problem.

2.5 Multi-level Pipelining Optimization

Pipelining. Pipelining, as a GPU kernel optimization, is frequently used in GPU libraries like CUTLASS [**cutlass**]. CUTLASS implements pipelining in matrix multiplication and convolution kernels. However, being a template-based kernel library, CUTLASS is unable to provide automatic pipelining for any tensor programs; this is only possible with our compiler-based solution.

The term pipelining in distributed deep learning training [**huang2019gpipe**,

narayanan2019pipedream, **barham2022pathways**, **zheng2022alpa**] refers to operator-wise parallelism. Pipelining is also a hardware design technique widely used in accelerator designs [**liu2016cambricon**, **jouppi2017datacenter**, **sohrabizadeh2020end**, **liao2021ascend**], or hardware generation languages [**wei2017automated**, **lai2019heterocl**, **wang2021autosa**, **parashar2019timeloop**]. Software-pipelining has been studied to exploit instruction-level parallelism [**ning1993novel**, **govindarajan1996framework**] and multithread parallelism [**wei2012software**]. Compared to all these pipelining scenarios, ALCOP's task is more challenging because it must support multi-level pipelining and must automatically split code into a load- or compute-blocks using IR analysis, whereas, in other settings, the pipeline stages are straightforward. There is a rich amount of work on analytical performance models for GPUs [**hong2009analytical**, **volkov2016understanding**, **wang2020mdm**, **huang2014gpumech**, **zhang2011quantitative**, **baghsorkhi2010adaptive**, **lym2019delta**]. The most relevant is DELTA [**lym2019delta**], which builds a model to predict the latency of Conv2D kernels on GPUs. However, ALCOP is the first to model how pipelining stage numbers affect performance and trade-offs between pipelining and tiling. Recently, static analysis has arisen to supplement the standard ML-based schedule tuning, whose cost model lacks hardware knowledge. Tuna [**wang2021tuna**] builds a performance model for CPU and GPU to replace the ML-based schedule tuning in AutoTVM [**chen2018learning**]. We show that a combination of an analytical model and machine learning can achieve greater search efficiency than the Tuna technique.

2.6 Domain-Specific Accelerator

Gemmini Accelerator Generator. Gemmini is an open-source, full-stack generator for deep learning models, spanning across different hardware architectures,

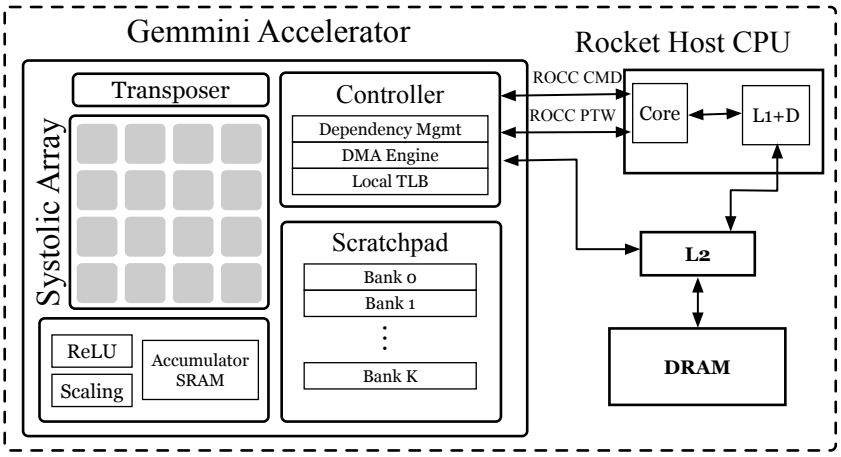


Figure 2.4: *Gemmini hardware architectural template.*

programming stacks, and system integration incorporating customized RISC-V instructions. A typical DNN hardware accelerator has a few key components, as described in Figure 2.4.

- *Off-chip DRAM* used for holding the weights and activation results of the entire DNN models, which needs to be large enough to hold all of the model weights and activation results.
- Smaller on-chip memory, referred to here as the scratchpad memory, which needs to be large enough to hold a subset of the weights and inputs in order to feed the processing elements (PEs).
- An array of PEs, each of which has the capacity to perform matrix multiplication and accumulation operations, and which often contains one or more small local memories called register files (RFs) that can store data with lower per-access energy than scratchpad memory.
- An internal *network-on-chip* (NoC) that can transfer data between all of the PEs.

The Gemmini systolic array-based accelerators was selected as a test platform

because of its open-source nature. Gemmini was developed by the UC Berkeley and is part of the Chipyard ecosystem. It works in a tighly-coupled manner with a RISC-V CPU, using Rocket Chip Coprocessor (RoCC) interface to control the accelerator with help from custom RISC-V instructions. The Gemmini project has two different host CPU configurations. One is a low-power in-order Rocket core and the other is a high-performance out-of-order BOOM core. We first generate a fairly typical DNN accelerator, which is illustrated in Figure 2.4, using the Gemmini accelerator-generator. The accelerators performs general matrix multiplication using a fixed-size systolic array, which can implement the weight-stationary or the output-stationary dataflow. When performing convolutions, the dimensions of the output-channels and input-channels are spatially unrolled. The `8`-bit integer weights and inputs are stored in a 256 KB local scratchpad memory, and the `32`-bit partial sums are stored in a dual-ported 64 KB accumulator SRAM which performs matrix additions. When DNN layers are too large to fit into the local scratchpad, the fall back onto an external L2 cache and DRAM which are shared with CPUs and other accelerators on the system-on-chip (SoC). Gemmini also provides an easy-to-use programming interface so that end users can quickly program the applications for the generated accelerators. Notably, different developers would prefer different software design environments based on the targets or research interests. For instance, model designers would prefer that the hardware programming environment be hidden by model development frameworks like TensorFlow or PyTorch so that the do not need to worry about low-level development details, as in the case of VTA and DNNWeaver. At the same time, framework developers and system programmers may want to interact with the hardware at a low level, in either C or C++ with assembly code, to accurately control hardware states and squeeze every bit of efficiency out, as in the case of MAGNet and Maeri. Unlike other DNN generators that tend to focus on one of these

requirements, Gemmini provides a multi-level programming interface to satisfy users with different requirements. With the assistance of Gemmini, researchers have the capability to generate a wide range of accelerators, ranging from low-power edge devices to high-performance cloud platforms, incorporating customized RISC-V instructions. equipped with Rocket Chip Coprocessor (RoCC) interface to control the accelerator via custom RISC-V instructions. The Gemmini project has two different host CPU configurations. One is a low-power in-order Rocket core and the other is a high-performance out-of-order BOOM core. For a systolic array-based domain specific accelerator, we focus on the low-level instructions because these instructions can model the fine-grained control over the optimization schedules provided by the programmer. The Gemmini systolic array-based accelerators was selected as a test platform because of its open-source nature. Gemmini was developed by the UC Berkeley and is part of the Chipyard ecosystem. It works in a tighly-coupled manner with a RISC-V CPU, using Rocket Chip Coprocessor (RoCC) interface to control the accelerator with help from custom RISC-V instructions. The Gemmini project has two different host CPU configurations. One is a low-power in-order Rocket core and the other is a high-performance out-of-order BOOM core.

Architecture Template. Figure 2.4 presents an overview of the architecture template employed by Gemmini. The primary computation units are spatial processing elements (PEs), responsible for executing dot product operations. The spatial array retrieves data from a locally managed scratchpad, composed of banked SRAMs. The results of the computations are then written to a local accumulator register file, which possesses a higher bitwidth compared to the input data. Regarding other commonly-used kernels in DNNs, such as maxpooling or non-linear activations, they can be seamlessly incorporated with a RISC-V based host CPU for programming purposes. In the Gemmini architecture, the spatial array adopts a two-level hierarchical design,

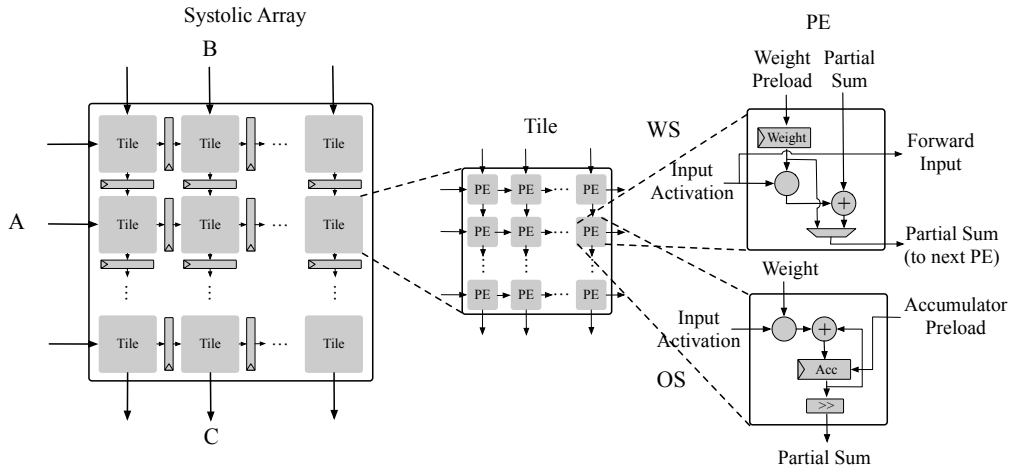


Figure 2.5: *Micro-architecture of Gemmini's two-level spatial array and dataflow.*

which offers a versatile template suitable for diverse micro-architectures. The spatial array consists of "Tilings" which are interconnected through a set of explicit pipeline register files. Each individual Tiling can be further divided into an array of PEs. PEs within the same Tiling are connected combinationaly without the use of pipeline register files. Each PE performs a single multiplication and accumulation (MAC) operation per cycle, utilizing weight-stationary and output-stationary dataflows in Figure 2.5.

Programming Model. Gemmini not only encompasses its own hardware stack but also incorporates a sophisticated hand-tuned library for software performance. This library supports a multi-level software flow to accommodate various programming scenarios. At the high-level, a customized version of ONNX-runtime has been developed. It empowers users to input an ONNX model file and generate executable binaries. It strives to map as many kernels as possible onto the Gemmini-generated hardware accelerator using dynamic dispatch. In cases where no accelerators are available, the execution kernel falls back to the host CPU. On the other hand, Gemmini provides a C/C++ library of kernels, including general matrix multiplication

and convolution layer. These kernels employ loop tiling with heuristics to maximize buffer usage. The Gemmini hardware generators can also be programmed through C/C++ APIs, with tuned implementations for common DNN kernels. The performance of each implementation is dependent on specialized memories, such as the sizes of the scratchpad and register files. When the Gemmini accelerator is generated, the programming stack can produce an accompanying header file that contains various parameters, including the dimensions of the spatial array, dataflows and computation units. The programming stack utilizes pre-defined templates with different tiling sizes to maximize the amount of data moved into the scratchpad per iteration, based on the input dimensions and hardware parameters of the accelerator instantiation during the runtime. Gemmini also enables researchers to integrate different types of RISC-V CPUs with Gemmini-generated accelerators in the Chipyard toolchain, including out-of-order BOOM or in-order Rocket CPUs. These generated accelerators and multiple host CPUs can serve as crucial components of a System-on-Chip (SoC), enabling the parallel execution of multiple computation-intensive workloads.

Hardware Accelerator ISA. For the low-level programmable paradigm, the Gemmini ISA has two versions: complex and primitive instructions. For the primitive instructions, Gemmini has `mvin`, `preload`, `compute`, `mvout`. These primitive instructions are used to operate on DIM $\times$ DIM submatrices. We can use these instructions to explicitly manage scratchpad memories. In the mid-level programming paradigm, Gemmini has some interfaces to integrate these functions into loops for the computation. As for the complex instructions, Gemmini hardcodes these loops with `loop_matmul` and `loop_conv`. In this paradigm, the scratchpad memory is not explicitly managed by programmers. In the above complex instructions, all of the hyper-parameters are fixed.(maybe we can use some dynamic scheduling on it.

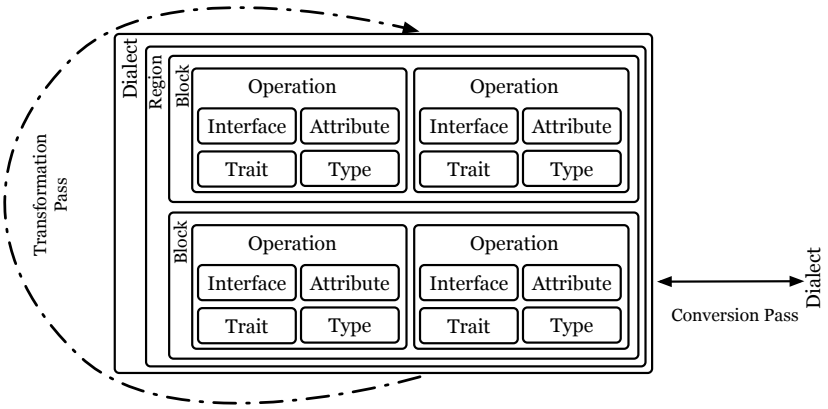


Figure 2.6: *Key concepts related to MLIR compilation framework.*

2.7 MLIR Compilation Infrastructure

MLIR [**lattner2021mlir**] is a collection of modular and reusable software components that enable the progressive lowering of high-level operations to efficiently target hardware in a common way. For the implementation of BACO, we use the open-source MLIR infrastructure. MLIR is motivated by the observation that early lowering to a low-level intermediate representation (IR) in a conventional compiler loses much of the high-level semantics and structure of an workload, because it cannot be described in the low-level IR. To enable the compiler to reason about high-level semantics and achieve better compilation results, it is desirable to represent the application by a hierarchy of representations, gradually lowering them to a low-level IR. To facilitate the implementation of such representations at various levels of abstraction and avoid the repeated implementation of infrastructure components, the MLIR framework aims to provide a reusable infrastructure for the compiler researchers.

The basic organizational unit for IRs are the `Dialects`, which can define a set of "Operations" and "Types". While the MLIR framework also uses the

static single assignment (SSA) form, its representations are more flexible than the traditional control flow graph (CFG) representations in LLVM IR. Unlike the previous CFG structures, MLIR allows for loop-nesting inside the IR information. Each “Operations” does not only produce typed values with SSA, but can also have multiple “Blocks”. Additionally, "Attributes" can be attached to “Operations” to represent additional compilation information. In general, a typical compilation flow utilizes multiple `Dialects`, and mixtures of them to represent the workload at different abstraction levels. As the flow progresses, and moves to low-level `Dialect`, lowering is used to translate between different `Dialects`. MLIR provides a number of generic transformations in compilation, such as constant folding, through the use of “Traits” and “Interfaces” that can be attached to "Operations". These are provided in addition to other common infrastructure, such as pass managers. Next, we will highlight some important concepts for the key terms related to MLIR mentioned above as described in Figure 2.6.

Dialects. A `Dialect` defines a namespace for a group of related operations, attributes, and types. MLIR provides multiple built-in `Dialect` to represent common functionalities, and also features an open infrastructure that allows users to define new `Dialect` at different abstraction levels.

Attributes. Attritbutes are used to identify specific features of operations. An operation that initializes a constant may have an attribute that defines its value. The `std.const` operation which uses the `value` attribute for this purpose.

Blocks. Blocks are defined similarly to other IRs with a few differences. First, unlike the classical definition in compilation [**lattner2004llvm**], blocks in MLIR have operations that can contain regions. Another difference is the concept of block arguments. These arguments abstract control-flow dependent values that indicate which SSA values are available in each block.

Regions. Regions in MLIR are a combination of blocks that can map to either a classical reducible CFG or not, depending on how the input language generates its MLIR representation. For instance, some languages such as Fortran implement classical reducible CFG regions as an MLIR region with a single basic block, while irreducible parts of the CFG are implemented using a region with multiple basic blocks.

Pass. Passes are a key compownent of compilation that traverses the intermediate representation (IR) for the purpose of analysis and optimization. Similar to the LLVM infrastructure, compiler researchers can design *transform* and *analysis* passes in MLIR to perform IR transformations and analysis, respectively. Note that *transform* typically indicates the transformation within a specific dialect. The transformation between different levels of dialects is defined as *conversion*, while the transformation between MLIR dialects and external representation is referred to as *translation*. Additionally, *Lowering* is a terminology that refers to the process of lowering the different abstraction level of the IR in the compilation flow.

2.8 Programmable Spatial Accelerator

Hand-optimized Libraries on Accelerators. Popular DL acceleration libraries such as cuDNN [**chetlur2014cudnn**], cuBLAS [**cublas**]and CUTLASS [**cutlass**, **markidis2018nvidia**] are developed to use NVIDIA Tensor Core for the tensor computation. As for the Intel CPUs, oneDNN [**li2023onednn**] are developed to use special instructions to perform high-performance computation. These libraries are implemented by manual optimization and need lots of engineering efforts. DL frameworks such as TensorFlow [**abadi2016tensorflow**], MxNet [**chen2015mxnet**] and PyTorch [**paszke2019pytorch**] all leverage these libraries to optimize performance. The Gem-

mini generator [**genc2021gemmini**] produces not just a hardware stack, but also a tuned software stack, boosting developer's productivity for different hardware accelerator specifications including import a DNN and computation graph in the ONNX [**onnx-mlir**] can also be programmed via C/C++ code.

Automatic Generation Tensor on Accelerators. Halide [**ragan2013halide**] introduces the concept of *algorithm* and *schedule* to represent the description and optimization, while TVM [**chen2018tvm**] generalizes this concept and allows performance engineers to use *tensorize* primitive to lower part of the *for-loop* fragments on the spatial accelerators manually. Automatic schedulers, such as Halide scheduler [**halide-scheduler**], FlexTensor [**zheng2020flextensor**], ALT [**zeng2020alt**], Rammer [**ma2020rammer**], Roller [**zhu2022roller**], DGP [**sun2021fast**], AutoGTCO [**bai2021autogtco**], ALCOP [**huang2023alcop**] ,and Ansor [**zheng2020ansor**], have focused on optimizing general-purpose hardware and have not addressed code generation for systolic array-based accelerators [**kung1979systolic**] like Gemmini. Meanwhile, existing compilers that generate code for spatial accelerators with intrinsic functionality, such as XLA [**sabne2020xla**], AutoTVM [**chen2018learning**], ISA Mapper [**sotoudeh2019isa**], and UNIT [**weng2021unit**], rely on hand-written templates for schedules without the support of RISC-V instructions. Unfortunately, creating these templates is a time-consuming process, which limits the range of supported operators. TensorIR [**feng2023tensorir**] automatic schedules algorithm and performs tensorization jointly with other optimizations and generates performant programs on GPU. Hidet [**ding2023hidet**] constructs an efficient hardware-centric schedule space, which is agnostic to the program input size and greatly reduces the tuning time with task mapping Paradigm on Tensor Cores. The VeGen [**chen2021vegen**] compiler can automatically generate templates for intrinsics on Intel CPU with AVX-512 instruction. Polyhedral compilers such as AKG [**zhao2021akg**] relies on

a combination of polyhedral model and templates to map software onto spatial accelerators. However, above methods are limited in generalizing to RISC-V based hardware accelerators. Exo [**ikarashi2022exocompilation**] is an embedded DSL and schedules are written as meta-programs in the Python. It transforms a simple program into an equivalent, but more complex and high-performance version, targeted to the Gemmini and x86 CPUs via successive rewriting of the application via scheduling.

Instruction Selection Traditional method of instruction selection [**lam2006compilers**] applies local pattern matching rules to replace small IR fragments with equivalent instructions. However, it struggles to effectively exploit accelerator instructions which correspond to large, complex program fragments. Recently, some work apply more powerful search technique to target more complex SIMD instructions using program synthesis [**phothilimthana2019swizzle**] and equality saturation [**vanhattum2021vectorization**]. TVM offers users a tensorization directive that allows them to replace loop fragments with equivalent instructions. However, it does not possess the flexibility to combine automation and verification, which is a key feature of BACO's unified compilation flow.

2.9 Computation on Spatial Accelerator

Figure 2.7 illustrates an implementation of matrix multiplication $\boldsymbol{C} = \boldsymbol{A} \times \boldsymbol{B}$ on the spatial array with weight-stationary (WS) dataflow. Firstly, we decompose the matrix multiplication into independent sub-blocks by tiling the M and N dimensions. After the tiling, it has $\frac{M}{Tile_M} \times \frac{N}{Tile_N}$ independent sub-blocks. Each sub-block is a matrix multiplication with size: $Tile_M \times Tile_N \times K$. Then, the fragments of $\boldsymbol{A}$ with size of $Tile_M \times K$ and $\boldsymbol{B}$ with $K \times Tile_N$ are loaded from DRAM to scratchpad memory. As

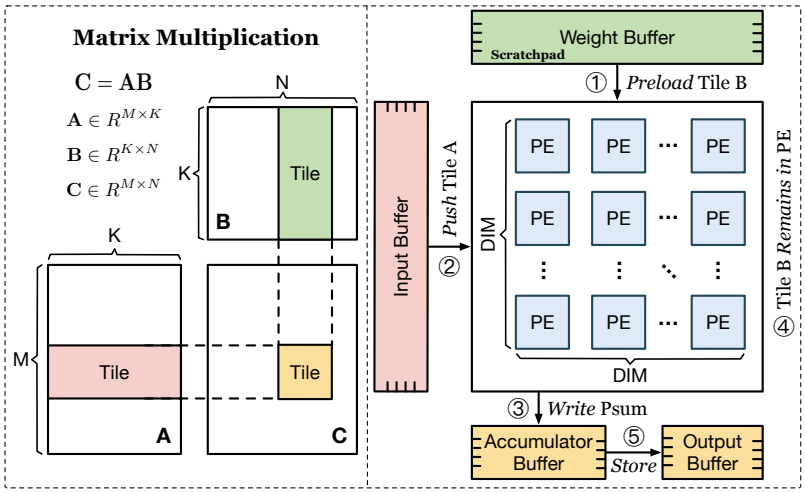
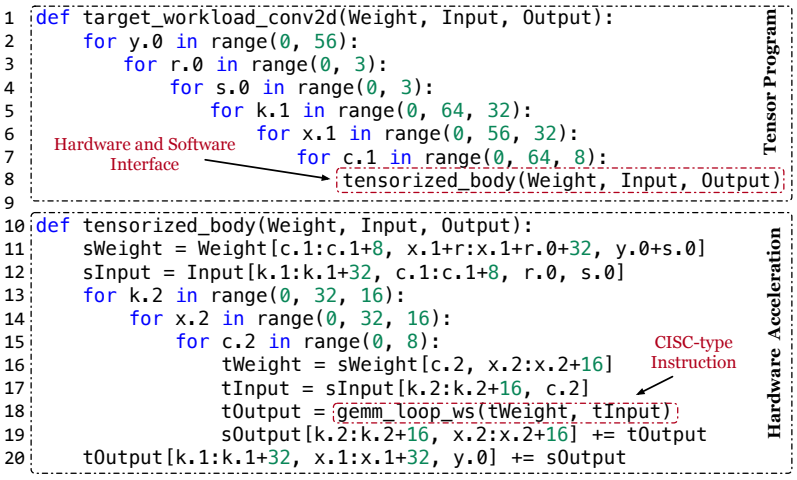


Figure 2.7: *Matrix multiplication with weight-stationary dataflow.*

shown in step ①, the elements of the ***B*** need to be pre-filled and stored into each PE before the computation for the weight-stationary dataflow. Then the elements of the ***A*** is pushed through the array in step ②, and each PE generates one partial sum every cycle in step ③. The users can reuse the already loaded weights ***B*** in the ④. The generated partial sums are then reduced along each column in parallel to generate the output and store the results to DRAM in step ⑤. There are two ways to implement the kernel: *i)* directly write the primitive instructions to explicitly manage data movement and computation, or *ii)* use CISC-type instructions to hardcode all of the for-loop optimization. The CISC instructions have too many operands to fit into a single primitive instruction. Therefore, they are implemented as a sequence of many primitive instructions which must be called consecutively by the users.




```
1  def target_workload_conv2d(Weight, Input, Output):
2      for y.0 in range(0, 56):
3          for r.0 in range(0, 3):
4              for s.0 in range(0, 3):
5                  for k.1 in range(0, 64, 32):
6                      for x.1 in range(0, 56, 32):
7                          for c.1 in range(0, 64, 8):
8                              tensorized_body(Weight, Input, Output)
9
10 def tensorized_body(Weight, Input, Output):
11     sWeight = Weight[c.1:c.1+8, x.1+r:x.1+r.0+32, y.0+s.0]
12     sInput = Input[k.1:k.1+32, c.1:c.1+8, r.0, s.0]
13     for k.2 in range(0, 32, 16):
14         for x.2 in range(0, 32, 16):
15             for c.2 in range(0, 8):
16                 tWeight = sWeight[c.2, x.2:x.2+16]
17                 tInput = sInput[k.2:k.2+16, c.2]
18                 tOutput = gemm_loop_ws(tWeight, tInput)
19                 sOutput[k.2:k.2+16, x.2:x.2+16] += tOutput
20     tOutput[k.1:k.1+32, x.1:x.1+32, y.0] += sOutput
```

Figure 2.8: *Manually writing a convolution kernel with CISC-type instructions.*

2.10 Complex Hardware and Software Interfaces

The design of accelerated hardware and tensor program presents a distinctive engineering task, characterized by several unusual aspects. Firstly, unlike traditional programs running on general-purpose processors, the hardware-software interfaces for domain-specific accelerators are complex, encompassing specialized memories, exposed configuration state, and intricate operations. Moreover, these interfaces exhibit a high level of diversity, with each accelerator having its unique complexities. We use an example of executing convolution kernel from ResNet-50 on the spatial accelerator with CISC instruction in Figure 2.8. Secondly, the rates of change within the stack are inverted compared to conventional systems. Accelerator architectures evolve at a faster pace than the essential kernels they execute, and the optimization of these kernels to effectively utilize the hardware undergoes even more rapid iterations. This phenomenon is particularly pronounced during accelerator design, where the target application workloads often remain fixed, while both the hardware architecture and program transformation are co-designed iteratively to achieve

Table 2.1: *Comparsions with existing optimization frameworks for hardware and software design.*

Frameworks	Productivity			Performance			Automation		
	Frontend	IR Level	User-specific	Target Usecase	ISA Support	Hardware	Co-search	Multi-layer	Auto-Tuner
MLIR [lattner2021mlir]	Nelli [**levental2023nelli**]	Multiple	✓	Compilation	General	General	-	-	-
TVM [chen2018tvm]	DSL	Single	✓	Compilation	General	General	-	-	Evolutionary Search
AKG [zhao2021akg]	DSL	Single	✓	Compilation	General	Spatial	-	-	Genetic Search
TensorIR [feng2023tensorir]	DSL	Single	✓	Compilation	General	General	-	-	Evolutionary Search
AMOS [zheng2022amos]	DSL	Single	✓	Compilation	General	General	-	-	Genetic Search
Exo [ikarashi2022exocompilation]	DSL	Single	✓	Compilation	RISC-V	Spatial	-	-	-
Timeloop [parashar2019timeloop]	-	-	-	Co-explore	-	Spatial	-	-	Genetic Search
FAST [zhang2022full]	-	-	-	Co-explore	-	Spatial	Two-loop	-	Integer Linear Programming
MAGNet [venkatesan2019magnet]	-	-	-	Co-explore	-	Spatial	Two-loop	-	Heuristics
HASCO [xiao2021hasco]	-	-	-	Co-explore	-	Spatial	Two-loop	-	Reinforcement Learning
Interstellar [yang2020interstellar]	-	-	-	Co-explore	-	Spatial	One-loop	-	Heuristics
DiGamma [kao2022digamma]	-	-	-	Co-explore	-	Spatial	One-loop	-	Genetic Search
DOSA [hong2023dosa]	-	-	-	Co-explore	-	Spatial	One-loop	✓	Gradient Descent
BACO	DSL	Multiple	✓	Co-explore with compilation	RISC-V	Spatial	One-loop	✓	Gradient Descent

optimal performance and efficiency. Therefore, we need a more productive method to perform hardware and software co-exploration.

2.11 Hardware-Software Co-design Framework

There has been a growing body of research as shown in Table 2.1 focused on optimizing the hardware and software design with the goal of achieving performance and efficiency. Most prior work treats the program transformation and hardware co-exploration as a *two-loop search* and utilizes a combination of various optimization techniques to solve each space independently. The inner loop demonstrates the software optimization and the outer loop represents the hardware optimization. In the first stage, the optimizer samples accelerator configurations from hardware space and then optimizing for optimal program for that specialized hardware within the inner loop. In the second stage, the best program is used for generating the hardware performance feedback for the outer loop optimization. There are three common optimizers: heuristics, block-box (BB) and white-box optimization.

There are also some works to use *one-loop search*. The mechanism tackles it from a mapping-first approach that infers the minimal accelerator configuration

with *instruction-agnostic* program found in the mapping space, and the hardware space is similar in size to the mapping size. [**kao2022digamma**] employs BB-GA which treats the mapping performance as a block-box and needs to evaluate many unique hardware and software points iteratively to achieve a good performance. DOSA [**hong2023dosa**] takes a novel technique by formulating the analytical performance model as a differentiable white-box model. It then employs gradient descent to optimize the mapping variables in the direction of steepest descent.

Chapter 3

GTCO: Graph and Tensor Co-Design for Transformers on GPUs

3.1 Motivation

Recent years have witnessed the success of deep learning in the industry-scale application, ranging from language translation, virtual reality, and recommendation systems to computer vision and autonomous driving. In particular, convolutional neural networks (CNN) remain dominant in computer vision [**IMGC-CVPR2016-ResNet**, **IMGC-ICLR2015-VGG**, **IMGC-CVPR2016-Inception**, **CV-ICME-Pose**]. Despite their significant success, attention has been increasingly focused on integrating self-attention techniques with CNN-based models [**ATTN-NIPS2017-Vaswani**], inspired by their success in neural language processing (NLP). Many model architectures have entirely replaced CNNs with transformer-based models [**ATTN-2021-TransformersSurvey**], promoting the further application of these models in various vision tasks [**ATTN-ECCV2020-DETR**, **ATTN-2021-SelfSupervisedTransformer**, **ATTN-CVPR2021-SETR**, **ATTN-arXiv2020-ViT**]. This trend paves the way for using a unified transformer architecture in future re-

search developments.

Optimizing the performance of different operators with diverse hardware platforms requires a significant engineering effort when using these vendor-provided libraries. As a result, researchers and engineers often concentrate on enhancing the performance of compute-intensive primitives, such as GEMM and convolution, which are frequently employed in CNN architecture. Alternatively, they turn to a search-based compilation approach [**Compiler-SIGGRAPH2012-Halide**, **Compiler-OSDI2018-TVM**] that involves separating kernel definition from computation scheduling to automatically generate tensor programs.

While prior research has primarily focused on optimizing the deployment of CNN models, the potential of transformer-based vision models on modern accelerators has not been effectively leveraged due to the specialized self-attention modules. Although existing libraries have the capability to fuse element-wise operators into compute-intensive kernels, they are not as effective in optimizing memory-intensive workloads, such as matrix multiplication with softmax operators. Furthermore, transformer-based models typically involve a large number of fine-grained operators, which can result in significant overheads when implemented on specific hardware, such as GPUs. Meanwhile, due to the predefined rules for optimizing compute-intensive operators, traditional techniques such as Halide [**Compiler-SIGGRAPH2012-Halide**, **Compiler-OSDI2020-Ansor**] cannot effectively utilize the inference efficiency of specialized modules in transformer models.

Numerous techniques have been proposed to enhance the efficiency of models by optimizing them at the graph-level. Lots of work are proposed to facilitate the optimization from the graph-level to improve the efficiency of the model. TensorRT [**GPU-NVIDIA-TensorRT**] utilizes a two-step fusion policy to optimize the

computation graph. Firstly, specific operators such as fully-connected, convolution, batch normalization [**CV-ICML2015-BN**], and ReLU [**CV-CoRR-ReLU**] are fused vertically using rules designed by high-performance computing engineers. Secondly, the operators are fused horizontally within the same stage. This two-step optimization method works well for CNN architectures with multiple parallel branches and kernels of the same size. Greedy rule-based subgraph substitutions are employed to optimize the computation graph in classical frameworks such as TensorFlow [**DL-OSDI2016-TensorFlow**], PyTorch [**DL-NIPSW2017-PyTorch**], TVM [**Compiler-OSDI2018-TVM**], and Ansor [**Compiler-OSDI2020-Ansor**]. While template-based substitutions may enhance the efficiency of computations, they are not suitable for long-term maintenance. With new operators continuously being proposed from a high-level model perspective, the rule-based graph substitution approach becomes increasingly unsuitable for real-world production, as it requires significant engineering effort. To search potential substitutions, search-based methods are introduced by TASO [**Compiler-SOSP2019-TASO**] and IOS [**Compiler-MLSys2021-IOS**] to exploit a large enough search space. TASO generates graph substitutions with a formal verification to verify the correctness of the optimized graph substitutions automatically. Search-based methods have been introduced to tackle the issue of finding potential substitutions for new operators. TASO [**Compiler-SOSP2019-TASO**] and IOS [**Compiler-MLSys2021-IOS**] are two examples of search-based methods that are capable of exploring a vast search space. TASO generates graph substitutions and verifies the correctness of the optimized graph substitutions through formal verification techniques. While IOS is able to leverage inter-operator and intra-operator parallelism to schedule operators with CUDA stream, thus maximizing the benefits of both software and hardware accelerators, it is not able to fully exploit the code generation capabilities from a

compilation perspective for the implementation of each operator. This is due to the IOS using the vendor-provided library cuDNN to do the runtime, which provides a fixed template for each operator with limited runtime performance. As a result, searching for optimal solutions with operator fusion from a comprehensive search space is not possible using this paradigm.

The NVIDIA GPUs' domain-specific accelerators (Tensor Cores) are programmed using instruction set architecture, which allows for algorithmic specification to be separated from hardware architectural details. These instructions are commonly referred to as intrinsic, and using them for tensor computation is known as tensorization. While intrinsics provide programmability, mapping specific intrinsics remains a challenging task. For example, there are 35 different ways to map the seven for-loops of a 2D convolution implementation on the Tensor Cores. Mapping performance is crucial to configurations that can impact data locality and parallelism on GPUs. However, current compilers [**Compiler-OSDI2018-TVM**, **Compiler-OSDI2020-Ansor**] rely heavily on manual programming with hardware intrinsic to develop high-performance implementation, which may overlook optimal mapping choices between the complex memory hierarchy. To efficiently support algorithms on domain-specific accelerators, an automated mapping solution is necessary to explore software and hardware co-design. While the baseline framework is presented in [**bai2021iccad**], our framework is called GTCO.

