University of California
Santa Barbara

System for Efficient Big Data Analytics

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

Doctor of Philosophy
in
Computer Science

by

Boyuan Feng

Committee in charge:

Professor Yufei Ding, Chair
Professor Chandra Krintz
Professor Lingqi Yan

March 2022

The Dissertation of Boyuan Feng is approved.

Professor Chandra Krintz

Professor Lingqi Yan

Professor Yufei Ding, Committee Chair

March 2022

System for Efficient Big Data Analytics

by

Boyuan Feng

To my family

Acknowledgements

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

I would like to thank the rest of my thesis committee members – Professor Chandra Krintz and Professor Lingqi Yan. Their feedback and advice are essential for completing this dissertation. I want to thank Professor Yuan Xie for creating an inclusive lab environment and providing strong support. We always have the best GPU machines to experiment with cutting-edge ideas. I also want to thank Professor Rich Wolski, Amr El Abbadi, and Yu-Xiang Wang, who taught me a lot on systems and theories.

I am grateful to have Dr. Weifeng Zhang and Dr. Guoyang Chen for their guidance on my first GPU project and the great internship experience in their group at Alibaba. I want to thank Dr. Ang Li and Dr. Tong Geng for their insightful suggestions. I would also thank Dr. Shumo Chu for bringing me to the exciting world of zero-knowledge proof. I want to thank Dr. Zhou Li and Dr. Yu Zhang for helpful discussions. I learn a lot from each of them and appreciate their help during this journey.

I want to thank my lab mates and friends: Yuke Wang, Xin Ma, Zheng Wang, Gushu Li, Tianqi Tang, Xinfeng Xie, Lei Zhang, Liu Liu, Zhaodong Chen, Lianke Qin, Kun Wan, Xiaolei Liu, Fei Shi, and Lingwei Xie. I enjoy every moment that we have worked and played together.

Finally, I want to thank my wife, Shu Yang, for this great journey during the last nine years. From Nanjing to Madison. From Madison to Santa Barbara. During some days, I cannot publish a single paper and she cannot find a single job. I asked her to practice more LeetCodes and she asked me to do more research. We encourage each other and solve problems together. I would also thank my parents for their unconditioned support. Their kindness and self-motivation set role models for me.

Curriculum Vitæ

Boyuan Feng

Education

2017 - 2022	Doctor of Philosophy in Computer Science, University of California, Santa Barbara
2016 - 2017	Master of Science in Statistics, University of Wisconsin, Madison
2012 - 2016	Bachelor of Science in Statistics, Nanjing University

Experience

2017 - 2022	Research Assistant, University of California, Santa Barbara, CA
2021	Machine Learning Engineer (Ph.D.) Intern, Facebook, CA
2020	Research Intern, Alibaba, CA

Publications

[1] Yuke Wang*, **Boyuan Feng***, and Yufei Ding. (* co-primary authors) "QGTC: Accelerating Quantized GNN via GPU Tensor Core". Symposium on Principles and Practice of Parallel Programming (PPoPP) 2022.

[2] **Boyuan Feng**, Yuke Wang, Guoyang Chen, Weifeng Zhang, Yuan Xie, and Yufei Ding. "EGEMM-TC: Accelerating Scientific Computing on Tensor Cores with Extended Precision". Symposium on Principles and Practice of Parallel Programming (PPoPP) 2021.

[3] **Boyuan Feng***, Yuke Wang*, Tong Geng, Ang Li, Yufei Ding (* co-primary authors). "APNN-TC: Accelerating Arbitrary-Precision Neural Networks on Tensor Cores". The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC) 2021.

[4] **Boyuan Feng**, Yuke Wang, Gushu Li, Yuan Xie, Yufei Ding. "Palleon: A Runtime System for Efficient Video Processing toward Dynamic Class Skew". USENIX Annual Technical Conference (ATC) 2021.

[5] **Boyuan Feng**, Yuke Wang, and Yufei Ding. "Uncertainty-aware Attention Graph Neural Network for Defending Adversarial Attacks". Association for the Advancement of Artificial Intelligence (AAAI) 2021.

[6] **Boyuan Feng**, Yuke Wang, and Yufei Ding. "SAGA: Sparse Adversarial Attack on EEG-based Brain Computer Interface". The international Conference on Acoustics, Speech, and Signal Processing (ICASSP) 2021.

[7] **Boyuan Feng**, Lianke Qin, Zhenfei Zhang, Yufei Ding, and Shumo Chu. "ZEN: Efficient Zero-Knowledge Proofs for Neural Networks". Technical Report.

[8] **Boyuan Feng**, Zheng Wang, Yuke Wang, Lianke Qin, Shu Yang, Shumo Chu, Yuan Xie, Yufei Ding. "ZENO: A Type-based Optimization Framework for Zero Knowledge Neural Network Inference". Technical Report.

[9] **Boyuan Feng**, Tianqi Tang, Yuke Wang, Zhaodong Chen, Zheng Wang, Shu Yang, Yuan Xie, Yufei Ding. "Faith: An Efficient Framework for Transformer Verification on GPUs". Technical Report.

[10] Yuke Wang, **Boyuan Feng**, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, Yufei Ding. "GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs". USENIX Symposium on Operating Systems Design and Implementation (OSDI) 2021.

[11] Yuke Wang, **Boyuan Feng**, and Yufei Ding. "DSXplore: Optimizing Convolutional Neural Networks via Sliding-Channel Convolution". International Parallel and Distributed Processing Symposium (IPDPS) 2021.

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

[13] Yuke Wang, **Boyuan Feng**, Gushu Li, Lei Deng, Yuan Xie, Yufei Ding. "STPAcc: Structural TI-based Pruning for Accelerating Distance-related Algorithms on CPU-FPGA Platforms". IEEE Transactions on Computer-Aided Design of Integrated Circuits And System (TCAD) 2021.

[14] Yuke Wang, **Boyuan Feng**, Xueqiao Peng, Yufei Ding. "An Efficient Quantitative Approach for Optimizing Convolutional Neural Networks". The Conference on Information and Knowledge Management (CIKM) 2021.

[15] Yuke Wang, **Boyuan Feng**, Gushu Li, Georgios Tzimpragos, Lei Deng, Yuan Xie, Yufei Ding. "TiAcc: Triangle-inequality based Hardware Accelerator for K-means on FPGAs". IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid) 2021.

[16] Fei Shi, Hongrui Cao, Yuke Wang, **Boyuan Feng**, and Yufei Ding. "Chatter Detection in High-speed Milling Processes based on ON-LSTM and PBT". The International Journal of Advanced Manufacturing Technology. 2020.

[17] Yitong Huang, Yu Zhang, **Boyuan Feng**, Xing Guo, Yanyong Zhang, Yufei Ding. "A Close Look at Multi-Tenant Parallel CNN Inference for Autonomous Driving". Annual IFIP International Conference on Network and Parallel Computing (NPC) 2020.

[18] Lingwei Xie, Song He, Zhongnan Zhang, Kunhui Lin, Xiaochen Bo, Shu Yang, **Boyuan Feng**, Kun Wan, Kang Yang, Jie Yang, Yufei Ding. "Domain-Adversarial Multi-Task Framework for Novel Therapeutic Property Prediction of Compounds". Bioinformatics 2020.

[19] Kun Wan, Shu Yang, **Boyuan Feng**, Lingwei Xie, Yufei Ding. "Reconciling Feature-Reuse and Overfitting in DenseNet with Specialized Dropout." The IEEE International

Conference on Tools with Artificial Intelligence (ICTAI) 2019.

[20] Yuke Wang, Zhaorui Zeng, **Boyuan Feng**, Lei Deng, and Yufei Ding. "KPynq: A Work-Efficient Triangle-Inequality based K-means on FPGA". The IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM) 2019.

Please Note: Text and figures from these papers are used and appear in this dissertation.

Abstract

System for Efficient Big Data Analytics

by

Boyuan Feng

Big data analytics enjoy increasingly wide applications in the real world enabled by the development of model, data, and hardware. However, the development of these three components usually shows a significant imbalance. While there are many new models and data, commercialized hardware usually provides only limited support. My research mitigates this gap by building systems for efficient big data analytics that stitch model, data, and hardware together. In particular, we find that given specialized hardware with limited compute primitives, system optimizations are the keys to generalizing such specialized hardware and efficiently supporting diverse model and data workload.

This thesis discusses systems for efficient big data analytics under three aspects. The first aspect introduces hardware-aware kernel tuning. Based on limited hardware compute primitives, my research builds more middle-level libraries to efficiently support diverse big data analytic workloads. The second aspect proposes runtime systems for efficient neural network inference. While larger neural networks are used for general workload, a specialized workload is usually observed in a specific scenario. My research builds runtime systems to automatically detect the scenario information during runtime and exploit such information for efficient big data analytics. The third aspect is to build secure deep learning frameworks to efficiently support diverse workloads such as zero-knowledge neural networks and neural network verification. My research abstracts the key computing patterns in secure deep learning and automatically optimizes diverse NN operators with framework supports.

Contents

Chapter 1

Introduction

1.1 Motivation

Big data analytics are changing the landscape of many real-world applications thanks to its human-level accuracy. This success is powered by the advancement of diverse model designs and large-scale datasets. For example, convolutional neural networks (CNNs) [1, 2] achieve human-level performance on image recognition and power self-driving cars in Waymo [3] and TuSimple [4]. Transformers [5, 6] achieve state-of-the-art performance on many natural language processing (NLP) tasks and have been widely deployed for hate speech detection in Facebook [7] and question answering in Alex [8]. Recent advancement also includes secure deep learning [9, 10, 11, 12, 13] to protect the privacy and improve the robustness of neural networks.

Despite its wide deployment, big data analytics usually come with high latency and energy consumption due to two reasons. The first reason is that models in big data analytics usually involve intensive computation and memory access. VggNet-16 has 30 giga floating-point operations (GFLOPs) when processing a single image. This lead to 1.4-second latency and 3.6W energy consumption which can drain the power of a large

smartphone battery (e.g., 2.7-Ah battecy in iPhoneX [14]) in 2 hours. The second reason is that models designed for different tasks usually show significantly different computing patterns, which makes it challenging to efficiently map the high-level algorithmic design to low-level hardware backends. Convolutional neural networks are usually composed by dense matrix-matrix multiplications while graph neural networks incorporate sparse graph computation. Recent secure models such as zero-knowledge neural networks involve specialized security computation (*e.g.*, *circuit computation*) which is significantly different from plaintext models.

Many hardwares, such as GPUs, have been developed to efficiently support big data analytics by improving computation and memory access performance. GPUs are first developed to efficiently animate graphics especially various video games in 1970s [15]. During 2000s, GPUs are introduced to accelerate neural networks especially the shallow models with two fully connected layers [16, 17, 18, 19]. Since then, many hardware efforts in modern GPUs have been made to efficiently support big data analytics, such as Tensor Cores [20] to accelerate tensor computation and mobile GPUs (*e.g.*, Jetson Nano [21]) to facilitate big data analytics on edge devices. However, there is still a large gap between the limited hardware supports and the diverse computing patterns in big data analytics. For example, many quantized models have been designed with 2-bit or 3-bit computation [22, 23, 24], while Tensor Cores on Ampere architecture provides limited precision supports (*e.g.*, 1-bit and 4-bit computation).