This work presents a solution to the automatic mapping problem on domain-specific accelerators (Tensor Cores) by introducing a novel abstraction above the hardware intrinsics. The abstraction comprises two components: *computation* and *memory* abstraction, which describe the computation and data movement behaviors within an intrinsic. Based on the proposed abstraction, this work develops a two-step mapping generation method that can map software computations to a virtual

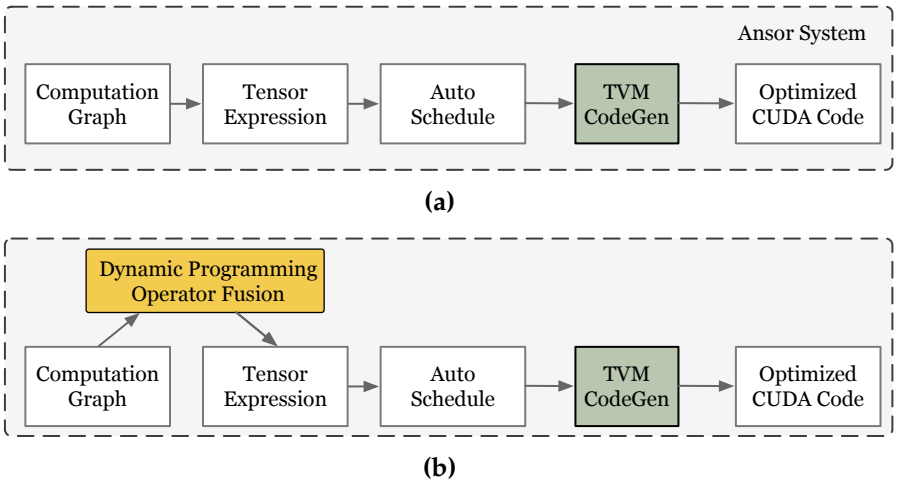


Figure 3.1: *The overview of (a) Ansor [***Compiler-OSDI2020-Ansor***] and (b) [***bai2021iccad***]. Initially, the input is a computation graph, which is converted into tensor expression language. Subsequently, the auto-schedule module is capable of automatically searching for the optimal schedule for each operator. Finally, TVM code generation is performed to generate optimized CUDA code on GPU. However, the primary distinguishing factor between the two systems is the Dynamic Programming Operator Fusion module.*

hardware accelerator without hardware constraints and then modify the mapping configurations based on actual physical constraints. This work is based on Ansor, shown in Figure 3.1. Additionally, the authors efficiently explore the search space to achieve low inference latency on domain-specific accelerators (Tensor Cores). Finally, the authors implement all of the techniques and integrate them into an end-to-end tensor compilation named GTCO.

3.2 System Overview

Figure 3.2 illustrates the overall architecture of our tensor compilation system, which comprises four essential modules: dynamic programming operator fusion, subgraph scheduler, program sampler, and performance tuner. Starting with the input of the transformer-based model lacking the operator fusion technique, each operator is

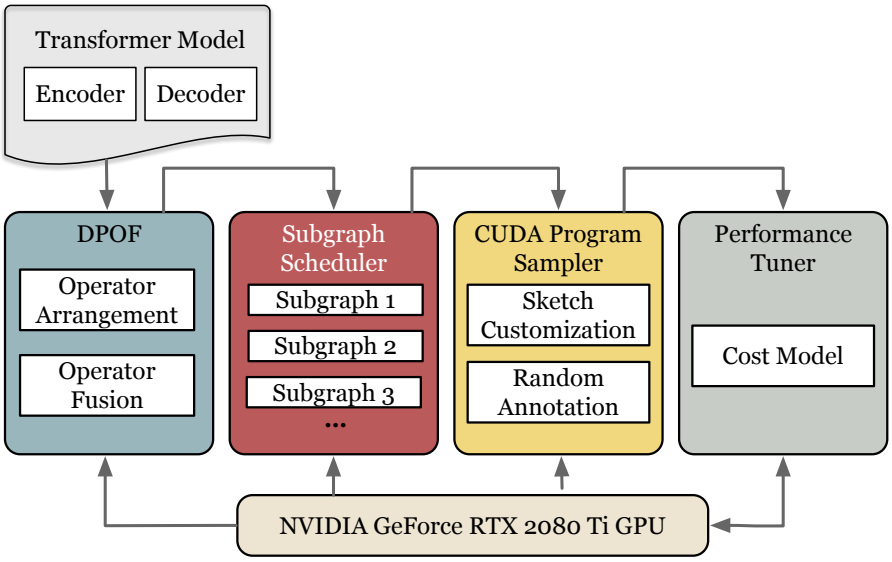


Figure 3.2: *The workflow and components of our framework. The input is the transformer-based vision models and the output is the tensor programs generated on the GPU platform.*

initially labeled with a pattern that denotes the relationship between the connected operators. After applying the operator fusion technique, each operator is assigned a new pattern, and the types of the connected operators are also altered based on the predicted labels that arise from the operator fusion. High-performance tensor programs with hardware intrinsic are generated for the fused operators on the GPU platform. All in all, our framework includes the following components:

- A DPOF module that can find an optimized operator fusion schedule for the transformer-based vision model.
- A subgraph scheduler module assigns time slots for optimizing fused subgraphs optimized by the DPOF.
- A program sampler module that constructs a comprehensive search space and samples different kinds of tensor programs from it randomly to ensure diversity.

- A performance tuner module trains a regression-based model to predict the performance of sampled programs.
- An automatic mapping flow with compilation techniques that can find the optimal implementation of the fused operators in transformer-based models on Tensor Cores with floating point 16 (`FP16`) datatype.

3.3 Problem Formulation

This section defines the operator fusion strategy and formulates the problem.

Computation Graph. The model is defined using a computation graph $G = (V, E)$, where V is the set of vertices and E is the edge set. Each vertex in the graph represents an operator such as GEMM or softmax. As for the edge $(u, v) \in E$, it represents a tensor that can store the input of operator v and the output of operator u. Each vertex can represent an operator such as GEMM and softmax operation in the computation graph. Each edge $(u, v) \in E$ is a tensor that can store the output of operator u and the input of operator v. Computation graphs are a common way to represent deep learning models in frameworks or compilers. Figure 2.1 shows the computation graph in the transformer-based model.

Operator Pattern. Operator fusion is a very efficient technique to optimize memory-bound workloads. To circumvent the need for storing intermediate results in global memory, operator fusion merges multiple connected operators into a single computation kernel. This optimization technique can substantially enhance the inference speed-up, particularly in throughput-oriented architectures like GPUs. In order to effectively do operator fusion, the operator patterns must first be defined. We categorize operators into five distinct patterns: (1) *injective*, (2) *element-wise*, (3) *opaque*, (4) *reduction*, and (5) *complex-out-fusable*. Meanwhile, generic rules are provided to

fuse the operators. More details can be found in [**Compiler-OSDI2018-TVM**]. We observe that matrix transposition, layer normalization, batch matrix multiplication, softmax, and fully connected layers frequently occur in transformer-based models. Furthermore, the default configuration of them adheres to these guidelines: i) softmax are marked as the *opaque* pattern; ii) Dense and batch matrix multiplication are identified as the *complex-out-fusible* pattern; and iii) Layer normalization [**CV-CoRR-LN**] can be decomposed into a set of fundamental operators (multiple, add, subtract), which are labeled as the *element-wise* pattern.

Fusion Scheduling. Based on each computation graph G extracted from the transformer-based model, a corresponding schedule S is defined to optimize the inference latency on the GPU platform as follows:

$$\mathrm{S} = \{(\mathrm{V}_1, \mathrm{F}_1), (\mathrm{V}_2, \mathrm{F}_2), ..., (\mathrm{V}_k, \mathrm{F}_k)\},$$

where V_i denotes a set of computation operators in the i-th stage and F_i denotes the paired variable that describes the fusion relationship between any two nodes. The execution of G with the schedule S is carried out consecutively from the first stage (V_1, F_1) to the last stage (V_k, F_k).

Problem Formulation. Given a computation graph G and fusion schedule S on GPU, our objective is to search for a schedule S^*:

$$\mathrm{S}^* = \underset{S}{\operatorname{argmin}}\ \mathrm{Cost}(\mathrm{G}, \mathrm{S}),$$

where Cost is the execution time of G with the schedule S.

3.4 Workflow of GTCO

3.4.1 Dynamic Programming-based Operator Fusion

Initially, the execution order of computation operators in the original graph is determined by a topological sorting algorithm, which serves as our starting point for finding an optimized schedule. Next, we use a computation queue to store the selected operators from the topological sorting algorithm. Rather than using a more complex graph data structure, we employ a queue data structure to identify and store the optimal schedule. The variables in the queue can be divided into two categories: placeholder and computation variables. Placeholder variables are used to store input and output results, which do not affect the execution time of the entire computation graph and are therefore not taken into account. As introduced in Section 3.3, the same operator with various labels makes the difference in the execution stage. Thirdly, we assume that no fusion relationship exists between the operators at the beginning stage, and all operators are assigned to an opaque state. The size of the queue is determined by the maximum number of the stage defined in the scheduling Section 3.3. As discussed in Section 3.3, the same operator with different patterns can have a significant impact on the execution stage.

Operator Fusion. The computation graph $G = (V, E)$ is initially partitioned into two sets: V' and $V - V'$ based on the execution order of the computation variables and the maximum number of the stages during the scheduling. In a set of V', the edges in a set of $V - V^{'}$ have a pointing relationship with the directed edges. Specifically, the start points of the edges are in $V - V^{'}$ and the end points are in V'. The set of vertices V' is defined as the segmentation set. The interplay between $V^{'}$ and $V - V^{'}$ is illustrated in Figure 3.3. We observe that the computation graph

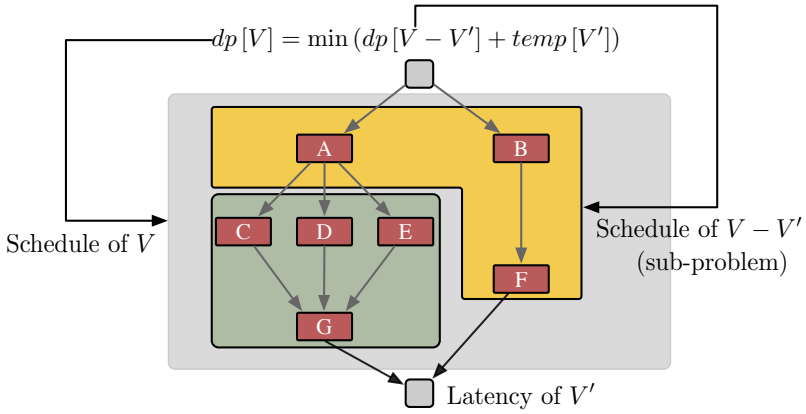


Figure 3.3: *The design of dynamic programming operator fusion.*

exhibits numerous segmentation sets. According to the dynamic programming algorithm, we can systematically enumerate the elements in the segmentation sets V' of V. This technique transforms the original problem into a sub-problem that aims to determine the optimal schedule for $V - V^{'}$. Consequently, the computation graph G can be recursively optimized for each element in the segmentation set. The dynamic programming approach defines $dp[V]$ as the execution time of the computation graph G with an optimal schedule S in the nodes set V. Additionally, $temp[V^{'}]$ represents the execution time of the subgraph composed of the nodes in the stage $(V^{'}, F)$. Here, F represents the optimal fusion strategy in the segmentation set $V^{'}$. The state transition equation can be defined as follows:

$$dp[V] = \min_{v \in V^{'}}(dp[V - V^{'}] + \sum_{v} temp[v]). \tag{3.1}$$

In Algorithm 1, v represents a node in the segmentation set $V^{'}$, and $dp[\varnothing]$ is the boundary value of the state transition equation, set to 0. The optimal solution is obtained by storing each node v in the segmentation set $V^{'}$ and measuring the execution time of each V through $action[V]$.

Algorithm 1 Operator Fusion Strategy

Require: A computation graph $G = (V, E)$ with the **opaque** type for $\forall v \in V, pattern(v) = 0$;
Ensure: A operator fusion strategy with the type of each operator $v \in V, pattern(v)$;

Defining $dp[\varnothing] \leftarrow 0$, $dp[V] \leftarrow +\infty$, $action[V] \leftarrow \varnothing$;
Defining $S \leftarrow [\varnothing]$ (A Stack data structure to store the phase of optimal schedule for operator fusion);

function SELECTSCHEDULE(G)
    V = all operators in computation graph G;
    `Scheduler`(V);
    while $V \neq \varnothing$ **do**
        $V^{'}, F = action[V]$;
        Put phase $(V^{'}, F)$ into the stack S;
        $V = V - V^{'}$;
    return the Fusion Strategy S;

function SCHEDULER(V)
    if $dp[V] \neq +\infty$ **then**
        return dp[V];
    for all $v \in V^{'}$ **do**
        $T_{V'}, F_{V'} =$ `PhasePartition`$(V^{'})$;
        $T_V =$ `Scheduler`$(V - V^{'}) + \sum_{v_i \in V'} T_{V'}$;
        if $T_V \leq dp[V]$ **then**
            $dp[V] = T_V$;
            $action[V] = (V^{'}, F_{V'})$;
    ;
    return $dp[V]$;

function PHASEPARTITION($V^{'}$)
    for all operators $v_i \in V^{'}$ **do**
        if $pattern(v_i, v_j) \neq opaque$ **then**
            $T_{\text{fused}(i,j)} = \text{Runtime}(pair(v_i, v_j))$;
        else
            $T_{\text{fused}(i,j)} = +\infty$;
    return $T_{\text{fused}(i,j)}, pattern(v_i, v_j)$;

3.4.2 Subgraph Scheduler

It is evident that partitioning a model into different subgraphs is a crucial step before performance optimization. However, it is futile to invest significant time in tuning subgraphs without the possibility of enhancing the execution performance of the model during optimization. Therefore, we opt to dynamically assign varying amounts of time slots to different types of subgraphs. For transformer models,

a subgraph may occur multiple times. As a result, achieving a well-optimized transformer-based model requires resolving numerous scheduling tasks during compilation.

We integrate three objectives during the tuning process: i) reducing the total execution time of the transformer-based models; ii) meeting requirements of execution time for various subgraph; iii) decreasing the overall tuning time when certain subgraphs already meet the requirement and cannot be significantly improved. To achieve this, we define an assignment vector as t, where t_i denotes the time slots assigned to the *i-th* task. Initially, all t values are set to $(1, 1, ..., 1)$. We then define $g_i(t)$ as the minimum execution time required for the *i-th* subgraph under task t_i. The subgraph execution times $f(g_1(t), g_2(t), ..., g_n(t))$ represent the end-to-end execution time of a transformer-based model, and our objective is to minimize the following function. To minimize the end-to-end execution time of the transformer, the objective function is defined as follows:

$$f = \max \left[\sum_{i=1}^{n} w_i \times \max(g_i(t), ES(g_i, t)), L_j \right]. \tag{3.2}$$

The number of search task occurrences is denoted as w_i, where i is the task index. It is important to note that if the latency requirement is already satisfied, no tuning time slots will be allocated for a subgraph i.

Thus, the latency requirement of subgraph j is represented as L_j. Additionally, we define a function $ES(g_i, t)$ to enable early stopping by utilizing the historical log information of the i-th task. Our framework differs from other frameworks in that it compares the execution and early stopping configurations. Furthermore, we optimize each search task sequentially. Finally, a scheduling design based on the gradient descent method is developed to efficiently solve the objective function.

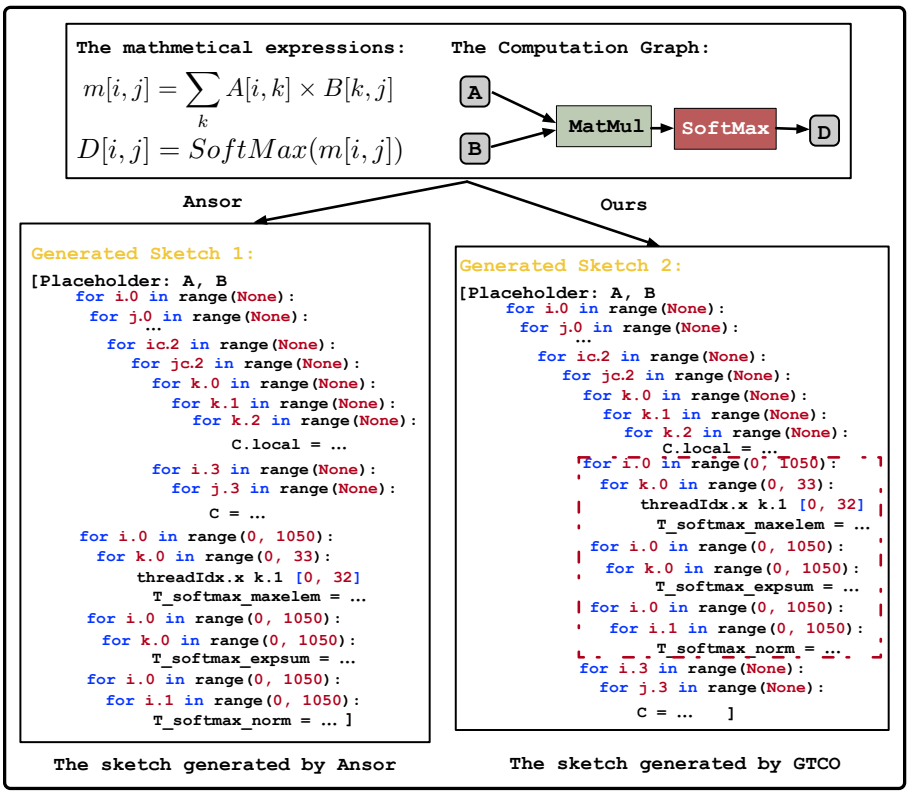


Figure 3.4: *Sketch generation for the subgraph of MHA. This figure shows two generated sketches. The left one is generated by the default Ansor [***Compiler-OSDI2020-Ansor***] and the right one is generated by [***bai2021iccad***]. The difference between these two sketches is that the operator fusion occurs in GTCO with "red-dotted". The code example is pseudo-code in a Python-like syntax.*

3.4.3 Program Sampler

A hierarchical search space is defined to sample the tensor program, which is based on two techniques: high-level sketch generation and low-level annotation sampling. The high-level information is encapsulated in the sketches, and millions of low-level choices are made to obtain specific optimization, such as blocking size, virtual thread tiling, and cooperative fetching, as the final annotations. To generate high-level sketches for each subgraph, computation nodes are visited in topological order, and a generation structure is built iteratively with multi-level for-loop nests. For computation-intensive nodes with a higher likelihood of data reuse, such as batch

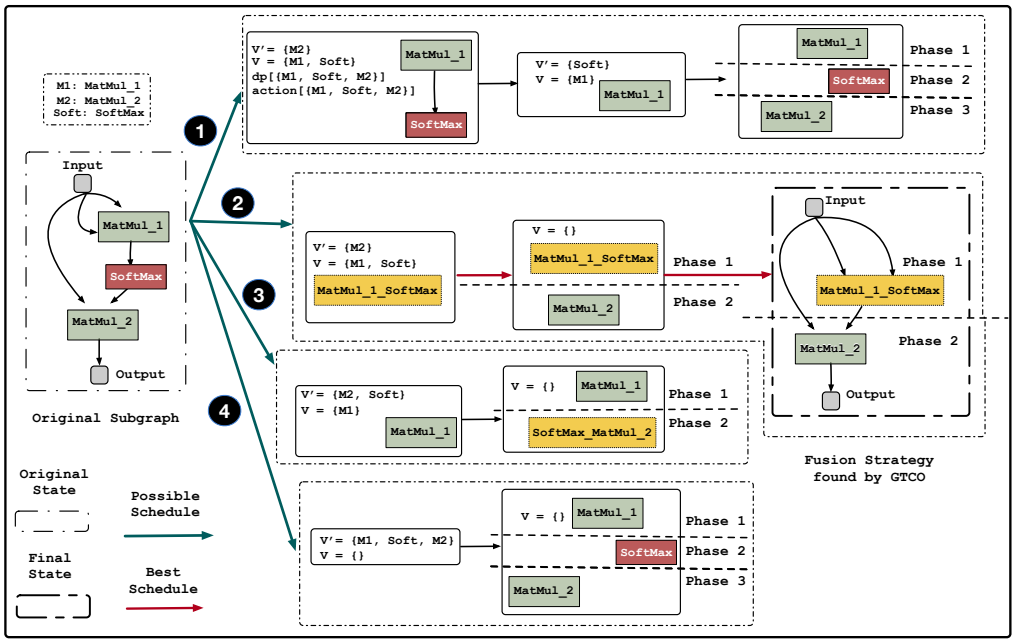


Figure 3.5: *An example to illustrate how DPOF finds the fusion strategy. The original computation graph is shown on the left. It has three operators, M1,M2, and Soft. There are 4 states during the dynamic programming algorithm and each transition is shown in the figure. Any transition starts from the state $V = \{M1, Soft, M2\}$ to $V = \{\}$. The best fusion strategy can be obtained by the dynamic programming process.*

matrix multiplication, standard loop tiling, and fusion strategies are implemented as the sketch generation technique.

As illustrated in Figure 3.4, a running example is presented to elucidate the process of generating high-level sketches for a common subgraph that comprises matrix multiplication ($[1050, 8, 32] \times [32, 8, 1050]$) and softmax operators in the multi-head attention mechanism. The computation nodes are sorted in the order of $(\boldsymbol{A}, \boldsymbol{B}, MatMul, Soft, \boldsymbol{D})$. Starting from the output node **D**, we utilize the generation rules to obtain the sketches of the subgraph. The generated sketch `1` depicts that the softmax operators and matrix multiplication are implemented separately, and are not integrated into a single CUDA kernel. To fully exploit the potential of execution efficiency, batch matrix multiplication and softmax operators with new derivation

rules are designed in the transformer-based model to optimize numerous operators as a unified computation kernel. In the end, we integrate all of the techniques with existing optimization rules seamlessly to improve execution efficiency.

Sketch Customization. The default sketch generation rules for the GPU backend with a multi-level tiling structure in Ansor are denoted by the string "SSSRRSRS". Here the letters 'S' and 'R' indicate the spatial and reduction dimensions, respectively. The first three "S" correspond to BlockIdx,Virtual Thread, and ThreadIdx in GPU programming. The consecutive 'S' in the tiling structure "SSSRRSRS" describes the matrix multiplication process, which transforms the original 3-level for-loop into a 19-level for-loop, as illustrated in the Figure 3.4. Additionally, the special multi-level tiling structure can take loop order into consideration during the tensor transformation. Therefore, we have designed an effective operator fusion strategy, denoted as "SSSR-RSRS". For more details, please refer to Figure 3.5. This customized tile structure is specifically designed for batch-matrix multiplication and softmax operators in transformer-based models. By incorporating a caching node with the optimized loop tiling structure, we are able to fully utilize the computation resources on GPUs and fuse multiple operators together. Finally, the computation structure is sent to the sketch of the softmax operator to obtain the fused subgraph implementation.

In Figure 3.5, the optimized fusion strategy for a subgraph containing batch-matrix multiplication and softmax operators is shown. This strategy was discovered using Algorithm 1 in DPOF. There are four states ❶, ❷, ❸ and ❹ in the process. The initial state has three operators in the subgraph $V = \{M_1, M_2, S\}$. For each state, we can get the ending of $V^{'}$ of V. Thus, the execution efficiency of V can be composed of the latency of $V^{'}$ and $V - V^{'}$. The latency of $V^{'}$ can be measured on the GPU directly and the optimized latency of $V - V^{'}$ can be solved with

dynamic programming. ❶ to ❹ show the four different states during the dynamic programming. ❹ is the final state with the best fusion strategy and it also has the optimal inference latency compared with other states. In this state, M_1 and S are fused into a single computation kernel in the first phase and M_2 is executed after that in the second phase.

Annotation Sampling. The sketches generated by the customization rules are incomplete tensor programs because they only have parallel thread structures without the specific value. In order to generate the complete tensor programs which can be successfully executed on the GPU, an auto-tuning technique is employed to find optimal parameters for these optimized parallel thread structures. To achieve this, a performance tuner is developed. a performance tuner is developed to make up for the incomplete tensor programs with optimal values. To illustrate this, we randomly select a sketch from a list of generated sketches using our customization rules. Parallel intrinsic functions are used to generate complete tensor programs for the outer for-loop optimization, while vectorize and unroll intrinsic functions are employed to optimize the inner for-loops. It is worth noting that all valid hyper-parameters are sampled from a uniform distribution and assigned random values during the tuning process.

3.4.4 Performance Tuner

ML-Based Cost Model. Auto-tuners [**Compiler-NIPS2018-LOTP**] provide a means to search for the optimal scheduling of a tensor program from a vast search space. A crucial component of this process is the use of a learned cost model to evaluate the performance of all sampled tensor programs. The cost model is trained on a wide range of extracted features, including arithmetic and memory access features that represent the number of floating-point and integer operations, vectorization,

unrolling, parallelization, buffer access, allocation, and GPU thread binding-related features. We adopt the same feature extraction scheme as that used in Ansor [**Compiler-OSDI2020-Ansor**]. The loss function of the prediction model f on a set of sampled programs P with throughput y is defined as the weighted squared error. Specifically, the loss function is given as:

$$loss(f, P, y) = w_p(\sum_{s \in S(P)} f(s) - y)^2, \tag{3.3}$$

where $S(P)$ denotes a set of innermost non-loop statements in P. To predict the performance of the sampled tensor programs, we train a gradient-boosting decision tree [**ENL-KDD2016-XGBoost**] as the underlying prediction model f. In the training process, we set y to be approximately equal to w for the actual calculation, which is consistent with Ansor's approach.

Evolutionary Search. To collect training data, a search policy is required for the performance tuner. Evolutionary search is utilized, which repeatedly generates a new set of candidates through `mutation` and `crossover` mechanisms for multiple iteration rounds during the search. The objective of the performance tuner is to select a set of tensor programs with the highest prediction scores for optimization. If a generated tensor program has a higher prediction score, it indicates that it will run faster on the platform. The generated tensor programs are compiled and measured on the actual GPU platform to obtain the execution time as the training labels. In addition, the collected data with the highest prediction scores from the previous training is incorporated into the training dataset to enhance the quality of the cost model.

3.5 Hardware Abstraction and Mapping on Tensor Cores

3.5.1 Domain Specific Accelerators on GPU

The recent advancements of GPU hardware technology have resulted in a significant increase in computing power, particularly with the introduction of the Tensor Cores on NVIDIA GPUs. Unlike the scalar-to-scalar primitives found in CPUs or the general CUDA Cores in GPUs, Tensor Cores provide specialized tensor computation capacities, which can deliver over $10\times$ higher throughput. Notably, the initial version of Tensor Core is designed for handling the GEMM with half-precision input and full-precision output. Recently, new features supporting different datatypes such as `int8`, `int4` and `int1` input variables have been introduced in the latest architecture (Truing and Ampere). Listing 1 demonstrates several essential instructions utilized in NVIDIA GPU Tensor Cores. It is worth noting that Tensor Cores are capable of executing Fused-Multiply-Add (FMA) operations in each instruction cycle. These FMA operations process input values in half-precision, while the output values can be in either half-precision (`FP16`) or full-precision (`FP32`).

Tensor Cores enable the computation primitive of $\boldsymbol{D} = \alpha(\boldsymbol{A} \times \boldsymbol{B}) + \beta\boldsymbol{C}$, where tiling of matrix $\boldsymbol{A}$ and $\boldsymbol{B}$ are required to be a certain type of precision, while the type of matrix $\boldsymbol{C}$ and $\boldsymbol{D}$ are also be determined. The shape of tiling $\boldsymbol{A}(M \times K)$ and $\boldsymbol{B}(N \times K)$ may have multiple configurations which depend on the input data precision and GPU architecture. Unlike CUDA Cores, which require users to define the execution flow of each thread, Tensor Cores only require the collaboration of a warp of threads. Using Tensor Cores for computation involves the following steps: i) before calling Tensor Cores, all registers of a warp of threads collaboratively store the tiling into a memory component called `fragment`, which allows for data sharing

Algorithm 2 Standard Attention Implementation in the MHA

Require: Input Matrices $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} \in \mathbb{R}^{N \times d}$ in the global memory.
Ensure: Output Matrics $\boldsymbol{O} \in \mathbb{R}^{N \times d}$
1: Load $\boldsymbol{Q}$, K with thread blocks from the global memory;
2: Compute $\boldsymbol{S} = \boldsymbol{Q}\boldsymbol{K}^{\top}$ with CUDA cores;
3: Store $\boldsymbol{S}$ back to the global memory;
4: Read $\boldsymbol{S}$ from the global memory;
5: Compute $\boldsymbol{P} = softmax(\boldsymbol{S})$ with the CUDA cores;
6: Write $\boldsymbol{P}$ back to the global memory;
7: Load $\boldsymbol{P}$ and $\boldsymbol{V}$ with thread blocks form the global memory;
8: Compute $\boldsymbol{O} = \boldsymbol{P}\boldsymbol{V}$ with the CUDA cores;
9: Store $\boldsymbol{O}$ back to the global memory;

across all registers. The intra-warp sharing mechanism provides opportunities for fragment-based memory optimizations; ii) the loaded matrix fragment components serve as the input variables of the Tensor Cores to generate the output fragment, which also consists of the registers from each thread in a warp and data movements among these registers are also managed collaboratively by a warp of threads.

In this section, we present a compilation-based approach to optimize tensorization programs and introduce the abstractions utilized in our design. Our primary aim is to convert high-level tensor expressions into low-level hardware intrinsics with optimal inference performance. We define the register-level abstraction using hierarchical mapping, which is divided into two categories: computation and data movement (memory) abstractions. The purpose of these abstractions is to formalize the behavior of domain-specific accelerators, which enables our system to automatically analyze and optimize different compute-intensive workloads. In this work, we focus solely on the NVIDIA Turing 2080Ti GPU with the `FP16` datatype via Tensor Cores. Turing architecture supports two common matrix multiplication shapes from the instruction-level parallelism, with $8 \times 8 \times 4$ and $16 \times 8 \times 8$. An overview of our framework can be found in Figure 3.6. We illustrate the whole process by using the matrix multiplication operator in Figure 3.7 and demonstrate

how to define the abstractions, generate tensorization candidates, and explore the optimal mapping step-by-step on the Tensor Cores. Tensor Cores are supported by different architectures such as Volta, Turing, and Ampere, and they can handle various datatypes such as `TF32`, `FP16`, `INT8`, `INT4` and `INT1`.

3.5.2 Standard Attention Implementation on CUDA Cores

Memory Access Overheads. Memory-aware design involves carefully managing the movement of data between the different levels of hierarchical memory on a GPU. Since most operations in transformer-based models are memory-bound [**ivanov2021data**], it is crucial to implement memory-aware optimizations to achieve an efficient performance. However, popular frameworks like TensorFlow, PyTorch, and TVM lack the necessary fine-grained control for memory-bound optimization. A computation paradigm that can compute exact operators in transformer-based models with fewer data movements would be highly beneficial. Such a paradigm would reduce the number of required memory accesses and improve overall performance. Operation fusion, which involves using the output values from one operation such as a matrix multiplication layer as the input directly to the following operation such as a softmax layer, without writing intermediate values to off-chip memory, is one way to achieve this optimization.

Opportunities. Given the inherent limitations of the standard attention mechanism, it is imperative to minimize the number of memory accesses to improve performance on GPUs. However, a naive approach may lead to increased data movement overheads between on-chip and off-chip memory, even with the register files. Fortunately, the MHA computation process can be deconstructed into distinct steps, allowing the softmax reduction to be computed incrementally without accessing the entire input. As such, the attention computation can be restructured by partitioning

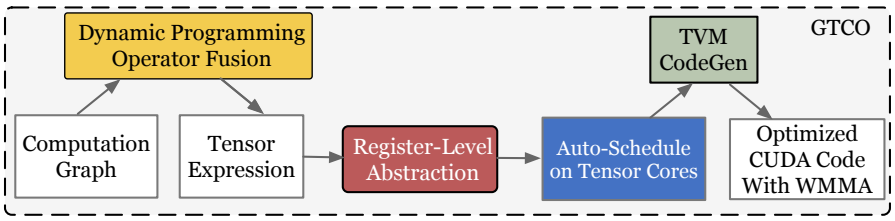


Figure 3.6: *Register-Level Abstraction is defined to enable optimal configuration of tensor computation with WMMA instructions on Tensor Cores. The Input of the GTCO is the computation graph extracted from a deep learning framework. The Dynamic Programming Operator Fusion (DPOF) technique, as introduced in [***bai2021iccad***] with Section 3.4.1, is utilized for graph-level optimization. With Register-Level Abstraction, the original tensor expressions can be encoded with hardware intrinsic. Subsequently, a tensorization-aware auto-schedule, which includes code generation is developed to generate high-performance tensor programs on Tensor Cores.*

the input into blocks and performing several passes over these blocks. Our approach leverages a compiler to enable precise control over memory access and combines the matrix multiplication and softmax operations into a single kernel using specialized hardware intrinsic to accelerate inference.

3.5.3 Register-Level Abstraction

Regarding computation abstraction, we utilize the function to represent arithmetic operations such as addition and multiplication operators. Specifically, we represent Tensor Core computations as `mma_sync` intrinsic, which is capable of computing a matrix multiplication for a special shape. The abstraction can be defined as follows:

$$\boldsymbol{dst}\left[m,n\right] = \texttt{multiply}(\boldsymbol{src_1}\left[m,k\right], \boldsymbol{src_2}\left[k,n\right]). \tag{3.4}$$

Register-Level abstraction is a list of statements that specify the scope, operands, and memory access indices. The notation used for this abstraction includes $\boldsymbol{dst}$ to represent the output tensor, and $\boldsymbol{src_{1,2}}$ for the input tensors. The terms "global", "shared", and "register" are used to indicate the different memory hierarchies, while $\vec{i}$, $\vec{j}$, $\vec{k}$ denote the indices for tensor storage in each hierarchy. Specifically, $\vec{i}$ refers

Algorithm 3 Optimized Implementation with Tensor Cores

Require: Input Matrices $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} \in \mathbb{R}^{N \times d}$ in the global memory, shared memory of size mem_1 and register files of size mem_2.
Ensure: Output Matrics $\boldsymbol{O} \in \mathbb{R}^{N \times d}$.
1: Divide $\boldsymbol{Q}$ into a numbers of blocks $\boldsymbol{Q}_1, \boldsymbol{Q}_2, \ldots, \boldsymbol{Q}_i, \ldots, \boldsymbol{Q}_{\frac{mem_1}{4d}}$;
2: Divide $\boldsymbol{K}$ into a numbers of blocks $\boldsymbol{K}_1, \boldsymbol{K}_2, \ldots, \boldsymbol{K}_i, \ldots, \boldsymbol{K}_{\frac{mem_1}{4d}}$;
3: Load $\boldsymbol{Q}_i$ and $\boldsymbol{K}_i$ with blocks from global memory to shared memory and store in the fragment;
4: Divide $\boldsymbol{Q}_i$ into blocks $\boldsymbol{Q}_{i1}, \boldsymbol{Q}_{i2}, \ldots, \boldsymbol{Q}_{ij}, \ldots, \boldsymbol{Q}_{i\frac{mem_2}{4d}}$;
5: Divide $\boldsymbol{K}_i$ into blocks $\boldsymbol{K}_{i1}, \boldsymbol{K}_{i2}, \ldots, \boldsymbol{K}_{ij}, \ldots, \boldsymbol{K}_{i\frac{mem_2}{4d}}$;
6: Load $\boldsymbol{Q}_{ij}$ and $\boldsymbol{K}_{ij}$ from the fragment to the register files;
7: Compute $\boldsymbol{S}_{ij} = \boldsymbol{Q}_{ij}\, \boldsymbol{K}_{ij}^{\top}$ with the Tensor cores;
8: Compute the max value of row in $\boldsymbol{S}_{ij}$ via $\boldsymbol{m}_{ij}$;
9: Compute the $\boldsymbol{p}_{ij} = \exp(\boldsymbol{S}_{ij} - \boldsymbol{m}_{ij})$ for softmax;
10: Compute the sum value of row via $\boldsymbol{l}_{ij} = \sum \boldsymbol{p}_{ij}$;
11: Compute $\boldsymbol{P} = softmax(S)$ with $\boldsymbol{m}_{ij}$, $\boldsymbol{p}_{ij}$ and $\boldsymbol{l}_{ij}$;
12: Write $\boldsymbol{P}$ back to the shared memory and store in the fragment;
13: Load $\boldsymbol{V}$ to the shared memory and store in the fragment;
14: Compute $\boldsymbol{O} = \boldsymbol{P}\boldsymbol{V}$ with the tensor cores;
15: Store $\boldsymbol{O}$ back to the global memory;

to the indices in the global memory, $\vec{j}$ in the shared memory, and $\vec{k}$ in the register files. It is important to note that the same operand may have different indices within different memory scopes. For example, the intrinsic `load_matrix_sync` is used to load data from shared memory to register files, while the intrinsic for storing data from registers to global memory can be expressed in the same way. The whole process of the data movement can be formulated as follows:

$$shared.Src\left[\vec{j}\right] = global.Src\left[\vec{i}\right], \tag{3.5}$$

$$reg.Src\left[\vec{k}\right] = shared.Src\left[\vec{j}\right], \tag{3.6}$$

$$global.Src\left[\vec{i}\right] = reg.Src\left[\vec{k}\right]. \tag{3.7}$$

Both the computation and memory abstractions are designed to capture the behavior of hardware intrinsics with the behavior of Tensor Cores.

3.5.4 Auto-Scheduling on Tensor Cores

To generate the abstraction with hardware intrinsics, we propose a hierarchical mapping approach. First, we map the software iterations to a virtual accelerator component composed of the load/store unit and computation units, without considering memory allocation or computation resource assignment. Next, we consider constraints such as memory capacity and hardware intrinsic with six parameters, as shown in Figure 3.7. These parameters can represent three-level parallel optimization on the GPU with Tensor Cores. The first three parameters B_m, B_n, B_k can map the computation on the thread-block-level concurrency. The last three parameters, W_m, W_n, W_k represent warp-level optimization and parallelism in the implementation of the main loop. In the Tensorized Body component, the shapes of the WMMA instruction and datatype differ between GPU architectures. For this work, we focus solely on the Turing architecture with `FP16` datatype, with the option of using 8 $\times$ 8 $\times$ 4 and 16 $\times$ 8 $\times$ 8 shapes. In order to obtain a valid physical mapping from software iterations to the domain-specific accelerators with a more complex memory hierarchy, we propose modifications to the previous auto-schedule introduced in **[bai2021iccad]**. We present a practical example to illustrate our hierarchical mapping approach for matrix multiplication on Tensor Cores. Specifically, we consider the computation $\boldsymbol{S}[m,n] += \boldsymbol{Q}[m,k] \times \boldsymbol{K}[k,n]$, which can be easily extended to the batch matrix multiplication used in vision transformer models on Tensor Cores.

The initial step assumes that GTCO is capable of loading data of arbitrary volume into on-chip memory (register files and shared memory), and has sufficient hardware resources to directly perform all the tensor computations. The primary challenge at this stage is to map software-defined tensor operations to their corresponding hardware intrinsic. The second step involves considering two types of constraints:

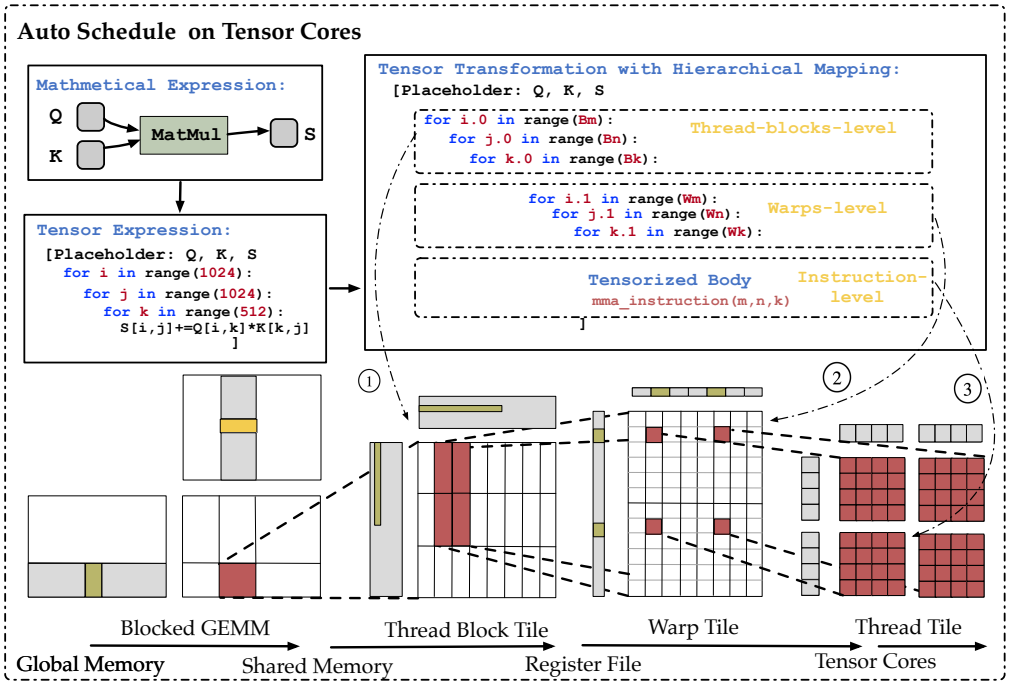


Figure 3.7: *During auto-tuning, the matrix multiplication is mapped to Tensor Cores with hardware intrinsic via a hierarchical mapping described in Algorithm 3. It involves three levels of optimization, namely thread-blocks, warps, and instruction-level. It employs six parameters ($B_m, B_n, B_k, W_m, W_n, W_k$) and two sets of WMMA instruction for tensor transformation. Additionally, the double buffering technique is employed during kernel execution. It is worth noting that the primary difference between GTCO and [**bai2021iccad**] is that the former utilizes more comprehensive optimization techniques that span across all three levels of parallel programming models, while the latter only uses thread-blocks-level optimization with shared memory on CUDA cores.*

memory capacity and intrinsic size. Hardware accelerators have a fixed capacity for computing results at any given time, which is limited by the problem size of an intrinsic extracted from its indices range represented in the computation abstraction. In our running example, the matched software iterations are limited by a factor of 4 due to the problem size of the example Tensor Core shapes being $8 \times 8 \times 4$ and $16 \times 8 \times 8$. (1), (2), and (3) show the proposed mapping across different memory hierarchies from high-level mathematical expression to low-level hardware intrinsic. Step (1) represents that the original matrix is divided into tiling data, which are loaded from the global memory to the shared memory and then stored in

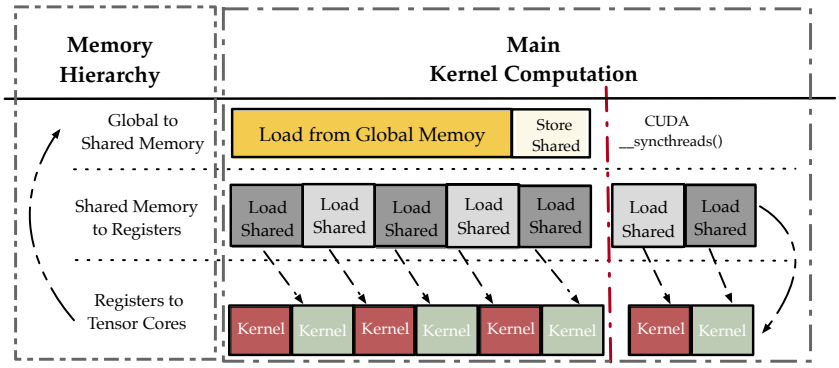


Figure 3.8: *The fusion of softmax and matrix multiplication kernel computation with data movement across a complex memory hierarchy is implemented with the double buffering technique to improve overall execution efficiency.*

the fragment with thread block. It corresponds to the operation defined in Equation (3.5). Step (2) describes the process of moving tiling data from the fragment to the register files for the computation, which corresponds to Equation (3.6). Step (3) indicates that performing matrix multiplication within one clock cycle, using the tiling data stored in the register and scaling it according to the dimensions (m $\times$ n $\times$ k) specified by the WMMA instruction. It corresponds to the data movement operations defined in Equation (3.7). To alleviate the impact of memory latency, software pipelining is utilized to overlap memory accesses with other computations within a thread. The pipelining stage is set to 2 throughout the process. Further details on the pipelining of various kernel fusions, such as softmax and matrix multiplication operators in transformer-based models, can be found in Figure 3.8.

Thus, in order to identify the optimal implementation, valid software-hardware mappings need to be selected from the design search space. However, when optimizing these schedules with tiling or parallelization primitives on domain-specific accelerators, it is challenging to determine the performance of these mappings. The reason is that these optimizations, combined with different software-hardware mappings, exhibit varying performance due to differences in computation and memory

utilization. Moreover, the search space for performance tuning is too vast. Therefore, to address this challenge, we employ a combination of tuning techniques and an ML-based cost model to explore the mapping and schedule space. For a more detailed description of the performance tuner design, please refer to Section 3.4.4.

It enables our computation and data movement in one kernel, loading input from global memory, performing all the computation operations such as softmax and matrix multiplication on the register-level files, then writing the result back to global memory. It can avoid repeatedly reading and writing inputs and outputs with the memory access overhead. More details about the proposed optimization technique can be found in Section 3.5.3.

3.6 Evaluation

3.6.1 Experimental Setup

Our framework is implemented on top of PyTorch and the HuggingFace Transformers library. We evaluate the effectiveness of our approach, such as operator fusion optimization, kernel generation mechanisms, and Tensor Cores acceleration on three modern vision transformers: ViT [**ATTN-arXiv2020-ViT**] for image classification, DETR [**ATTN-ECCV2020-DETR**] for object detection and SETR [**ATTN-CVPR2021-SETR**] for semantic segmentation. The DETR and ViT pre-trained models are downloaded from the Hugging Face datasets hub. The SETR models are downloaded from the original GitHub repo [**SETR-repo**]. PyTorch 1.7.1, CUDA 10.0, cuDNN V7.6.5, NVIDIA driver 460.67, TVM 0.8.dev0 [**Compiler-OSDI2018-TVM**] and TensorRT V7.0.0.11 [**GPU-NVIDIA-TensorRT**] are set as baselines for fair comparisons. Note that all evaluation results are

Table 3.1: *Detailed information of the benchmark and experiment models*

model	ec	dc	width	mlp-dim	nh	input shape	patch	mha	encoder	decoder	Params
DETR-ResNet50-E3	3	6	256	2048	8	[1,3,800,1333]	N/A	q [1050,1,256] k [1050,1,256] v [1050,1,256]	[1050,1,256]	t [100,1,256]	37.40M
DETR-ResNet50-E6	6	6	256	2048	8	[1,3,800,1333]	N/A	q [1050,1,256] k [1050,1,256] v [1050,1,256]	[1050,1,256]	t [100,1,256]	41.30M
DETR-ResNet50-E12	12	6	256	2048	8	[1,3,800,1333]	N/A	q [1050,1,256] k [1050,1,256] v [1050,1,256]	[1050,1,256]	t [100,1,256]	49.20M
SETR-Naive-Base	12	1	768	4096	12	[1,3,384,384]	16	q [576,1,768] k [576,1,768] v [576,1,768]	[576,1,768]	t [576,1,768]	87.69M
SETR-Naive	24	1	1024	4096	16	[1,3,384,384]	16	q [576,1,1024] k [576,1,1024] v [576,1,1024]	[576,1,1024]	t [576,1,1024]	305.67M
SETR-PUP	24	1	1024	4096	16	[1,3,384,384]	16	q [576,1,1024] k [576,1,1024] v [576,1,1024]	[576,1,1024]	t [576,1,1024]	310.57M
ViT-Base-16	12	0	768	3072	12	[1,3,224,224]	16	q [197,1,768] k [197,1,768] v [197,1,768]	[197,1,768]	N/A	86.00M
ViT-Large-16	24	0	1024	4096	16	[1,3,224,224]	16	q [197,1,1024] k [197,1,1024] v [197,1,1024]	[197,1,1024]	N/A	307.00M
ViT-Huge-14	32	0	1280	5120	16	[1,3,224,224]	14	q [257,1,1280] k [257,1,1280] v [257,1,1280]	[257,1,1280]	N/A	632.00M

Table 3.2: *The information of subgraphs and scheduling weights with graph partition*

	n_1	Weight-Encoder	n_2	Weight-Decoder	n_3	Weight-Transformer
Ansor **[Compiler-OSDI2020-Ansor]**	9	$\{[6*7],[12*2]\}$	13	$\{[6*10],[18*3]\}$	22	$\{[6*17],[13*2],[19*2],[18*1]\}$
[bai2021iccad]	6	$\{[8*4],[10*2]\}$	11	$\{[8*6],[20*5]\}$	17	$\{[9*12],[20*3],[16*2]\}$
GTCO	6	$\{[8*4],[10*2]\}$	11	$\{[8*6],[20*5]\}$	17	$\{[9*12],[20*3],[16*2]\}$

collected on 2080Ti GPU by different batch sizes.

Workflow. Our workflow can be categorized into the following two patterns: 1) For the inference engine like TensorRT, PyTorch is used to build the models and then the ONNX export interface is used to export ONNX models. To avoid some redundant operators caused by conversion, ONNX-Simplifier **[onnx-sim]** is used to simplify the ONNX model. Finally, the model is converted into an executable file with the CUDA runtime environment; 2) As for the compiler flow like Ansor, **[bai2021iccad]** and GTCO, the model is converted into the TorchScript format first

Table 3.3: *End-to-end network execution performance benchmark (ms)*

	PyTorch JIT [**DL-NIPSW2017-PyTorch**]	TVM-CUDA [**Compiler-OSDI2018-TVM**]	TVM-cuDNN [**Compiler-OSDI2018-TVM**]	TensorRT [**GPU-NVIDIA-TensorRT**]	Ansor [**Compiler-OSDI2020-Ansor**]	[**bai2021iccad**]	GTCO
DETR-ResNet50-E3	18.62	54.73	54.43	6.97	5.85	5.32	**4.18**
DETR-ResNet50-E6	23.67	93.59	88.25	7.73	6.78	5.60	**4.46**
DETR-ResNet50-E12	33.01	171.96	157.97	15.79	14.29	13.18	**11.08**
SETR-Naive	68.26	753.25	742.21	33.71	34.22	28.65	**22.34**
SETR-Naive-Base	31.06	186.13	187.39	16.97	15.44	14.21	**11.45**
SETR-PUP	37.62	199.42	189.21	18.61	17.89	16.01	**12.88**
ViT-Base-16	24.92	91.86	96.31	5.87	8.57	8.43	**5.08**
ViT-Large-16	52.96	329.74	334.38	18.45	18.99	18.41	**14.85**
ViT-Huge-14	76.07	846.87	846.27	34.14	32.53	29.89	**24.08**

and then is imported into the Relay interface to read the TorchScript model into our compilation system. In terms of the subgraphs, the corresponding tensor programs are generated by the code generation part. As for the Tensor Cores settings, we only use the basic instructions provided by CUDA such as `fragment`, `load_matrix_sync`, `mma_sync`, `store_matrix_sync` without extensions of customized instructions.

3.6.2 End-to-End Performance

Workloads. The configurations of the models are described in Table 3.1, including the number of encoders, decoders, attention heads, the shape of inputs, and the outputs. All of the results are reported with batch size `1` on an NVIDIA 2080Ti.

Baselines and Configurations. PyTorch JIT [**DL-NIPSW2017-PyTorch**], TensorRT [**GPU-NVIDIA-TensorRT**], TVM [**Compiler-OSDI2018-TVM**], and Ansor [**Compiler-OSDI2020-Ansor**] are used as our baseline frameworks. More specially, there are two common ways to improve the efficiency of execution time on GPUs. Optimizing operators via the vendor-provided libraries such as PyTorch and TensorRT is the first way. On the other hand, another strategy is to use a regression-based model to search the schedule for each kernel such as TVM. In the meantime, TVM also supports calling external libraries such as cuBLAS/cuDNN to optimize some kernels in the computational graph. PyTorch JIT is a just-in-time compiler in PyTorch, which is a way to create serializable and optimizable models from PyTorch code to a production

Table 3.4: *ViT-Base-16 with different optimization settings*

Setting	GTCO					
	(a)	(b)	(c)	(d)	(e)	(f)
TC		✓	✓	✓	✓	✓
DPOF			✓	✓	✓	✓
Sketch Customization				✓	✓	✓
Subgraph Scheduler					✓	✓
Auto-Schedule Register						✓
Speedup	1.00×	1.22×	1.44×	1.56×	1.59×	1.68×

environment where Python programs may be disadvantageous for performance and multi-threading reasons. Auto-tuning trails are set to `10,000` measurement unless execution time converges to a stable value. The goal of the subgraph scheduler is to minimize the total execution time. Finally, optimized tensor programs for subgraphs are generated for measurement. In the Turing architecture GPUs, they have two recommended types of GEMM shapes with $8 \times 8 \times 4$ and $16 \times 8 \times 8$. We use $8 \times 8 \times 4$ in all of the experiments and we find that $16 \times 8 \times 8$ is not better than the former after a certain number of experiments.

Results. Table 3.3 shows the results on a NVIDIA RTX 2080Ti GPU. In general, [**bai2021iccad**] surpasses all of the baseline frameworks except the ViT-Base vision model. Compared with vendor-specific engine TensorRT, [**bai2021iccad**] consistently outperforms all benchmarks with 1.01 to 1.38× speedup except for the ViT-Base vision model. The reason for the drop in execution time is that ViT-Base is composed of a number of encoders, and the input shape $(197, 1, 768)$ of the encoder in ViT-Base is relatively limited compared with ViT-Large and ViT-Huge. Thus, it limits the search space of specific operators in transformer-based models such as batch matrix multiplication and softmax fusion in MHA. Compared with the tuning-based method [**Compiler-OSDI2020-Ansor**], [**bai2021iccad**] outperforms all

benchmarks with 1.01 to 1.21× speedup. That is, our framework equipped with operator fusion technique and sketch customization rules has achieved good performance on transformer-based vision models. TVM-cuDNN/BLAS means we use the TVM as the compiler and then call the operators defined in the cuDNN and cuBLAS library to optimize the execution time of each kernel. Compared with TVM-cuDNN/BLAS [**Compiler-OSDI2018-TVM**], [**bai2021iccad**] consistently outperforms all benchmarks with significant speedup.

Obviously, TVM-cuDNN/BLAS uses the operator fusion patterns defined in Relay to partition the graph into lots of subgraphs. Each operator in the subgraph is replaced with the implementations in cuDNN/cuBLAS. Neither the search-based optimization for the tensor program nor fine-tuning the performance of each kernel by a regression-based cost model is implemented during the optimization. Therefore, [**bai2021iccad**] has more advantages on the emerging new operators or uncommon operator fusion patterns because it is not easy for vendor-specific static libraries such as cuDNN/cuBLAS to optimize for all the cases manually. The only difference between TVM-CUDA and TVM-cuDNN/BLAS is that the implementation of each operator in the subgraph is done by the default scheduling template defined in the deep learning compiler. Note that the GTCO means all of the transformer-based vision models are conducted on the Tensor Core with floating-point `16` data type. From the last column in Table 3.3, we can find that our abstraction design and mapping strategies on Tensor Cores for GTCO perform the best in the end-to-end benchmark. In the DETR-ResNet series of vision models, GTCO can achieve up to 1.27 × speedup compared with [**bai2021iccad**] on 2080Ti GPU. In the SETR series of vision models, GTCO can achieve up to 1.28 × speedup compared with [**bai2021iccad**].

All in all, our design can fully take advantage of the Tensor Cores on the modern

GPU to accelerate the vision transformer execution with specific datatype `FP16`.

Ablation study. To explore the function of each module, we run variants of GTCO on the ViT-Base-16 benchmark. "DPOF ✓" means dynamic programming operator fusion technique is used to optimize the computation graph from graph-level rather than the template-based method designed in Relay. "Sketch Customization ✓ " means sketch generation rules and search policy defined in GTCO is used to generate the tensor program rather than default configurations defined in Ansor. "Subgraph scheduler ✓" means we use the object function defined in Equation (3.2) to optimize our auto-tuning. Obviously, *Design (a)* is the native Ansor with Tensor Core support. We set the execution time of *Design (b)-(d)* to be the speedup against Ansor equipped with Tensor Core. As shown in Table 3.4, *Design (e)* performs the best in speedup performance among all of the designs. "TC" means that we only use the Tensor Cores to accelerate the operators in the models without register-level abstraction and hierarchical mapping. "Auto-Schedule Register" represents the proposed register-level abstraction and auto-schedule on Tensor Cores. It can verify that the graph and tensor co-design is very significant for the transformer-based model execution. At the graph-level, GTCO utilizes the "DPOF" to optimize the operator fusion and then uses a subgraph scheduler designed for the transformer-based model to assign different search tasks with various time slots. At the tensor-level, the tensor programs are generated by our sketch generation rules with search policy. The design of register-level abstraction of computation and memory can boost the final performance on Tensor Cores with hierarchical mapping.

3.6.3 Subgraph Benchmark

Baselines and Configurations. Three common subgraphs in DETR-ResNet-50-E6 are conducted to verify the subgraph benchmark including MHA, Encoder, and Decoder.

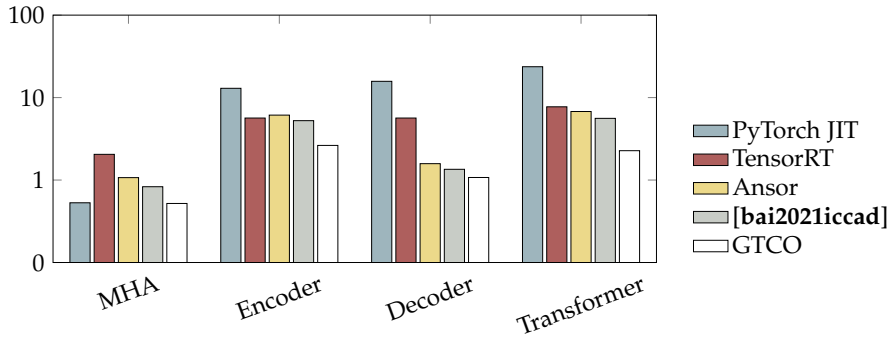


Figure 3.9: *Sugraph performance benchmark. The y-axis is the throughput-based log 10 and then plus 1.*

The measurement trails per test case are set to `20,000` during the auto-tuning for Ansor and we use the consumed time in the whole process to demonstrate the final performance. We use the same set of baseline frameworks and run benchmarks with the approximate converged latency.

Results. Figure 3.9 shows that [**bai2021iccad**] outperforms PyTorch JIT on the Encoder and Decoder by 2.47× and 11.67× speedup. For the high-performance computing library TensorRT, [**bai2021iccad**] can achieve 2.47×, 1.08×, and 4.19× speedup on MHA, Encoder, and Decoder. For the compiler-based search algorithm Ansor, GTCO can achieve 1.29×, 1.17×, and 1.17× speedup on MHA, Encoder, and Decoder. It can prove that [**bai2021iccad**] can generate efficient tensor programs for these subgraphs on the NVIDIA GPU platform. Meanwhile, GTCO performs best in all subgraph benchmarks compared with Pytorch-JIT, TensorRT, Ansor, and [**bai2021iccad**] under the same inference configuration. GTCO can achieve 1.83×, 1.44×, and 1.29× speedup on MHA, Encoder, and Decoder compared to [**bai2021iccad**].

Table 3.5: *The number of measurement trails*

	Ansor [**Compiler-OSDI2020-Ansor**]	[**bai2021iccad**]	GTCO
Multi-Head Attention	1,600	1,408	1,264
Encoder-3-Layer	3,008	2,816	2,416
Encoder-6-Layer	4,992	4,096	3,464
Encoder-12-Layer	6,528	5,760	5,022
Decoder-6-Layer	2,688	2,432	1,986
DETR-ResNet-50-E6	8,640	6,784	6,112
Average	100%	87%	**75%**

3.6.4 Graph Partition and Tuning Time

The graph partition on the DETR-ResNet50-E6 benchmark is shown in Table 3.2. "n_i" means the number of subgraphs in the encoder, decoder, and transformer models, respectively. The meaning of "Weight-*" can be explained with two important values. For example, $\{[6 * 7], [12 * 2]\}$ means there are 7 subgraphs with weight value 6 and 2 subgraphs with weight value 12. Therefore, the total number of subgraphs in the encoder is 9. Compared to the rule-based method in Ansor, we can find that our graph partition and subgraph scheduler methods can achieve a more effective operation fusion strategy for the number of subgraphs and weight values. In addition, GTCO can not change the partition methods defined in our optimization on encoders and decoders compared with [**bai2021iccad**] from Table 3.2, and it only accelerates the runtime on the Tensor Cores.

Table 3.5 shows the search time needed for [**bai2021iccad**] and GTCO to match the execution time of Ansor on the same benchmark."number of measurement trails" are used to evaluate the search time. From the table, GTCO can match the performance of Ansor with fewer measurement trails. It can prove that the efforts saving in search time come from the techniques we introduced before including a subgraph scheduler, the dynamic programming operator fusion at the graph level, the sketch generation rules for tensor programs generation, and register-level abstraction with hierarchical mapping on Tensor Cores. From the average

Table 3.6: *The time used in the total compilation phase*

Network	batch	**[bai2021iccad]**	GTCO	Ratio
DETR-ResNet50-E3	1	1,625	1,430	88%
	4	3,652	3,324	91%
	8	6,322	5,436	86%
DETR-ResNet50-E6	1	1,825	1,624	89%
	4	4,314	3,796	88%
	8	6,792	6,112	90%
DETR-ResNet50-E12	1	1,997	1,698	85%
	4	4,798	4,220	87%
	8	7,001	5,880	84%
SETR-Naive	1	2,158	1,770	82%
	4	4,334	3,640	84%
	8	6,698	5,626	84%
SETR-PUP	1	4,160	3,452	83%
	4	5,026	4,272	84%
	8	8,298	6,804	82%
ViT-Base-16	1	1,682	1,312	78%
	4	4,146	3,358	81%
	8	7,019	5,826	83%
ViT-Large-16	1	2,767	2,214	80%
	4	5,310	4,354	82%
	8	7,631	6,410	84%
ViT-Huge-14	1	3,158	2,748	87%
	4	6,620	5,564	84%
	8	9,012	7,388	82%

performance of six benchmarks, we can find that GTCO outperforms the Ansor and **[bai2021iccad]**. CTCO can generate high-performance tensor programs on Tensor Cores with less effort.

Table 3.6 shows the total compilation time needed for GTCO to match the nearly uniform latency performance of **[bai2021iccad]** on the end-to-end performance. All the data used in this table are recorded in seconds. We use "Ratio" to demonstrate the efficiency of our compilation technique. Compared with the time `1,625` seconds, the total compilation time used in GTCO with DETR-ResNet-E3 is `1,430` seconds. If we use the GTCO with `FP16` datatype on Tensor Cores, we can get the final latency performance compared with the dense CUDA Core version of GTCO in 88% of the total time. From all of the experimental results in the Table 3.6, we can find that the techniques used in the hardware abstraction and automatic mapping

on Tensor Cores can accelerate the optimization time during the total compilation phase among all of the vision transformer models.

3.7 Summary

Existing deep learning compilers optimize operator fusion based on the rule designed by experts, which is strictly improving execution performance for the new operators on hardware platforms. However, they fail to consider the potential performance improvements that more effective operator fusion strategies could provide. This work addresses this issue by tackling the problem from two perspectives. Firstly, a dynamic programming algorithm is introduced to explore operator fusion patterns. Secondly, a search policy is proposed that includes new sketch generation rules and a novel hardware abstraction with register-level optimization, enabling more flexible mapping for tensor computation and better performance. This approach is applied to optimize fused matrix multiplication and softmax operators with WMMA instructions. To achieve an end-to-end flow, a regression-based learned model is used to fine-tune the performance of each kernel. Overall, GTCO achieved up to 1.73× inference speedups compared to the high-performance inference engine TensorRT with Tensor Cores, and 1.38× speedups with CUDA Cores.

Chapter 4

ATFormer: A Learned Performance Model with Transfer Learning Across Devices for Deep Learning Tensor Programs

4.1 Motivation

Recently, there has been a significant improvement in model performance for deep neural networks (DNNs) [**CNN-CVPR2016-He**, **MobileNetV2-CVPR2018-sandler**, **Semantic-Seg-Shan**, **Bert-NACCL2019-Devlin**, **FBNet-CVPR-WU**, **VQA-CVPR2019-STVQA**, **bello2019-attentionaugmentedcnn**]. However, this progress has been accompanied by a significant increase in the number of operators and, consequently, the computational complexity of DNNs. As a result, it has become increasingly challenging to efficiently deploy DNNs with optimized tensor programs on certain hardware accelerators like CPUs, GPUs and TPUs [**TPU-V1-Google**].

To overcome the limitations, mainstream search-based tensor compilers [**Chen-TVM-OSDI**, **Ansor-OSDI2020-Zheng**, **AutoGTCO-ICCAD2021-Bai2021**, **li2020deep**, **MLSYS2021_182be0c5**] are developed. These compilers automatically search for the optimal deployment configuration of each operator on increasingly heterogeneous platforms. Conducting on-device measurements is extremely time-consuming, making it impossible to place all the generated tensor programs on the target platform for measurement during the compilation process. Therefore, the prediction via an optimal cost model is crucial in reducing the time-consuming measurements during the compilation which can significantly improve search efficiency and quality.

Nevertheless, the existing cost models are capable of selecting nearly optimal configurations but suffer from excessively long optimization time. These long optimization times not only impede the deployment period but also raise concerns about the practicality of search-based compilers. Furthermore, statistic cost models trained on one hardware platform exhibit significant performance degradation on different hardware, making them unusable across different platforms. It is noteworthy that the execution times of tensor programs can vary significantly on different platforms due to domain gaps, making it challenging to deploy optimized models on multiple platforms. This is further compounded by the significant differences in the features extracted from various platforms. Even when extracted on GPUs, the feature's stability and performance cannot be guaranteed across different GPU architectures such as Volta, Turing, and Ampere. Therefore, additional engineering efforts are necessary to account for the differences in hardware architectures, resulting in a laborious and cumbersome feature extraction process.

To address these challenges, we propose a powerful yet simple approach that uses attention-inspired blocks to enhance the performance of cost models. These blocks can capture global and long-range dependencies among tensor program statements.

Additionally, transferable features with pre-trained parameters are used to expedite search convergence across different hardware platforms. These techniques can be easily incorporated into existing search algorithms and improve efficiency in an end-to-end fashion. Our performance model, consistently outperforms popular DNN benchmarks, including small and large-scale models. Furthermore, our techniques enable cross-platform transfer learning, resulting in more efficient deployment.

4.2 Problem Formulation

We describe a DNN model as a computation graph and then define some important terminologies in the compilation flow.

Subgraph. Computation Graph G is partitioned into a set of subgraphs S based on the graph-level optimizer [**UW-Relay-PLDI**]. Each search task is extracted from an independent subgraph S_i on a specific hardware platform $\mathbb{H}$. Thus, we define search task Q as follows:

$$Q_{\mathbb{H}(S|G)} = \left\{ Q^1_{(S_1|G)}, Q^2_{(S_2|G)}, \ldots, Q^n_{(S_n|G)} \right\}, \tag{4.1}$$

where n is the number of subgraphs in G. Note that each subgraph S_i contains a computation-intensive operator σ and $\sigma \in S_i$. Therefore, we use $Q^i_{(S_i|G)}$ to represent the $i-$th search task in G. Each subgraph S_i has its own search space, which is determined by the input and output shapes, data precisions, memory layout, and the hardware platform. The search space is usually large enough to cover almost all kinds of tensor candidates.

Hierarchical Search Space. A tensor program, denoted by p, represents an implementation of the subgraph using low-level primitives that are dependent on the hardware platform. Each tensor program can be considered as a candidate in the

search space. We define the hierarchical search space $\phi_{1,2}$, which decouples high-level structures ϕ_1 from low-level details ϕ_2, allowing for the efficient exploration of potential tensor candidates during the tuning process. Here, we can transform a tuning problem into an optimization problem that explores the potential tensor programs in a hierarchical search space.

Given code generation function $_1$, high-level structure generation parameters ϕ_1, low-level detail sampling parameters ϕ_2, computation-intensive operator σ and operator setting k (*e.g.,* kernel size), our goal is to use $\phi_{1,2}$ to build a hierarchical search space and generate tensor program p to achieve the optimal prediction score y^* on a specific GPU hardware platform $\mathbb{C}$ with CUDA cores.

$$\begin{aligned} \phi_{1,2}^* &= \arg\max_{\phi} y_c, \\ y_c &= f_{\mathbb{C}}(_1(\phi_1, \phi_2 | \sigma, k)). \end{aligned} \tag{4.2}$$

The cost model $f_{\mathbb{C}}$ predicts score y_t of the tensor program p on GPU CUDA cores. The accuracy of the cost model $f_{\mathbb{C}}$ is crucial in finding ideal optimization configuration.

4.3 Performance Model

The process of optimization using our design is outlined in Algorithm 4. The input is a set of optimized operators or subgraphs with different configurations. To implement our workflow, three functions are defined: `GenerateHighSketch()`, `Sampling()`, and `EvolutionSearch()`, as shown in Algorithm 4. The function of `GenerateHighSketch()` takes ϕ_1, σ, and k as input and returns the high-level generation sketch GS_1 as output. `Sampling()` takes GS_1, ϕ_2, σ, and k as input

Algorithm 4 Search-based Framework on CUDA Cores

Require: Search space ϕ_1, ϕ_2 with operator σ and setting k.
Ensure: Tensor program p^* with best configuration c^*.

while nTrials $<$ eachSubgraphTrials **do**
  $GS_1 \leftarrow$ `GenerateHighSketch(`ϕ_1, σ, k`);`
  $GS_2 \leftarrow$ `Sampling(`GS_1, ϕ_2, σ, k`);`
  `P` $\leftarrow$ `EvolutionSearch(`GS_1, GS_2`);`
  for $p \in$ `P` **do**
    $c \leftarrow f_{\mathbb{C}}(_1(\phi_1, \phi_2 | \sigma, k))$;
  nTrials $\leftarrow$ nTrials $+$ batchSize;
$c^* \leftarrow$ best tensor program configurations on CUDA cores;

and returns the low-level annotation samples GS_2 as output. `EvolutionSearch()` takes the high-level generation sketch GS_1 and the low-level annotation samples GS_2 as input and returns a group of tensor candidates for the cost model training. Next, an evolutionary search strategy is used along with a learned cost model to fine-tune the performance of the generated tensor programs. By iteratively mutating high-quality tensor programs, it can generate new programs with potentially higher quality. After a number of measurement trials, the best tensor program configurations can be identified.

Hierarchical Feature Generation. The input of ATFormer is a series of mix-grained feature vectors extracted from p_σ, where p_σ is the full tensor program to implement operator σ. Each vector represents a single computation statement within p_σ. These mix-grained feature vectors are composed of two important components: *(i) Coarse-Grained operator embedding features* that capture the high-level structure of the operator σ and *(ii) Fine-Grained statement features* that capture the low-level details of each statement within program p_σ. Each operator in the subgraph S can be classified into a few categories, and we represent each operator with a one-hot embedding feature vector that covers all possible operator types. In practice, we use feature vectors of length 10 for the operator embedding and length 164 for the statement features, consistent with the approach used in Ansor [**Ansor-OSDI2020-Zheng**].