My research builds systems to efficiently support diverse model and data workload with commercialized hardware that only has limited compute primitives. The key feature is to characterize the computing patterns in diverse big data analytics and build systems to efficiently support such computing patterns. In this thesis, I will show how my research efficiently supports big data analytics from three perspectives. The first perspective is *hardware-aware kernel tuning*, which builds more middle-level libraries for diverse work-

load based on limited hardware compute primitives (chapter 2, chapter 3). The second perspective is *runtime system for efficient NN inference*, which identifies workload characteristics during runtime and automatically specializes NNs to reduce latency and energy consumption (chapter 4). The third perspective is *secure deep learning framework* which abstracts the key computing patterns from diverse secure deep learning workloads and automatically optimizes such computing patterns with framework supports (chapter 5, chapter 6, chapter 7).

1.2 Overview of the Dissertation

1.2.1 Hardware-aware Kernel Tuning

Many hardwares have been designed to accelerate big data analytics especially by supporting tensor computation. Tensor Core from NVIDIA GPUs is one of the most popular machine learning accelerators and has been widely adopted across various domains such as deep learning, high-resolution climate simulation, earthquake simulation, and bioinformatics [25]. Comparing with CUDA Cores, Tensor Cores achieve 4× to 8× higher throughput on NVIDIA A100 GPUs. To exploit this increased throughput, PyTorch uses Tensor Cores as the default compute primitive when available [26].

While Tensor Cores have been successfully deployed in many domains, there is still a large gap between the limited hardware compute primitives and the diverse model designs. Different from previous CUDA Cores with scalar computation, Tensor Cores only support tensor computation (*e.g.*, multiplying two input matrices of shape 8×16 and 16×8 to generate an output matrix of shape 8×8) and a limited set of precision (*e.g.*, `int1`, `int4`, and `fp16`). However, models for diverse big data analytics usually show significantly different computing patterns, which makes it challenging to exploit Tensor

Cores. My research fills this gap by building middle-level libraries that efficiently map diverse workload towards the limited hardware compute primitives.

We start our exploration with APNN-TC [27] (chapter 2) that accelerates arbitrary-precision neural networks on Tensor Cores. Arbitrary-precision neural network [22, 23, 24] is an important category of fast and efficient neural networks that replaces high-precision data (*e.g.*, `fp32`) in neural networks with low-precision quantized representation (*e.g.*, `int2`). It finds its strengths in the minimum modification of the original model architecture, lower memory consumption, and potentially better runtime performance. While quantized neural networks usually require arbitrary precisions (e.g., 1-bit weight and 2-bit activations), Tensor Cores only support a limited range of precisions (*e.g.*, `int1` and `int4`) and limits the potential performance benefits.

To tackle this problem, we propose APNN-TC to support arbitrary precision neural networks with the limited precisions on Tensor Cores. We learn that the key is to develop an AP-BIT algorithmic emulation design to support arbitrary-precision computation and an efficient AP-Layer kernel implementation to achieve high performance for individual NN layers. We further propose an efficient APNN design to minimize data movement across NN layers. Our experiments show that APNN-TC can achieve up to 3.78× speedup over CUTLASS kernels and 3.08× speedup over CUBLAS kernels.

We further develop EGEMM-TC [28] (chapter 3) to accelerate scientific computing on Tensor Cores with extended precision. In NVIDIA Volta and Turing achitectures, Tensor Cores achieve high performance with half-precision matrix inputs tailored towards deep learning workloads, based on the fact that deep learning workloads are usually robust to low-precision computation [29, 30, 31]. Since GEneral Matrix-Matrix multiplication (GEMM) is also one essential building block of many scientific computing applications, we expect to bring this performance benefits to scientific computing domain. However, many scientific computing applications (*e.g.*, kNN and kMeans in large-scale physical

simulations [32] and mathematical computations [33]) are rather sensitive to computation precision for generating valid results. Such a restriction on precision prevents them from exploiting powerful Tensor Cores to improve performance.

To tackle this problem, we design Emulated GEMM on Tensor Cores (EGEMM-TC) to accelerate GEMM-based scientific computing on Tensor Cores with both high performance and extended-precision computation. Our key insight is that exploiting high-precision intermediate results from hardware computation can effectively mitigate the emulation overhead. Evaluation shows that EGEMM-TC achieves 3.13× and 11.18× speedup on average over single-precision kernels on CUDA Cores from cuBLAS and CUDA-SDK, respectively. On a set of GEMM-based scientific computing application, EGEMM-TC also achieves 1.8× speedup on average compared to hand-tuned code on CUDA Cores.

1.2.2 Runtime system for efficient NN inference

Another important technique for accelerating big data analytics is automatically detecting and exploiting the runtime information to reduce energy consumption while still achieving the desired accuracy. One typical runtime information in big data analytics is the *temporal locality* in video streams. Considering a video stream collected from a continuous camera feed, it is common that only a small number of classes keep appearing in a large number of consecutive frames. For example, in a film scenario, only a small number of people would come to the master shots frequently, generally lasting for a few minutes, and another group of people will not appear until the scenario has changed.

We turn such an abstract concept, temporal locality, into something concrete and measurable, *class skew*. To fully exploit the class skews, we build Palleon [34] (chapter 4), a runtime system that dynamically adapts and selects a CNN model with the least energy

consumption based on the automatically detected class skews, while still achieving the desired accuracy. Extensive experiments confirm the effectiveness of Palleon and show that it could achieve up to 6.7× energy saving and 7.9× latency reduction while achieving an equivalent or better accuracy.

1.2.3 Secure deep learning frameworks

Secure deep learning frameworks are important to efficiently support models with diverse computing patterns. Early neural networks usually involve with only standard tensor computations such as matrix-matrix multiplication for convolution layers and can be accelerated with vendor libraries such as cuBLAS and CUTLASS. However, recent private and secure models show significantly different computing patterns and usually can hardly be accelerated with existing vendor libraries. For example, zero-knowledge neural network requires circuit computation that is intrinsic to the zero-knowledge proof security scheme, which are significantly different from standard tensor computation. Accelerating these private and secure models usually requires both expertise from the machine learning domain and the security domain, making it challenging to optimize the performance. My research builds secure deep learning frameworks to automatically accelerate these secure and private models by efficiently supporting diverse computing patterns.

We first develop ZEN [35] (chapter 5) to generate efficient verifiable, zero-knowledge neural network (zkNN) inference schemes. We generate two zkNN schemes: ZEN_{acc} and ZEN_{infer}. Used in combination, these verifiable computation schemes ensure both the privacy of the sensitive user data as well as the confidentiality of the neural network models. We further propose two kinds of optimizations to efficiently map neural networks to the zero-knowledge proof security schemes. Evaluation shows that ZEN produces verifiable neural network inference schemes with 5.43× to 22.19× (15.35× on average)

less R1CS constraints.

Then, we develop ZENO (chapter 6) to significantly reduce the latency of zkNN inference schemes. Different from ZEN that reduces the number of constraints, ZENO focuses on system optimizations to further reduce latency. We observe that system optimizations play an important role in reducing the latency of zkNNs while the computation complexity of zero-knowledge proofs is proportional to the number of constraints in theory. Our key insight is to exploit the high-level privacy and tensor semantics to reduce the cost of low-level zero-knowledge circuit computation. Evaluation shows that, based on the same security schemes from ZEN, ZENO can achieve 8.5× end-to-end speedup over state-of-the-art zkSNARK systems.

Finally, we build an efficient framework, Faith (chapter 7), to accelerate transformer verification on GPUs. Transformer verification is an important technique to formally verify the robustness of a transformer against adversarial attacks [11, 9, 10, 36]. However, transformer verification contains intensive bound-centric computation, which are significantly different from tensor computation in standard neural networks and lead to high latency. To tackle this problem, we build Faith (chapter 7), the first efficient framework to optimize the performance of transformer verification on GPUs. We propose a set of verification tailored system optimizations. In particular, we design a semantic-aware computation graph transformation to identify and exploit novel fusion opportunities for transformer verification, a verifier-specialized kernel crafter to effectively map transformer verification kernels to GPU backends, and an expert-guided autotuning to incorporate a set of expert knowledge on modern GPU architecture to guide large design space exploration. Extensive experiments show that Faith achieves up to 3.4× speedup (2.6× on average) over state-of-the-art frameworks.

Chapter 2

APNN-TC: Accelerating Arbitrary-Precision Neural Networks on Tensor Cores

In this chapter, we present APNN-TC to accelerate Arbitrary-Precision Neural Networks (APNN) on Tensor Cores (this work [27] has been published in SC 2021). Over the years, accelerating neural networks with quantization has been widely studied. Unfortunately, prior efforts with diverse precisions (e.g., 1-bit weights and 2-bit activations) are usually restricted by limited precision support on GPUs (e.g., int1 and int4). To break such restrictions, we introduce the first framework, APNN-TC, to fully exploit quantization benefits on Ampere GPU Tensor Cores. Specifically, APNN-TC first incorporates a novel emulation algorithm to support arbitrary short bit-width computation with int1 compute primitives and XOR/AND Boolean operations. Second, APNN-TC integrates arbitrary precision layer designs to efficiently map our emulation algorithm to Tensor Cores with novel batching strategies and specialized memory organization. Third, APNN-TC embodies a novel arbitrary precision NN design to minimize memory access across

layers and further improve performance. Extensive evaluations show that APNN-TC can achieve significant speedup over CUTLASS kernels and various NN models, such as ResNet and VGG.

2.1 Problem Statement

Over the recent years, demands to improve the performance of deep neural networks (DNNs) have never been satisfied. Prior work approaches faster and more efficient DNNs from different aspects, such as model pruning [37, 38, 39, 40], kernel factorization [41, 42, 43, 44], and data quantization [45, 46]. Among those efforts, quantization-based DNN acceleration [47, 45, 46] finds its strengths in minimum modification of the original model architecture, lower memory consumption, and better runtime performance.

To accelerate quantized DNNs, many specialized cores have been introduced to support low-precision dense matrix-matrix multiplications, such as Tensor Processing Units (TPUs) [48], Neural Network Processors (NNPs) [49], and GPU Tensor Cores [50]. For example, NVIDIA introduces Tensor Cores in Volta architecture [51] that support `FP16` matrix-matrix multiplication. In Turing architecture, NVIDIA extends architecture support for more precisions (*e.g.*, `int1` and `int4`) and bit-level operations (*e.g.*, `XOR`) [52]. Recently in the Ampere architecture, we find there is additional support for more precision and bit-level operations (*e.g.*, `AND`). However, these specialized cores still support a limited range of precisions with only architecture-level efforts, while quantized DNNs usually require arbitrary precisions (*e.g.*, 1-bit weight and 2-bit activations). In this chapter, our key question is *whether we can support arbitrary precision neural networks with the limited precisions on Tensor Cores.*

We identify two major challenges in accelerating arbitrary precision DNNs on Ampere GPU Tensor Cores.

Lack of mathematical emulation design. To support arbitrary precisions (*e.g.*, `int1` weights and `int2` activations), one naive approach is to represent these low-precision values with the supported high-precision values (*e.g.*, `int4`). However, this approach introduces extra overhead and prevents efficient quantized DNNs on Tensor Cores. Another approach is to emulate with `int1` compute primitives. However, with `int1` precision, Tensor Cores only support two bit-level operations (*i.e.*, `XOR` and `AND`) and mathematical emulation designs are required to support multiplication and addition in quantized DNNs. Moreover, quantized DNNs may have diverse input data (*e.g.*, -1/+1 or 0/1), where different data may require different emulation designs.

Lack of efficient implementation for arbitrary precision NN layers. To accelerate APNN on Tensor Cores, we need to efficiently map arbitrary precision NN layers to Tensor Cores with specialized compute primitives and memory architectures. Existing works on accelerating binary neural networks simply split NN layers into small matrix tiles (*e.g.*, 8×8) to match Tensor Core compute primitives and improve the parallelism. However, naively borrowing these strategies fails to exploit the data locality during NN layer computation especially for our emulation workload. Moreover, arbitrary precision computation usually computes at the bit-level (*e.g.*, `int3` or `int5`) while existing hardware devices such as CPUs and GPUs usually operate at the word or byte level. Specialized bit operations and data organization are required to support efficient bit-level computation and avoid uncoalesced memory access.

Lack of efficient NN framework designs. One standard approach to build quantized neural networks is to stack a sequence of NN layers, such as a convolution layer followed by a pooling layer and a quantization layer. However, this approach ignores the data reuse opportunity across NN layers and leads to unnecessary memory overhead. For example, on NNs with n 2-bit activations, there are two semantic equivalent implementations – quantization after reading 32-bit activations from the previous layer or

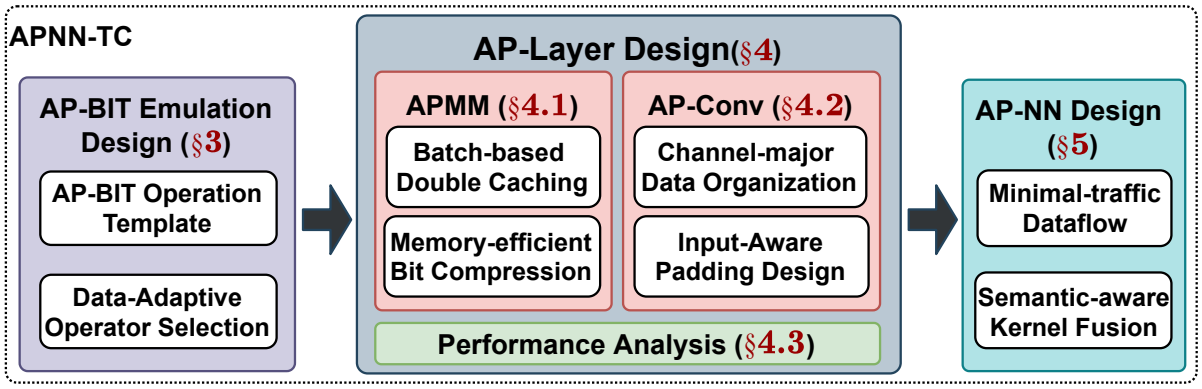


Figure 2.1: The overview of APNN framework.

quantization to 2-bit ones before writing to global memory for the next layer. While these two implementations provide the same semantic, the former requires memory access of $32n$ bits while the latter only requires memory access of $2n$ bits.

2.2 Overview of Proposed Solution

To this end, we propose APNN-TC to accelerate Arbitrary Precision Neural Networks on Ampere GPU Tensor Cores, as illustrated in Figure 2.1. First, we propose an *AP-BIT emulation design* to support arbitrary-precision computation with 1-bit compute primitives. Our AP-BIT algorithm can adaptively select operators (*e.g.*, `XOR` or `AND`) to support diverse input data (*e.g.*, -1/1 or 0/1). Second, we build efficient *AP-Layer design* including an arbitrary-precision matrix-matrix multiplication (APMM) layer for fully connected layers and an arbitrary-precision convolution (APConv) layer for convolution layers. We propose a set of memory and computation designs (*e.g.*, batch-based double caching and channel-major data organization) to fully exploit Tensor Core computation and minimize memory access. We also incorporate a performance analysis to automatically tune the hyper-parameters in APMM and APConv. Third, we propose

an efficient *APNN design* to improve the performance at the framework level. It includes a minimal-traffic dataflow to support various precisions over APNN layers and a semantic-aware kernel fusion to minimize the data movement across layers.

Extensive experiments show that APNN-TC can achieve up to 3.78× speedup over CUTLASS kernels and 3.08× speedup over CUBLAS kernels. APNN-TC can also consistently outperform NNs implemented with built-in int8, half, or single precision. For example, with 2-bit weights and 8-bit activations, APNN-TC can achieve more than 4× latency reduction and 3× higher throughput than the single-precision NN with only 2% accuracy drop.

2.3 Related Works

2.3.1 APNN algorithm designs

Arbitrary precision (lower than INT8) neural network (APNN) algorithms have been widely studied [53, 52, 54, 22, 24, 23, 55, 56, 57] to fully explore the spectrum of NN performance and NN accuracy and cater to diverse application requirements. In addition to widely supported precisions on modern GPUs (*e.g.*, `int1`, `int4`, and `int8`), these APNNs usually utilize more diverse precisions such as `int2`, `int3`, and `int5`. APNNs may also have different precisions for weights and activations (*e.g.*, 1-bit weights and 2-bit activations). Comparing with INT8 quantized neural networks, APNNs provide better performance and memory efficiency at the cost of (slightly) degraded accuracy. Popular APNNs include DoReFa-Net [22] for 1-bit weights and 2-bit activations, LQ-Nets [24] for 1-4 bits, HAQ [23] for 1-8 bits, OLAccel [55] for 4 bits, BSTC [56] and TCBNN [52] for 1 bits. In this chapter, we follow LQ-Nets [24] that starts from a full-precision NN and adopts the quantization error minimization (QEM) strategy to generate quantized NNs.

2.3.2 APNN Hardware Supports

While many APNN algorithms have been designed, the hardware supports are still limited. One direction is to build FPGA and ASIC based implementations [23, 55] to demonstrate the performance benefits of APNNs. However, these implementations usually require specialized hardware designs to support arbitrary-precision computation and cannot be applied to GPUs. Another direction is to utilize built-in precisions on GPUs for quantized neural networks. Taking the most famous Pytorch [58] framework as an example, it supports FP32, FP16, and BF16 models on GPUs and int8 quantization on x86 CPUs with AVX2 support. Recently, BSTC [56] and BTC [52] accelerates binary neural networks on GPUs by exploiting the int1 compute primitive. However, existing works can only build on the limited precision supported on GPUs (*e.g.*, `int1`, `int4`, and `int8`) and cannot fully exploit the performance benefits from APNNs. In this chapter, we build the first generalized framework to accelerate arbitrary-precision neural networks on Ampere GPU Tensor Cores.

2.3.3 Tensor Cores

Tensor Cores are specialized cores for accelerating neural networks in terms of matrix-matrix multiplications. Tensor Cores are introduced in recent NVIDIA GPUs since Volta architecture [59]. Different from CUDA Cores that compute scalar values with individual threads, Tensor Cores compute at the matrix level with all threads in a warp [60]. For example, the 1-bit Tensor Core compute primitive takes two `int1` input matrices A and B of shape 8×128 and generates an `int32` output matrix C of shape 8×8 [52]. In Volta architecture, Tensor Cores support only half-precision computation [61]. To support more quantized neural networks, Tensor Cores add more precisions including `int1`, `int4`, and `int8` in Turing architecture [62]. Regarding `int1` precision, Tensor Cores support only

`XOR` logical operation in Turing architecture and recently add `AND` logical operation in Ampere architecture [63]. Despite these hardware efforts on supporting more precisions, arbitrary precisions are still not supported. This is the first work to support arbitrary precision computation on Ampere GPU Tensor Cores with `int1` precision and support for both `XOR` and `AND` operations.

2.4 AP-Bit Emulation Design

In this section, we design an AP-BIT emulation on Tensor Cores to support arbitrary-precision computation. We first design an AP-Bit operation template that supports arbitrary-precision computation with 1-bit compute primitive on Tensor Cores. Then, we propose a data adaptive operator selection to automatically support various input data (*e.g.*, -1/+1 and 0/1) with bitwise `XOR` and `AND` on Tensor Cores. Here, we focus on the algorithm design on small matrices (*i.e.*, input matrices of 8×128 and output matrix of 8×8) that can fit directly on Tensor Core compute primitives. We will discuss the efficient computation of large matrices in the next section.

2.4.1 AP-Bit Operation Template Design

The AP-Bit operation template takes a matrix W with p-bit elements and a matrix X with q-bit elements, and computes with 1-bit operations on Tensor Cores to generate a 32-bit output matrix $Y = WX$. Our key observation is that each arbitrary-bit scalar digit can be decomposed to a sequence of 1-bit scalar digits and the arbitrary computation can be conducted with only 1-bit operations and shift operations. Formally, to support scalar-level arbitrary precision computation wx of a 1-bit weight w and a 2-bit feature $x = x^{(1)}x^{(0)}$ with $w, x^{(i)} \in$ `int1`, we can first decompose 1-bit values $x^{(1)}$ and $x^{(0)}$ from

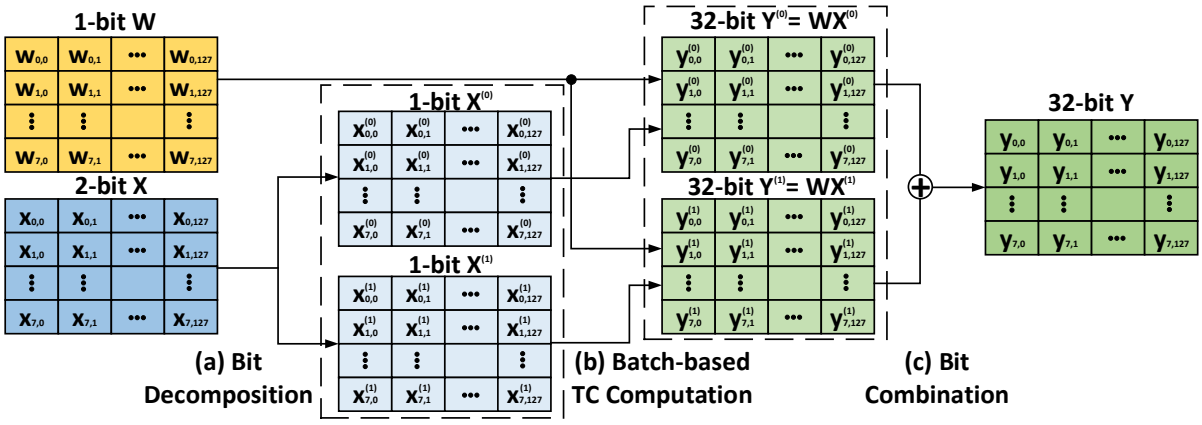


Figure 2.2: Illustration of AP-Bit Operation Template with 1-bit weight W and 2-bit feature X, which can be generalized to arbitrary weight bits and feature bits. Note that $X^{(0)}$ and $X^{(1)}$ in the dashed box are batched into a single large matrix during computation, which will be discussed in Section 2.5.1.