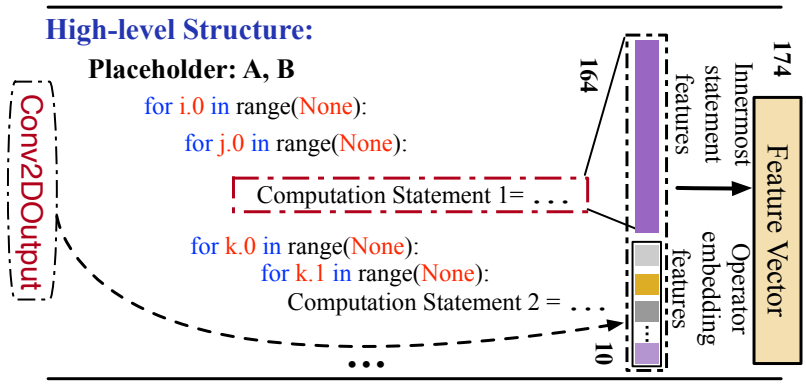


Figure 4.1: *Hierarchical features of convolution with a full tensor program representation.*

The prediction score for a subgraph is computed as the sum of the prediction scores for each innermost non-loop statement within the loop nests of the full tensor program. More details can be found in Figure 4.1.

Model Architecture. Our proposed ATFormer model consists of three layers: (i) a kernel embedding layer, which extracts a compact feature representation; (ii) a computation processing layer, which captures essential information from the innermost non-loop computation statements in the neighborhood; and (iii) a simple regression layer for making the final prediction. ATFormer can be easily integrated into existing search algorithms and consistently improve the efficiency of auto-tuning. We believe that the simplicity of our method will attract more research attention to the field of tensor operator optimization, further enhancing training and inference efficiency. The feature processing of computation and regression in ATFormer is illustrated in Figure 4.2. The kernel embedding layer is composed of two fully connected layers with ReLU activation. The function of the kernel embedding layer is to project the features from low dimension space to a new embedding space for similarity measurement. Starting from the batched tensor programs $\mathcal{I} \in \mathbb{R}^{L \times D_{in}}$ representing a specific type of operator σ, where L is the accumulated number of the feature statements within $\mathcal{I}$. A kernel embedding

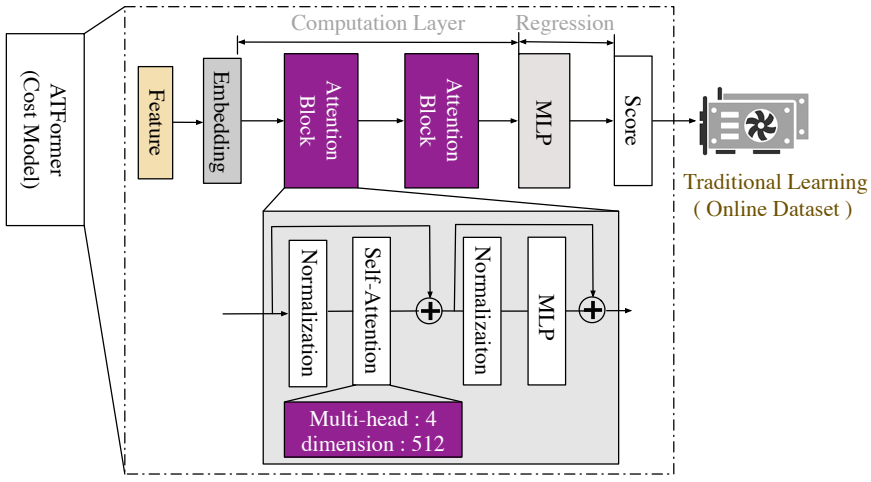


Figure 4.2: *The architecture of performance model includes two attention blocks that extract coarse and fine-grained features of the tensor program, as well as a lightweight MLP layer for directly predicting the score.*

layer then generates a set of feature statements $\mathcal{E} \in \mathbb{R}^{L \times D_{out}}$ in embedding space. Typically, we use $D_{out} = 512$. The value L is determined by the parameters of high-level structures ϕ_1 and the low-level details sampling ϕ_2 for each subgraph S.

As for the computation layer, a set of feature statements $\mathcal{E} \in \mathbb{R}^{L \times D_{out}}$ should be split into M stacks of feature statements $\mathcal{Z} \in \mathbb{R}^{M \times N \times D_{out}}$ firstly. Each stack contains N feature statements of innermost non-loop computation within a full tensor program p. We adopt the self-attention mechanism for feature statements aggregation. With the parameter tensors written as $\boldsymbol{W}^Q, \boldsymbol{W}^K, \boldsymbol{W}^V$, a full tensor program with a set of innermost non-loop feature statements $\boldsymbol{Z}$ is first encoded into query $\boldsymbol{Q}$, key $\boldsymbol{K}$, and value $\boldsymbol{V}$ by three identical linear transformations: $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} = \boldsymbol{Z}^\top \boldsymbol{W}$. Then it will be further calculated by the self-attention as:

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{Softmax}\left(\frac{\boldsymbol{Q}^\top \boldsymbol{K}}{\sqrt{d_k}}\right)\boldsymbol{V}. \tag{4.3}$$

The final prediction of these M tensor programs is computed by a regression layer with a dimension from 512 to 1. The predicted score is $\boldsymbol{y} \in \mathbb{R}^{M \times 1}$.

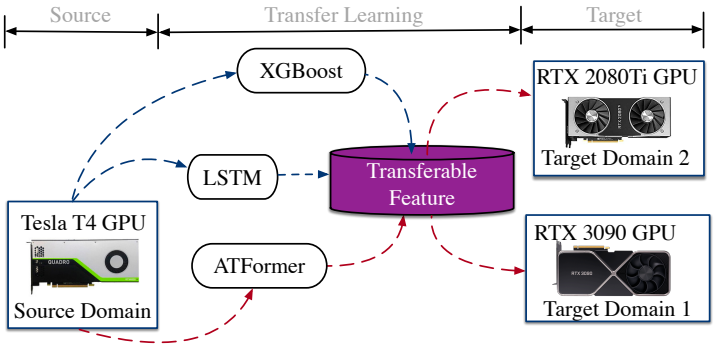


Figure 4.3: *Transfer learning among different platforms with ATFormer.*

Loss Function. The model ranks the performance of potential candidates in a large search space. Therefore, the model can be trained with ranking losses or regression losses to predict relative or absolute scores. To explore the loss function to train ATFormer, a common choice is to use the squared error function as a regressor which can mostly care about identifying the well-performing tensor programs. The loss function of the model f on a full tensor program p with throughput h is $\text{MSELoss}(f, p, h) = (\sum_{s \in S(p)} \hat{f}(s) - y)^2$, where $S(p)$ is the set of innermost non-loop computation statements in tensor program p. We train ATFormer as the performance model f. However, we only care about the relative order of tensor program runtime rather than their absolute values during the compilation. We instead use the following RankLoss [**Learning2rank-ICML2007-Cao**] to rank the performance of candidates in the large design space. This can fully exploit the optimal candidates to reduce the impact of the search algorithm on final prediction results. The loss function is defined as follows:

$$\text{RankLoss} = \sum_{s(i),s(j) \in S(p)} \log(1 + e^{f(i,j)}); \tag{4.4}$$

$$f(i,j) = -\text{sign}(y_i - y_j)(\hat{f}(s_i) - \hat{f}(s_j)). \tag{4.5}$$

We can use the prediction $\hat{f}(x)$ to select the top-performing implementations of a

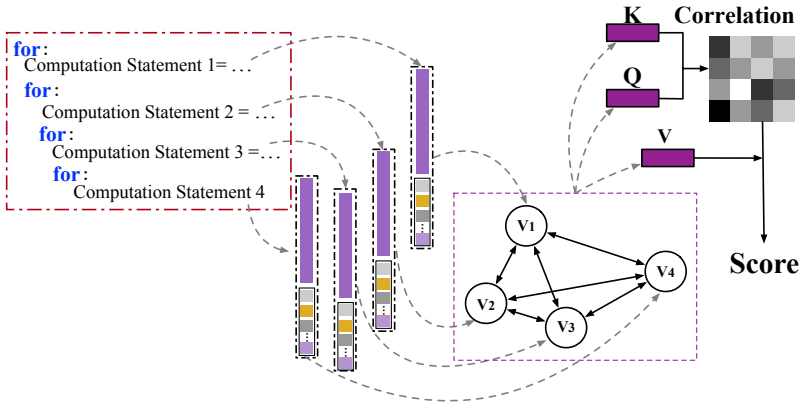


Figure 4.4: *Self-attention between statement vector features during the compilation*

full tensor program p. The computation graph G is trained for tensor programs extracted from all subgraphs. The throughput of all tensor programs is normalized to be in the range of $[0, 1]$.

4.4 Transfer Learning

The trade-off between search time and performance improvement is interesting to explore and exploit, as long search times may not always be acceptable. Our current focus is on developing a cost model for optimizing tensor operators on a specific hardware platform. However, in practical settings, we require a cost model that can be used across various hardware platforms. This would allow us to reuse a single-cost model for multiple platforms by providing it with new online data during auto-tuning. To achieve this, we pre-train the cost model with an offline static dataset and exploit transferable features that are invariant to both source and target domains to speed up the optimization process, as depicted in Figure 4.3. The use of transferable features greatly contributes to the success of transfer learning, as different designs may have varying degrees of invariance. By training the cost

model offline using a dataset, we can significantly reduce the frequency of on-device measurements and use the pre-trained parameters as a starting point for new search tasks via transfer learning. In Figure 4.4, each tensor program is transformed into a sequence of vectors, with each vector representing a tensor computation statement. During training, all sequences are of the same length, and any shorter sequences are padded with zeros at the end. The padded items are masked out and excluded from the loss computation.

4.5 Extension with Tensorized Instruction

4.5.1 Tensorized Instruction on NVIDIA GPUs

The surge in hardware specialization has added further complexity to the problem at hand. In order to accelerate machine learning, modern hardware backends have introduced specialized instructions for efficient tensor computations. Examples of such advancements include Nvidia Tensor Core [**nvidia2017tensorcore**] and Google TPU [**jouppi2017datacenter**]. Additionally, experts in various domains have begun developing micro-kernel primitives, which employ highly optimized instructions to perform sub-computations and expedite domain-specific tensor operator libraries. These hardware instructions and micro-kernel primitives primarily operate on multi-dimensional tensor regions, enabling efficient tensor operations such as multi-dimensional loads, dot products, and matrix multiplications. To fully leverage the potential of these hardware backends, modern machine learning systems must optimize programs that consist of hierarchical loop nests, multi-dimensional loads, and tensor intrinsics. We refer to this optimization challenge as *tensorized instruction optimization*. Currently, most tensorized programs are optimized by

domain experts who combine tensorized primitives with multi-dimensional loops, threading patterns, and data caching techniques to create specialized kernel libraries like Intel MKL-DNN [**intel2017mkldnn**], ARM Compute Library [**acl**], and NVIDIA cuDNN [**chetlur2014cudnn**]. These libraries are then utilized by popular machine learning frameworks such as TensorFlow [**abadi2016tensorflow**], PyTorch [**paszke2019pytorch**], and MXNet [**chen2015mxnet**]. However, the support for an expanding range of models and backends requires significant engineering efforts, and the adaptation of these libraries to the ever-changing and expanding landscape of machine learning applications takes time, hindering the development of new machine learning models.

Notably, the Tensor Core [**nvidia2017tensorcore**] on NVIDIA GPUs stands out as a significant breakthrough. Unlike the scalar-to-scalar primitives in CPUs or the general primitives on GPUs CUDA Cores, Tensor Core offers specialized tensorized instructions that can achieve higher throughput. It is noteworthy that the original tensorized instructions on Tensor Core was designed to handle basic GEMM operations with half-precision input and full-precision output. More recently, new features have been introduced on the latest GPU architecture to support different data types such as `int8`, `int4`, and `int1`. Each cycle of a Tensor Core can carry out Fused-Multiply-Add (FMA) operations, whereby the input values are in half-precision, while the output values can either be in half precision (`FP16`) or full precision (`FP32`).

The matrix tiling shapes of $\boldsymbol{A}$ with $(M \times K)$ and $\boldsymbol{B}$ with $(N \times K)$ can have various configurations depending on the data type and the different GPU micro-architecture. For instrance, if we set the datatype as 1-bit, the Tensor Core requires $M = N = 8$ and $K = 128$. Unlike the CUDA Cores, which necessitate users to define the execution flow of each thread, the Tensor Core only requires the collaboration of a

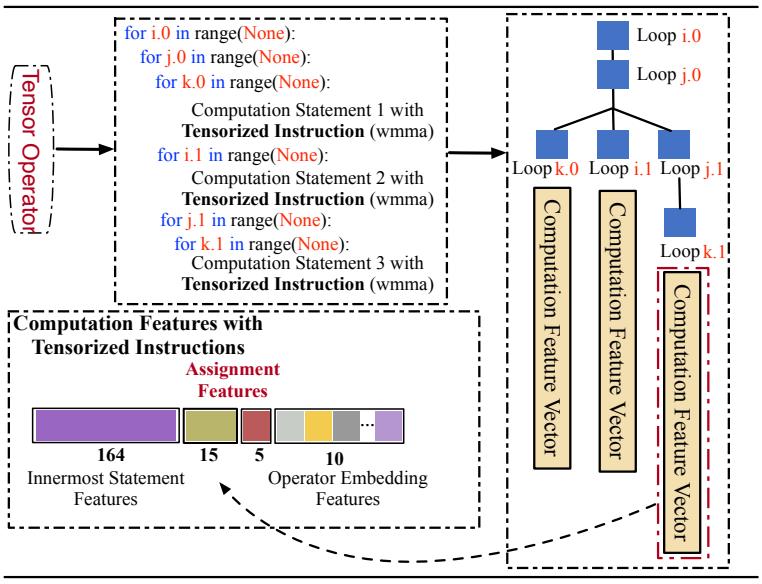


Figure 4.5: *Feature vectors with tensorized instructions for a tensor operator during the compilation.*

warp of 32 threads. The process of utilizing Tensor Cores can be summarized as follows: *i)* prior to invoking Tensor Cores, all registers within a warp of threads must collectively store the matrix tiling in memory called a `fragment`, which enables data sharing across all registers. This intra-warp sharing mechanism presents numerous opportunities for fragment-based memory optimizations; *ii)* the loaded matrix fragment serves as inputs for the Tensor Core, generating the output fragment, which also encompasses the registers from each thread in a warp. Data movements among these registers are managed collaboratively by the warp of threads.

4.5.2 Assignment features for Tensor Program Characterization

In order to get the high-performance implementation of each operator by compilation, we can transform the generation of tensor program with tensorized instruction into an optimization problem that explores the potential configurations in a hierarchical scheduling space.

Problem Formulation. Given code generation function $_2$, high-level structure generation parameters ϕ_1, low-level detail sampling parameters ϕ_2, register-level computation abstraction ϕ_3, data movement abstraction ϕ_4, computation-intensive operator σ and operator setting k (*e.g.*, kernel size), our goal is to use $\phi_{1,2,3,4}$ to build a hierarchical scheduling space and generate tensor program p to achieve the optimal prediction y^* on a specific GPU platform $\mathbb{T}$ with tensorized instruction.

$$\begin{aligned} \phi^*_{1,2,3,4} &= \arg\max_{\phi} y_t, \\ y_t &= f_{\mathbb{T}}(_2(\phi_1, \phi_2, \phi_3, \phi_4 | \sigma, k)). \end{aligned} \tag{4.6}$$

The performance model $f_{\mathbb{T}}$ predicts score y_t of the tensor program p with tensorized instruction and the accuracy of the $f_{\mathbb{T}}$ is crucial in searching ideal program configuration in the large scheduling space. Our tensor program characterization is based on the representation of intermediate representation. A tensor program is caracterized as an ordered tree of computation feature vectors as shown in Figure 4.5. A computation feature vector that includes three pieces of information: *i)* innermost statement features; *ii)* assignment features; *iii)* opeartor embedding features. We describe each of these components, the key features we aim to encode by each one, and how to combine them in a compact way during the tensor program compilation. Compared with the previous compilation framework with auto-tuning, the tensorized instructions are quite new and there is a lack of understanding of the behavior during the execution. In order to achieve a fully automatic compilation flow for generating the tensor program with tensorized instruction, the core of this flow is the performance model which estimates the latency of tensorized program candidates on the hardware. Therefore, it is important to design the performance model which can clearly specify the behavior of tensorized instruction. Based on

Algorithm 5 Extension with Tensorized Instructions.

Require: Search space $\phi_1, \phi_2, \phi_3, \phi_4$ with operator σ and setting k.
Require: Algorithm and iteration matching access matrix $\boldsymbol{A}, \boldsymbol{I}$.
Require: Tensorized instruction access matrix $\boldsymbol{T}$.
Ensure: Tensor program p^* with best configuration c^*.

while nTrials $<$ eachSubgraphTrials **do**
  $GS_1 \leftarrow$ `GenerateHighSketch(`ϕ_1, σ, k`);`
  $GS_2 \leftarrow$ `Sampling(`GS_1, ϕ_2, σ, k`);`
  $GS_3 \leftarrow$ `RegisterCompute(`GS_2, ϕ_3, σ, k`);`
  $GS_4 \leftarrow$ `RegisterDataMove(`GS_3, ϕ_4, σ, k`);`
  `P` $\leftarrow$ `EvolutionSearch(`GS_1, GS_2, GS_3, GS_4`);`
  for $p \in$ `P` **do**
    $c \leftarrow f_{\mathbb{T}}(_2(\phi_1, \phi_2, \phi_3, \phi_4 | \sigma, k))$;
    `Validation(`c`);`
  nTrials $\leftarrow$ nTrials $+$ batchSize;
$c^* \leftarrow$ best tensor configurations with tensorized instructions;
function VALIDATION(**G**, **C**)
  $\boldsymbol{A}^{'} = \boldsymbol{T} \times \boldsymbol{I}$;
  $\boldsymbol{T}^{'} = \boldsymbol{A} \times \boldsymbol{I}^{T}$;
  Return $(\boldsymbol{A}^{'} = \boldsymbol{A})$ and $\boldsymbol{T}^{'} = \boldsymbol{T}$;

the previous design experience in [**bai-2023-atformer**], we also use the innermost statement features to handle the general for-loop nests based optimization in a full tensor program and opeartor embedding features for the tensor category. The main difference is that we design the a set of novel assignment features between previous computation feature vectors. The idea behind the assignment feature is to make the performance model handle the tensorized instructions into the compilation process. That is, these assignment features can reform the low-level tensorized instructions into equivalent high-level progarm representation. We category assignment features into two components: 1) data movement abstraction; 2) computation abstraction. The goal of this design is to formally define the behavior of data movement and compuation so that our performance model can automatically analyze the tensorized instruction on the hardware during the compilation.

Data Movement Abstraction. Memory abstraction is a list of statements in GPU

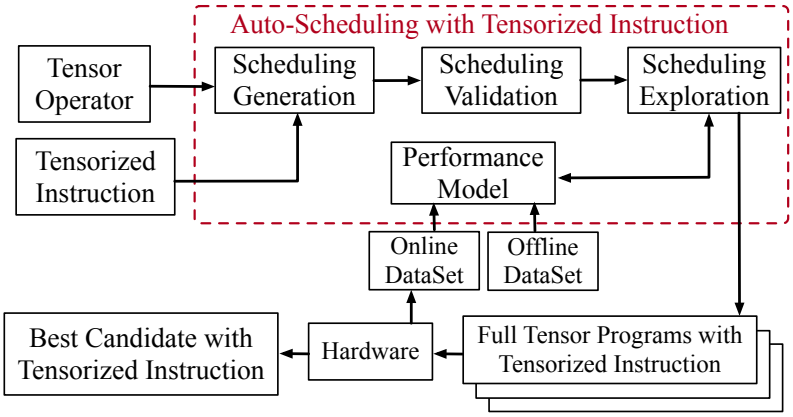


Figure 4.6: *Automatic compilation flow for the tensor program with a learned performance model. Our framework take a tensor operator and tensorized instruction as inputs and generate the best low-level implementation on domain-specific accelerators.*

programming. Each statement specifies the scope, operands, and memory access indices. For each buffer statement access, we extract features for it. While different statements can access different number of buffers, we conduct feature extraction for at most five buffers. Then we pad zeros if a statement accesses less than five buffers and remove small buffers if a statement accesses more than five buffers. On the Tensor Core computation, instruction such as `wmma::store_matrix_sync`, `wmma::load_matrix_sync`, and `wmma::fragment` are used to describe the data movement among the complex memory hierarchy. The register-level abstraction requires the `fragment` to fill six variables to formally execute this instruction. Take the tensorized instruction `wmma::fragment<Matrix_a,M,N,K,Dtype,Row>` as example, the `Matrix_a` parameter in the context specifies the type of matrix $\boldsymbol{A}$, while the dimensions M and N specify the number of rows and columns of matrix $\boldsymbol{A}$, respectively. Additionally, the parameter K specifies the size of each slice, and `Dtype` specifies the precision or data type of the tiling. The `Row` parameter indicates that the matrix elements are arranged in row-major order in fragement memory.

Collectively, these parameters form the WMMA fragment object. This object

represents a tiling of matrix that is loaded into the shared memory of the GPU. The purpose of this operation is to facilitate matrix computations using tensorized instructions. By organizing the data into tilings and storing them in shared memory, we can take advantage of the high-performance capabilities of the GPU's Tensor Cores for efficient matrix operations. In our assignment feature, we have formulated 15 designated sites to symbolize diverse tensorized instructions, encompassing `wmma::store_matrix_sync` and `wmma::load_matrix_sync`. These sites can be employed to indicate the category, placement, and sequence of distinct instructions, along with other attributes linked to them during the compilation.

Computation Abstraction. We extract the above features for one innermost non-loop statement in the context of a full tensor program. The features include categorical features and numerical features. We use one-hot encoding to encode category features. Computation abstraction is a statement that specifices the operands, arithmetic operations among operands, and data access indices of operands for each tensorized instruction. We use `wmma::mma_sync<C_frag,A_frag,B_frag,C_frag>` tensorized instructions during the computation. The instruction corresponds to a total of four register variables, namely `C_frag`, `A_frag`, and `B_frag`. These variables represent different fragments of matrices in the computation. Specifically, the first `C_frag` refers to the output matrix fragment, `A_frag` represents the fragment of the first input matrix, and `B_frag` represents the fragment of the second input matrix. It is important to note that the second `C_frag` refer to the same memory block with the first `C_frag`. This instruction synchronously performs the multiplication and accumulation operations between the fragments of `A_frag` and `B_frag` matrices, and the resulting values are written into the `C_frag` fragment. In our assignment feature, we have incorporated a flag to distinguish the computation abstraction. As a result, a total of 5 designated sites are utilized to encode the

parameters within this feature.

4.5.3 Auto-Scheduling with Tensorized Instruction

In the last section, we introduce assignment features with data movement and computation abstraction. In order to fully take advantage of the set of improvements, we need an automatic solution to optimize over a set of schedulings and map computations to the standard tensorized instructions. In this section, we introduce a tensorized instruction-aware automatic scheduler to solve this problem.

Scheduling Generation. Figure 4.6 provides an overview of our auto-scheduling for tensorized instruction. Our compilation framework takes as input a description of the workload operator from a neural network, as well as the tensorized instructions specific to the domain-specific accelerators. The auto-scheduler initially generates candidates for tensorized instructions by analyzing the computation patterns. It then generates candidates for tensor program sketches that incorporate the tensorized operations, while also determining the data movement based on the computation patterns. Within the search space defined by the tensorized instructions, we employ an evolutionary search guided by a performance model with assignment features. The entire process revolves around tensorization and makes use of abstractions for both data movement and computation. During compilation, a program sketch is constructed, where certain parts of the program structure are fixed, while allowing for flexibility in parameter choices such as loop tiling size and computation caching decisions. We generate sketches by applying predefined rules for sketch generation that operate on tensorized instructions. Consequently, the key challenge in the generation step lies in correctly mapping algorithm-defined tensor operators to their corresponding tensorized instructions. In other words, it involves establishing the correspondence between iterations in the algorithm and iterations in the tensorized

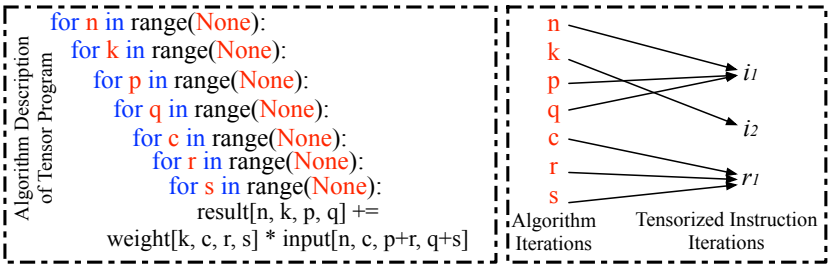


Figure 4.7: *The relationship between algorithm iterations and tensorized instruction iterations.*

instructions.

As depicted in Figure 4.7, we present a motivating example that demonstrates the mapping of algorithm iterations n, p, q to tensorized instruction iterations i_1, algorithm iterations k to i_2, and algorithm iterations c, r, s to r_1. This matching, illustrated in the figure, enables the transformation of the original convolution tensor operators into an equivalent matrix multiplication. The tensor core is designed to load two matrices, each with dimensions 16×8 and 8×8, into registers. It then performs the matrix multiplication concurrently for the $16 \times 8 \times 8$ dimensions, followed by storing the resulting output back to the global memory. During this process, we take into consideration two types of constraints: the index range of each tensorized instruction and the memory capacity. The domain-specific accelerator is capable of computing or handling only a fixed size of tensors at a time. It is worth noting that the problem size of a tensorized instruction can be determined from its index range, which is exemplified in the computation abstraction. As for memory capacity, each register fragment on the GPU can only accommodate a limited amount of data. Consequently, we divide the entire input/output data into smaller tiles and load/store these data tiles multiple times. This necessitates updating the base address and strides accordingly.

Scheduling Validation. However, it is important to note that not all generated

mappings may be valid with tensorized instructions. To address this, we have designed a validation algorithm that ensures only valid tensorized candidates are considered. The details of this algorithm can be found in Algorithm 5. The Access matrix is a binary matrix that describes the data access relationship between indices and tensors. Each row of the matrix represents a tensor, while each column represents an index. If an index and column are used to access the data of a tensor at a specific row, the corresponding value in the access matrix is set to 1; otherwise, it is set to 0. Similarly, the Matching matrix is a binary matrix that represents the matching relationship between algorithm iterations and tensorized instruction iterations. In equations 4.7 and 4.8, we use the convolution operator to illustrate the access matrix, tensorized instructions, and the matching matrix between them. These binary matrices contain crucial access and matching information that is essential for validation. The algorithm's access relationship defines which algorithm iterations access which registers using tensorized instructions, while the tensorized instructions access the algorithm-defined operator tensors. Initially, the algorithm employs the tensorized instruction access matrix $\boldsymbol{T}$ and the iteration matching matrix $\boldsymbol{I}$ to calculate the algorithm's access relationship using $\boldsymbol{A'} = \boldsymbol{T} \times \boldsymbol{I}$. If $\boldsymbol{A'}$ is equal to $\boldsymbol{A}$, it indicates that the access behavior remains consistent for all indices and every pair of input/output tensors from the algorithm and tensorized instructions. Similarly, we can calculate the tensorized instruction access relationship using $\boldsymbol{T'} = \boldsymbol{A} \times \boldsymbol{I}^T$. The resulting $\boldsymbol{T'}$ is also a binary matrix and is expected to be the same as the tensorized instruction access matrix $\boldsymbol{T}$ if the iteration matching matrix $\boldsymbol{I}$ is valid. Our validation algorithm, while simple, is highly efficient in guaranteeing the exploration of a valid scheduling space. This ensures the preservation of the original semantics when utilizing tensorized instructions.

$$A = \begin{array}{c} \\ input \\ weight \\ result \end{array} \begin{array}{c} \begin{array}{ccccccc} n & k & p & q & c & r & s \end{array} \\ \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix} \end{array} \tag{4.7}$$

$$I = \begin{array}{c} \\ i_1 \\ i_2 \\ i_3 \end{array} \begin{array}{c} \begin{array}{ccccccc} n & k & p & q & c & r & s \end{array} \\ \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} \end{array}, T = \begin{array}{c} \\ s_1 \\ s_2 \\ out \end{array} \begin{array}{c} \begin{array}{ccc} i_1 & i_2 & r \end{array} \\ \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix} \end{array} \tag{4.8}$$

Scheduling Exploration. After generating valid scheduling during the compilation process, determining the performance of each scheduling option with specific code transformation and optimization passes poses a challenge. Following the tensor program generation and validation with tensorized instructions, we obtain a comprehensive scheduling space in conjunction with induced programs. To identify the best candidate with tensorized instructions, we employ an evolutionary search approach. Our search procedure commences with random initializations of choices for the given tuning knobs. Subsequently, we introduce mutations to the current set of tensor programs, fostering diversity and variation within the search. From the mutated candidates, we select the most promising tensor programs and assess their real performance on the hardware with tensorized instructions. This evaluation phase involves benchmarking and collecting a substantial amount of data. These data serve a crucial role in updating the performance model, integrating assignment features, to enhance our understanding of the performance characteristics associated with different scheduling options. By iteratively refining the performance model and leveraging the insights gained from the evaluation phase, we steadily converge

towards identifying superior tensorized instruction candidates. This iterative process not only optimizes the overall performance of the compiled code but also ensures that the selected tensorized instructions are the best fit for the given workload and architectural constraints.

4.6 Evaluation

4.6.1 Implementation Details

The performance model is implemented on the top of Ansor [**Ansor-OSDI2020-Zheng**] and evaluated from two aspects: end-to-end search efficiency and quality, as well as performance portability. We compare the proposed performance model against the state-of-the-art methods, including the statistic, DNN-based cost models and [**bai-2023-atformer**]. The items labeled with XGBoost represent the Ansor default configuration. We also provide a detailed ablation study of the model architecture, accuracy, loss function, convergence speed, and training scheme, with insights and qualitative results. The generated tensor programs are evaluated on two different GPU architectures: Turing RTX 2080Ti and Ampere RTX 3090, with float32 data types used for all evaluations. We train the cost model using the Adam optimizer for 50 epochs, with a starting learning rate of $7e^{-4}$ that decays to $1e^{-6}$, and a training batch size set to 512. We use TVM v0.8dev in TenSet [**Tenset-NIPS2021-Zhang**], LLVM 11.0, and CUDA 11.0 for compilation, while XGBoost 1.5.0 and PyTorch 1.7.1 are used for training models. The use of a "mask" is a widely adopted technique for training transformers. Our ablative models, including MHA, ATFormer-1L, ATFormer, and ATFormer-M, were also experimented with. MHA is the basic Multi-Head Attention layer, ATFormer-1L only has one encoder layer, ATFormer has two

encoder layers, and ATFormer-M uses the "mask" scheme during training. Ours means the ATFormer combined with assignment features for tensorized instruction compilation.

4.6.2 Dataset and Benchmark

We evaluated our design using TenSet, a large-scale and challenging dataset for search-based tensor compilers. TenSet comprises 52 million performance records of tensor programs obtained from real measurements on different hardware platforms. Various randomly generated tensor programs for popular workloads are compiled via the TVM compiler and executed on the target hardware platforms. To ensure the inclusion of diverse workloads essential for generalization ability, we collected tensor programs from 120 networks with 13,848 tasks on the NVIDIA Tesla T4 GPU. This dataset serves as a series of static offline datasets. We use float32 as the data type for all evaluations. We train our model with the Adam optimizer for 50 epochs with a starting learning rate of $7e^{-4}$, the learning rate decays to $1e^{-6}$, and the training batch size is set to 512. We use TVM v0.8dev in TenSet, LLVM 11.0 and CUDA 11.0 for compilation. Meanwhile, we use XGBoost 1.5.0 and PyTorch 1.7.1 for training models. To explore transferable features and fast adaptation of ATFormer between different hardware platforms, ATFormer is pre-trained using offline learning with a number of samples from TenSet, and then fine-tuned using online learning on different platforms. For the offline learning, we randomly sample 50, 100, 200, 300, 500 search tasks from TenSet NVIDIA Tesla T4 GPU. We train 40 models including XGBoost, LightGBM, LSTM, TabNet, Multi-head attention, ATFormer-1L, ATFormer, ATFormer-Mask for all of experiment evaluation in this chapter.

4.6.3 End-to-End Execution Evaluations

Workloads. We evaluate the performance of ATFormer on various DNNs, including small and large-scale models. For small-scale models, we use AlexNet, VGG-16, MobileNet-V2, ResNet-18/50 and Bert-Tiny to evaluate the design. As for the large-scale models, we use BERT and GPT-3 models, specifically BERT_{base}, BERT_{large}, GPT-2_{large} and GPT-3_{350M}. We report the the end-to-end inference latency with batch size 1 on RTX 2080Ti.

Baselines and Settings. For statistic model, we use XGBoost as a baseline which has proven to be a state-of-the-art feature-based model in auto-tuning framework [**Ansor-OSDI2020-Zheng**]. For DNN-based learning, we use LSTM with eight heads and 1024 hidden dimensions, and TabNet is implemented in TenSet as another baseline. Note that the search algorithm uses the default configurations, and the search terminates when it runs out of allowed measurement trials. We keep the rest of the factors the same for a fair comparison.

Main Results on CUDA Cores. Figure 4.8 shows the final optimized *total latency* results on the RTX2080Ti GPU. Overall, the ATFormer-series model performs the best in all cases. Compared with the tree-based model XGBoost, ATFormer outperforms them in all cases with $1.15 - 1.61\times$ speedup. Compared with the DNN-based model TabNet, ATFormer outperforms them in all cases with $1.14 - 2.14\times$ speedup. Compared with LSTM, ATFormer performs equally the best and achieves $0.96 - 1.48\times$ speedup. Although LSTM surpasses ATFormer a little in finding the best configuration on Bert-Tiny and VGG-16, the amount of computation that can be parallelized in ATFormer leads to a shorter used time. Overall, the experiment results from the GeoMean verify the effectiveness of the attention-based modules over the tree- and DNN-based performance models.

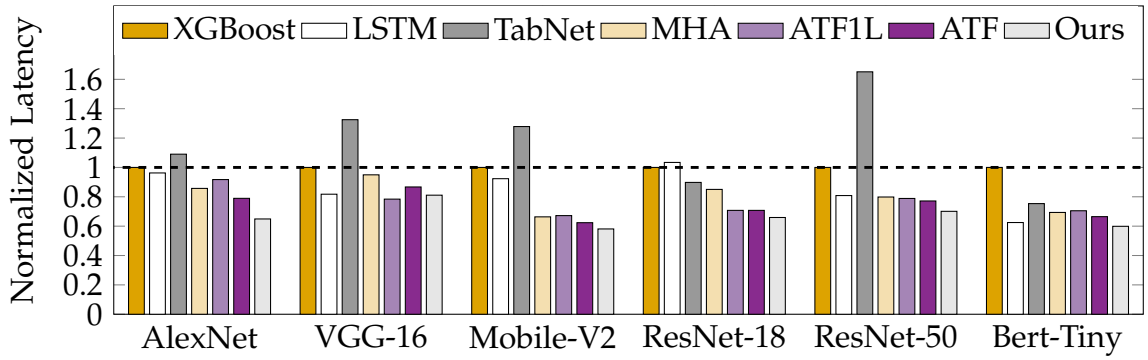


Figure 4.8: *End-to-end performance comparison of cost models across DNNs and normalized by the XGBoost.*

Main Results with Tensorized Instructions. The dataset collection process equipped with tensorized instruction takes 5 days with a server equipped with an Intel Core i9-12900K CPU and NVIDIA GeForce RTX 3090 GPU. The sampling selection process for the operator is conducted in a manner similar to that on the GPU's CUDA Cores. We use the floating point 16 (`fp16`) as the computation datatype in the whole evaluation. In the end-to-end latency evaluation experiment, we conducted tests on our performance model on a GPU using the same DNN benchmark as ATFormer [**bai-2023-atformer**]. Specifically, we targeted AlexNet, VGG-16, ResNet-18, ResNet-50, and Bert-Tiny, achieving latency values of 1.22×, 1.11×, 1.13×, 1.14×, and 1.13×, respectively.