the 2-bit feature x as:

$$x^{(1)} = (x \gg 1)\&1, \quad x^{(0)} = (x \gg 0)\&1$$

Suppose we have an 1-bit operation $OP(a, b)$ (*e.g.*, the `bmma` API of Tensor Cores) that takes 1-bit inputs and generate 32-bit outputs, we can compute wx as

$$wx = OP(w, x^{(1)}) * 2 + OP(w, x^{(0)})$$

We illustrate our AP-Bit operation template in Figure 2.2. Here, we focus on a 1-bit weight matrix W of shape 8×128 and a 2-bit feature matrix X of shape 8×128 to illustrate our algorithm design. A naive approach is to use 4-bit integers to represent each 1-bit element $w_{i,j}$ and 2-bit element $x_{i,j}$, and then use the $int4$ compute primitive on Tensor Cores. However, this approach would lead to unnecessary memory and computation overhead. Instead, we propose to exploit the $int1$ compute primitive on Tensor Cores to support arbitrary-precision computation by dynamically adjusting the memory and computation requirement. In particular, the first step is to conduct **bit decomposition**

by splitting a 2-bit $x_{i,j}$ to two 1-bit elements $x_{i,j}^{(0)}$ and $x_{i,j}^{(0)}$:

$$x_{i,j}^{(1)} = (x_{i,j} \gg 1)\&1, \quad x_{i,j}^{(0)} = (x_{i,j} \gg 0)\&1$$

These 1-bit elements are then packed into 1-bit matrix X^0 and $X^{(1)}$. The second step is to conduct **batch-based Tensor Core computation** on these 1-bit matrices with the `bmma` API and generate 32-bit output matrices

$$Y^{(0)} = \texttt{bmma}(W, X^{(0)}), \quad Y^{(1)} = \texttt{bmma}(W, X^{(1)})$$

These matrices can be computed directly with the `bmma` API since all of them have the shape of 8×128. We also note that Tensor Core primitives for `int1`, `int4`, and `int8` generate 32-bit output matrices to accumulate a large number of bit-operation outputs and avoid overflow. The third step is to conduct **bit combination** and generate the final output matrix Y

$$Y_{i,j} = Y_{i,j}^{(1)} * 2 + Y_{i,j}^{(0)} \tag{2.1}$$

Here, $Y_{i,j}$, $Y_{i,j}^{(1)}$ and $Y_{i,j}^{(0)}$ refer to the $(i,j)^{th}$ scalar elements of matrix Y, $Y^{(1)}$ and $Y^{(0)}$, respectively. For notation simplicity, we abbreviate Equation 2.1 as $Y = Y^{(1)} * 2 + Y^{(0)}$ in the following sections to represent the scalar multiplication and elementwise addition. We note that $Y = WX$ mathematically.

It is not hard to see that this computation can be generalized to matrices with arbitrary bits p and q. Formally, given a p-bit weight matrix W and a q-bit weight matrix X, we can first decompose into 1-bit matrices $W^{(s)}, s \in \{0, 1, ..., p-1\}$ and $X^{(t)}, t \in \{0, 1, ..., q-1\}$. For each element, we have

$$w_{i,j}^{(s)} = (w_{i,j} \gg s)\&1, \quad x_{i,j}^{(t)} = (x_{i,j} \gg t)\&1 \tag{2.2}$$

Then, we compute the `bmma` API for pq times for each combination of s and t:

$$Y^{(s,t)} = \texttt{bmma}(W^{(s)}, X^{(t)})$$

Finally, we conduct bit combination to generate the 32-bit output matrix Y:

$$Y = \sum_{s=0}^{p-1} \sum_{t=0}^{q-1} Y^{(s,t)} * 2^{s+t}$$

Cost Analysis. The cost of arbitrary-precision computation comes from three parts: bit decomposition, tensor core computation, and bit combination. Given a p-bit weight matrix and a q-bit data matrix of shape $n \times n$, bit decomposition shows complexity of $O((p+q)n^2)$ since we need $O(pn^2)$ operations to split each p-bit element from A into p 1-bit elements and another $O(qn^2)$ operations to split each q-bit element from B into q 1-bit elements. The bit combination shows complexity of $O(pqn^2)$, since we have pq matrices $Y^{(s,t)}$ of shape $n \times n$ and need to add elementwisely. This overhead is negligible compared with the $O(n^3)$ complexity in the Tensor Core computation. Note that only 1-bit compute primitives are used for this expensive matrix-matrix multiplication, which significantly reduces the overall latency.

2.4.2 Data Adaptive Operator Selection

While we compute with bit-0 and bit-1 in arbitrary-precision computation, these two values may actually encode diverse values. For example, the 1-bit weight matrix in neural networks may encode -1 and 1, instead of 0 and 1, in order to improve the accuracy of neural networks. In this case, bit-0 indicates the value -1 and bit-1 indicates the value 1. To support this diversity in the encoded data, we introduce *data adaptive operator selection* by adopting different bit operations in Tensor Cores (*i.e.*, `XOR` and `AND`). In particular, we support three cases, where we first conduct bit operations and then accumulate with `popc` (*i.e.*, population count [64] that counts the number of set bits). The *Case-I* is that both W and X encode 0 and 1, where we choose logical `AND` operation. For example, given a 1-bit vector $W = [0, 1]$ and a 1-bit vector $X = [1, 1]$, we

use `AND` operation to compute as

$$WX = \texttt{popc}(\texttt{AND}([0,1],[1,1])) = \texttt{popc}([0,1]) = 1$$

The *Case-II* is that both W and X encodes -1 and $+1$, where we select logical XOR operation. For example, given two 1-bit vectors $W = [-1, 1]$ and $X = [1, 1]$, we first map -1 to 0 and compute as

$$WX = n - 2 * \texttt{popc}(\texttt{XOR}([0,1],[1,1])) = n - 2 * \texttt{popc}([0,1]) = 0$$

Here, $n(=2)$ is the length of the vector.

The *Case-III* is that W encodes -1 and $+1$, while X encodes 0 and 1. For example, we may need to compute the multiplication of two 1-bit vectors $W = [-1, 1]$ and $X = [1, 0]$. This case happens frequently in neural networks with a 1-bit weight matrix W and a q-bit feature matrix X with $q > 1$. In this case, naively adopting `XOR` or `AND` does not work, since there are three values -1, 0, and 1 that cannot be easily encoded with 1 bit. To this end, we incorporate a linear transformation on W and compute with only `AND` operation. Our key observation is that W can be transformed into a vector with only 0 and 1 by adding a constant vector $\mathbf{J}_2 = [1, 1]$:

$$\hat{W} = \frac{W + \mathbf{J}_2}{2} = [0, 1]$$

Then, we compute $\hat{W}X = 0$ with `AND` operation as Case-I. Finally, we recover the value WX by another linear transformation:

$$WX = 2\hat{W}X - \mathbf{J}_2 X = 2 * 0 - 1 = -1$$

Note that $\mathbf{J}_2$ is a constant vector that can be cached in Tensor Core fragment and does not introduce extra memory overhead.

2.5 Arbitrary Precision Layer Design

In this section, we propose the Arbitrary-Precision Matrix Multiplication (APMM) for fully connected layers and Arbitrary-Precision Convolution (APConv) for convolution layers.

2.5.1 Arbitrary-Precision Matrix Multiplication

Arbitrary-Precision Matrix Multiplication (APMM) takes the decomposed 1-bit weight matrix $W^{(s)}$, $s \in \{0, ..., p-1\}$, the decomposed 1-bit feature matrix $X^{(t)}, t \in \{0, ..., q-1\}$, and computes output matrix $Y = \sum_{s=0}^{p-1} \sum_{t=0}^{q-1} Y^{(s,t)} * 2^{s+t}$. By default, APMM generates 32-bit output to avoid data overflow for large matrices and match the 32-bit output in Tensor Core compute primitives. APMM also supports arbitrary-precision output (*e.g.*, `int2`) when APMM is used as a hidden layer in neural networks (NNs) and the output is consumed by the next APMM-based NN layer.

Considering that APMM essentially computes an arbitrary precision GEneral Matrix-Matrix multiplication (GEMM) kernel with multiple Binary Matrix-MAtrix multiplication (BMMA) kernels, one naive strategy is to build upon existing BMMA kernels [52, 56]. In particular, we can use existing BMMA kernels to multiply each pair of $W^{(s)}$ and $X^{(t)}$ and accumulate $W^{(s)}X^{(t)}$ to the output matrix Y. However, this approach shows significant inefficiency due to two reasons. First, this approach ignores the data reuse opportunity since the same weight matrix tile from $W^{(s)}$ can be multiplied with different feature matrix tiles from X_{t1} and X_{t2}. Second, this approach requires extra communication across BMMA kernels, such that reducing $W^{(s)}X^{(t)}$ into Y leads to significant global memory access. We show our efficient APMM design in Figure 2.3. It includes a *batch-based double caching* to facilitate the data reuse and a *memory-efficient bit combination* to accelerate the accumulation and optionally generate the arbitrary-precision output.

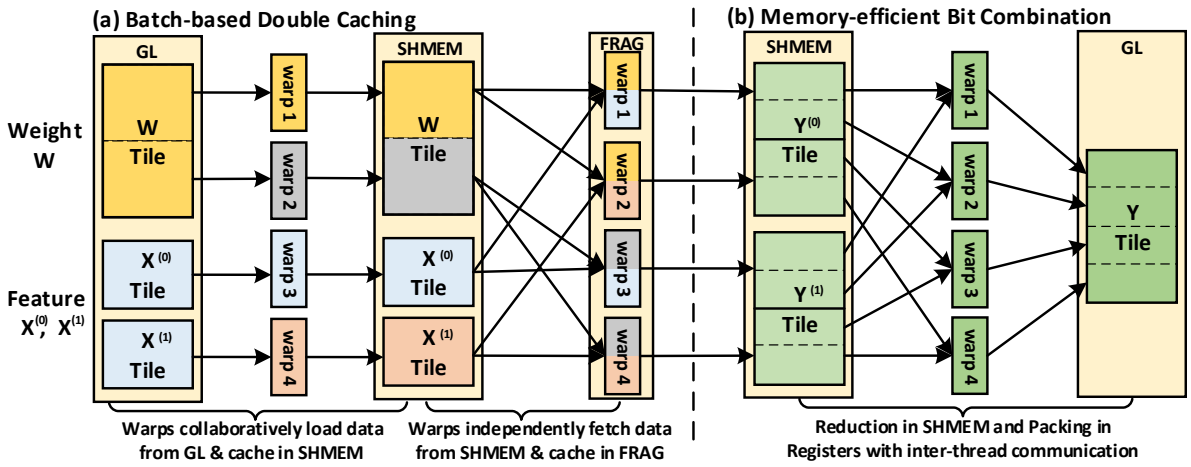


Figure 2.3: Illustration of APMM. GL: GLobal memory. SHMEM: SHared Memory. FRAG: FRAGment.

Here, we illustrate the design with 1-bit W and 2-bit X for notation simplicity while arbitrary-precision W and X are supported.