4.6.4 Transfer Learning Evaluations

As mentioned in Section 4.4, we use RTX 2080Ti and 3090 GPUs as different platforms to verify our design by two typical metrics: *i) Fix the measurement trails and compare the total latency* and *ii) Fix a converged latency, and then compare the search time to reach it*. To explore transferable features and fast adaptation of auto-tuning between different hardware platforms, ATFormer [**bai-2023-atformer**] is

Table 4.1: *Transferable adaptation evaluation between different GPU platforms on ResNet-18.*

cost model		XGBoost		LightGBM		LSTM		TabNet		MHA		ATFormer-1L [**bai-2023-atformer**]		ATFormer [**bai-2023-atformer**]		ATFormer-M [**bai-2023-atformer**]		**Ours**	
(ms/s)		latency	time	latency	time	latency	time	latency	time	latency	time	latency	time	latency	time	latency	time	latency	time
ResNet-18-2080Ti		1.47	573	1.58	770	1.29	604	1.52	748	1.32	687	1.25	706	1.04	787	1.23	762	**0.78**	**622**
RTX 2080Ti Transfer	TenSet-50	0.86	535	0.98	527	1.02	614	1.13	583	1.01	595	1.00	602	0.97	600	1.00	611	**0.72**	**588**
	TenSet-100	0.96	533	0.98	526	1.07	615	0.82	596	0.87	602	1.00	602	0.85	594	0.84	611	**0.69**	**575**
	TenSet-200	0.99	536	0.86	525	1.07	611	0.88	582	0.83	602	0.82	612	0.82	604	0.82	632	**0.68**	**561**
	TenSet-300	0.89	538	0.85	526	1.02	622	0.83	583	0.85	600	0.81	609	0.89	612	0.87	607	**0.63**	**542**
	TenSet-500	0.96	530	0.81	529	1.03	622	0.82	574	0.83	593	0.87	598	0.84	612	0.79	615	**0.61**	**535**
ResNet-18-3090		1.07	589	1.11	676	1.24	762	1.64	741	1.11	658	0.97	661	1.02	677	3.01	665	**0.75**	**642**
RTX 3090 Transfer	TenSet-50	0.70	537	0.74	524	0.88	593	0.75	581	0.75	610	0.77	605	0.78	599	0.79	604	**0.68**	**589**
	TenSet-100	0.71	540	0.73	526	0.83	599	0.67	620	0.65	607	0.68	601	0.66	606	0.69	614	**0.62**	**574**
	TenSet-200	0.78	534	0.68	526	0.87	582	0.70	589	0.65	612	0.73	599	0.64	596	0.66	611	**0.59**	**557**
	TenSet-300	0.70	536	0.68	531	0.83	616	0.66	585	0.64	617	0.67	595	0.71	607	0.66	613	**0.58**	**537**
	TenSet-500	0.72	535	0.67	540	0.85	618	0.69	587	0.67	591	0.68	581	0.67	607	0.63	609	**0.52**	**529**

Table 4.2: *The performance of Transformer models on TenSet-500 with transfer learning.*

cost model		XGBoost		LSTM		MHA		ATFormer-1L [**bai-2023-atformer**]		ATFormer [**bai-2023-atformer**]		Ours		Speed up	
performance (ms / s)		latency	time	latency	time	latency	time	latency	time	latency	time	latency	time	latency	time
$BERT_{base}$	Traditional Learning	24.51	3028	32.89	3246	19.13	2890	18.77	2996	17.56	2874	14.52	2681	1.21×	5.17×
	Transfer Learning	23.82	654	33.35	880	19.98	602	19.51	648	18.72	578	15.78	519		
$BERT_{large}$	Traditional Learning	51.63	5016	59.81	5540	53.21	5218	54.32	5312	46.54	5232	42.18	5011	1.10×	5.18×
	Transfer Learning	52.49	1098	60.33	1302	55.88	1084	56.58	1192	47.76	1026	43.52	967		
GPT-2_{large}	Traditional Learning	489.12	6240	502.22	6531	467.22	6311	452.56	6380	445.52	6268	401.72	5996	1.11×	5.98×
	Transfer Learning	491.24	1392	503.52	1594	468.29	1375	454.18	1272	447.31	1102	408.58	1003		
GPT-3_{350M}	Traditional Learning	513.61	7789	542.23	8582	479.42	8082	468.59	7982	442.02	7891	389.97	7542	1.13×	6.97×
	Transfer Learning	514.42	1857	543.59	2302	480.12	1890	470.52	1920	443.62	1296	392.48	1081		

Table 4.3: *Pre-trained models on TenSet-500 via transfer learning with converged latency.*

cost model		XGBoost		LSTM		MHA		ATFormer-1L [**bai-2023-atformer**]		ATFormer [**bai-2023-atformer**]		ATFormer-M [**bai-2023-atformer**]		Ours	
performance (ms / s)		latency	time	latency	time	latency	time	latency	time	latency	time	latency	time	latency	time
RTX 2080Ti	Traditional Learning	1.26	1026	1.02	1487	1.03	1172	1.20	1269	1.02	1382	1.71	1124	0.89	1057
	Transfer Learning	1.23	281	1.05	348	0.99	261	1.15	264	0.99	271	0.93	266	0.86	206
RTX 3090	Traditional Learning	0.96	1004	1.03	1235	0.79	1125	0.87	1141	0.74	2054	0.94	2018	0.69	1878
	Transfer Learning	0.98	287	1.02	270	0.77	261	0.83	269	0.76	267	0.65	264	0.70	228

pre-trained with a number of samples from TenSet and then fine-tuned using online datasets on different platforms. Therefore, we divide our experiment settings into "traditional learning" and "transfer learning" parts. In order to further validate the compilation performance of our performance model under above settings, and to make a fair comparison with ATFormer [**bai-2023-atformer**], we adopted the same dataset partitioning method. The dataset with tensorized instruction is also divided into combinations of TenSet-50/100/200/300/500 and then fine-tuned using online datasets.

Traditional Learning. In Table 4.1, our performance model achieves the best total

Table 4.4: *Total latency and tuning time of different methods with ResNet-18, MobileNet-V2 and Bert-Tiny networks for end-to-end evaluation. The relative gains obtain for batch size = 1 with 300 measurement trials. "Register Abstraction" means the optimization for the tensorized instruction during the compilation.*

Methods	ResNet-18							MobileNet-V2							Bert-Tiny						
	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(a)	(b)	(c)	(d)	(e)	(f)	(g)
mask?			✓	✓			✓			✓	✓			✓			✓	✓			✓
pre-trained?				✓	✓		✓				✓	✓		✓				✓	✓		✓
RMSE Loss?	✓							✓							✓						
Rank Loss?		✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓
AutoTVM?						✓							✓							✓	
Assignment Features?							✓							✓							✓
total latency (ms)	1.42	1.04	1.23	0.81	0.83	1.92	**0.72**	0.53	0.51	0.76	0.39	0.40	1.29	**0.31**	4.18	3.41	3.97	2.32	2.46	5.07	**1.98**
search time (s)	781	787	762	620	611	3274	**599**	962	1000	958	617	604	2996	**587**	1127	1141	1150	818	816	3826	**784**

latency on RTX 2080Ti. ATFormer performs almost equally best with ATFormer-1L about *total latency* with a fixed measurement trail on 3090 GPU. The results show that self-attention based models perform best in the final performance compared with others without this design on different types of GPUs with tensorized instruction.

Transfer Learning. In Table 4.1, experiment results on RTX 2080Ti and 3090 show that the pre-trained parameters make the search convergence much faster. With the increasing number of training tasks in the offline dataset from 50 to 500, the learning ability of cost models with self-attention blocks, including MHA, ATFormer-1L, and ATFormer-Mask, become more stable, and they can adapt to the new tasks via transfer learning. Our performance model performs better than the statistic and DNN-based model XGBoost, LSTM in optimized *total latency* with the parameters trained from TenSet-100 to TenSet-500. All large-scale models are exported from Hugging Face, with a batch size of 1 and a maximum input sequence length of 512. As shown in Table 4.2, our performance model achieves latency speedups of 1.21×, 1.10×, 1.11×, and 1.13× on the RTX 3090 GPU compared with [**bai-2023-atformer**]. In terms of end-to-end tuning time, our performance achieves speedups of 5.17×, 5.18×, 5.98×, and 6.97× compared to traditional learning.

As for the TenSet-50 datasets, curves start from different points at the beginning,

and we can find that XGBoost performs best. It means that the transferable features in the ATFormer-series models are not fully exploited on the limited dataset (task#50) during the training. Obviously, the adaptation skills amplify rapidly with the increasing number of tasks on the offline dataset. From TenSet-100 to TenSet-500, we can find that ATFormer-series models show fast adaptation and generalization ability across hardware platforms and operators compared with XGBoost and LSTM models.

In Table 4.3, we make the traditional learning and transfer learning on different platforms for ResNet-18 have an approximate converged latency. Our performance model reduces the *search time* by up to 5.13× while maintaining the same search quality on RTX 2080Ti. This is the best speedup compared with 3.65× by XGBoost, 4.27× by LSTM, 4.81× by ATFormer-1L, 5.09× by ATFormer and 2.2× MHA, respectively. Under the same conditions, our performance model also performs the best with reducing the *search time* by up to 8.24× on RTX 3090 compared with 3.49× by XGBoost, 4.57× by LSTM, 4.24× by ATFormer-1L, respectively. Traditional learning with a mask-guided training scheme degrades the performance on *total latency* and *search time*. Comprehensive experiments show that it is not easy to make ATFormer-Mask have the approximate converged latency on RTX 2080Ti and 3090 compared with traditional learning and transfer learning. It means that ATFormer-Mask with pre-trained parameters has better task generation for tensor programs and achieves better performance during the compilation.

Overall, our performance model takes full advantage of transferable features learned from the source domain Tesla T4 GPU and transfers the knowledge to the different target domains RTX 2080Ti and RTX 3090 to accelerate the convergence speed with a fixed number of measurement trails. Fast convergence is desirable for many users of auto-tuning to have better control of the optimization cost and

Table 4.5: *Different architecture about performance model.*

architecture	n_head	hidden_dim	latency (ms)	search time (s)
MHA	2	512	3.71	652
	4	256	1.58	647
	4	512	1.24	641
	4	1024	1.29	652
	6	768	1.48	658
	8	512	1.19	658
ATFormer-1L [**bai-2023-atformer**]	4	512	1.25	706
ATFormer [**bai-2023-atformer**]	4	512	1.04	777
ATFormer-3L [**bai-2023-atformer**]	4	512	1.23	788
Ours	4	512	0.89	716

Table 4.6: *Accuracy of the cost models on TenSet.*

Architecture	XGBoost	ATFormer-1L [**bai-2023-atformer**]	ATFormer [**bai-2023-atformer**]	ATFormer-M [**bai-2023-atformer**]	Ours
TenSet-50	91.31	85.98	93.48	93.28	93.51
TenSet-300	92.24	90.41	93.82	93.33	93.85
TenSet-500	93.08	91.98	94.06	93.71	94.12

good performance. For instance, deployment engineers may want to obtain an optimized model as soon as possible or quickly get an upper-bound estimation of total inference latency in real-world production. They can use the cost model like ours with strong generalization as decent pre-trained parameters to accelerate not only the convergence speed but also the total execution inference time. Finally, comprehensive experiments with pre-trained parameters on different sizes of the TenSet dataset show that our design enable fast adaptation in not only cross-operator but also cross-platform scenarios.

4.6.5 Ablation Study

Various designs are evaluated in this section. We report the performance about *total latency*, *search time* on ResNet-18 and MobileNet-V2 and *accuracy* on the static datasets.

Loss Functions. Table 4.4 shows two different loss functions in our experiments. Method `(a)` is ATFormer with Root Mean Square Error (RMSE) loss function while

Table 4.7: *Hierarchical features and model architecture improvements for end-to-end evaluation.*

Methods	ResNet-18							
	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)
Hierarchical features?	✓		✓		✓		✓	
XGBoost?	✓	✓						
LSTM?			✓	✓				
ATFormer? **[bai-2023-atformer]**					✓	✓		
Ours?							✓	✓
w/o Transfer total lantency (ms)	1.47	1.63	1.29	1.58	1.04	1.18	0.88	0.97
w/o Transfer search time (s)	573	618	604	648	787	796	724	752
w/ Transfer total latency (ms)	0.96	0.98	1.03	1.12	0.84	0.91	0.76	0.81
w/ Transfer search time (s)	530	599	622	689	612	632	545	578

method (b) is with lambdaRank loss function. Compared with method (a) and method (b), we find that lambdaRank loss always outperforms RMSE in our design for different workloads of DNNs. It shows that the goal of a decent cost model is to rank the performance of different tensor programs by relative scores in a given search space.

Convergence Speed. In Table 4.4, method (d) is the proposed ATFormer, which adapts the pre-trained parameters to the new task via transfer learning into method (c). Note that ATFormer with the pre-trained parameters minimizes the *total latency* of all subgraphs in three DNNs as much as possible and the *search time* as quickly as possible. The proposed ATFormer improves the *total latency* by 4.66× speedup and convergence speed by 1.55× speedup. Method (f) is the AutoTVM with lambdaRank loss function. The performance is inferior to the baseline configuration. Method (g) is the ATFormer with introduced assignment features for the tensorized instructions. The performance of optimized inference latency and the search time are the best in all of the different configurations.

Training Schemes. In Table 4.4, method (c) incorporates the mask module into method (b) during traditional learning. Method (d) imports the mask module into

method (e) during transfer learning, resulting in a notable increase in convergence speed. It's worth noting that adding a mask scheme during traditional learning is not very helpful and can even cause a decrease in the total latency. However, for transfer learning with pre-trained parameters, incorporating the mask module is crucial for achieving faster convergence speed. The introduced techniques do not require expensive training resources in terms of both time and computation power. The only difference between method (d) and method g is the introduced assignment features for register-level abstraction during the tensorized instruction compilation. In the same training strategy, method (g) can outperform the original ATFormer on the three benchmark from total inference latency and search time.

Model Architectures. Table 4.5 lists ATFormer with various architectures. To achieve high accuracy while minimizing the model parameters, we find that the self-attention block, which contains four heads with 512 hidden dimensions, performs the best on the total latency and search time. Note that ATFormer does *not* benefit from deeper encoder layers in the Transformer model. Thanks to its simple and efficient architecture, the inference latency of ATFormer is consistently lower than that of the DNNs it optimizes. Thus, we set the two encoder layers as the final decision. Table 4.7 shows the relationship between the hierarchical-level features and different architectures to affect *total latency* and *search time* on ResNet-18. Method g in Table 4.7 means hierarchical features combined with assignment features for the tensorized instruction during the compilation for ATFormer. The performance of final optimized latency with transfer learning is greater than Method e. Method h means we only use the assignment features without the hierarchical features.

Accuracy. Table 4.6 presents the pairwise comparison accuracy of ATFormer and XGBoost on various scales of static datasets. The findings indicate that ATFormer outperforms XGBoost, demonstrating the highest measurement accuracy and pro-

Table 4.8: *The training time of the ATFormer series cost models during the offline optimization.*

Cost Model	TenSet-50	TenSet-100	TenSet-200	TenSet-300	TenSet-500
ATFormer-1L [**bai-2023-atformer**]	258	362	549	685	916
ATFormer [**bai-2023-atformer**]	299	384	588	712	951
ATFormer-M [**bai-2023-atformer**]	324	416	605	749	972
Ours	303	389	598	723	963

Table 4.9: *Pre-trained models with the converged latency on the Intel CPU platform.*

cost model		XGBoost		LSTM		MHA		ATFormer-1L [**bai-2023-atformer**]		ATFormer [**bai-2023-atformer**]	
performance (ms / s)		latency	time	latency	time	latency	time	latency	time	latency	time
ResNet-18	Traditional Learning	6.13	658	6.16	731	6.12	642	6.22	633	6.15	661
	Transfer Learning	6.16	334	6.25	451	6.19	346	6.29	419	6.18	304
ResNet-50	Traditional Learning	19.59	652	21.23	697	17.50	630	17.52	614	16.90	643
	Transfer Learning	20.01	342	21.99	461	18.11	338	17.91	362	17.02	318
VGG-16	Traditional Learning	36.92	891	39.85	1004	35.69	839	34.51	826	30.01	840
	Transfer Learning	37.51	395	40.17	422	36.79	326	34.87	318	34.88	270
BERT-Tiny	Traditional Learning	17.98	1012	19.22	1433	17.55	1126	16.09	1168	15.11	1232
	Transfer Learning	18.05	396	19.57	498	17.91	401	16.41	416	15.16	388

viding optimal search quality during the tuning. Table 4.8 presents the specific training times (s) of the ATFormer series models on static datasets. Note that our approach is also suitable for scenarios involving large batch sizes. By incorporating the proposed design into the new performance model, it can be observed that the overall training time of the performance model has not increased significantly. During the offline training process, the time required is comparable to that of ATFormer, maintaining a similar duration.

4.6.6 Other Platforms: Intel CPUs

We use the dataset from Intel Platinum-8272 to verify transferability on Intel E5-2698 CPU with a fixed converged latency (6.13ms) by the same measurement trials for ResNet-18. More details can be found in Table 4.9. Therefore, ATFormer also works well for CPU with lots of different DNN benchmarks including ResNet-50, VGG-16, BERT-Tiny with batch size 1. As for the ResNet-18, we fix the converged

latency to 19.59ms, the traditional learning will cost 658s to search the optimal configuration with XGBoost performance model. But the ATFormer can search the optimal implementation of ResNet-50 with 643s by the same measurement trials under the 16.90ms converged latency. We can get the same conclusions from the VGG-16 and BERT-Tiny neural networks.

4.7 Summary

This chapter introduces a novel performance model for optimizing tensor programs with tensorized instructions. We propose a set of novel assignment features that compiles tensor program with computation and memory abstraction to represent the intrinsics of the hardware accelerators. It enables systematic exploration of scheduling space and efficient matching with better performance for various tensor operators. The proposed learned performance model achieves significant speedup for model deployment on GPU Tensor Cores. Through transfer learning, the performance model achieves faster-converged latency and superior transferability across different hardware platforms, outperforming previous state-of-the-art benchmarks.

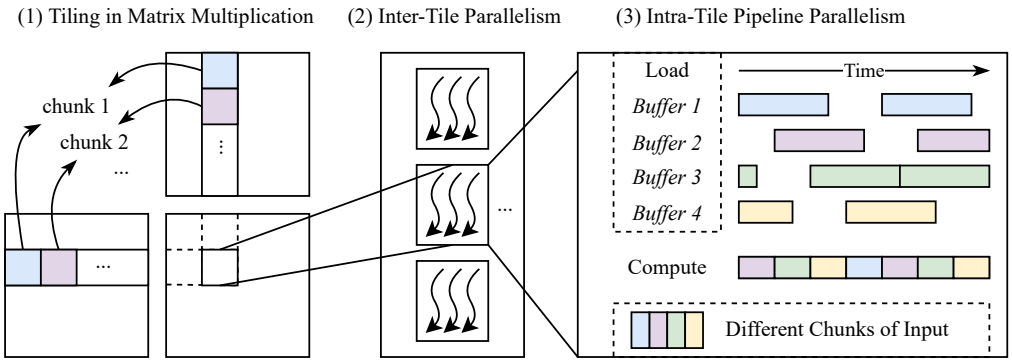
Chapter 5

ALCOP: Automatic Load-Compute Pipelining in Compiler for GPUs

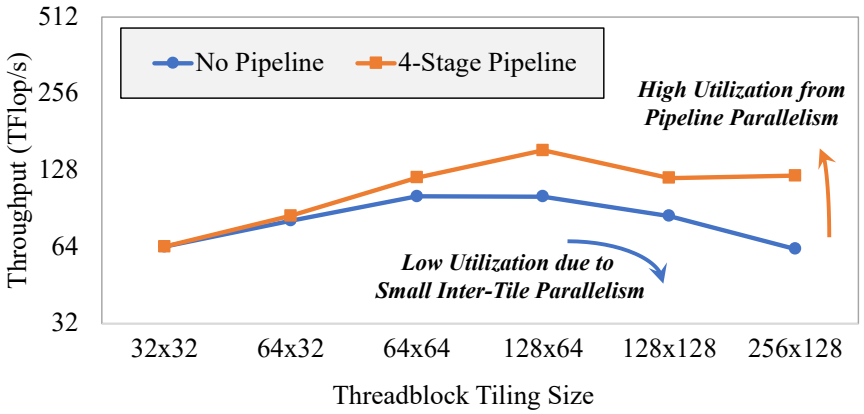
5.1 Motivation

Deep learning (DL) has achieved great success in a variety of application fields, spanning computer vision, natural language processing, and recommendation systems [**he2016deep**, **devlin2018bert**, **naumov2019deep**]. The widespread use of GPUs [**nvidiav100**, **nvidiaa100**] to accelerate DNNs makes an indispensable contribution in this AI era.

High-performance tensor programs on GPUs require complex optimization efforts. When Tensor Core was introduced to GPUs to accelerate deep learning, harnessing the power of Tensor Cores became the center of GPU software optimization, motivating the development of a number of libraries and compilers [**cutlass**, **yan2020demystifying**, **dakkak2019accelerating**, **feng2021egemm**, **chen2018tvm**, **katel2022mlir**]. Because Tensor Core throughput continued to increase but memory bandwidth lagged, research on tiling and fusion to improve data



(a) *Concept of tiling, inter-tile parallelism and pipeline parallelism.*

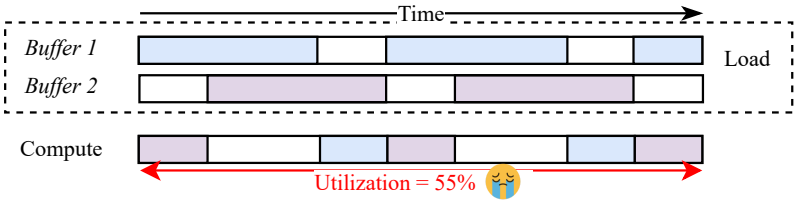


(b) *Motivating example: performance of a* $2048 \times 2048 \times 2048$ *matrix-multiplication tested on NVIDIA A100 with different tiling and pipelining choices.*

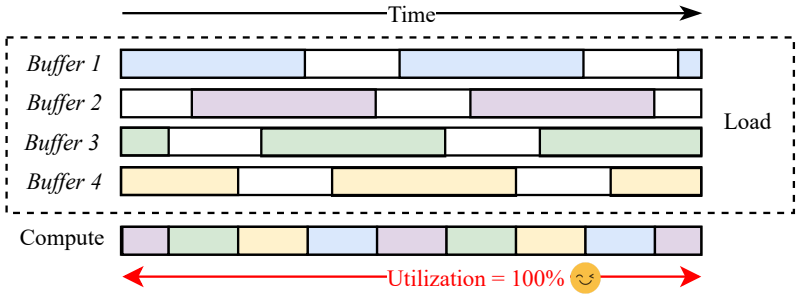
Figure 5.1: *Motivation of automatic pipelining. (a-3) explains the concepts of pipelining, which is overlapping data loading with computation. (b) gives a motivating example. With tiling only, the performance is always sub-optimal. Pipelining unleashes intra-tile parallelism and increases the performance under large tiling.*

re-use surged [**niu2021dnnfusion**, **zhao2022apollo**, **asplos22-zhenzheng**, **zheng2020fusionstitching**

However, aggressively large tiling limits the number of tiles and hinders inter-tile parallelism, a crucial GPU mechanism to achieve high utilization. Restoring the parallelism lost due to aggressive tiling becomes an important task. **Pipelining** – the overlap of data loading and computing – is an ideal mechanism for unleashing intra-tile parallelism. Figure 5.1 depicts the concept of pipelining and its performance advantages. As the difficulty of capitalizing on the ever-growing parallelism



(a) *Two-stage pipelining*

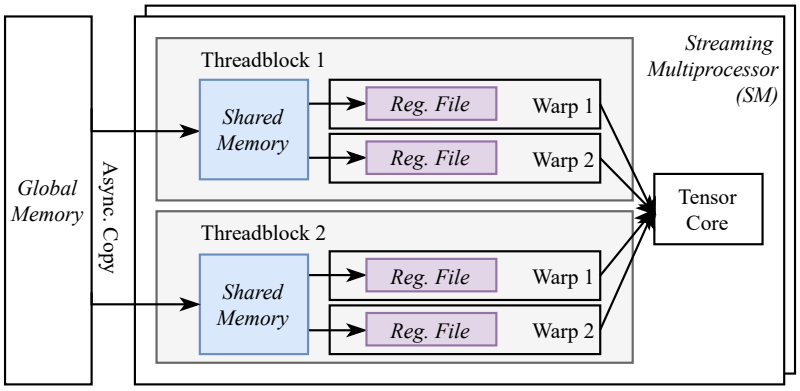


(b) *Multi-stage pipelining (four-stage here)*

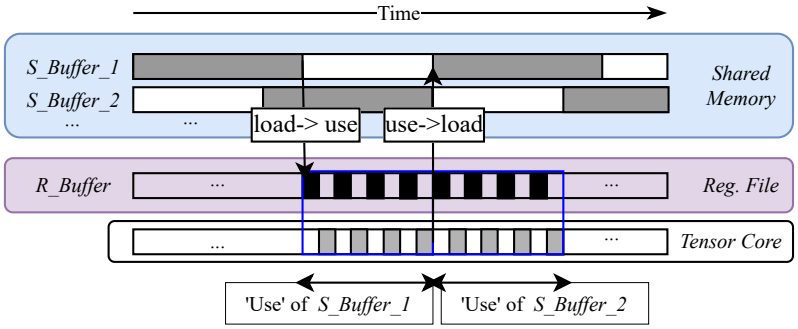
Figure 5.2: *Concept of multi-stage pipelining. (a) two-stage pipelining (or called double-buffering) is not enough to hide the data loading latency. (b) Four-stage pipelining can hide the data loading latency and achieve full utilization of the computing units. ALCOP supports multi-stage pipelining.*

in GPUs increases, the study of pipelining becomes essential.

Despite the necessity of pipelining optimization, existing approaches are either limited in their design space coverage or their degree of automation. Prior work [**katel2022mlir**] has studied double-buffering, a common but far less complicated case of pipelining. However, double-buffering is only a two-stage, one-level instance of the entire multi-stage, multi-level pipelining design space shown in Figure 5.2, and simplifying pipelining to double-buffering hinders a major performance gain (which will be evaluated in Sec. 5.6.1). Although deep learning systems can access comprehensive pipelining optimization from hand-written libraries [**cublas**] or compiler-integrated libraries [**xing2022bolt**], due to their fundamental difference from the tensor program generation workflow of DL compilers, they are inextensible to new operators and indecomposable with prior compiler passes such as



(a) *GPU memory hierarchy.*



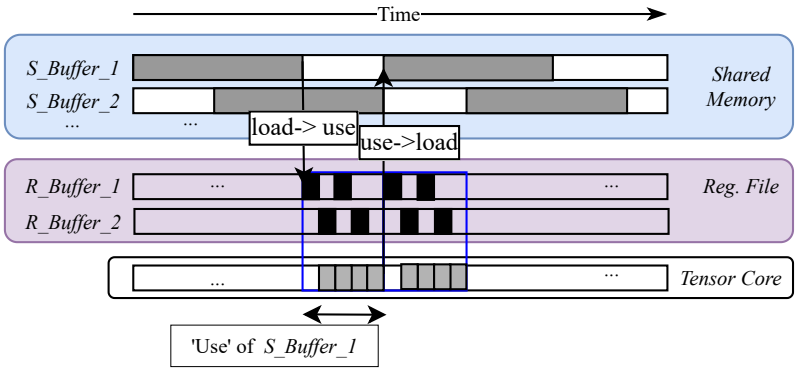
(b) *Single-level pipelining.*

Figure 5.3: *Concept of multi-level pipelining and inner-pipeline fusion. (a) shows the GPU memory hierarchy, with two levels of buffers: the shared memory and the register file. (b) shows the execution timeline of single-level (only shared memory) pipelining.*

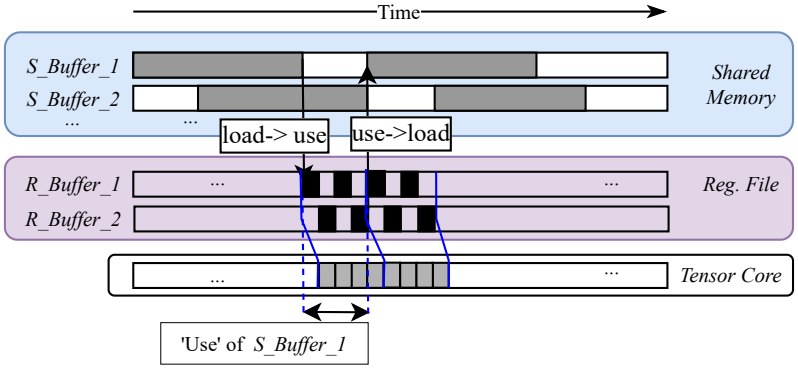
auto-fusion and auto-tiling.

Automatic pipelining presents three distinct challenges: workload complexity (diverse DL operators), hardware complexity (multi-level memory hierarchy), and design space complexity (coherent performance tuning factors).

Our key insight is that, instead of solving everything in a monolithic compiler pass, we should exploit the progressive lowering structure of DL compilers and the information exposed at each level. Specifically, we address the aforementioned three challenges through three decoupled and collaborative compilation modules: pipeline buffer detection, pipeline program transformation, and analytical-model



(a) *Multi-level pipelining without inner-pipeline fusion.*



(b) *Multi-level pipelining with inner-pipeline fusion.*

Figure 5.4: *(a) improves over (b) by pipelining the inner loop: register loading and computing. (b) improves over (c) via inner-pipeline fusion, which treats the repeated inner loop as a holistic loop and pipeline it. ALCOP supports multi-level pipelining with inner-pipeline fusion.*

guided design space search. Pipeline buffer detection addresses the workload complexity because it occurs during the scheduling phase when the entire dataflow is visible. The second module addresses the hardware complexity. During the program transformation stage, the intricate for-loop structure and data movement are revealed and modified. This module utilizes the safety check of the preceding module to execute the robust transformation. The third module addresses the design space complexity. It happens at the auto-tuning stage, where pipelining and other techniques are co-optimized. This module makes use of the preceding parameterized module. We further design an analytical hardware model to expedite

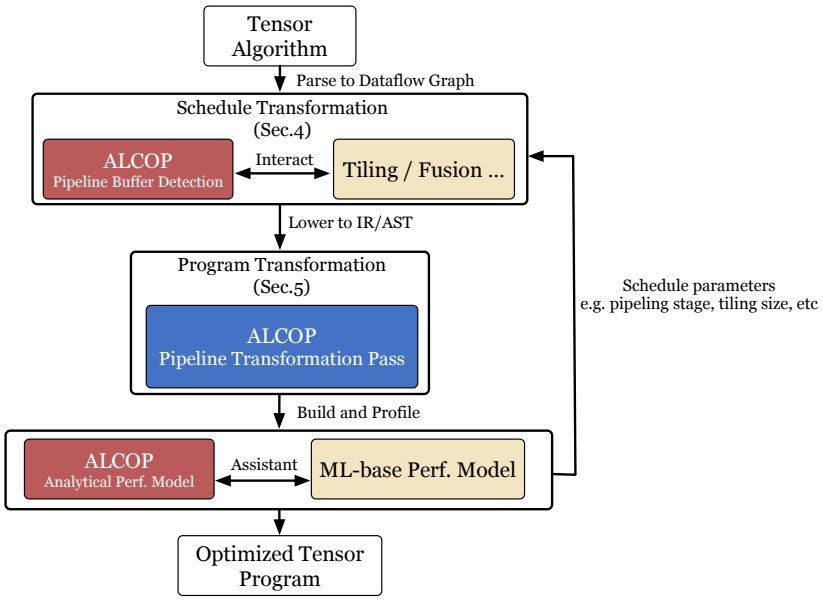


Figure 5.5: *The overview of ALCOP. It has three important modules including schedule transformation, program transformation and performance auto-tuning.*

the design space search.

5.2 System Overview

In this chapter, we propose ALCOP (Automatic Load-COmpute Pipelining), the first DL compiler solution (auto-scheduler, program transformation, auto-tuner) for automated multi-stage, multi-level pipelining; its architecture is shown in Figure 5.5. Additional contributions include:

- We design methods to examine each buffer for applying pipelining, including the ordering of pipelining and other schedule transformations to avoid mutual interference. (Sec. 5.3)
- We design a program transformation pass that handles index manipulation, synchronization injection, and prologue injection, among other transforma-

tions. (Sec. 5.4)

- We propose a pipeline-aware analytical performance model. Combining it with an existing machine-learning (ML) based tuning algorithm significantly improves the efficiency of schedule tuning. (Sec. 5.5)

Experiments show that ALCOP pass can bring on average 1.23×, and up to 1.73× speed-up on DL operators over TVM [**chen2018tvm**]. ALCOP brings 1.02-1.18× end-to-end inference speed-up for six DL models over TVM [**chen2018tvm**], and 1.01-1.64× over XLA [**xla**]. Through combining the analytical model with ML-based tuning, we can identify schedules with 99% performance compared to the best schedule in the entire design space while reducing the number of trials by 40×.

5.3 Scheduling Transformation

Automatic pipelining begins by identifying potential pipelining possibilities. We implement it through a schedule transformation pass in the compiler, which attaches the pipelining primitive to buffer variables in the program.

Pipelining can be applied to a *load-and-use* loop in which the *load* step copies data into a buffer and the *use* step reads data from the buffer. Consequently, the purpose of the schedule transformation is to identify and record "load-and-use" structures in a program. The pass marks the buffer variables within such load-and-use loops as pipelined buffers. Later on, a program transformation pass described in Section 5.4 will turn the load-and-use structure into its pipelined version.

Two important questions must be addressed: First, we must determine is what rules we should apply to identify the buffers that can be pipelined. The second one is determining the *ordering* if pipelining in relation to other schedule transformations, such as tiling, aware of their mutual effect.

5.3.1 Identification of Buffers for Pipelining

Constraints of pipelining come from not only the algorithm, *i.e.,* how the buffer is used, but also the hardware capabilities, *i.e.,* what forms of memory copy can be executed asynchronously. For each buffer variable, the following three rules are evaluated to determine whether pipelining can be applied. Firstly, we do not pipeline a buffer that is not produced by asynchronous memory copy. An asynchronous memory copy indicates that the memory copy is non-blocking, so we can initiate memory copies for future loop iterations in advance and meanwhile continue with the computation in the present iteration. Only when an explicit synchronization instruction is encountered does the program block to wait for the completion of the memory copy. If the data in a buffer is not produced by direct memory copy but rather by some compute operation, the buffer does not meet this condition.

Secondly, we do not pipeline a buffer produced outside of a *sequential* loop. The purpose of pipelining is to overlap the data-loading operation of future iterations with the computation of the current iteration. This load-and-use loop must be sequential and cannot be parallelized (bound to parallel threads) or unrolled. This condition is typically violated by stencil algorithms that use tiling to increase the reuse of the input tensor, but the buffer is only filled and used once, as opposed to being generated in a sequential loop. Consequently, the pipelining approach cannot be applied to these buffers.

The final rule is about *synchronizing the pipeline:* If the hardware platform supports only scope-based synchronizations, we inspect all buffers within the same scope and refuse to pipeline them if their synchronization positions do not match. Synchronizing the pipeline requires special memory barriers that await certain

loading instructions (*e.g.*, instructions issued in the fourth-last iteration in a 4-stage pipeline). On NVIDIA Ampere GPUs, such memory barriers are provided for the shared memory scope. Hence, the hardware is incapable of resolving this conflict if two buffers are both in the shared memory scope, but their barriers must be inserted at distinct positions in the program. If this conflict occurs, our schedule transformation refuses to pipeline these buffers.

5.3.2 Ordering of Schedule Transformations

Pipelining is applicable to three schedule transformations already in existence: cache-reading, tiling, and fusion. We will briefly introduce these transformations and then determine whether pipelining should be applied before or after them.

Cache-reading. It means inserting a read buffer for a tensor input. Given an algorithm and computation tensor `S2` from tensor `S1`, applying cache-reading means inserting a new tensor `S1_buf` which is an identical copy of `S1` but with a buffer scope. Cache-reading should be applied before pipelining since pipelining needs to be applied to buffers generated by the former.

Tiling. It is the process of dividing the output tensor into blocks. In conjunction with cache-reading, it can cache data within buffers to improve data reuse. Tiling should also be performed before pipelining. The second condition for a buffer to qualify pipelining, *i.e.,* whether there exists a sequential load-and-use loop, must be inspected based on the for-loop sketch after tiling.

Fusion. It means avoiding writing back intermediate data between two operators. Inlining, a specific type of fusion, should come after pipelining. Inlining a tensor means producing the value of the tensor precisely where it is used; this technique is often used on lightweight element-wise operations like datatype casting. Figure 5.6 shows an example in which originally `S2` is produced by applying element-wise

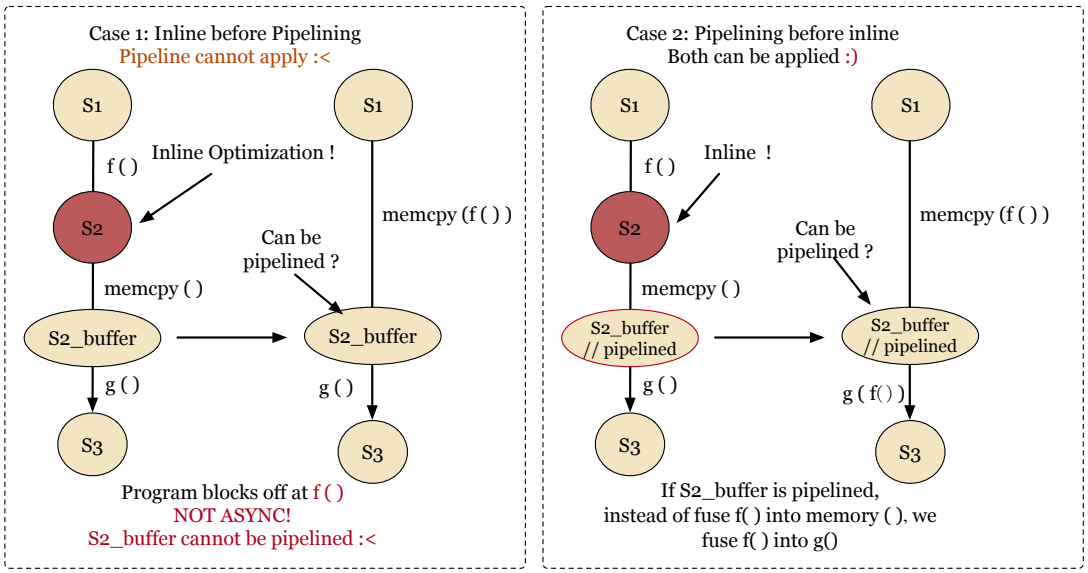


Figure 5.6: *The effectiveness study on the optimization order of inlining and pipelining. In case 1, after inlining, `S2_buf` can no longer be pipelined because it is no longer produced by an asynchronous memory copy. In case 2, after pipelining, inlining can still be applied.*

function $f(\cdot)$ to `S1`, and a buffer tensor `S2_buf` is injected after `S2` via cache-read. Inlining `S2` is equivalent to applying $f(\cdot)$ first and then copying the data directly into `S2_buf` without writing the data back to memory. According to our first rule outlined in the previous subsection, a pipelined memory buffer should be produced from an asynchronous memory copy. However, for `S2_buf` here, the operation to produce it is no longer asynchronous, as the explicit $f(\cdot)$ forces the program to stall, waiting for data to be loaded. And since buffer `S2_buf` is not produced asynchronously, it cannot be pipelined. Here in case 1, inlining impedes the opportunity of pipelining. Nevertheless, if pipelining is applied before inlining, like in case 2, the inlining of `S2` can still be applied, but in a different manner: Instead of inlining `S2` into `S2_buf`, we cache-read `S1` and fuse the computation $f(\cdot)$ into the production of `S3`. Thus, we ensure both sides are satisfied: the buffer is produced through an asynchronous copy and can be pipelined, while computation $f(\cdot)$ is fused and we avoid explicitly generating an intermediate tensor.

5.4 Program Transformation

In this section, we introduce the second component of automatic pipelining: transforming the program IR (Intermediate Representation) to implement pipelining. After the schedule transformation outlined in Section 5.3, the program is lowered to its IR form, composed of for-loops and load/store/compute operations. Figure 5.8 gives a sample input and transformed IR of the pipelining pass. Figure 5.7 also depicts the transformation steps.

5.4.1 Analysis

The First Step. Given a program IR, pipelining begins with the collection of pipelining hints inserted by the schedule transformation, including the buffer to be pipelined and the number of stages for each buffer.

The Second Step. Given a set of buffers we want to apply pipelining, the second analysis task is to reconstruct the producer tensor and consumer tensor(s) of these buffers. Then we can derive if there are multi-level buffers by deciding if the producer of a pipeline buffer is also a pipelined buffer. Since pipelined buffers are always produced via asynchronous memory copy, to determine the producer tensor, it suffices to retrieve which tensor it copies from. The decision of consumers happens when IR traversal encounters a load operation from this buffer.

The Third Step. This step is to determine the sequential load-and-use loop for each pipelined buffer. This identifies the iteration variable to be pipelined and is required by all the index shifting operations in the transformation steps. The sequential loop can be determined as follows: starting from the instruction that copies data into the buffer, traversing all the for-loops from inside to outside, and finding the first

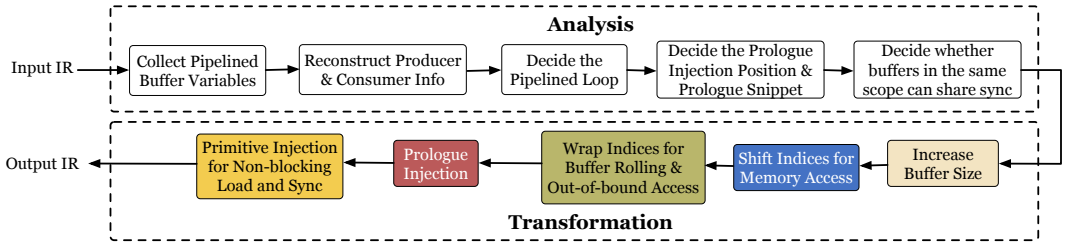


Figure 5.7: *Workflow and example input and output of the pipelining program transformation*

sequential loop whose iteration variable is not used to index inside this buffer. This means the buffer is reused for each iteration of this loop, which is the loop we want to pipeline. Take Figure 5.8 as an example, the pipelined loop for `A_shared` is with iteration variable `ko`, and the pipelined loop for `A_reg` is with variable `ki`.

The Fourth Step. We should document the pieces of code that *loads* and *uses* this buffer. This information is required for the injection of synchronization primitives and prologues. In the *Input IR* in Figure 5.8, the "loading" part for `A_shared` is Line `8`, and the "using" part is the `ki` loop and everything inside. The loading part for `A_reg` is Line `13`, and the using part is Line `15`.

The Fifth Step: We also need to decide where to inject prologues. Since we transform the program to issue memory copy for future iterations while doing computation for the current iteration, we need to move the first few stages of memory copy ahead of the start of the main load-and-use loop. This pre-posed loading code block is a prologue. Typically, prologues can be injected simply before the pipelined loop. However, when a multi-level pipeline appears, the prologues of inner pipelines must be injected into the sequential loop of the outer-most pipeline, in order to build a holistic pipeline as opposed to a recursive one as shown in Figure 5.4a.

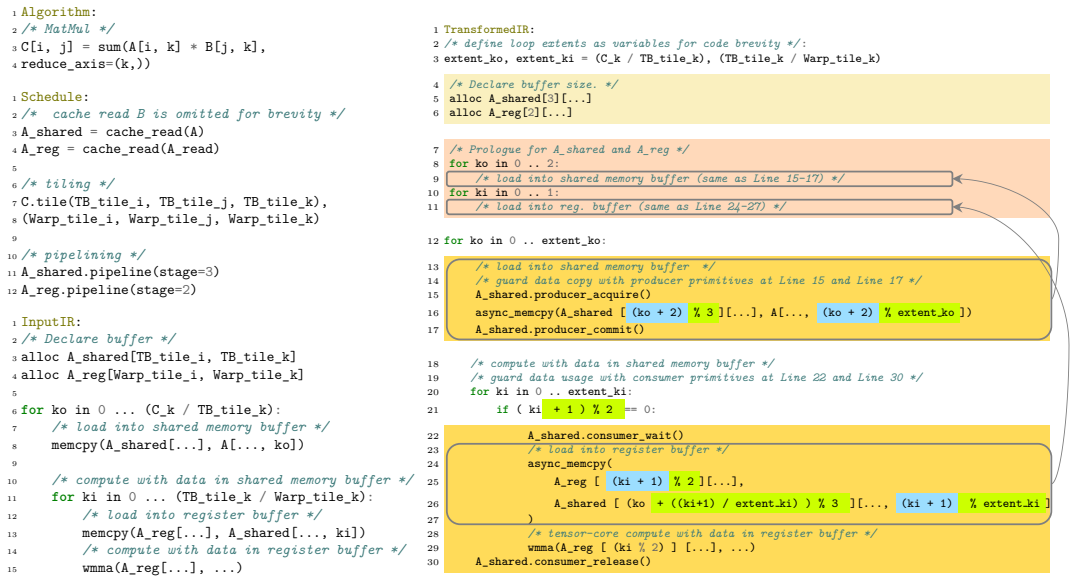


Figure 5.8: *An example to illustrate how to transform an original Tensor-IR (left) to its pipelined version (right).*

5.4.2 Transformations

Five steps are required to transform a *load-and-use* loop into a pipelined loop. The *Transformed IR* in Figure 5.8 shows the transformed version of the *Input IR* in the same figure.

The First Step. This step increases the size of the memory buffer by the number of pipeline stages. Relevant transformed code is highlighted in light yellow.

The Second Step. This step shifts the indices used in memory access. The eelevant code is highlighted in blue. In each *load-and-use* iteration, we issue asynchronous memory copy for *future* iterations rather than the present iteration. Therefore, we need to increase the pipelining loop variables in the memory access indices. If it is a 3-stage pipeline, for instance, we should load data 2 iterations ahead.

The Third Step. This step handles indices for buffer rolling (circular access) and out-of-bound wrapping. Relevant code is highlighted in green. There are two cases

that we need to wrap indices: first, when we use the pipelining variable to index a chunk of the buffer, we should use the modulo of pipeline iteration variable divided by pipeline stages. Secondly, since we increase the pipelining variable, it is possible that we index out of the bound of its producer tensor. We must take the modulo of the pipelining variable divided by its own extent to avoid indexing out-of-bound. A complicated case is in a multi-level pipeline when the overflow of the inner pipeline causes the increase of the outer pipeline variable. Line `26` in the transformed IR handles this case.

The Fourth Step. This step injects prologue primitives. The contents of prologues are the memory copy of the first `n_stage -1` chunks of data, where `n_stage` is the number of the pipeline stage. We inject prologue at the positions we record in the preceding analysis pass.

The Fifth Step. The final step injects synchronization primitives. The pipeline is guarded by four primitives: `producer_acquire` or `commit`, `consumer_wait`, and `consumer_release`. `producer_commit` commits a batch of asynchronous loading operations. `consumer_wait` blocks until a previous batch of loading is completed. When the pipeline is full, `producer_acquire` blocks until the execution of `consumer_release`[1]. The pairs of producer/consumer primitives are put around the loading/using part of the buffer, respectively, as shown in Line `15, 17, 22,` and `30` of the transformed IR.

[1] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#with-memcpy_async-pipeline-pattern-multi

Table 5.1: *Analytical Performance Model in ALCOP*

Category	Model
Kernel Latency Model	$T_{kernel} = T_{threadblk} \times N_{threadblk\_batch}$
Pipeline Latency Model	Input: $T_{load}, T_{use}, N_{loop}, N_{pipe}, N_{mplx}$ Output: $T_{load\_use\_loop}$ If $T_{load} \leq (N_{pipe} \times N_{mplx} - 1) \times T_{use}$: $T_{load\_use\_loop} = T_{use} \times N_{loop}$ Else: $T_{load\_use\_loop} = (T_{load} + T_{use}) \times N_{loop} \div N_{pipe}$
Threadblock Latency Model	$T_{threadblk} = T_{init} + T_{main\_loop} + T_{epilogue}$ $T_{init} = T_{smem\_load} + T_{reg\_load}$ $T_{main\_loop} = \texttt{PipelineLatencyModel}(T_{smem\_load}, T_{smem\_use}, N_{smem\_loop}, N_{smem\_pipe\_stage}, N_{threadblk\_per\_SM})$ $T_{smem\_use} = \texttt{PipelineLatencyModel}(T_{reg\_load}, T_{compute}, N_{reg\_loop}, N_{reg\_pipe\_stage}, N_{warp\_per\_threadblk})$
Computation Latency Model	$T_{compute} = \frac{FLOPs_{one\_reg\_loop}}{Throughput_{SM} \times \texttt{Util}(N_{warp\_per\_threadblk}, N_{threadblk\_per\_SM})}$
Memory Latency Model	$T_{snen\_load} = MAX(T_{LLC\_load}, T_{DRAM\_load})$ $T_{LLC\_load} = LAT_{LLC\_read} + \frac{Bytes_{one\_smem\_loop} \times N_{threadblk\_per\_threadblk\_batch}}{BW_{LLC}}$ $T_{DRAM\_load} = LAT_{DRAM\_read} + \frac{Bytes_{threadblk\_batch\_workset}}{BW_{DRAM}}$
Epilogue Model	$T_{epilogue} = LAT_{DRAM\_write} + \frac{Bytes_{output\_tile} \times N_{threadblk\_per\_threadblk\_batch}}{BW_{DRAM\_write}}$

5.5 Static Analysis Guided Tuning

This section introduces how we combine a static analytical performance model with existing machine-learning (ML) based auto-tuning [**chen2018learning**] to choose schedule parameters. The key component is a novel performance model aware of pipelining and its interaction with other optimizations, as illustrated in Figure 5.9.

5.5.1 Top-Level Model

Our analytical model is shown in Table 5.1. At the top level, the threadblocks are grouped into threadblock-batches (*threadblk_batch*), and one threadblock-batch occupies all Streaming Multiprocessors (SMs) at a time. Since all threadblocks execute the same program, the latency of a kernel equals the threadblock latency multiplied by the number of batches. The number of threadblock-batches in a kernel depends on the GPU scheduling policy, which we learn through performance

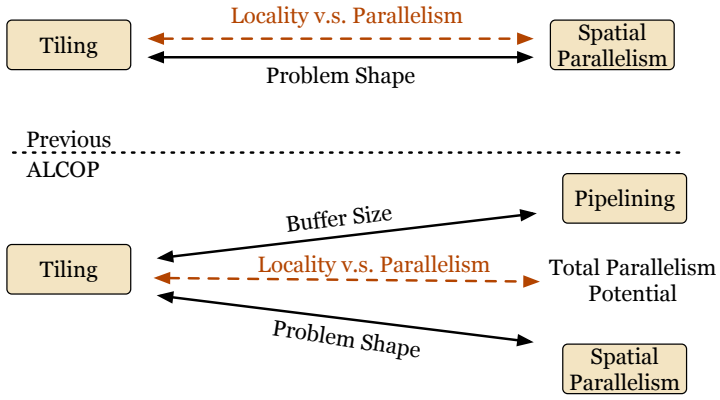


Figure 5.9: *A high-level view of the performance model. Compared to prior work [***lym2019delta***], our model takes into account the constraints and trade-offs among pipelining, tiling and spatial parallelism.*

profiling. The maximum number of threadblocks per SM is limited by the size of shared memory and register files that each SM can provide, as well as the request of threadblock. Our simulated GPU scheduling policy considers all these factors to decide $N_{threadblk\_batch}$.

At the threadblock level, we estimate its final performance by summing the latencies of three phases: (1) the initial phase T_{init}, in which the first chunk of data is requested and the pipeline waits for it to arrive; (2) the main loop T_{main_loop}, in which the load-and-use pipeline advances at a steady rate; (3) the epilogue phase $T_{epilogue}$, in which the final results are written back into the global memory. $T_{epilogue}$ is determined using the Epilogue Model equation proposed in DELTA [**lym2019delta**].

Let us consider T_{main_loop}. It illustrates load-and-use loop at the shared memory level, which comprises copying data from the device memory to the shared memory, reading the data into the register, and doing computations with tensor cores. We employ a Pipeline Latency Model, which is described in the next subsection, to calculate the latency of the loop. This model considers the pipelining and multiplexing factors, N_{pipe}, N_{mplx}, which means the number of stages the pipeline has, and

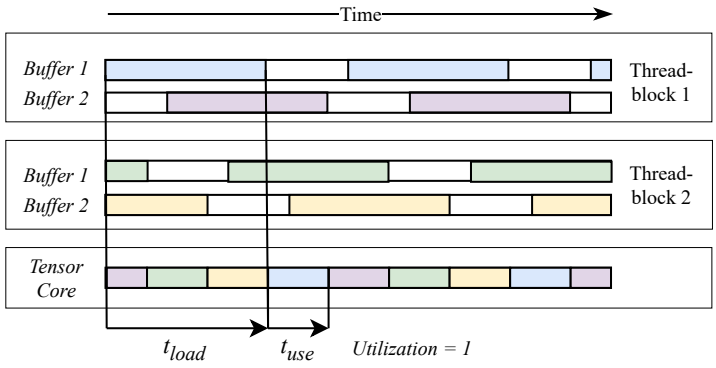
the number of parallel workers that can be multiplexed to hide the memory copy latency. At the shared memory level, these two parameters equal to the number of stages at the outer load-and-use loop, $N_{smem\_pipe\_stage}$, and the number of parallel threadblocks in an SM, $N_{threadblk\_per\_SM}$.

Calculation of T_{main_loop} still needs the latency of the use phase in this loop. However, the use phase is another pipeline that loads data into the register files and performs computations with tensor cores. We can calculate the latency of the use phase by estimating the stable state latency of the inner pipeline through inner-pipeline fusion. For this inner load-and-use loop, the use latency refers to the latency of performing arithmetic operations inside one loop on tensor cores. The pipeline and multiplex factors are determined by the number of stages of this inner load-and-use loop and the number of parallel warps in a threadblock.

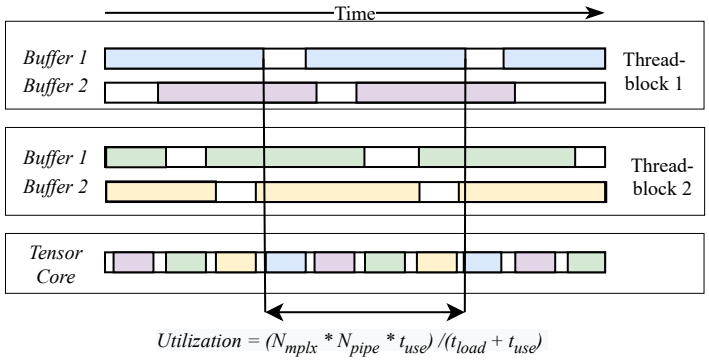
5.5.2 Obtaining Detailed Latencies

Pipeline Latency Model. Now we address the core issue of estimating the latency of a load-and-use loop in its stable state. Intuitively, the prediction should differ depending on whether the bottleneck is loading or using. Line 3 in the Pipeline Latency Model in Table 5.1 is the criterion for determining the bottleneck. Figure 5.10 illustrates the two scenarios in which computation or loading is the bottleneck. The intuition is that, during the loading of one data chunk, the computation units can be used to compute other chunks of data in this pipeline (N_{pipe}), or used for other parallel workers (N_{mplx}). If the latency of data loading exceeds the latency of all computations that can overlap with it, the loading becomes the bottleneck, making the loop latency equal to the latency of one load-and-use iteration, divided by the number of overlapping streams, i.e., $(T_{load} + T_{use})/N_{pipe}$.

Computation and Memory Latency Model. To obtain the computation latency, we



(a) *Case 1:* $t_{use} \cdot N_{mplx} \cdot N_{pipe} \geq (t_{load} + t_{use})$



(b) *Case 2:* $t_{use} \cdot N_{mplx} \cdot N_{pipe} < (t_{load} + t_{use})$

Figure 5.10: *Explanation of the pipeline latency model. A load can be overlapped by computing in other threadblocks, or in other stages of the same threadblock.*

can simply divide the number of float-point operations performed inside a loop by the tensor core throughput in an SM. When determining the latency of memory copies, four parameters must be considered: the amount of data transferred, the available bandwidth, the number of parallel workers (threadblocks or warps) to share with the bandwidth, and a constant round-trip latency LAT . Note that GPU LLC is shared by all SMs. Hence the DRAM traffic cannot be computed by the sum of data loaded by all threadblocks because the data may hit in LLC. We model DRAM traffic by deciding the working set of a threadblock-batch.

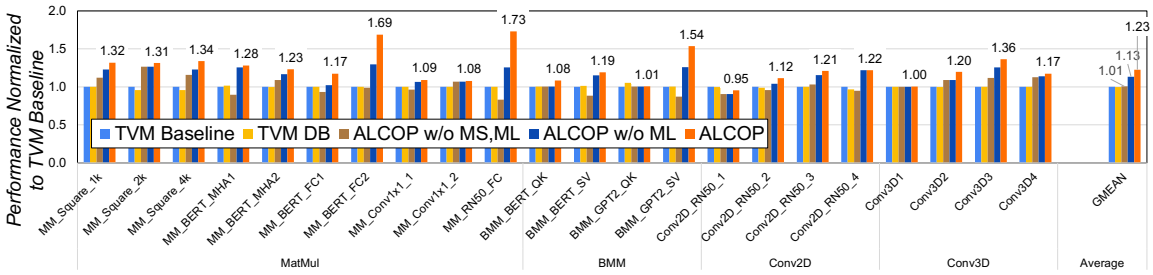


Figure 5.11: *Single operator performance normalized to TVM on A100.*

5.5.3 Model-Guided Auto-Tuning

Now, we will discuss how to use the analytical performance model for scheduled tuning. The workflow of auto-tuning is composed of a *cost model* to predict performance from schedule, and a *sampling method* to propose new *trials*. Unlike the analytical model we developed, TVM uses not an analytical model but a machine learning (ML)-based cost model that only learns from the profiled performance results. Analytical model and ML-based tuning offer complementary benefits: analytical model does not require the complexity of compiling and running sampled schedules but cannot be very accurate because it is difficult to capture hardware factors such as memory system thoroughly. ML-based tuning learns the cost model from measured performances that incorporate these complex factors, but it requires a large amount of sampled data, leading to a lengthy tuning process.

Finally, we leverage the analytical performance model's prediction to pre-train the ML-based model, allowing the ML model to acquire previous knowledge while still utilizing profiled data. Table 5.2 compares our method (Model-Assisted XGB) with other available auto-tuning approaches.

Table 5.2: *Comparison of compiler search methods.*

	Grid Search	**XGB**	**Anal. Only**	**Anal. + XGB (ours)**
Cost Model	N.A.	ML	Analytical	ML
Prior Knowledge?		No	Yes	Yes
Update Cost Model?		Yes	No	Yes
Sampling	Enumerate	Simulated Annealing	Cost-Model Ranking	Simulated Annealing

5.6 Evaluation

5.6.1 Single Operator Performance

This part evaluates pipelining speedup on single operators. Our benchmarks extracted from real DNN workloads contain four operators with a variety of shapes. All operators use half-precision and run on Tensor Cores. We run all experiments on NVIDIA Ampere GPU, as prior generations lack the asynchronous memory-copy hardware feature. Our evaluation platform is NVIDIA A100-SMX4 with 40GB device memory. The software we use is CUDA v11.4.

We implement our pipelining framework based on TVM [**chen2018tvm**] v0.8 and compare it against the vanilla TVM. We augment both ALCOP and baselines with shared memory swizzling to avoid bank conflict limitation. We also manually insert double-buffering primitives into TVM and use it as the second baseline (TVM DB). We also compare against two downgraded versions of our compiler for ablation study: ALCOP without multi-level (ML), meaning just pipelining in shared memory level, and ALCOP without ML and multi-stage (MS), meaning only allowing two-stage pipelining. Here we exhaustively search the schedule space and give the best schedule for ours and all baselines.

Figure 5.11 shows the performance of different compilers normalized to TVM.

Table 5.3: *Model speedup from pipelining*

Model	Speedup over TVM	Speedup over XLA
BERT	1.15	1.27
BERT-Large	1.18	1.16
GPT-2	1.15	1.34
ResNet-18	1.02	1.64
ResNet-50	1.06	1.02
VGG-16	1.10	1.01

Our compiler produces operators that are 0.95-1.73×, on average 1.23×, faster than TVM. Pipelining is especially effective for operators with small output shapes but long reduction axis. Take matrix-multiplication (MatMul) as an example, MM_RN50_FC, the operator that gives the largest speedup, has an output shape of 1024×64, and a reduction axis of 2048. Also, for Batched Matrix Multiplication (BMM), the operators with short reduction axis (e.g., BMM_BERT_QK) show much smaller speedup than those with long reduction axis (e.g., BMM_BERT_SV).

Insights about when pipelining works well. Problems with small output shapes (*e.g.,* MM_BERT_FC2, MM_RN50_FC) have limited spatial parallelism, so they benefit more from pipelining since pipelining uncovers extra parallelism. For problems with large output shapes (*e.g.,* MM_Conv1x1_1), or with small reduction dimensions (*e.g.,* BMM_GPT2_QK), pipelining provides limited benefit since the former already have abundant parallelism and the latter cannot amortize the latency of initial loading stages in the pipelining schedule.

Ablation study. Multi-level and multi-stage pipelining are both critical to final speedup. As shown in Figure 5.11, TVM DB does not bring obvious speedup over TVM. Without multi-level pipelining, ALCOP can only provide an average 1.13× speedup. Without multi-level and multi-stage pipelining, ALCOP can only give 1.01× speedup over TVM.

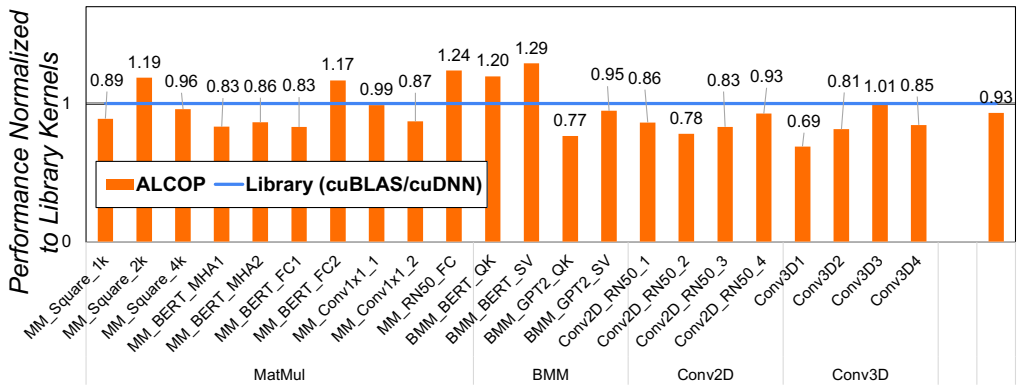


Figure 5.12: *Single operator performance versus libraries.*

5.6.2 End-to-End Performance

To evaluate end-to-end model acceleration, we compare against two baselines: TVM [**chen2018tvm**] and XLA [**xla**] (TF v2.9.1). XLA is a compiler integrated into the Tensorflow framework to optimize models in an end-to-end fashion. We evaluate six popular deep learning models. BERT, BERT-Large [**devlin2018bert**] and GPT-2 [**radford2019language**] are popular models in Natural Language Processing (NLP). ResNet-18, ResNet-50 [**he2016deep**] and VGG-16 [**simonyan2014very**] are three convolution neural networks widely used in vision tasks. Pipelining can be applied to MatMuls, BMMs and Conv2Ds, which are the most computation intensive operators and consumes a great proportion of the inference latency in these models. Table 5.3 shows the end-to-end speedup in real models. We achieve 1.02-1.18× end-to-end speedup over TVM and 1.01-1.64× speedup over XLA.

5.6.3 Comparison with Libraries

We compare with kernels in vendor libraries (cuBLAS [**cublas**]/cuDNN [**cudnn**]), which are heavily hand-optimized for the typical problem shapes we evaluate. Note

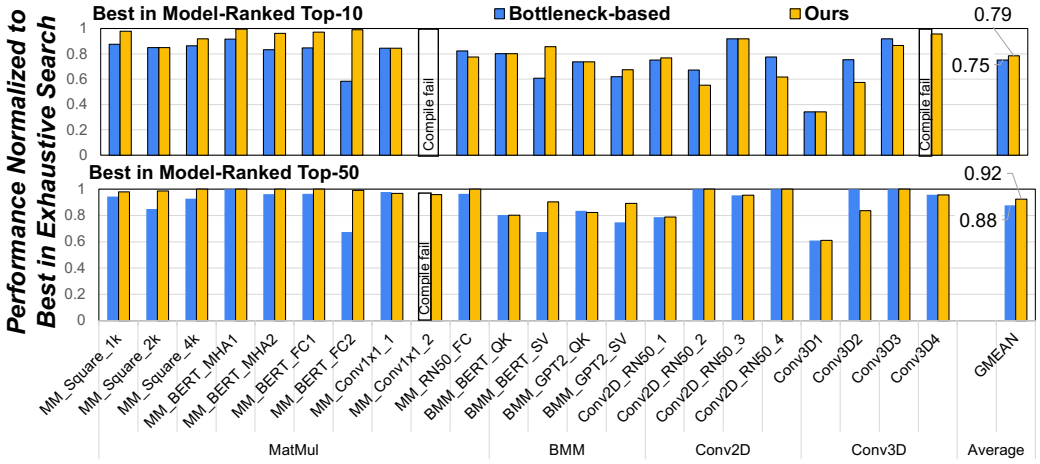


Figure 5.13: *Best-in-top-k performance of two analytical performance models. The mark 'compile fail' means the first 10 or 50 proposed schedules fail to compile into executables.*

that despite their high performance, libraries take huge manual efforts due to low modularity and cannot replace compilers in AI-GPU optimization. Figure 5.12 shows the performance of ALCOP normalized to library kernels. We can achieve on-par, on average 93% normalized, performance compared with library kernels. For some operators like BMM_BERT_QK, our compiler even generates faster kernels than cuBLAS because our compiler can search the entire schedule space and find the best schedule for input operators.

5.6.4 Performance Model Accuracy

The metric we use to evaluate our performance model is best performance in model-ranked top-k schedules, or *best-in-top-k* in short. It means the best performance within the top k schedules is predicted by the performance model. Compared to mean-absolute-error among the entire schedule space, best-in-top-k is more meaningful to schedule tuning because tuning cares about finding efficient schedules within a limited number of trials.

We compare against bottleneck-based analysis, a simple model that takes the

maximum of computation, shared memory loading and device memory loading time, assuming full utilization of computation throughput and bandwidth. It is over-simplified in the following ways: (1) assumes an aggregated computation unit, but in GPUs the Tensor Cores are distributed in different SMs and occupancy of SMs matters. (2) agnostic to the latency hiding effect, which is what pipelining mainly benefits.

Figure 5.13 shows the best-in-top-k results for our analytical model and bottleneck-based analysis for $k = 10, k = 50$. All results are normalized to exhaustive search, *i.e.*, the best performance in the entire schedule space. Within the top-10 trials, our performance model achieves an average of 79% performance compared to the best in exhaustive search, but the bottleneck-based method only achieves 75%. Within the top-50 trials, which is a 40× saving of trials compared to exhaustive search, our model achieves an average 92% performance, whereas the bottleneck-based method only achieves 88%. Our model also achieves more than 95% performance for all matrix-multiplication operators.

5.6.5 Analytical-Model-Guided Schedule Tuning

This part evaluates our technique to combine the analytical model with machine learning (ML) based schedule-tuning. The metric is *best-in-k-trials* similar as in the last part. We compare our method with the other three methods, as detailed in Table 5.2: (1) Grid-Search, which simply grid-search all the parameter configurations and does not learn anything from the collected performance data. (2) XGB, which is the default method in TVM [**Tavarageri2021**], and uses XGBoost [**chen2016xgboost**] as a cost model to fit the collected data and uses simulated annealing to propose new trials. (3) Analytical-only, which ranks all schedules according to their *predicted* performance via our analytical model (4) Analytical+XGB, which first pretrains the

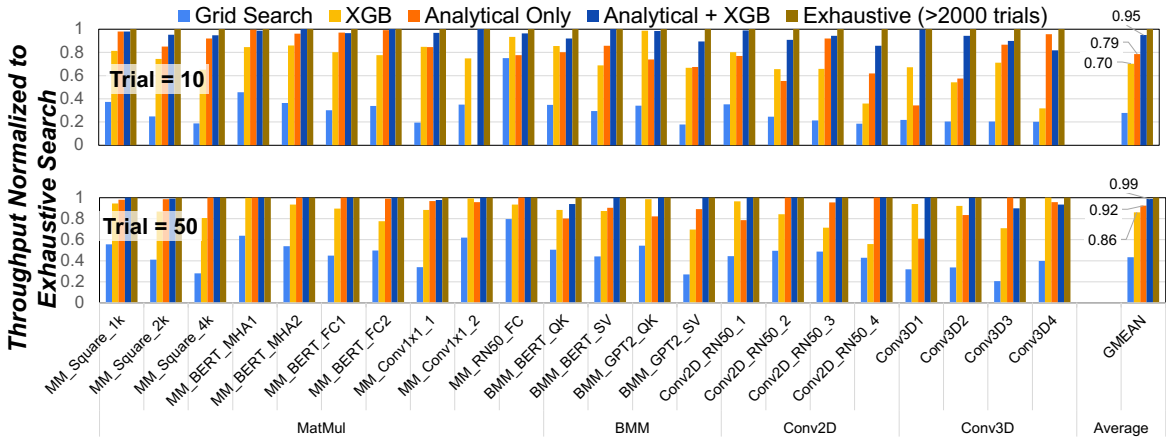


Figure 5.14: *Search efficiency of schedule tuning methods.*

XGB model offline with pairs of schedules and their predicted performance from the analytical model, and next follows the same workflow as XGB.