(a) Batch-based Double Caching. Batch-based double caching exploits two GPU memory hierarchies (*i.e.*, shared memory and fragment located in registers) to cache matrix tiles and facilitate data reuse in APMM computation, as illustrated in Figure 2.3(a). Considering the limited size of shared memory and fragment, we tile weight matrices $W^{(s)}$ and feature matrices $X^{(t)}$ such that these tiles can be cached in GPU memory hierarchies. Formally, given $W^{(s)}$ of shape $M \times K$ and $X^{(t)}$ of shape $N \times K$, we first tile $W^{(s)}$ along the M dimension into block matrix tiles of shape $b_m \times b_k$. Similarly, we tile $X^{(t)}$ along the N dimension into block matrix tiles of shape $b_n \times b_k$. Here, each GPU block will multiply one pair of block matrix tiles and generate an output matrix tile of shape $b_m \times b_n$. Considering that Tensor Cores compute at the warp level, we further tile $W^{(s)}$ into warp matrix tiles of shape $w_m \times w_k$ and X_s into $w_n \times w_k$ such that each warp computes an output tile of shape $w_m \times w_n$. To match with the $8 \times 8 \times 128$ `bmma` compute primitive of Tensor Cores, each warp will slide along w_m, w_n, and K dimension during

computation. Note that these tiling sizes have a significant impact on the performance, which will be analyzed in Section 2.5.3.

Batch-based double caching first adopts a batch strategy to improve inter-thread parallelism and achieve high performance. Existing works on binary neural networks [56, 52] report that the GEMM size in NN workload is usually small (*e.g.*, 512×512) and use small matrix tiling sizes (*e.g.*, 32×32) to improve the inter-thread parallelism. However, this approach leads to low intra-thread parallelism and prevents data reuse. Instead, our batch strategy virtually transforms multiple small BMMAs into a large BMMA. In particular, given $W^{(s)}, s \in \{1, ..., p-1\}$ of shape $M \times K$ and $X^{(t)}, t \in \{1, ..., q-1\}$ of shape $N \times K$, we batch these small matrices into W_B of shape $pM \times K$ and X_B of shape $qN \times K$ and compute using a single large BMMA. Here, we implement a "virtual" batch strategy during the data loading procedure by dynamically deciding the global memory address of the corresponding matrix tile such that no additional memory movement is involved.

Batch-based double caching then exploits two GPU memory hierarchies to facilitate data reuse at different levels. The first level is shared memory caching to reuse matrix tiles from $W^{(s)}$ and $X^{(t)}$. Here, a naive strategy is that each warp independently loads a weight tile and a feature tile for computation. However, we observe that the same weight tile may be multiplied with feature tiles from different 1-bit feature matrices $X^{(0)}$ and $X^{(1)}$, as illustrated in Figure 2.3(a). To this end, our design requires all warps to first collaboratively load $b_m \times b_k$ weight data and $b_n \times b_k$ feature data from global memory to shared memory. Then, each warp fetches its own matrix tiles from shared memory. In this way, we can significantly reduce global memory access by exploiting fast shared memory.

The second level is fragment caching to continuously store output tiles in the same Tensor Core fragment. Since Tensor Core compute primitives require to accumulate in

32-bit Tensor Core fragments, the output tiles usually consume a large memory space compared with the 1-bit input data. Moving output tiles between shared memory and Tensor Core fragment may lead to heavy shared memory access. Moreover, existing dissecting works [62, 61] reveal that fragment is composed of registers and one GPU block of 8 warps can provide up to 256 KB Fragment, which is much larger than shared memory. To this end, as iterating through the K dimension during computation, we continuously use multiple fragments to cache output tiles of shape $b_m \times b_n$ for reducing shared memory access and caching more feature and weight tiles in shared memory.

(b) Memory-efficient Bit Combination. Bit combination consumes 32-bit BMMA outputs $Y^{(s,t)} \in \texttt{int32}^{M \times N}$ and generates 32-bit APMM outputs $Y \in \texttt{int32}^{M \times N}$ as $Y = \sum_{s=0}^{p-1} \sum_{t=0}^{q-1} Y^{(s,t)} * 2^{s+t}$. 'Bit combination can also generate arbitrary precision output when it is utilized as a NN hidden layer and its output is consumed by the next NN layer. Overall, bit combination takes only $O(pqMN)$ computation complexity, which is significantly lower than the computation complexity of GEMM operations. However, there are two potential memory bottlenecks in bit combination, which have a significant performance impact. The first one is global memory access when reducing 32-bit BMMA outputs to 32-bit APMM outputs. In a naive implementation that independently conducts BMMA and bit combination, bit combination usually introduces similar latency as the BMMA kernel. The main reason is that, while Tensor Cores provide significantly higher computation throughput than CUDA Cores, the global memory bandwidth remains the same. The second one is the shared memory access when converting 32-bit APMM outputs to arbitrary-precision outputs. In this procedure, we usually need to pack low-bit values (*e.g.*, 2-bit) in registers from different threads to a single memory-aligned value (*e.g.*, 32-bit) before storing to global memory. Relying on shared memory for data exchange across threads may lead to heavy shared memory access.

Memory-efficient bit combination includes two novel designs to mitigate memory over-

head. The first design includes a semantic-aware workload allocation and an in-shared-memory reduction. In particular, at the data loading phase of BMMA, we load feature tiles and weight tiles of the same spatial location such that their multiplication outputs can be reduced directly. As illustrated in Figure 2.3, instead of loading a $b_n \times b_k$ feature tile of $X^{(0)}$ or $X^{(1)}$, we load two $0.5b_n \times b_k$ feature tiles of both $X^{(0)}$ and $X^{(1)}$ with the same matrix index. In this way, we can reduce $WX^{(1)}$ and $WX^{(0)}$ directly in shared memory and mitigate global memory access while not degrading the BMMA performance.

The second design incorporates an element-wise routine and an inter-thread communication to pack low-bit values and mitigate shared memory overhead. The element-wise routine is a user-defined interface to provide diverse support of quantization and batch normalization across NN layers. This routine applies to individual 32-bit reduced values in registers. Given a 32-bit value in a register, this routine may quantize it into a p-bit value that is still stored in the 32-bit register with the first $32 - p$ bits as zeros. This routine also includes bit decomposition (Equation 2.2) that splits this p-bit value in a register to 1-bit values in p registers. After that, we use a `__ballot_sync` API to enable inter-thread communication and directly pack the 1-bit values across threads into 32-bit values that can be stored to the global memory.

2.5.2 Arbitrary-Precision Convolution (APConv)

APConv takes the decomposed 1-bit weight matrix $W^{(s)}$ of shape $C_{out} \times C_{in} \times K \times K$, the decomposed 1-bit feature matrix $X^{(t)}$ of shape $BS \times C_{in} \times Height \times Width$, and generates output matrix Y. Here, C_{out} is the number of output channels, C_{in} is the number of input channels, K is the kernel size, BS is the batch size. Existing works on bit-level convolution usually adopt a direct convolution design [56, 52] to improve the GPU utilization. However, these methods ignore the data reuse opportunity and

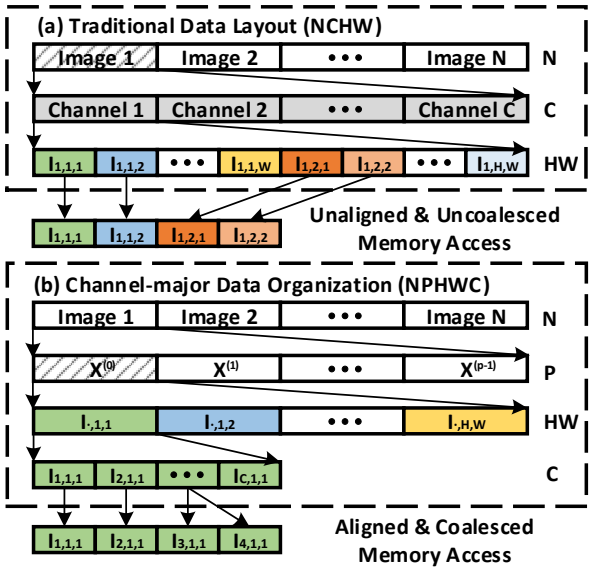


Figure 2.4: Illustration of Channel Major Data Organization (NPHWC). P indicates the number of bits. I_{chw} indicates the image pixel at the c-th channel, h-th height, and w-th width.

introduce heavy global memory access. In addition, APConv on a p-bit weight and a q-bit feature usually has pq times workload than the BConv on the same weight and feature size, which can easily contribute to high GPU utilization. To this end, APConv incorporates the batch-based double caching design as APMM to mitigate the global memory access. However, there are still two key challenges that distinguish APConv from APMM. The first is the data organization where naively reading the $K \times K$ feature map may easily lead to un-coalesced memory access. The second is the data padding where simply padding zeros may lead to erroneous results. To tackle these challenges, we propose *channel-major data organization* and *input-aware padding design*.

(a) Channel-Major Data Organization. Channel-major data organization transforms un-coalesced and unaligned memory access to a coalesced and aligned one for improving performance. Traditional data organization for 32-bit convolution usually employs a NCHW design, as illustrated in Figure 2.4(a). However, naively borrowing this design to APConv leads to un-aligned and un-coalesced memory access due to two rea-

sons. First, multiple P-bit (*e.g.*, 3-bit) elements usually cannot be packed into an aligned 32-bit element, which is required for valid GPU reads and writes. Using a 32-bit element to store a P-bit element will introduce extra memory overhead. Second, convolution operations usually read only K continuous elements (or KP bits) due to the $K \times K$ kernel size, which may lead to un-coalesced memory access.

We design a channel-major data organization as illustrated in Figure 2.4(b). There are two key design choices. First, we split a P-bit feature matrix into P 1-bit feature matrices and store each 1-bit feature matrix consecutively. In this way, we can provide aligned memory access for each 1-bit feature matrix and support arbitrary precision P. Second, we consecutively store all channels of elements with the same spatial location. Since convolution layers usually have $128C, C \in \mathbb{N}$ channels, this usually leads to coalesced memory access during computation.

(b) Input-aware Padding Design. Input-aware padding design adaptively adjusts padding values according to input values. As mentioned in Section 2.4.2, when the weight W encodes -1 and 1 with 0 and 1, we cannot naively padding 0 since 0 represents -1.

We propose three padding strategies according to the input data. First, when both weight and feature encode 0 and 1, we simply pad zeros for features. In this case, padding 0 for features will only add extra 0's for arbitrary weight values, which does not change the computation result. Second, when both weight and feature encode -1 and 1, we pad 1 for features and use an extra `counter` flag to track the number of 0's when the convolution weight moves outside the input image frame. We will subtract `counter` to amend the corresponding convolution results. Third, when weight encodes -1 and 1 and feature encodes 0 and 1, we pad 0 to features and do not change the convolution results.

2.5.3 Performance Analysis

In our APNN-TC kernel design, there are six tuning knobs – the block tiling sizes b_m, b_n, b_k, and the warp tiling sizes w_m, w_n, w_k. These tiling sizes bring a trade-off between the Thread-Level Parallelism (TLP) and the Instruction Level Parallelism (ILP), especially the compute intensity (CI). Here, we focus on block tiling sizes, since we empirically observe that utilizing 8 warps per block and splitting the block workload evenly across warps provide the best performance (*i.e.*, $w_m = b_m/4$, $w_n = b_n/2$, $w_k = b_k$). In this subsection, we first analyze the performance impact of individual tuning knobs. Then, we propose an autotuning strategy to maximize the performance. Since APMM and APConv share the same batch-based double caching strategy, we use the same autotuning strategy for these two kernels.