Figure 5.14 shows the *best-in-k-trials* of the four searching methods, normalized to the best performance in exhaustive search. At a budget of 10 trials, our Model-Assisted XGB finds schedules that reach 95% of the best performance in exhaustive search, while sampling purely based on an analytical model or non-pretrained XGB gives 79% and 70% of the best possible performance accordingly. At a budget of 50 trials, which achieves more than 40× saving of trials compared to an exhaustive search, our method reaches 99% of the best possible performance, while Model-Ranking and XGB only obtain 92% and 86%, respectively.

To sum up, we find that (1) analytical model helps ML: Model-Assisted XGB is better than XGB because it incorporates prior knowledge about the hardware, and (2) ML helps analytical model: Model-Assisted XGB is better than pure analytical model because it uses the actual profiled data to fine-tune the performance model.

5.7 Summary

This chapter addresses the important need for *automatic pipelining* in deep learning compilers. Due to the large tiling size required to mitigate bandwidth constraints, inter-tile parallelism is inadequate for achieving high utilization, and intra-tile pipelining becomes essential. We propose the first compiler solution that supports *multi-stage, multi-level* pipelining. Through introducing automatic pipelining, our compiler can generate GPU programs with an average 1.23× and maximally 1.73× speedup over vanilla TVM [**chen2018tvm**]. Additionally, we develop an analytical performance model which significantly improves the search efficiency of the schedule tuning process.

Chapter 6

BACO: Co-exploration of Hardware Acceleration and Tensor Programs with Unified Compilation Interface

6.1 Motivation

In recent years, there has been a notable increase in the utilization of domain-specific accelerators [**dally2020domain**] to enhance computation performance and improve energy efficiency in modern machine learning systems. Among the diverse array of hardware accelerators available, spatial architecture has emerged as a particularly efficient option. This architecture organizes numerous processing elements in a structural and hierarchical manner, delivering exceptional efficiency across various applications, including deep learning [**he2016deep**, **ren2015faster**] and scientific computing [**manavski2008cuda**]. Developing efficient accelerators with high-performance kernel libraries in a cost-effective manner has emerged as a powerful technique.

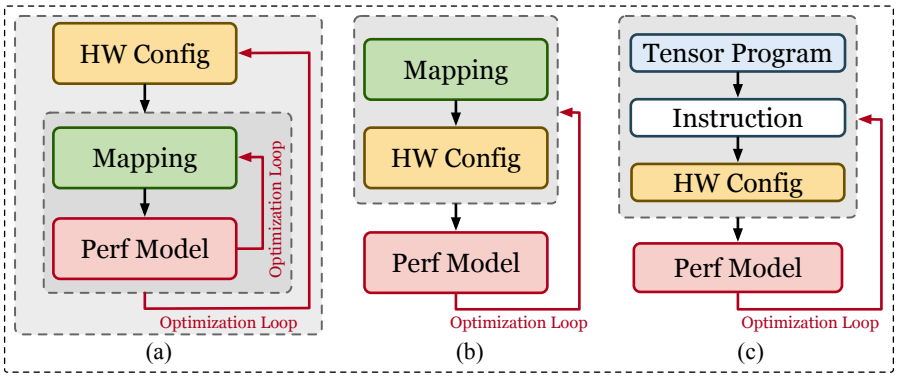


Figure 6.1: *(a) Hardware-first exploration with two-loop search; (b) Mapping-first exploration with one-loop search; (c) Instruction-aware exploration with one-loop search.*

The development process entails exploring two distinct design spaces: the hardware space and the software space. The hardware space encompasses accelerator design parameters, such as interconnect topology, buffer capacity, and spatial array sizes. On the other hand, the software space involves determining how workloads are executed on the target accelerator, including decisions related to computation instructions and data movement orchestration. In both spaces, the objective is to optimize a performance metric, such as the energy-delay product (EDP). The developed kernels can then be executed on the selected hardware configuration, assuming that the memory buffers are adequately sized to accommodate the data movement optimized by the kernel libraries. Given that these constraints encompass both hardware and software design aspects, it is essential to optimize these two spaces simultaneously.

Both the hardware and software design spaces are characterized by high dimensionality and consist of categorical and discrete variables. Additionally, evaluating the performance of each design point can be computationally expensive. The size of the combined optimization space, along with the cost of evaluating each point, presents significant challenges. As depicted in Figure 6.1, prior re-

search [**sakhuja2023leveraging**, **huang2022learning**, **zhang2022full**, **lin2021naas**, **venkatesan2019magnet**] has addressed this challenge through a *two-loop search* approach. These methods directly explore the design space of potential hardware configurations. The performance of each hardware configuration is determined by constraining the programs compatible with that particular hardware configuration. Program optimization is performed iteratively, resulting in a two-loop search that iterates over both the hardware and software design spaces. Consequently, they must grapple with a combinatorial explosion of possible configurations.

Despite the advantages of the *one-loop search* [**kao2022digamma**, **hong2023dosa**, **yang2020interstellar**], it primarily optimizes the program without considering hardware-specific instructions or assembly-level ISA. We refer to this approach as *instruction-agnostic* co-exploration. While program transformations with these instructions play an increasingly critical role, there is a lack of programming language or compiler support for users to write them. As a result, the state-of-the-art kernels are still predominantly manually written in low-level C or assembly with metaprogramming techniques. This limitation restricts the range of accelerated routines and creates barriers to exploring new or improved accelerators. Consequently, the performance achieved in the software space is sub-optimal, thereby limiting the overall performance of the co-exploration process in the *one-loop search* approach.

Our Goal: We aim to establish an instruction-aware co-exploration framework that employs a high-level and flexible interface. This involves taking a set of kernels, written in a domain-specific language (DSL) and annotated to specify the desired functionality, as input. These kernels serve as a representation of generality. The desired outcome is to obtain optimized hardware configurations that are specifically tailored to the characteristics of the input kernels, along with accelerated programs. Ideally, a compiler should be capable of targeting the given design. Ultimately, the

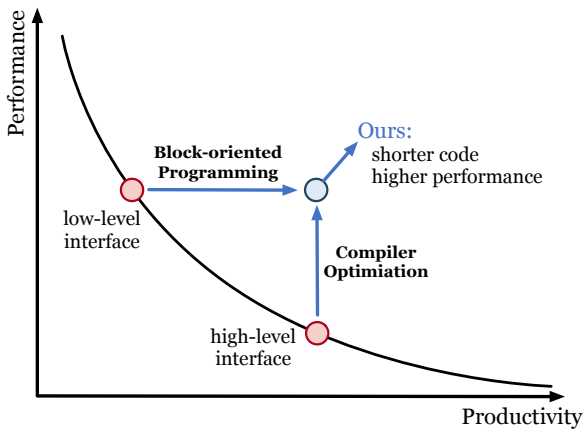


Figure 6.2: *The goal of BACO is to achieve both the productivity of a high-level programming paradigm and the high-performance of manually optimized design.*

framework aims to achieve both high-performance and ease of programming for co-exploration, as illustrated in Figure 6.2.

Challenges: Designing a unified interface for hardware and software co-exploration presents several challenges. Firstly, it requires the design of a new programming paradigm and the determination of a suitable granularity for accelerator programming. This paradigm should effectively shield users from the intricate hardware-specific instructions and details, providing a user-friendly experience. Additionally, for a given target kernel, an automatic mechanism needs to be devised to construct an appropriate design space for co-exploration. Secondly, it is crucial to develop efficient search methods that can effectively navigate and explore the design space. These methods should be capable of minimizing the computational burden associated with evaluating numerous design points. Lastly, when targeting multi-layer DNNs, it is imperative to identify and prioritize the layers that have a significant impact on the overall end-to-end performance. This prioritization allows for more focused exploration and optimization efforts within the co-exploration framework.

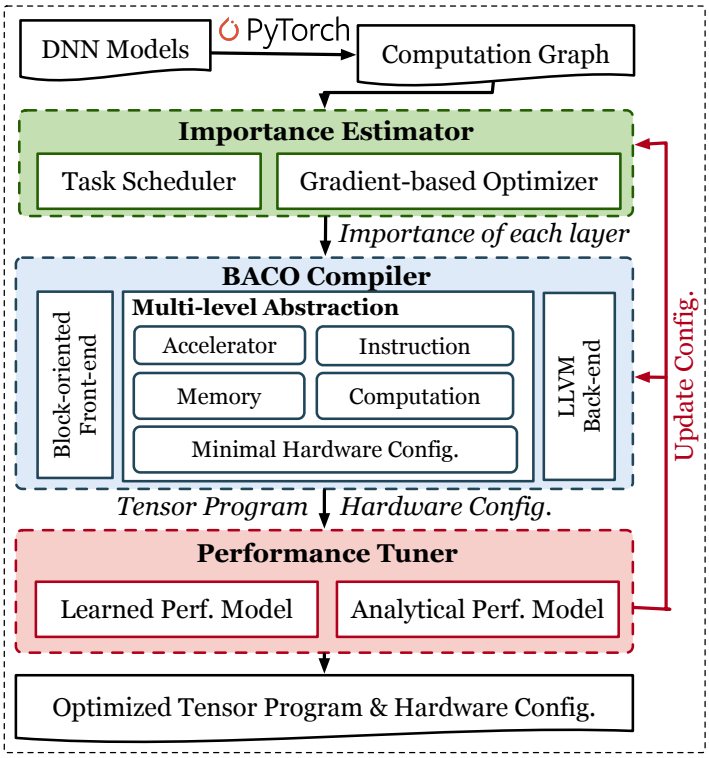


Figure 6.3: *Overall design of BACO. The black arrows show the flow of extracting layers with different importance scores from DNNs and generate optimized hardware and software configurations. The red arrows means the performance tuning updates the status of all components in the framework.*

6.2 Overview of BACO

BACO is an end-to-end one-loop search framework which captures key relations between hardware and software in a unified compilation interface. Figure 6.3 shows the design of BACO. It has three major components: a compiler, a performance tuner and an importance estimator.

BACO Compiler. One of the primary challenges that BACO must overcome is to establish a unified interface that facilitates hardware-specific code generation and policies. To tackle this challenge, BACO introduces a DSL that revolves around the notion of a block. By the incorporation of block-level optimization passes, it seamlessly integrates hardware and software features, thereby enabling a unified

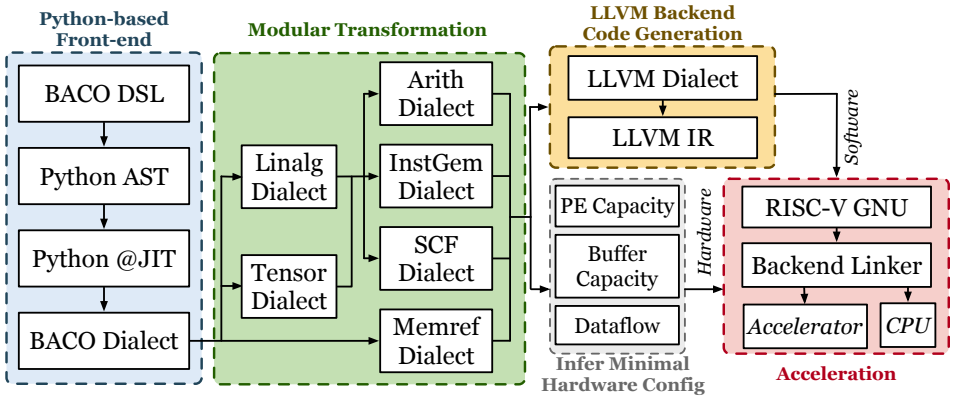


Figure 6.4: *The compilation flow with multi-level IR in BACO.*

interface. Meanwhile, a small accelerator-centric schedule space is constructed to infer the minimal hardware configurations.

Performance Tuner. The performance of compiled programs and inferred hardware configurations may not always meet desired expectations. The subsequent challenge lies in the fine-tuning process. In this regard, BACO adopts a novel approach by formulating the latency and energy performance model as a differentiable white-box model. This formulation enables the utilization of gradient descent to simultaneously update design parameters. By ensuring that the objectives are differentiable with respect to these parameters, BACO achieves high sampling efficiency.

Importance Estimator. Considering the whole DNN as a single computation graph has the potential to achieve optimal performance. However, it becomes inefficient due to the exponential expansion of the design space. To overcome this limitation, BACO adopts a strategy of dividing the computation graph into a collection of independent layers. Additionally, it incorporates an importance estimation strategy that allows for the prioritization of layers based on their significant impact on the end-to-end performance.

6.3 BACO Compiler

6.3.1 Compilation Flow

As depicted in Figure 6.4, the workflow begins by parsing the program into a Python abstract syntax tree (AST), drawing inspiration from previous Python-based DSLs [**hu2019taichi**, **tillet2019triton**]. The AST then undergoes type checking, inference, and conversions through a type system that utilizes user-specific annotations. Python AST functions that are selected for offloading to accelerators are transformed into the MLIR dialect representation. This process gives rise to the BACO dialect, which serves as the entry point into the MLIR dialect ecosystem. The BACO dialect functions as a comprehensive solution for program transformations, assuming the responsibility of determining the program regions to be offloaded onto hardware-specific components. Subsequently, the BACO dialect is primarily lowered to the MLIR Linalg and Tensor dialects for block values and the Memref dialects for memory pointers. Hardware-specific instructions are further lowered to the InstGem dialect. Finally, all the dialects, comprising a multi-level abstraction, are compiled into binary code using the LLVM backend. The compiler employs a hierarchy of abstractions that progressively lowers programs, facilitating program transformations and hardware optimization at the most appropriate abstraction level.

6.3.2 Problem Setup

The problem dimensions during computation are described using the following notations. Let $D = \{R, S, P, Q, C, K, N\}$ represent the optimization dimensions of a target workload. For instance, a convolution operator can be represented as a

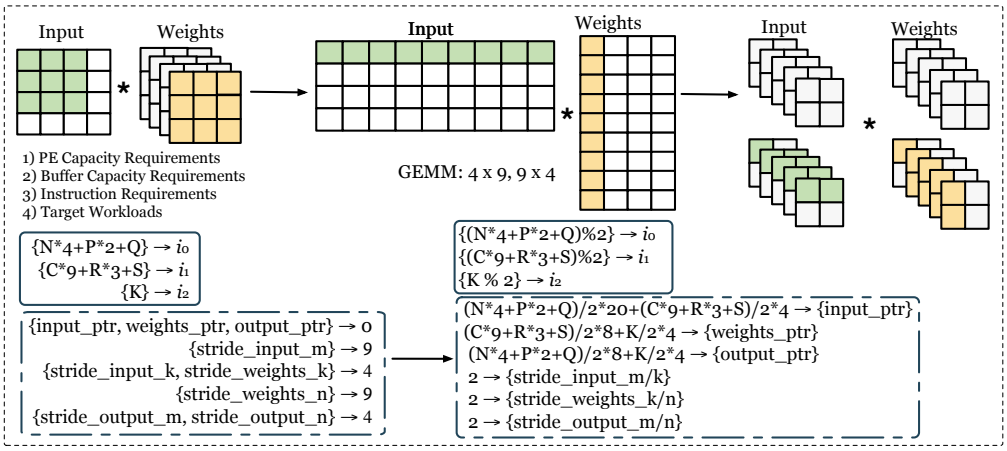


Figure 6.5: *Tensor to hardware matching with computation abstraction.*

7-level nested loop optimization over the height and width of the weight tensor (R, S), the output tensor (P, Q), the number of input and output channels (C, K), and the batch size N. It is worth noting that matrix-matrix multiplication (GEMM) can be represented as convolutions by setting R, S, P, Q to 1, and matrix-vector multiplication (GEMV) can be represented by setting R, S, P, Q, N to 1. We can define subsets of D as follows:

- $D_w = \{C, K, R, S\}$, which consists of the problem dimensions used to calculate the size for weights.
- $D_i = \{C, N, P + R - 1, Q + S - 1\}$, which consists of the problem dimensions used to calculate the size for inputs.
- $D_o = \{K, N, P, Q\}$, which consists of the problem dimensions used to calculate the size for outputs.

Optimization Objective. Given a target workload D, our objective is to explore a spatial accelerator s with an optimized hardware configuration, consisting of parameters such as PEs capacity N_{PE}, number of memory levels i, and buffer

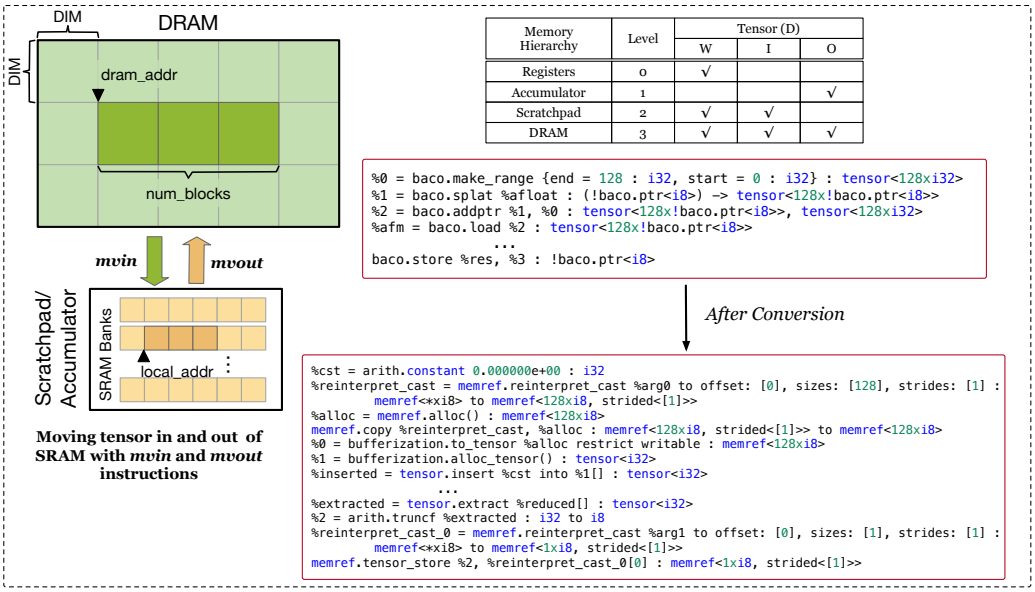


Figure 6.6: *Structured memory access patterns with memory abstraction.*

capacity of each memory C_i, as well as optimized program transformations f. Meanwhile, we aim to identify the optimal program transformations f^* and accelerator configuration s^*. It can be formulated as an optimization problem:

$$f^*, s^* = \underset{f \in F_f}{\operatorname{argmin}}\, Cost(f, s). \tag{6.1}$$

Our goal is to enable the maximum possible freedom to include different programming idioms to define specialized memories, custom instructions and configuration state for users. A general high-level programming interface serves this purpose better. Therefore, BACO provides a flexible interface which brings block as the first-class citizen to directly describe target workloads in Python.

6.3.3 Block-oriented Programming Model

Figure 6.7 exemplifies the implementation of the complete GEMM kernel using the BACO DSL, which necessitates less than 40 lines of Python code. This serves to the

```
1 @baco.jit
2 def matmul_kernel(
3     # Pointers to matrices and dimensions
4     a_ptr, b_ptr, c_ptr, M, N, K,
5     stride_am, stride_ak, # increase a_ptr by each dimension
6     stride_bk, stride_bn,
7     stride_cm, stride_cn,
8     # Meta-parameters
9     BLOCK_SIZE_M: baco.const, BLOCK_SIZE_N: baco.const, BLOCK_SIZE_K: baco.const):
10    # Map program ids `pid` to the block of C it should compute.
11    pid = baco.program_id(0)
12    num_pid_m = baco.cdiv(M, BLOCK_SIZE_M)
13    num_pid_n = baco.cdiv(N, BLOCK_SIZE_N)
14    pid_m = pid / num_pid_m
15    pid_n = pid % num_pid_n
16    # a_ptrs is a block of [BLOCK_SIZE_M, BLOCK_SIZE_K] pointers
17    offs_am = pid_m * BLOCK_SIZE_M + baco.arange(0, BLOCK_SIZE_M)
18    offs_bn = pid_n * BLOCK_SIZE_N + baco.arange(0, BLOCK_SIZE_N)
19    offs_k = baco.arange(0, BLOCK_SIZE_K)
20    a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] * stride_ak)
21    b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] * stride_bn)
22    # Iterate to compute a block of the C matrix
23    accumulator = baco.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype="int32")
24    for k in range(0, K, BLOCK_SIZE_K):
25        a = baco.load(a_ptrs)
26        b = baco.load(b_ptrs)
27        # We accumulate along the K dimension
28        accumulator += baco.dot(a, b)
29        # Advance the ptrs to the next K block
30        a_ptrs += BLOCK_SIZE_K * stride_ak
31        b_ptrs += BLOCK_SIZE_K * stride_bk
32    c = accumulator.to("int32")
33    # Write back the block of the output matrix C
34    offs_cm = pid_m * BLOCK_SIZE_M + baco.arange(0, BLOCK_SIZE_M)
35    offs_cn = pid_n * BLOCK_SIZE_N + baco.arange(0, BLOCK_SIZE_N)
36    c_ptrs = c_ptr + stride_cm * offs_cm[:, None] + stride_cn * offs_cn[None, :]
37    baco.store(c_ptrs)
```

Figure 6.7: *The kernel implementation only less than 40 LoC in Python.*

productivity benefits provided by the block-centric design approach in , especially when contrasted with the complexities associated with low-level assembly programming showcased in Figure 2.8. The GEMM kernel is successfully accomplished through the utilization of block-level matrix multiplication and multi-dimensional pointer arithmetic, effectively elevating the block to a prominent position in the programming paradigm. The programming primitives related to this concept are clearly illustrated in Table 6.1. The block-level matrix multiplication is implemented as follows (L24-31), where each iteration of the nested for-loop is executed by a dedicated program instance. The main challenge lies in computing the memory locations from which blocks of $\boldsymbol{a}$ and $\boldsymbol{b}$ must be read in the inner loop. To address this challenge, we utilize multi-dimensional pointer arithmetic and define blocks of pointers for $\boldsymbol{a}$ and $\boldsymbol{b}$ (L20-21). The initialization of the pointers for the blocks of $\boldsymbol{a}$ and $\boldsymbol{b}$ can be performed (L17-21). It is important to note that an additional modulo operation is necessary to handle cases where M is not evenly divisible

Table 6.1: *Instructions and programming primitives in BACO.*

Primitive	Description	Category
mvin($rs1, rs2$)	Scratchpad[rs2] $\Leftarrow$ DRAM[rs1]	Custom Hardware ISA
mvout($rs1, rs2$)	DRAM[rs1] $\Leftarrow$ Scratchpad[rs2]	
flush($rs1, rs2$)	execute without waiting for other instructions	
config_ld($rs1, rs2$)	stride $\Leftarrow$ rs2, scale $\Leftarrow$ rs1	
config_st($rs1, rs2$)	stride $\Leftarrow$ rs1	
config_ex($rs1, rs2$)	mode $\Leftarrow$ rs1, shift $\Leftarrow$ rs2	
fence()	Time and spatial synchronization between PEs.	
preload($rs1, rs2$) compute($rs1, rs2$)	Scrathpad[rs2] $\Leftarrow$ Scrathpad[rs3] * Scrathpad[rs4]+ Scrathpad[rs1]	
@jit	Decorator for JIT-compiling a function using$\sim${\name} compiler.	Domain-Specific Language
autotune(configs)	Automatic block partition by the specific configuration.	
program_id	The index of current program instance.	
make_block_ptr	Returns a pointer to a block.	

by *BLOCK_SIZE_M* or *N* is not evenly divisible by *BLOCK_SIZE_N*. In such scenarios, the data can be padded with values that have no impact on the final results. The pointers are then adjusted within the inner for-loops (L30-31). This exemplifies how multi-dimensional pointer arithmetic is employed to abstract the movement of data at the block level, enabling efficient iteration through the data blocks.

6.3.4 Multi-level Abstraction in BACO

Based on the principles of block-oriented programming, our objective is to establish a multi-level abstraction that enables the formal specification of specialized memories, custom instructions, and configuration state. This formal definition allows for the precise delineation of the behavior of the accelerator, enabling automatic analysis and optimization. The core idea behind this multi-level abstraction is to convert low-level accelerator instructions into higher-level representations that are based on scalar operations, which can then be organized by the block. To accomplish

this, BACO introduces four key abstractions: accelerator, computation, memory, and instruction. These abstractions work together harmoniously to facilitate the transformation process. To illustrate the effectiveness of these abstractions, we present a motivating example and provide a more comprehensive description of each abstraction.

A Motivating Example. We utilize a convolution example depicted in Figure 6.5 to provide a comprehensive description of our compilation flow. This flow is designed to generate optimized programs with specialized instructions tailored to specific accelerators. It is important to note that the proposed method is applicable to other target workloads as well. The convolution is expressed as:

$$D = \{R = 3, S = 3, P = 2, Q = 2, C = 1, K = 4, N = 1\}.$$

It is transformed into an equivalent GEMM operation with dimensions 4×9 and 9×4. Assuming there are no constraints regarding instructions or the accelerator, as long as a $4 \times 9 \times 4$ GEMM operation is executed, it would be sufficient to complete the convolution computation. The base addresses for the computation are initialized to 0, and the strides are determined based on the matrix shape illustrated in Figure 6.5. However, the computation capacity of the PEs is limited to generating results of a fixed shape at any given time. This limitation is determined by the configuration of specialized instructions and the available resources of the accelerator. The resources of a custom instruction in the accelerator can be derived from its index range, which is defined in the computation abstraction.

By taking into account the computation partition factors, which are determined by the available resources of the accelerator (such as the shape of the PEs), we can construct an accelerator-centric schedule space by enumerating possible values within the constraints of the hardware resources. This approach results in a sig-

nificantly smaller schedule space compared to traditional methods [**chen2018tvm**, **zheng2020ansor**]. In this example, the number of iterations is constrained by a factor of 2 due to the 2×2 shape of the processing elements. Furthermore, the memory capacity poses a limitation, with each SRAM only able to accommodate a certain size of tensor. As a result, it is necessary to partition the entire tensor into smaller blocks and perform multiple load/store operations on these blocks. The movement of these blocks within the memory is controlled by the base addresses and strides, which must be adjusted accordingly to facilitate proper tensor movement.

We flatten the base addresses and obtain the expression $\frac{(n\times 4+p\times 2+q)}{2} \times 20 + \frac{(c\times 9+r\times 3+s)}{2} \times 4$. Here, $4 = 2 \times 2$ represents the number of elements within a matrix, and $20 = 5 \times 4$ indicates that 5 sub-matrices are grouped together for the computation. The 4×9 matrix is split into 2×5 with 2×2 sub-matrices. Consequently, there exist trailing sub-matrices during the computation, and BACO automatically pads them with the value 0. When the number of PEs changes, the addressing and computation described above will also change accordingly. Hence, by combining the available resources, instruction configurations, and target workloads within the accelerator-centric schedule space, we can deduce the minimum hardware requirements.

Accelerator Abstraction. We specifically target the open-source RISC-V based DNN accelerator Gemmini [**genc2021gemmini**], which is composed of the following architecture templates: *i)* a systolic array of PEs, *ii)* an accumulator, *iii)* a scratchpad, and *iv)* a DRAM memory. In the context of Gemmini, the memory buffer levels are designated as 0, 1, 2, and 3 in Figure 6.6, where 0 corresponds to the per-PE registers. The hardware instructions are typically applied in conjunction with special memory scopes, layouts, and the corresponding load/store operations. For our purposes, we adhere to Gemmini's assembly-level instruction set, as presented in Table 6.1. This

instruction set consists of custom RISC-V instructions that allow for the specification of the computation and memory configuration. Notably, Gemmini's instruction set includes low-level instructions for moving strided matrices to and from the scratchpad, as well as instructions for calculating dot products. These accelerators are primarily designed to efficiently execute small and dense GEMM instructions. Consequently, a target accelerator is modeled based on the notions of the PE capacity N_{PE} and the buffer capacity C_i at different levels i. As observed in Table 6.1, the accelerator is equipped with instructions for memory access and computation. Therefore, when designing the multi-level abstraction, it is essential to decouple these instructions, as they control distinct hardware units within the accelerator.

Computation Abstraction. The computation abstraction refers to a statement that encompasses the operands, the arithmetic operation performed on those operands, and the indices used to access the tensor for the computation instruction in the accelerator. It is crucial to represent the range of indices within the abstraction, ensuring a comprehensive and complete specification of the computation. As depicted in Figure 6.6, D_W and D_I denote the input tensors, while D_O represents the output tensor. The function F denotes the arithmetic operation, such as a dot-product operation, and m, n, and k are indices used to access the tensors. Therefore, we can express the computation abstraction as $D_O[m,n] = F(D_W[m,k], D_I[k,n])$.

Memory Abstraction. The memory abstraction is presented as a list of statements, with each statement providing information about the scope, operands involved, and indices used for memory access. In Figure 6.6, detailed information is provided regarding the lowering programs that exhibit structured memory access patterns in the compilation process. Two crucial analyses are involved in this context. The first analysis is the pointer analysis, which plays a vital role in extracting structured memory access patterns from the BACO program during load and

store operations. This analysis traverses the intermediate representation (IR) and examines relevant instructions to construct strided memory accesses using the Memref dialect. Once the pointer analysis is performed, instructions involved in memory address calculations become redundant in the BACO program, as their semantics are effectively captured by the operations in the Memref dialect, representing the strided memory accesses. To safely remove these instructions, we employ an analysis that identifies and marks instructions used solely for address calculation purposes. The computation and memory abstractions in the design effectively represent the behavior of the accelerator and decompose the obscure specialized instructions into a sequence of comparable scalar operations. These operations are then structured into blocks, allowing for comprehensive analysis.

Instruction to Hardware Matching. First, we introduce the concept of tensor iterations. Tensor program transformation can be represented as optimal for-loop nests. Therefore, tensor iterations refer to all the instances within the for-loop nests. Meanwhile, we can define instruction iterations based on the computation abstraction discussed in Section 6.3.4. Instruction iterations correspond to scalar operations within the index range of the computation abstraction. For example, in systolic array computation, there exist three instruction iterations, namely (i_0, i_1, i_2), as illustrated in Figure 6.5. Given the definitions of tensor and instruction iterations, as well as computation and memory abstraction, we define the matching between tensors and instructions, and instructions and hardware.

$$Tensor[f] \rightarrow Instruction[i_0, i_1, i_2, \ldots] \tag{6.2}$$

$$Tensor[f] \rightarrow Operand[ptr_0, ptr_1, \ldots] \tag{6.3}$$

$$\{f\} \cup \{i_0, i_1, i_2, \ldots\} \cup \{ptr_0, ptr_1, \ldots\} \rightarrow HW\,Config. \tag{6.4}$$

The tensor programs are partitioned into scalar operations, leveraging computation

Table 6.2: *Analytical Performance Model in BACO.*

Category	Modeling
Read	$Read(l)$: amount of tensors read from $l-$level buffer. $BandwidthRead(l)$: bandwidth of reading tensors from $l-$level buffer. $Latency_r(l)$: latency of reading tensors from $l-$level buffer. $Latency_r(l) = Read(l)\ /\ BandwidthRead(l)$
Write	$Write(l)$: amount of tensors written to $l-$level buffer. $BandwidthWrite(l)$: bandwidth of writing tensor to $l-$level buffer. $Latency_w(l)$: latency of writing tensor to $l-$level buffer. $Latency_w(l) = Write(l)\ /\ BandwidthWrite(l)$
Latency	$Latency_c(l)$: latency of computation on $l-$level. $Seq(l)$: sequential loops for $l-$level buffer. $Latency(l) = \begin{cases} \max\limits_{i\in\{c,r,w\}} [Latency_i(l-1)] \times \prod Seq(l) & l > 0 \\ LatencyInstruction \times \prod Seq(l) & l = 0 \end{cases}$
Energy	$Energy = MACs \times EPA_{PE} + \sum_{i=0}^{3} [Access(i) \times EPA(i)]$.

and memory abstractions. This allows the BACO compiler to establish a correspondence between tensor and instruction iterations. Moreover, explicit control is exerted over the memory access indices for each operand across various memory scopes. By combining these factors with tensor program transformations f, the compiler is able to automatically generate matching between tensors and instructions, encompassing both computation and memory access. Ultimately, this enables the compiler to infer the optimal hardware configuration based on these matchings.

To address the obstacles posed by fragmentation, the BACO compiler introduces a modular and reusable abstraction that acts as a mediator between the hardware and software layers. This approach allows for the encapsulation of various operations and types within specific abstractions, while also facilitating the sharing of common optimizations across both hardware and software components.

```
module {
  llvm.func matmul_kernel(
    "InstGemdialect.intr.config_ex"
    (%64, %65) : (i64, i64) -> ()
    "InstGemdialect.intr.config_st"
    (%67, %68) : (i64, i64) -> ()
    "InstGemdialect.intr.config_ld"
    (%70, %69) : (i64, i64) -> ()
    "InstGemdialect.intr.loop_ws_config_bounds"
    (%79, %80) : (i64, i64) -> ()
    "InstGemdialect.intr.loop_ws_config_addrs_ab"
    (%57, %59) : (i64, i64) -> ()
    "InstGemdialect.intr.loop_ws_config_addrs_dc"
    (%63, %61) : (i64, i64) -> ()
    "InstGemdialect.intr.loop_ws"
    (%85, %86) : (i64, i64) -> ()
    "InstGemdialect.intr.flush"
    (%87, %87) : (i64, i64) -> ()
  llvm.return } }
```

Figure 6.8: *InstGem Dialect with hardware-specific instructions.*

6.3.5 Backend Code Generation

To facilitate the implementation of the multi-level abstraction, we introduce a conversion pass called `baco-to-mlir`. This pass is responsible for reducing the operations that are compatible with the MLIR core dialect.

Lowering to specialized instructions: When utilizing DSL, which incorporates a high-level block abstraction, the user-defined target workload and a set of optimizations revolving around the blocks are transformed through modular transformations. This process results in the lowering of the workload to the InstGem dialect. The InstGem dialect is responsible for modeling and configuring accelerator-specific instructions. More details can be found in Figure 6.8.

Lowering to LLVM IR: We present a LLVM extension that transforms all accelerator-specific RISC-V instructions into LLVM IR [**lattner2004llvm**]. Further details can be found in Figure 6.9. This conversion enables us to leverage the optimization capabilities offered by the LLVM backend, thereby augmenting the final code. Si-

```
1  define void matmul_kernel(
2    ptr %0, ptr %1, ..., i64 %19, i64 %20) {
3    call void @llvm.riscv.config.ex(i64 , i64)
4    call void @llvm.riscv.config.st(i64 , i64)
5    call void @llvm.riscv.config.ld(i64, i64)
6    ...
7    call void @llvm.riscv.loop.ws.config.bounds(i64 , i64)
8    call void @llvm.riscv.loop.ws.config.addrs.ab(i64, i64)
9    call void @llvm.riscv.loop.ws.config.addrs.dc(i64, i64)
10   call void @llvm.riscv.loop.ws.config.strides.ab(i64, i64)
11   call void @llvm.riscv.loop.ws.config.strides.dc(i64, i64)
12   call void @llvm.riscv.loop.ws(i64, i64)
13   call void @llvm.riscv.flush(i64, i64)
14   ...
15   ret void }
```

Figure 6.9: *LLVM IR Extension with hardware-specific instructions.*

multaneously, we develop a set of tools to facilitate the optimization and analysis process. The `baco-opt` tool functions as a modular optimizer and analyzer. It accepts source files as input, applies specified optimizations or analyses to them, and produces optimized files as output. The `baco-translate` tool is responsible for converting the source files generated from the MLIR dialect into the LLVM IR dialect. This conversion is necessary for importing the files into the LLVM backend. The `baco-llc` tool generates the object file needed for simulation purposes.