Performance Model TLP refers to the thread-level parallelism in terms of the number of threads in use. Intuitively, larger TLP can improve GPU utilization and kernel performance [65, 66]. Formally, given a p-bit weight matrix of shape $M \times K$, a q-bit feature matrix of shape $K \times N$ and the matrix tiling size $b_m \times b_n$, we define the TLP as

$$TLP = \frac{pM \times qN}{b_m \times b_n} \tag{2.3}$$

We ignore the number of threads for each block since it is a constant in our evaluation. Intuitively, smaller $b_m \times b_n$ may improve TLP, which suggests a small $b_m \times b_n$ especially for small matrices.

Compute intensity (CI) refers to the ratio of computation over memory access on each thread block. We aim to improve CI for two reasons. First, a higher CI indicates less memory access and better performance. While the amount of computation remains the same, the amount of memory access may be reduced significantly by data reusing and hyper-parameter tuning. Second, a higher CI on a thread block provides more

opportunities for latency hiding. Formally, for a matrix tile, we compute the amount of global memory access as $b_m \times b_k + b_n \times b_k$ when reading a $b_m \times b_k$ weight tile and a $b_m \times b_k$ feature tile. We compute the amount of computation as $2 \times b_m \times b_n \times b_k$ from the matrix-matrix multiplication. Finally, we compute CI as

$$CI = \frac{2 \times b_m \times b_n}{b_m + b_n} \tag{2.4}$$

Note that CI can be increased when b_m and b_n are increased. We also observe that CI is independent of b_k such that we can use smaller b_k to leave space for larger b_m and b_n, especially when the shared memory size is a limiting factor. In our evaluation, we fix b_k as 128 by default.

Auto-tuning During APNN-TC kernel design, there is a large search space on the complex interaction between matrix size (M, N, and K), weight bit p, feature bit q, and block tiling size b_m and b_n. Note that the selected parameters may also be different on various GPUs according to computation and memory capabilities. To this end, we propose a heuristic algorithm to provide a faster search procedure in this large search space. Formally, given the matrix size M, N, K, the weight bit p, the feature bit q, the algorithm selects $b_m, b_n \in \{16, 32, 64, 128\}$ in two steps. First, we compute the TLP of each combination of b_m and b_n. We put these combinations in a priority queue, where a higher TLP leads to a high priority. Second, we pop individual combinations in the priority queue. We stick to the first combination with the highest TLP if its TLP is already smaller than a threshold T. Otherwise, we continuously pop and select combinations in the priority queue to improve CI while ensuring TLP is larger than T. We empirically set T as 64 in our evaluation. Note that different block tiling sizes share the same data layout such that there is no overhead when consecutively executing two layers with different block tiling sizes.

2.6 Arbitrary Precision Neural Network Design

In this section, we introduce our Arbitrary Precision Neural Network (APNN) design. We first introduce a minimal-traffic dataflow on supporting various precisions across layers in APNN. Then, we incorporate a semantic-aware kernel fusion to minimize the memory access across layers.

2.6.1 Minimal-Traffic Dataflow

Given an `int8` RGB image, APNN computes a sequence of NN layers with p-bit weights and q-bit activations and finally generates an `int32` output logits. Here, all intermediate layers compute at arbitrary precision by taking a p-bit weights and q-bit activations and generate 32-bit outputs. Note that the `int1` Tensor Core compute primitive can only generate $int32$ outputs and an extra quantization layer is required to quantizing into q-bit activations for the next layer. For performance consideration, during the initialization of an APNN, we quantize all weights before the model inference computation. To effectively maintain and transfer arbitrary-bit data, we pack the data bit-by-bit for both weight and feature map, following the data organization discussed in Section 2.5.2.

The input layer and the output layer have different precisions from the intermediate layers. As is the common practice with `int8` image inputs, the input layer requires an extra quantization layer that quantizes 8-bit inputs into q-bit activations. The output of the input layer will also be the quantized arbitrary-bit feature map serving as the input for the following intermediate layers. In the output layer, Tensor Core computation results will be directly used for the final softmax logits computation. Thus, we do not apply quantization after the output layer.

2.6.2 Semantic-aware Kernel Fusion

Besides APMM and APConv discussed previously, there are still multiple important layers in APNN, including quantization, Batch Normalization (BN), pooling, and ReLU. Given all scalars $x_{i,j}$ in the i^{th} layer, quantization element-wisely converts `int32` values $x_{i,j}$ to `q`-bit values $y_{i,j}$:

$$y_{i,j} = \lfloor (x_{i,j} - z_i)/s_i \rfloor$$

Here, z_i is a 32-bit scalar zero-point, s_i is the scaling scalar, and $\lfloor \cdot \rfloor$ is the floor function. BN [67] is another major component in NNs for tackling the covariate shift problem and facilitating NN training:

$$y_{i,j} = \frac{x_{i,j} - \mathbb{E}[x_{i,*}]}{\sqrt{Var[x_{i,*} + \epsilon]}} \cdot \gamma_j + \beta_j \tag{2.5}$$

where $\mathbb{E}$ and Var are expectation and variance across the batch, γ_j and β_j are two learned parameters. Pooling splits the feature map spatially into $k \times k$ grids and generates 1 scalar output for each grid by computing the average or the maximum value in each grid. ReLU takes individual input values $x_{i,j}$ and generates output values $y_{i,j} = max(x_{i,j}, 0)$.

While these operations have linear time complexity to the size of feature maps and consume significantly less computation than APConv and APMM kernels, these operations may still introduce heavy latency due to the expensive memory access. Indeed, while Tensor Cores provides significantly improved computation capability, Tensor Cores share the same memory bandwidth with CUDA Cores on GPUs. Moreover, we observe that these values are usually computed element-wisely and do not require heavy communication across GPU threads. We propose a semantic-aware kernel fusion to minimize memory access. We first fuse APMM/APConv with its following quantization, BN, pooling, and ReLU kernels into a single kernel to minimize the global memory access. In particular, these following layers can be seamlessly applied once the convolution results become available at the shared memory. This can improve the computation intensity for

individual convolution kernels meanwhile reducing the global memory access from invoking an additional batch normalization kernel. Second, considering that these following layers usually compute at scalar level, we can further reduce shared memory access by directly reusing values in registers [68]. For example, when a APMM layer is followed by a BN layer, a quantization layer, and a ReLU layer, we directly compute the output scalar as

$$\lfloor max(\frac{x_{i,j} - \mathbb{E}[x_{i,*}]}{\sqrt{Var[x_{i,*} + \epsilon]}} \cdot \gamma_j + \beta_j - z_i, 0)/s_i \rfloor$$

Note that we only need to load a scalar once to a register and avoids unnecessary shared memory access.

2.7 Evaluation

In this section, we evaluate APNN-TC under diverse precisions and show the benefits of arbitrary-precision computation in performance and accuracy.

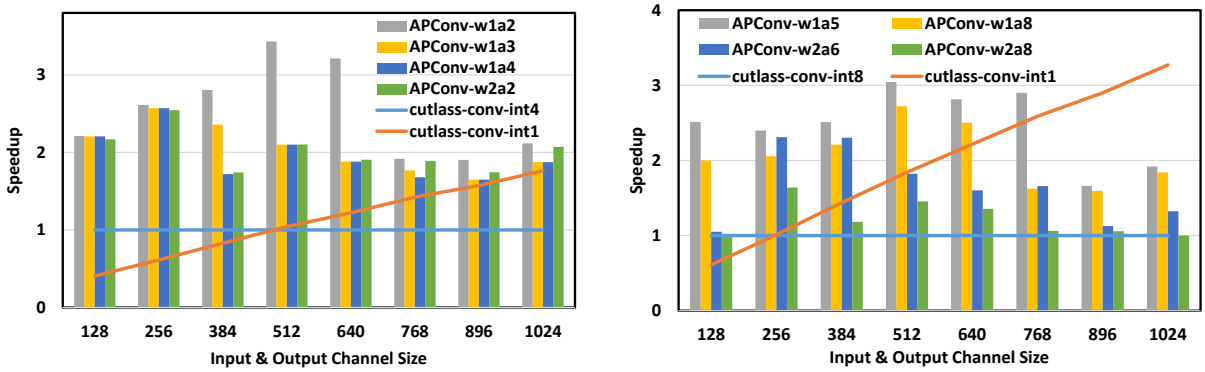
We evaluate on both Nvidia RTX 3090 and Nvidia Tesla A100. The RTX3090 GPU is in a ubuntu 16.04 system with Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz, 64 GB DDR3 DRAM, gcc-7.5.0, and using CUDA-11.1, CUTLASS-2.5, and CUBLAS-11.1. The A100 GPU is in a Linux 3.10.0 system with AMD EPYC 7742 64-core CPU, 1TB DDR4, gcc-9.1.0, and using CUDA-11.1, CUTLASS-2.5, and cuBLAS-11.3. All results reported are the average of 200 times execution.

2.7.1 APLayer Evaluation

APMM Performance We compare our APMM designs with NVIDIA implementations of low-bit gemm (*i.e.*, `int1`, `int4`, and `int8`) that are accelerated by Tensor Cores. For `int8`, we compare with cublas implementation, namely cublass-gemm-int8. Since

`int1` and `int4` are not supported in cublas, we compare with cutlass implementation, namely cutlass-gemm-int1 and cutlass-gemm-int4. Following popular settings in NNs, we compute matrix multiplication of a matrix with shape $B \times K$ and a matrix with shape $K \times N$, where $B = 64$ is a popular batch size and $K = N \in \{128, 256, ..., 1024\}$ covers typical fully connected layer dimensions. According to the precision of our APMM kernel, we name it APMM-wxay, where x indicates the weight bit and y indicates the activation bit. For example, APMM-w1a2 indicates 1-bit weights and 2-bit activations. While our APMM is general to support arbitrary precision, we show 8 popular bit combinations due to page limits. If both weight bits and activation bits are less than 4 (*e.g.*, w1a2, w1a3, w1a4, w2a2), we compare it against cutlass-gemm-int4. If either weight bits or activation bits are larger than 4, we compare it against cublas-gemm-int8. For each matrix size, we show a speedup of cutlass-gemm-int1 against cutlass-gemm-int4 and cublas-gemm-int8 as the performance benefit when sticking to binary neural networks [56, 52]. Since Tensor Core compute primitive supports only 32-bit outputs, all gemm kernels take low-bit input (*e.g.*, `int1`, `int4`, and `int8`) and generate 32-bit outputs.