6.4 Performance Fine-tuning

The design of domain-specific accelerators is a complex and resource-intensive task, involving the exploration of various hardware and software configurations to optimize the performance of specific workloads. In the case of BACO, our focus is on co-design, which involves minimizing the energy-delay product (EDP) for a given DNN model by considering both energy consumption and latency at a layer-wise level. The overall objective of BACO is to compute the product of the sums of energy and latency across all layers. The performance models for latency

Table 6.3: *Details modeling information of the accelerator.*

Component	Memory Level	Bandwidth (words/cycle)	Energy (EPA)
PE	-	-	0.561
Registers	0	$2N_{PE}$	0.487
Accumulator	1	$2\sqrt{N_{PE}}$	$1.94 + 0.1005 \times C_1/\sqrt{N_{PE}}$
Scratchpad	2	$2\sqrt{N_{PE}}$	$0.49 + 0.025 \times C_2$
DRAM	3	8	100

and energy are presented in Table 6.2. By treating the total EDP value as the loss term in a gradient descent optimization process, our aim is to minimize EDP for the entire DNN model, rather than optimizing individual layers in isolation. Thus, the performance model can be defined as follows:

$$EDP = Energy \times Latency = \left(\sum_{i=1}^{n} e_i\right) \times \left(\sum_{i=1}^{n} l_i\right), \tag{6.5}$$

where n represents the number of layers, and e_i and l_i denote the energy and latency of the i-th layer, respectively. Thanks to the scalability of gradient descent, we can optimize Equation (6.5) concurrently with respect to all program transformations, rather than tackling one program transformation at a time. Based on the accelerator abstraction defined in Section 6.3.4 and the notations introduced in Section 6.3.2, the performance model is also formulated on a level-by-level basis. The overall latency is determined by the maximum value among the read latency, write latency, and compute latency. The compute latency can be either the latency of inner-level computation or the latency of a single instruction. The single-instruction latency can be estimated using simulators. Additionally, the final latency is multiplied by the trip counts of sequential loops, as these loops are executed sequentially and are not bound to parallel cores. Further details on the performance model can be found in Table 6.2. The specific energy per access (EPA) values for each architectural component are provided in Table 6.3. Note that the differentiable performance

Algorithm 6 Unified Compilation Interface for Hardware and Software Co-Exploration in BACO.

Require: $\mathcal{K}$: Set of target workload kernels for multi-layers
Ensure: Optimized tensor programs and hardware configuration
1: Initialize tensor program $\mathcal{T}$ with hardware-specific instructions using BACO compiler
2: Initialize hardware config $\mathcal{H}$ with default settings
3: Assign equal importance weight w_i to each layer i
4: **repeat**
5: Compute HW requirements $\mathcal{R}$ for layerwise transformation
6: Derive minimal hardware config $\mathcal{H}_{min}$ from $\mathcal{R}$
7: Calculate arithmetic operations AO and memory accesses MA at each buffer level using analytical performance model
8: Predict EDP using AO, MA, and $\mathcal{H}_{min}$
9: Optimize $\mathcal{T}$ and $\mathcal{H}$ via gradient descent, update weights w_i
10: **until** Convergence criteria met

model of BACO is inspired by the existing design presented in [**hong2023dosa**].

6.5 Importance Estimation Strategy

As discussed in Section 6.4, BACO is capable of partitioning a DNN model into multiple independent layers. However, for certain layers, investing resources in exploring them does not yield significant improvements in the end-to-end EDP metric. There are two main reasons for this: *i)* the layer is not a performance bottleneck, and *ii)* the exploration of the layer only leads to marginal enhancements. To address this, BACO leverages the flexibility of the gradient descent loss function in Equation (6.5) to dynamically assign different weights to different layers, thereby avoiding the costly exploration of unimportant layers. Furthermore, for layers that appear multiple times, knowledge can be shared and reused. The task scheduler of BACO iteratively assigns different weights to each layer, optimizing the overall performance.

Problem Formulation. Our objective is to decrease the EDP metric for a given

DNN model by minimizing the exploration cost when further optimization does not significantly improve performance. Let us assume that there are n layers in the model. We introduce a variable $t \in \mathbb{Z}^n$, where t_i represents the importance weight assigned to layer i. The minimum EDP achieved by layer i is a function of the assignment variable $g_i(t)$, and the overall optimization cost of the DNNs is a function of each layer's values $(g_1(t), g_2(t), \ldots, g_n(t))$. Therefore, our objective function aims to minimize the end-to-end performance, taking into account the EDP and the assigned importance weights. It can be represented as follows:

$$\min f(g_1(t), g_2(t), g_3(t), \ldots, g_n(t)). \tag{6.6}$$

To minimize the end-to-end EDP, we can define the objective function as: $f = \sum_{j=1}^{n}(w_i \times g_i(t))$. Here, w_i represents the number of occurrences of the optimization task i in the DNN model. Therefore, our aim is to fulfill the minimum EDP requirement for the entire end-to-end DNN, while keeping the exploration cost to a minimum.

Solution. Based on the given formulation, we propose an approximation algorithm that utilizes gradient descent to optimize the objective function. To achieve this, we can compute the gradient of f_i with respect to t_i, denoted as $\frac{\partial f_i}{\partial t_i}$, using Taylor expansion. This allows us to make the following approximation:

$$\frac{\partial f_i}{\partial g_i} \approx \frac{\partial f}{\partial g_i}(\alpha \frac{g_i(t_i) - g_i(t_i - \Delta t)}{\Delta t} + (1 - \alpha)\frac{g_i(t_i + \Delta t) - g_i(t_i)}{\Delta t}). \tag{6.7}$$

In the given expression, Δt represents a small backward window size. The values of $g_i(t_i)$ and $g_i(t_i - \Delta t)$ can be obtained from the history log file. However, since we have not assigned $t_i + \Delta t$ units of resources to this layer, we need to predict the value of $g_i(t_i + \Delta t)$. To make this prediction, we consider two assumptions:

- Ideal condition: When t_i is sufficiently large, the value of $g_i(t_i)$ tends to become

very small, indicating that the EDP of the layer is approximately zero. Based on this observation, we can approximate $g_i(t_i + 1)$ as $g_i(t_i) + \frac{g_i(t_i)}{t_i}$.

- Similar structure: If two layers have a similar structure, they are likely to have similar FLOPS (floating-point operations per second). Therefore, we can assume that the value of $g_i(t_i + \Delta t)$ will be similar to $g_i(t_i - \Delta t)$ if the two layers share a similar structure.

Combining these two assumptions, we can approximate the value of $g_i(t_i + \Delta t)$ as follows:

$$g_i(t_i + 1) \approx min(g_i(t_i) + \frac{g_i(t_i)}{t_i}, \beta \frac{FLOPs(i)}{max_{k \in Y(i)} FLOPS(k)}), \tag{6.8}$$

where $Y(i)$ represents the set of layers similar to layer i, $FLOPs(i)$ denotes the number of floating-point operations for layer i, and $FLOPS(k)$ represents the computing performance of BACO in task k measured in floating-point operations per second. The $FLOPS$ value can be calculated using the formula $FLOPS = \frac{FLOPs \times Freq}{Cycles}$, where higher $FLOPS$ values indicate faster and more efficient computing performance. Additionally, β determines the weight given to the prediction based on layer similarity. To execute this algorithm, the BACO task scheduler begins with an initial allocation vector $t = (1, 1, \ldots, 1)$ during the warm-up phase. In each iteration, we compute the gradient for each task and select $argmax_i |\frac{\partial s}{\partial t_i}|$. We then assign the weight to task i and update it with $t_i = t_i + 1$. The optimization process continues until convergence is achieved.

The optimization process continues until convergence is achieved. The updated objective function is as follows:

$$f = \left(\sum_{i=1}^{n} w_i \cdot e_i \right) \times \left(\sum_{i=1}^{n} w_i \cdot l_i \right). \tag{6.9}$$

Table 6.4: *Benchmarked GEMV/GEMM with different shapes.*

Category	M	N	K	#Test Cases
Real-World	[1, 256]	[1, 256]	[1, 256]	100
	[1, 256]	[1, 256]	[257, 65536]	88
	[257, 1024]	[1, 256]	[1, 65536]	64
	[257, 1024]	[257, 65536]	[1, 65536]	98
	[1025, 8192]	[1, 256]	[1, 65536]	66
	[1025, 8192]	[257, 8192]	[1, 65536]	128

With the integration of the new objective function, gradient descent now traverses a gradient that places high importance on the initial weight of w_i. This allows for the optimization of EDP with a clear understanding of which layer is most suitable for the current exploration. Moreover, if the scheduler spends a significant number of iterations on a particular layer without observing a decrease in the EDP metric, it will abandon the optimization of that layer. This decision is based on the observation that $|\frac{\partial f}{\partial t_i}|$ is decreasing in comparison to other iterations.

The fundamental concept behind importance estimation is to estimate the impact of each task on the overall performance by making optimistic predictions and considering the similarity between tasks. The workflow of hardware and software co-exploration, where multi-layers are simultaneously compiled using the unified compilation interface in BACO, is outlined in Algorithm 6.

6.6 Evaluation

6.6.1 Experimental Setup

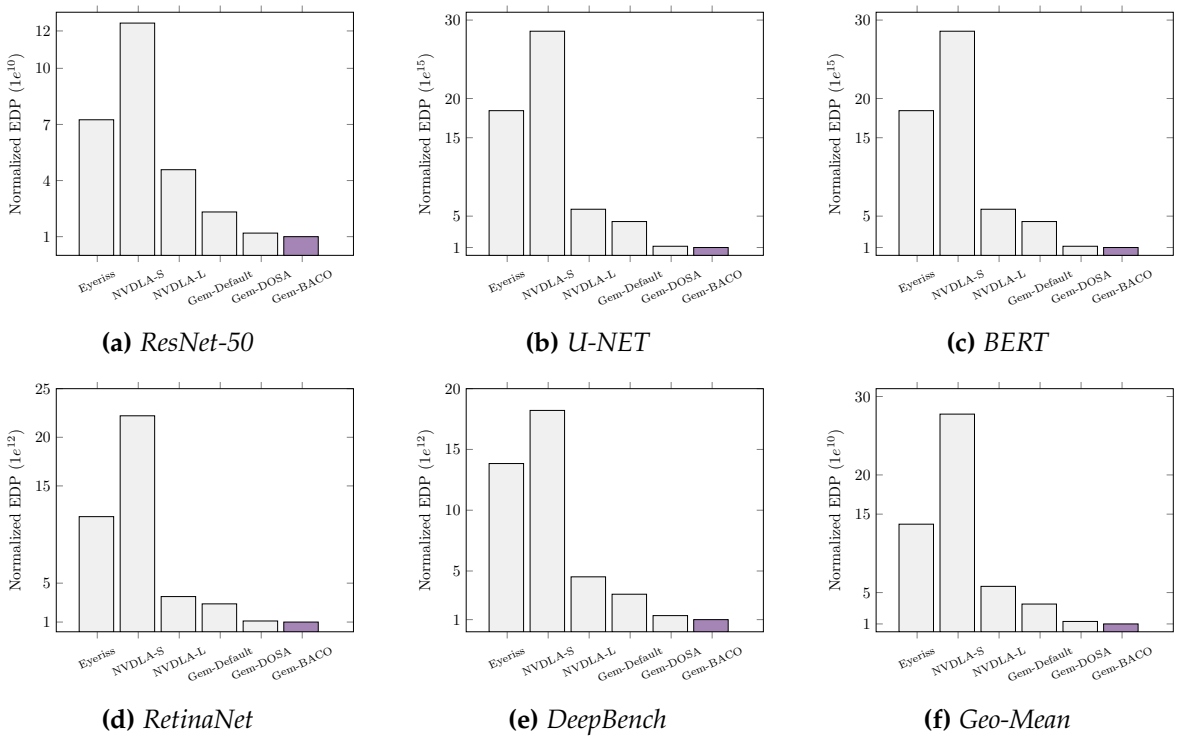
Implementation. The implementation of BACO consists of approximately 15,000 LoC in C++ and Python. This includes the DSL, MLIR dialect, and backend code generation. The tool supports DNNs as design entries, which can be specified

Table 6.5: *Benchmarked convolution with different shapes.*

Category	Fmap Size	Filter Size	#Test Cases
U-Net	[280, 568]	3×3	80
	[56, 138]	3×3	88
	[8, 52]	1×1	64
ResNet-50	[56, 120]	1×1	80
	[32, 80]	3×3	120
	[4, 40]	1×1	68
	[2, 30]	3×3	140
RetinaNet	[2, 28]	1×1	98
	[2, 28]	3×3	108

using PyTorch [**pytorch**] and the DSL. For PyTorch integration, we utilize the TorchInductor [**inductor**] compiler backend to generate Triton code [**triton-repo**]. Additionally, we make use of the Triton-shared library [**triton-shared**] to lower the Triton dialect into BACO and the MLIR core dialect for compilation. The important dialect and optimization passes in the compiler are implemented from scratch within BACO. All intermediate representations (IR) are compiled into the LLVM backend [**lattner2004llvm**] with accelerator-specific instructions. This allows us to generate executable binaries using the RISC-V GNU toolchain in chipyard [**amid2020chipyard**]. To incorporate differentiability, we utilize the PyTorch automatic differentiation mechanism. Random hardware configurations are used to generate starting points for gradient descent. The dimensions of processing elements (PEs) are limited to a maximum size of 128×128. Additionally, the sizes of SRAM are rounded up to increments of 1KB. Ensuring the correctness of the generated code is crucial, and BACO leverages the CPU backend to conduct functional simulation testing. The input data type is set to int8, while the accumulate data type is set to int32 for all evaluations.

Software and Hardware. The experiments are conducted on a server that is

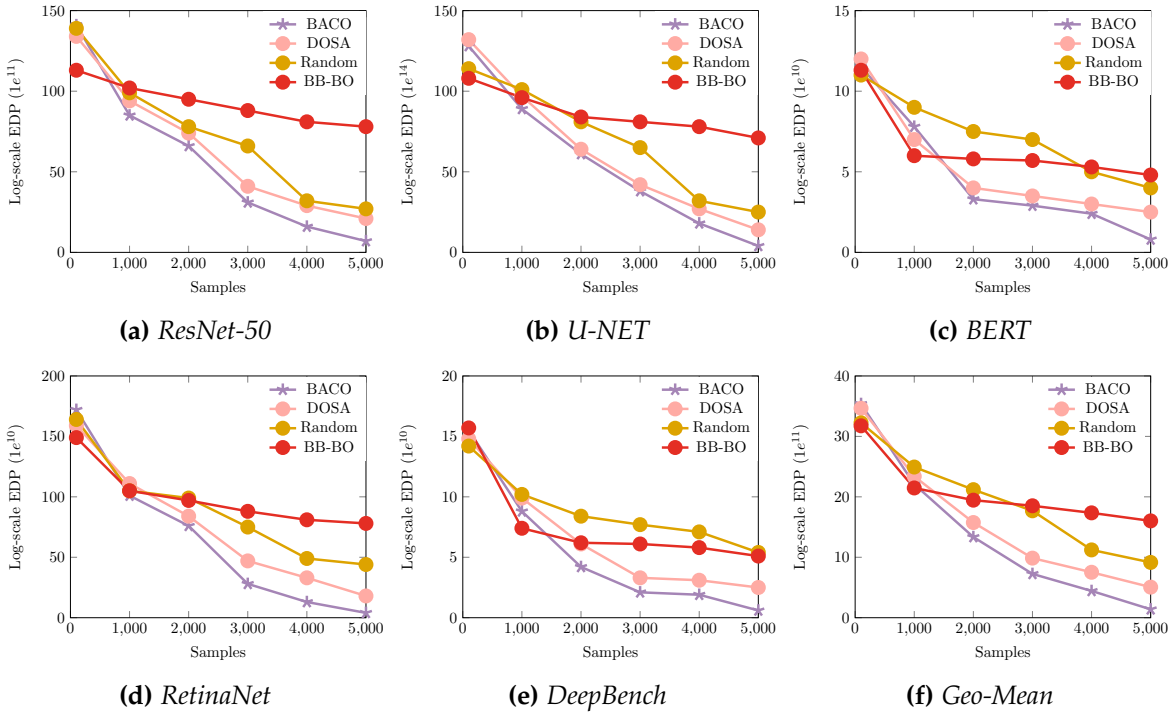


(a) *ResNet-50* **(b)** *U-NET* **(c)** *BERT*

(d) *RetinaNet* **(e)** *DeepBench* **(f)** *Geo-Mean*

Figure 6.10: *End-to-end benchmark for Energy-delay product (EDP) of baseline accelerators.*

equipped with a 16-core, 24-thread Intel i9-12900K CPU with hyper-threading. The server runs on Ubuntu LTS 22.04. For evaluation purposes, we employ both state-of-the-art performance models and real hardware designs. To assess latency and energy, we utilize the Timeloop-Accelergy [**wu2019accelergy**] and DOSA [**hong2023dosa**] performance models. In addition to the performance models, BACO also supports simulation using the Spike [**spike**] and Verilator [**verilator**]. Regarding the accelerator, we utilize the Gemmini accelerator generators in conjunction with Timeloop. These generators define the architectural specifications of the hardware accelerator, similar to Gemmini. For the RTL implementation of the Gemmini accelerator, we utilize Gemmini with the Chisel hardware description language (Gemmini-RTL)[**gemmini-repo**] and RISC-V custom instructions[**rocc-software-isa**]. To evaluate the latency of RTL simulation, we utilize FireSim [**karandikar2018firesim**].



(a) *ResNet-50* **(b)** *U-NET* **(c)** *BERT*

(d) *RetinaNet* **(e)** *DeepBench* **(f)** *Geo-Mean*

Figure 6.11: *End-to-end search efficiency for Energy-delay product (EDP) of baseline methods.*

6.6.2 Single Operator Benchmark

Workloads. We begin by evaluating BACO on a range of common DL operators, which include GEMM, GEMV, 1D convolution, and 2D convolution. Table 6.4 and Table 6.5 present the benchmarks for these operators. Each test case is characterized by a unique shape size, and the range of values is denoted as $[min, max]$. For matrix operations, we consider a total of 544 test cases derived from real-world applications, specifically Transformer-based models such as BERT **[devlin2018bert]**, DistilBERT **[sanh2019distilbert]**, RoBERTa **[liu2019roberta]**, Llama2-7B **[touvron2023llama]**, GPT-2 **[radford2019language]**, and ViT-B/16 **[dosovitskiy2021an]**. To evaluate efficiency in language model (LLM) scenarios, we select Llama2-7B from HuggingFace as the benchmark. The input sequence lengths range from 1 to 512,

Table 6.6: *Single Operator Benchmark on BACO.*

Benchmark	Accelerators	#Test Cases	Cycles	Energy	EDP
GEMM	Eyeriss	170	200433	41992.83	13.95×
	NVDLA-Small		21498335	810.24	28.87×
	NVDLA-Large		476818	5010.85	3.96×
	Gemmini-Default		3193494	923.87	4.89×
	Gemmini-DOSA		1630720	547.58	1.48×
	Gemmini-BACO		1358934	443.98	1.00×
GEMV	Eyeriss	174	74868	127692.71	16.72×
	NVDLA-Small		156342	130050.54	35.56×
	NVDLA-Large		37985	62921.04	4.18×
	Gemmini-Default		25874	119776.11	5.42×
	Gemmini-DOSA		13432	81304.068	1.91×
	Gemmini-BACO		11010	51930.26	1.00×
Conv1D	Eyeriss	68	226024	1888590.39	7.52×
	NVDLA-Small		521507	2155167.18	19.81×
	NVDLA-Large		49119	2981569	2.58×
	Gemmini-Default		72527	2715832.08	3.47×
	Gemmini-DOSA		65934	1231124.96	1.43×
	Gemmini-BACO		38374	1479228.38	1.00×
Conv2D	Eyeriss	128	315838.85	13864679.81	8.49×
	NVDLA-Small		493789.35	20943070.61	20.05×
	NVDLA-Large		240372.93	7338516.07	3.42×
	Gemmini-Default		231056	8906827.97	3.99×
	Gemmini-DOSA		101552	6501077.18	1.28×
	Gemmini-BACO		93168	5536073.53	1.00×

and the batch sizes range from 1 to 8. The output sequence length is fixed at 512, which aligns with common practices in serving LLMs.

Baselines and settings. We focus on optimizing the default architecture template of Gemmini, which consists of a 16×16 systolic array for block matrix multiplication, a 256KB scratchpad memory for input and output tensors, and a 64KB accumulator for partial sums. The low-level instruction format of Gemmini remains unchanged, including the movement of strided tensors to and from the scratchpad and the instructions for calculating dot products. In our evaluation, we compare the performance of Gemmini with different optimization approaches. These include Gemmini with the default heuristic-based optimization (Gemmini-Default), Gemmini with BACO optimization (Gemmini-BACO), Gemmini with DOSA optimization (Gemmini-DOSA), and other expert-optimized accelerator baselines such as Eyeriss [**chen2016eyeriss**] and NVDLA Small/Large [**zhou2018research**].

Table 6.7: *Ablation study of BACO on ResNet-50.*

Setting	BACO					
	(a)	(b)	(c)	(d)	(e)	(f)
Block Partition		✓	✓	✓	✓	✓
Infer HW Config			✓	✓	✓	✓
Analytical Perf. Model				✓	✓	✓
Learned Perf. Model					✓	✓
Importance Estimation						✓
Improvement	1.00×	1.44×	1.87×	1.98×	2.17×	2.32×

Results. We have randomly selected 170 cases from the GEMM benchmark, 174 cases from the GEMV benchmark, 68 cases from the 1D convolution benchmark, and 128 cases from the 2D convolution benchmark for our evaluation. The performance of the single operator benchmark is presented in Table 6.6. Our findings indicate that the mappings and hardware optimized by BACO exhibit greater consistency with the baseline compared to DOSA, with improvements of up to 1.91×, and the default Gemmini, with improvements of up to 5.42×. To evaluate these accelerators, we utilize Timeloop and perform a search of 5,000 valid mappings per operator using the random-pruned mapper. The reported results represent the geometric mean of the improvements in Energy-Delay Product (EDP) for all evaluations.

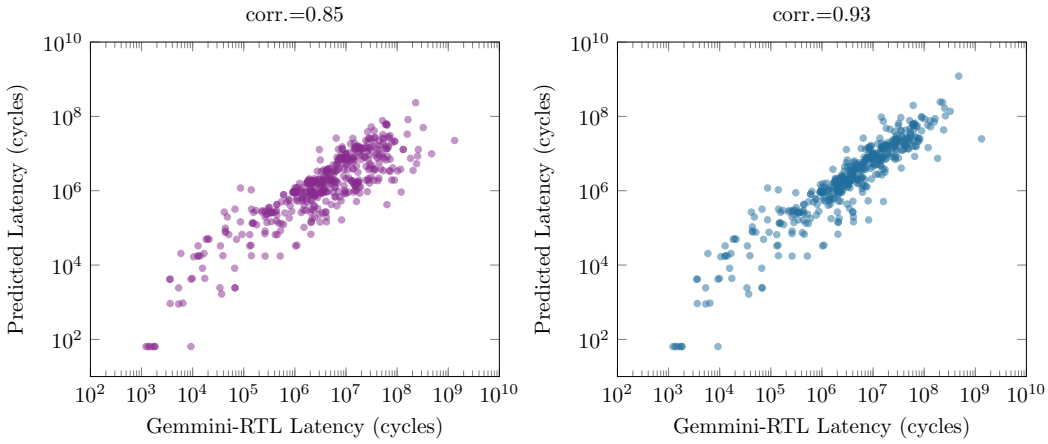
6.6.3 End-to-End Network Benchmark

Workloads. We assess the performance of our design across a diverse range of target DNN models that cover various tasks, including image classification, object detection, image segmentation, and natural language processing. The models include ResNet-50, U-Net, RetinaNet, BERT, and DeepBench with OCR and face recognition tasks [**deepbench**]. For these evaluations, we utilize the model implementations provided in the torchvision and transformers packages. The reported results are

obtained using a batch size of 1. It should be noted that the parameters α and β introduced in Section 6.5 are consistently set to 0.2 and 2, respectively, for all evaluations.

Results. We apply the same settings and accelerator baselines as described in Section 6.6.2 to evaluate the performance of the five target workloads. The results of the end-to-end performance with a single batch are presented in Figure 6.10. For these end-to-end workloads, we utilize Timeloop and search 10,000 valid mappings per layer using the random-pruner mapper. In terms of the EDP metric, BACO consistently outperforms the baselines. Compared to Gemmini-DOSA, BACO achieves a maximum improvement of 1.64×. Moreover, when compared to Gemmini-default, the highest improvement observed is 5.23×. Note that the primary distinction between single operator benchmarks and end-to-end benchmarks lies in the introduction of an importance estimator based on the gradient descent algorithm.

Ablation Study. We run five variants of BACO on the ResNet-50 and report the final improvement achieved in Table 6.7. "Block Partition": Autotuning is used to determine the optimal partitioning strategy. "Infer HW Config": The compiler is utilized to infer the minimal hardware configurations. "Learned Perf.Model": We employ a DNN-based model [**bai-2023-atformer**] to enhance the original performance model and capture real hardware performance characteristics. "Importance Estimation": Different weights are assigned to different layers to prioritize their impact on the end-to-end improvement. Compared to Gemmini-default, the combination of all proposed optimizations resulted in a significant improvement of 2.32× in the EDP metric.



(a) *Analytical Model* **(b)** *Analytical Model with DNN*

Figure 6.12: *Gemmini RTL latency v.s. predicted latency.*

6.6.4 Exploration Efficiency

Figure 6.11 illustrates the exploration efficiency of BACO compared to other state-of-the-art search methods, including *one-loop search* DOSA, random search, and black-box Bayesian optimization (BB-BO), for the five target workloads. All of the searchers are executed with 24 threads. Our findings demonstrate that BACO is capable of identifying optimal co-design points more effectively than other methods, despite using a similar number of samples. Initially, when the number of samples is relatively small, BB-BO outperforms the other methods. However, as the number of samples increases, BACO gradually exhibits superior performance. When the number of samples reaches 5,000, BACO has already converged to a very low EDP. This conclusion is further supported by the geometric mean of the results. To ensure a fair comparison, we employ the same setting as described in [**hong2023dosa**] for the other search methods. Our results clearly demonstrate that BACO not only reduces the convergence time but also generates high-quality schedules even with a large number of samples.

6.6.5 Performance Model Evaluation

To evaluate the quality of predictions, we generate a dataset consisting of 1,000 samples. These samples correspond to cycle-accurate Gemmini-RTL measurements obtained from FireSim. The samples are distributed evenly among the layers listed in Table 6.4 and Table 6.5. The dataset is divided into a training set (80% of the samples) and a test set (remaining 20%). Note that analytical models may not capture real hardware performance completely, as complex hardware-software interactions can be challenging to mathematically represent. To address this, one potential solution is to enhance analytical models with learned surrogate models. In our case, we employ an encoder-based DNN model [**bai-2023-atformer**] to augment the analytical model. The model is trained using the Adam optimizer for 30,000 epochs. Figure 6.12 shows the predicted performance compared to the Gemmini-RTL performance of ResNet-50 on the test set. The scatter points are distributed around the diagonal line, indicating that the model provides accurate predictions. The analytical model combined with the encoder-based DNN model achieves the highest accuracy, with a correlation coefficient of 0.93. The analytical model alone achieves a correlation coefficient of 0.85 on the test set.

6.7 Summary

This chapter proposes BACO, a comprehensive framework for co-exploration of hardware and software through an end-to-end one-loop search approach, accompanied by a unified compilation interface. We discover that block-level design granularity is well-suited for co-exploration purposes. Building upon this insight, BACO employs multi-level abstraction to clearly define the behavior of customized hardware

instructions, specialized memory management, and accelerator configuration state. Additionally, BACO provides programming language and compiler support to automatically explore diverse high-performance designs using differentiable analytical performance models based on gradient descent. We believe that BACO has the potential to greatly enhance co-design opportunities for performance engineers, compiler writers, computer architects, and simulation tool developers.

Chapter 7

Discussions

In the final section of this dissertation, we shall encapsulate the key insights gained from the works explored in the preceding chapters. Subsequently, we shall delve into the unresolved inquiries pertaining to the realm of design automation with compiler research.

7.1 Tensor Program Generation with Compilation

In the realm of automating tensor program generation through compilation optimization techniques, there exists the potential to broaden the scope of optimization beyond a single operator, thereby transcending the boundaries of deep learning alone. This can be accomplished by extending the design space to encompass the computation graph and integrating the interdependencies between operators within said graph. This expansion serves to augment the size of the solution space for compilation. Furthermore, in the context of time-intensive on-device testing, the combination of transfer learning can be leveraged to expedite the extension of compilation across diverse hardware platforms. By employing a performance

model as an intermediary, imbued with explicit instructions and characteristics of the hardware platform, the productivity of the compiler can be heightened.

7.2 Reforming Accelerator Design with Compilation

The research endeavors presented in this dissertation are primarily focused on pushing the boundaries of full-stack tensor program generation and specialized accelerator design through the use of a compiler. Our main objective is to propose a reformulation of the design paradigm for hardware specification, where previously co-designed software/hardware innovations are modularized and integrated into a cohesive design space for future reusability. By identifying commonalities between tensor programs and accelerators, we have designed a user-friendly workflow that serves as a unified programming interface with manageable extensions. However, it should be noted that this approach poses significant challenges in the development of high-level compilation techniques.

An alternative approach to constructing software stacks for novel specialized hardware accelerators is the development of domain-specific programming languages. By bridging the gap between software and hardware design, these languages inherently encode the intricate details of accelerator designs within the top-level computation application. Leveraging the capabilities of the compiler, this automated design flow enables the realization of specialized hardware accelerators. Moreover, our dissertation demonstrates how optimization techniques developed for these new specialized hardware accelerators and general-purpose processors can mutually enhance one another throughout the design process. For instance, tensorized instructions such as Intel AVX512, ARM NEON, and NVIDIA WMMA can be utilized to analyze and model the specific behavior of memory access patterns

during compilation.

7.3 Future Research and Open Questions

Another avenue for future research within this dissertation lies in expanding the applicability of full-stack specialized accelerators. We envision a promising area of exploration to be the design of mobile Systems-on-Chips (SoCs) that incorporate automatic generation of the software stack. Over the past decade, mobile SoCs developed by prominent companies like Apple and Qualcomm have integrated numerous specialized accelerator blocks, with their quantity continuing to proliferate. These specialized blocks consume a significant amount of on-chip power and occupy substantial area resources.

Previous research has demonstrated that a unified programmable accelerator has the potential to achieve performance levels comparable to those of individual specialized blocks, while still offering flexibility with a moderate power/area overhead, in contrast to a combination of specialized blocks. Therefore, we posit that the present moment presents an opportune time to reevaluate the design of specialized blocks on SoCs by embracing unified programmable compilation. This approach would not only save considerable effort in developing new accelerators for each generation but also enable scenarios where performance requirements can be met through software tuning alone.

7.4 Conclusion

The design of an accelerator and its corresponding software stack is not solely dictated by the application domain, but rather by the specific tensor program within

those applications. Building upon this understanding, this dissertation introduces the concept of automated and programmable tensor program and accelerator design. Through a meticulous co-design process, each software/hardware innovation is carefully constructed, allowing for modularization and encapsulation within a comprehensive and universally defined design space. Moreover, the relationship between software and hardware can be comprehended by a domain-specific compiler and programming language. This compiler serves a dual purpose: acting as a bridge between software and hardware, while also guiding the exploration of the hardware design space. By leveraging the compiler's understanding of the synergies between software and hardware, in conjunction with the design requirements inherent in the applications, effective exploration of the design space can be achieved. Beyond mere automation, we propose that the true transformative potential lies in reshaping the fundamental design principles of specialized accelerators. By expanding this design space, subsequent innovations can be realized, rendering them applicable across various application domains. We propose four key innovations that are centered around compiler-based approaches:

First, we describe GTCO [**bai2023gtco**] in Chapter 3, a new graph and tensor co-design compiler that addresses operator fusion issue from two perspectives. A dynamic programming algorithm is introduced to explore operator fusion patterns. Then, a search policy is proposed that includes new sketch generation rules and a novel hardware abstraction with register-level optimization, enabling more flexible mapping for tensor computation and better performance. We apply our approach to optimize fused matrix multiplication and softmax operators utilizing WMMA instructions. To achieve an end-to-end flow automatically, we utilize a regression-based learned model to fine-tune the performance of each kernel. Overall, GTCO demonstrates remarkable results. It achieves up to 1.73$\times$ inference speedups com-

pared to the high-performance inference engine TensorRT with Tensor Cores, and 1.38× speedups with CUDA Cores.

Second, we describe ATFormer [**bai2023atformer**] in Chapter 4, an innovative and efficient design for optimizing tensor programs. ATFormer leverages hierarchical features with different levels of granularity to effectively model the end-to-end compilation process. Additionally, self-attention blocks are employed to capture global dependencies within a complete tensor program, enabling high-quality evaluation. By utilizing transfer learning techniques, ATFormer achieves faster-converged latency and demonstrates superior transferability across various hardware platforms. Our approach outperforms previous state-of-the-art benchmarks, establishing its effectiveness and superiority in the field.

Third, we describe ALCOP [**huang2023alcop**] in Chapter 5, a deep learning compiler that addresses the crucial need for automatic pipelining. In order to mitigate bandwidth constraints, a large tiling size is often required. However, inter-tile parallelism alone is insufficient for achieving high utilization, making intra-tile pipelining essential.Our proposal is the first compiler solution that supports multi-stage, multi-level pipelining. By introducing automatic pipelining, our compiler is capable of generating GPU programs that exhibit an average speedup of 1.23× over vanilla TVM [**chen2018tvm**], with a maximum speedup of 1.73×. Furthermore, we develop an analytical performance model that significantly enhances the search efficiency of the schedule tuning process. This model contributes to the overall effectiveness and efficiency of ALCOP, enabling it to deliver impressive results in optimizing deep learning computations.

Finally, we describe BACO in Chapter 6, a comprehensive framework that facilitates the co-exploration of hardware and software through an end-to-end one-loop search approach. This framework is accompanied by a unified compilation

interface, providing a seamless and convenient experience for users. Through our research, we have discovered that block-level design granularity is particularly well-suited for co-exploration purposes. Building upon this insight, BACO adopts a multi-level abstraction approach to clearly define the behavior of customized hardware instructions, specialized memory management, and accelerator configuration state. Furthermore, BACO offers programming language and compiler support, enabling the automatic exploration of diverse high-performance designs. This is achieved through the utilization of differentiable analytical performance models based on gradient descent. We firmly believe that BACO has immense potential in enhancing co-design opportunities for performance engineers, compiler writers, computer architects, and simulation tool developers. By providing a comprehensive framework and powerful tools, BACO empowers these professionals to push the boundaries of performance optimization and innovation.

We aim for our contributions to serve as a foundational platform for future research endeavors focused on the development of advanced compilers with design automation, specifically aimed at achieving high efficiency.