Figure 2.7 shows the results of APMM on RTX 3090. We compare APMM with cutlass-gemm-int4 in Figure 2.7(a) and cublas-gemm-int8 in Figure 2.7(b). Overall, we have three major observations. First, APMM can usually achieve significant speedup over baselines. For example, APMM-w1a2 can achieve up to 2.35× speedup over cutlass-gemm-int4, while APMM-w5a1 can achieve up to 3× speedup over cublas-gemm-int8. This result demonstrates the performance benefits of emulating arbitrary-precision with `int1` compute primitives over sticking to `int4` or `int8` compute primitives. Second, APMMs with various weight and activation bits usually show similar performance on small matrices. For example, APMM-w1a2, APMM-w1a3, APMM-w1a4, and APMM-w2a2 achieves almost the same speedup when N=128 and N=256, even if these kernels have different computation overhead (*e.g.*, 2× from APMM-w1a2 and 4× from APMM-

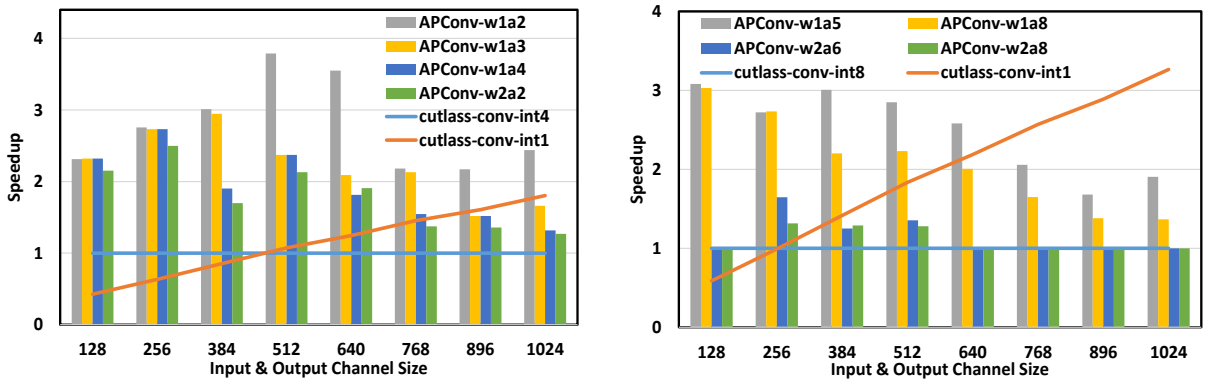


(a) Over CUTLASS-Conv-INT4. (b) Over CUTLASS-Conv-INT8.

Figure 2.5: APConv Performance on RTX 3090.

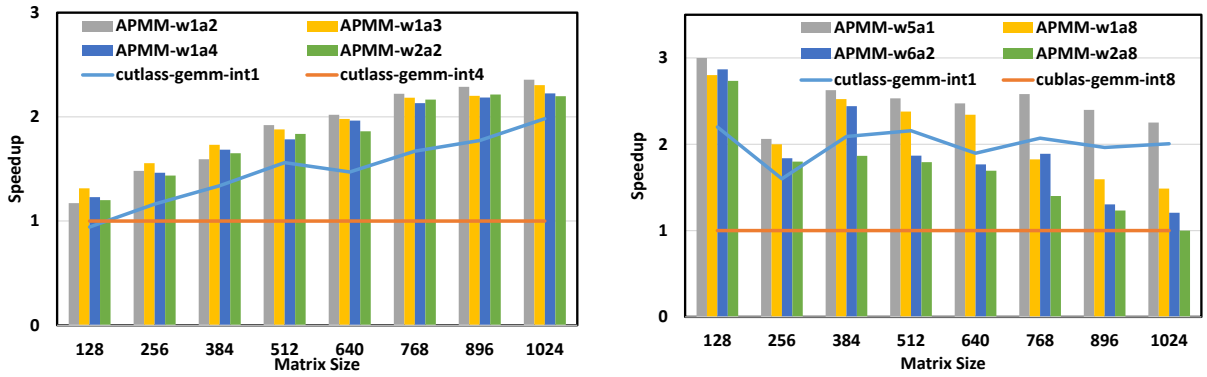
w2a2). This benefit comes from our batch-based double caching (Section 2.5.1(a)), where individual small BMMAs are batched into a large BMMA and computed simultaneously. Surprisingly, our arbitrary precision computation can even outperform cutlass-gemm-int1 in such cases due to the improved GPU utilization. Third, we observe a smaller speedup over cublas-gemm-int8 on large matrix sizes, when peak `int1` performance is achieved. Our investigation shows that, on RTX 3090, cutlass-gemm-int1 is only 5.9× faster than cublas-gemm-int8, such that emulation is slower than built-in int8 compute primitives on large matrices when peak int1 performance is achieved (*e.g.*, $64 \times 1024 \times 1024$ for APMM-w2a8). We argue that NN workload can still benefit significantly from our APMM since the fully connected layers in NNs usually have small matrix sizes (*e.g.*, $1 \times 512 \times 512$ in ResNet-18). We also show the results of APMM on A100 in Figure 2.8 with similar observations.

APConv Performance We compare APConv designs with NVIDIA implementations of low-bit convolution that are accelerated by Tensor Cores. Since cublas does not support `int1`, `int4`, AND `int8` convolution, we use kernels from cutlass. We name these kernels as cutlass-conv-int1, cutlass-conv-int4, and cutlass-conv-int8. Similar to APMM,



(a) Over CUTLASS-Conv-INT4. (b) Over CUTLASS-Conv-INT8.

Figure 2.6: APConv Performance on A100.

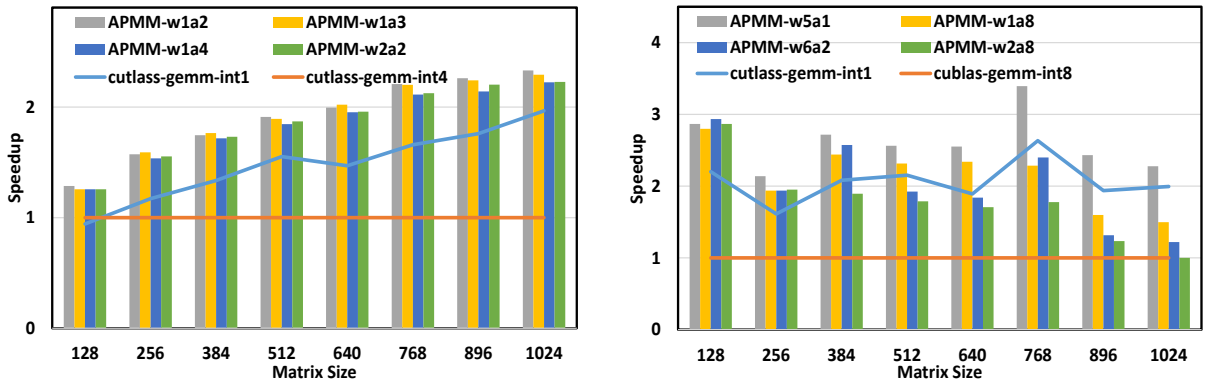


(a) Over CUTLASS-GEMM-INT4 (b) Over CUBLAS-GEMM-INT8

Figure 2.7: APMM Performance on RTX 3090.

we evaluate 8 types of precision with the name APConv-wxay. Since convolution kernels have much more hyperparameters than matrix-multiplication kernels, we show the performance under various input and output channels while fixing the input size as 16 (medium feature size), filter size as 3 (most frequently used), stride as 1 (most frequently used), and batch as 1 (for inference).

Figure 2.5 and 2.6 show the speedup of APConv on RTX 3090 and A100, respectively. APConv can achieve 3.78× speedup over cutlass-conv-int4 and 3.08× speedup over cutlass-conv-int8. This result shows the significant performance benefit from emulating arbitrary precision with `int1` over utilizing `int4` or `int8`. Similar to APMM, we



(a) Over CUTLASS-GEMM-INT4. (b) Over CUBLAS-GEMM-INT8.

Figure 2.8: APMM Performance on A100.

Table 2.1: APNN Evaluation Setting. We list the dataset, network, input size, output size, and the model accuracy under precisions of BNN (*i.e.*, int1), w1a2 (*i.e.*, 1-bit weights with 2-bit activations), and single-precision floating point.

Dataset	Network	Input Size	Output Size	Binary	w1a2	Single
ImageNet	AlexNet [69]	224x224x3	1000	46.1%	55.7%	57.0%
ImageNet	VGG-Variant [70]	224x224x3	1000	53.4%	68.8%	69.8%
ImageNet	ResNet-18 [2]	224x224x3	1000	51.2%	62.6%	69.6%

also observe a smaller speedup over cutlass-conv-int8 on larges channels due to the limitation of peak int1 performance. Since RTX3090 and A100 provide similar performance, we will focus on RTX3090 in the following evaluations.

2.7.2 APNN Evaluation

In this section, we evaluate the overall APNN performance on three mainstream neural network models with ImageNet dataset. The details of our evaluated NN models and their corresponding binarized neural network, low-bit (1-bit weight with 2-bit activation), single-precision accuracy precision are listed in Table 2.1.

We consider two types of configurations for evaluation. In the first setting, we focus on a specific low-bit configuration (1-bit weights and 2-bit activations, *i.e.*, w1a2) across different neural network models. We choose several baselines including neural networks

Table 2.2: APNN Inference Performance on NVIDIA Ampere RTX3090 GPU. Note that latency is measured under a batch of 8 images, throughput is measured under a batch of 128.

	ImageNet-AlexNet		**ImageNet-VGG_Variant**		**ImageNet-ResNet18**	
Schemes	**8 Latency**	**Throughput**	**8 Latency**	**Throughput**	**8 Latency**	**Throughput**
CUTLASS-Single	4.43ms	2.89×10^4fps	25.24ms	3.89×10^2fps	60.96ms	1.51×10^2fps
CUTLASS-Half-TC	3.79ms	3.38×10^4fps	24.19ms	4.67×10^2fps	57.33ms	1.89×10^3fps
CUTLASS-INT8-TC	13.10ms	9.77×10^3fps	25.77ms	6.52×10^2fps	57.09ms	2.85×10^3fps
BNN	0.69ms	1.37×10^4fps	2.17ms	3.91×10^3fps	0.68ms	1.89×10^4fps
APNN-w1a2	0.36ms	2.85×10^4fps	1.66ms	5.32×10^3fps	0.64ms	1.70×10^4fps

built with single-precision floating-point implementation from CUTLASS [71] running on CUDA Cores, half-precision implementation from CUTLASS running on Tensor Cores, INT8 precision implementation from CUTLASS running on Tensor Cores, and the 1-bit binarized neural network running on Tensor Cores based on the state-of-the-art design from [52]. As shown in Table 2.2, our APNN design running on Tensor Cores can achieve a significant speedup compared with CUTLASS INT8, half and single precision implementations. This indicates the practical usage of our APNN design in latency-sensitive applications. Meanwhile, on large batch sizes for throughput performance evaluation, our APNN design also demonstrates its high throughput advantage over these "standardized" bit (*e.g.*, 8-bit and half) precision baselines. Compared with the 1-bit binarized neural network running on Tensor Cores, our APNN design would demonstrate its significant accuracy improvement (an average 11.67%) as listed in Table 2.1. This can demonstrate the application of our APNN design in some application settings, where the BNN model accuracy performance fails to meet the demands. Overall, from the study, we can see that using our APNN design for arbitrary-bit precision computation is a potential way for balancing NN model accuracy and runtime performance.

In the second setting, we shift our focus towards model runtime performance tradeoff on the VGG network. We select several low-bit settings for comparison, including the 1-bit weight with 2-bit activation, 2-bit weight with 2-bit activation, and 2-bit weight with 8-bit activation. As shown in Table 2.3, APNN-TC significantly reduces latency

Table 2.3: Case Study: APNN of VGG on ImageNet.

Scheme	**8 Latency (ms)**	**Throughput (fps)**
Float	25.24	3.89×10^2
Half	24.19	4.66×10^2
INT8	25.77	6.52×10^2
BNN	2.17	3.91×10^3
APNN-w1a2	1.66	5.32×10^3
APNN-w2a2	3.08	2.59×10^3
APNN-w2a8	14.14	5.65×10^2

and improves throughput for w1a2 and w2a2 than INT8 which shows that APNN-TC can bring benefits for many arbitrary-precision computations. Comparing with INT8, APNN-TC with w2a8 shows lower throughput since we need to compute 16 (=2*8) 1-bit matrices to emulate arbitrary-precision computation, which require more computation than w1a2 with 2 1-bit matrices and w2a2 with 4 1-bit matrices. This also matches the performance on individual kernels (e.g., Figure 5, 6, 7, 8). This result indicates that APNN-TC can bring benefits for latency-sensitive applications.

2.7.3 Additional Studies

We perform several additional studies in this subsection, including the latency breakdown from individual NN layers and the benefit from kernel fusion. We show results from RTX 3090 and skip results from A100 since we observe similar trend on these two GPUs.

Latency Breakdown. Figure 2.9 illustrates the percentage breakdown of the latency for the inference of 8 images over three NNs on RTX-3090 GPU. Clearly, the first layer introduces the most delay since the input feature size for this layer is significantly larger than other layers. This percentage can be as high as 80.4% for AlexNet and 47.5% for VGG_Variant. On other layers, we observe a roughly balanced latency.

Benefits from Kernel Fusion. Figure 2.10 investigates the performance benefits

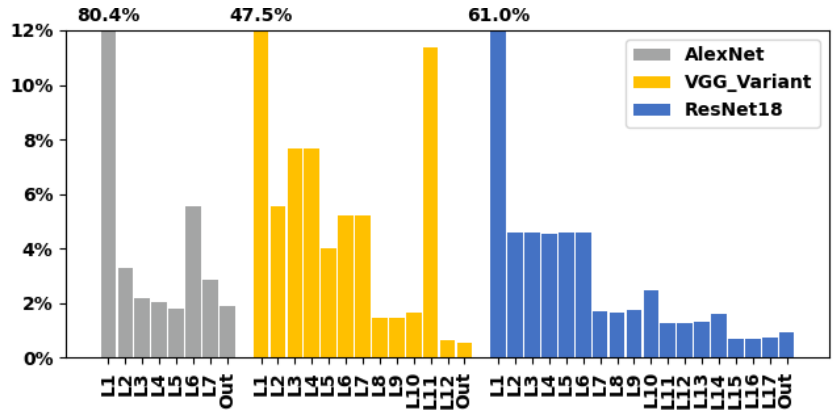


Figure 2.9: Per-layer latency breakdown of APNN models.

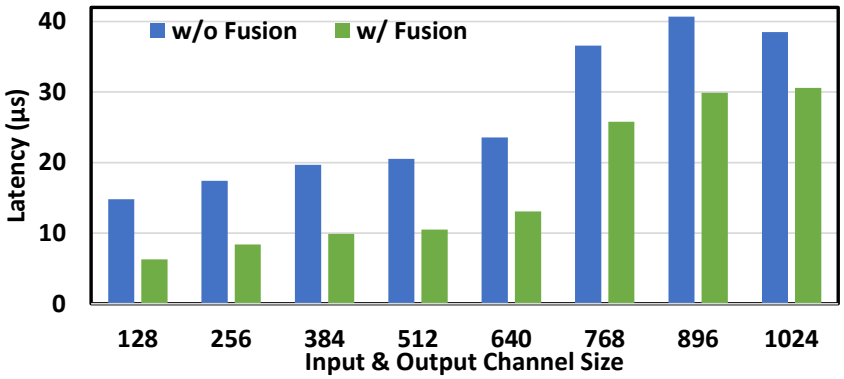


Figure 2.10: Speedup from APNN Kernel Fusion.

from fusing APConv-w1a2, pooling, and quantization into one kernel. Specifically, in the "w/o Fusion" implementation, we implement three global functions for APConv-w1a2 with 32-bit output, 2×2 pooling, and quantizing into 2-bit outputs, respectively. Here, each function read and write data to the global memory. In the "w/ Fusion" implementation, we conduct the same workload in a single kernel. Overall, we observe a latency reduction of 1.77× on average. The main reason is that, in "w/ Fusion", data across APConv, pooling, and quantization can be cached in shared memory and global memory access is significantly reduced.

Overhead from bit combination and bit decomposition. We show the overhead from bit combination and bit decomposition in Figure 2.11. We profile the overhead on

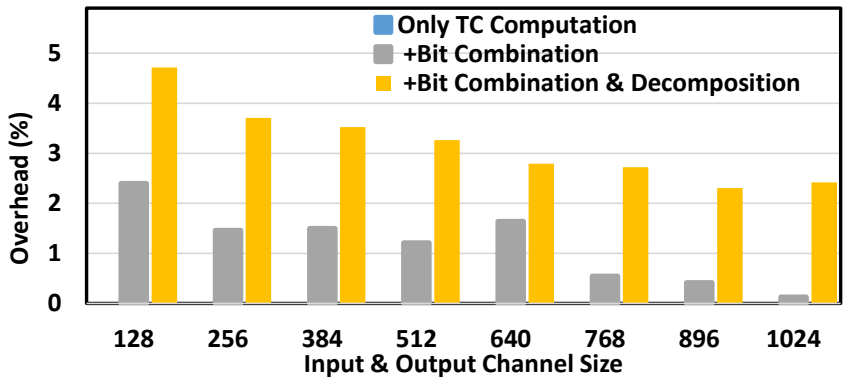


Figure 2.11: Overhead from bit combination and bit decomposition, relative to TC Computation.

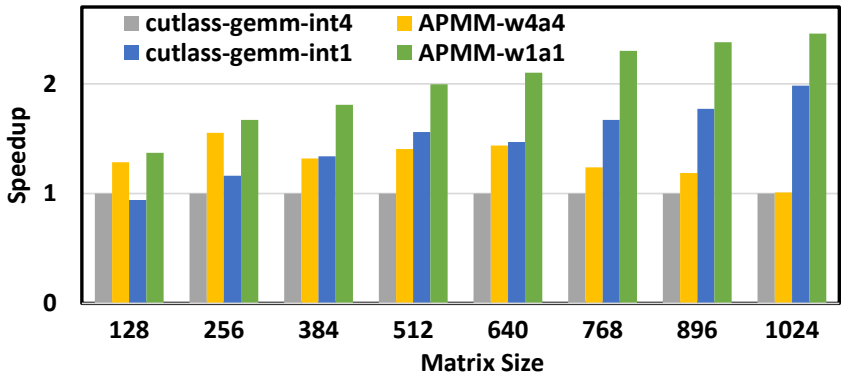


Figure 2.12: Comparing APMM and CUTLASS-GEMM.

APConv designs following the same setting as Section 2.7.1. We show results from APConv-w1a2 since we observe similar overhead across bit settings. On average, we empirically observe 1.16% overhead from bit combination and another 2.02% overhead from bit decomposition, compared to only TC computation. The main reason is that bit combination and bit decomposition introduce only quadratic time complexity, which is significantly smaller than the cubic time complexity from TC computation. Due to this difference in time complexity, the overhead from bit combination decreases from 2.4% to 0.12% as the channel size increases from 128 to 1024. We also observe similar trend for bit decomposition.

Comparing APMM and cutlass GEM under the same bits. Figure 2.12

Table 2.4: Raw latency of a typical fully-connected layer with batch size $M = 64$, input dimension $K = 1024$, and output dimension $N = 1024$. Unit: microsecond.

w1a2	**w1a3**	**w1a4**	**w2a2**	**cutlass-gemm-int4**	**cutlass-gemm-int1**
6.67	6.81	7.06	7.15	15.61	7.92

shows the performance comparison between APMM and cutlass-gemm when using the same bits. Overall, we observe that APMM-w4a4 can achieve 1.3× speedup over cutlass-gemm-int4. The main reason is that APMM-w4a4 can achieve better parallelism by using 16 int1 computations to emulate 1 int4 computation and achieving better GPU utilization, especially for small matrix sizes. We note that this speedup of APMM-w4a4 over cutlass-gemm-int4 decreases as the matrix size increases where more int1 computation resources are required for emulation. We also observe that APMM-w1a1 can achieve 1.35× speedup over cutlass-gemm-int1. This shows the benefit from our kernel-level optimizations.

Raw latency of a typical fully-connected layer. Table 2.4 shows the raw latency of a typical fully-connected layer with batch size $M = 64$, input dimension $K = 1024$, and output dimension $N = 1024$. Overall, we observe that we require only around 7 microsecond for such a layer. Comparing with cutlass-gemm-int4, we can achieve 2.27× speedup on average by using arbitrary-precision computation. We also note that the arbitrary-precision computation is even slightly faster than the cutlass-gemm-int1, which matches the result in Section 2.7.1.

2.8 Discussion

Practical usage of APNN. Arbitrary-precision neural networks have been widely studied to provide diverse tradeoffs between precision and efficiency [53, 52, 54, 22, 24, 23, 55, 56]. While arbitrary-precision may slightly reduce the precision, it shows merit in

many practical usages such as smart sensors [72, 73, 74], mask detection [75], and intelligent agriculture [76]. In these usages, when a certain accuracy bar is surpassed, other essential metrics such as real-time processing and resource consumption are more important. For example, BinaryCoP [75] utilizes low-power binary neural networks to detect facial-mask wear at entrances to corporate buildings and airports. Another example is XpulpNN [76] that uses quantized neural network on energy-efficient IoT devices.

Generality to other NNs. This paper reports the results of APNN-TC on two most time-consuming kernels, GEMM and Convolution, from the computer vision domain and showcases the performance on popular vision models (e.g., AlexNet, VGG, and ResNet). Yet, we expect that APNN-TC applies to NNs from various domains such as natural language processing (NLP). Intuitively, APNN-TC accelerates GEMM and dot products which is the building block of many NLP NNs [5, 77, 78], such as the attention layer and the feed-forward layer.

Generality to other processors. APNN-TC utilizes population count (i.e., `popc()`) and two logical operations (i.e., `XOR` and `AND`) to support arbitrary-precision computation on Nvidia GPUs. Considering the wide support for `popc()` and logical operations, APNN-TC can be easily adapted to diverse processors. For example, AMD GPUs [79] supports population count (i.e. `popcnt()` on AMD GPUs) and logical operations (e.g., bitwise `XOR`). Xeon phi [80] also supports population count and logical operations.

Chapter 3

Accelerating Scientific Computing on Tensor Cores with Extended Precision

In this chapter, we present EGEMM-TC for accelerating scientific computing on Tensor Cores with extended precision (this work [28] has been published in PPoPP 2021). Nvidia Tensor Cores achieve high performance with half-precision matrix inputs tailored towards deep learning workloads. However, this limits the application of Tensor Cores especially in the area of scientific computing with high precision requirements. To tackle this problem, we build Emulated GEMM on Tensor Cores (EGEMM-TC) to extend the usage of Tensor Cores to accelerate scientific computing applications without compromising the precision requirements. First, EGEMM-TC employs an extendable workflow of hardware profiling and operation design to generate a lightweight emulation algorithm on Tensor Cores with extended-precision. Second, EGEMM-TC exploits a set of Tensor Core kernel optimizations to achieve high performance, including the highly-efficient tensorization to exploit the Tensor Core memory architecture and the instruction-level optimizations to coordinate the emulation computation and memory access. Third, EGEMM-TC incorporates a hardware-aware analytic model to offer large flexibility for automatic performance

tuning across various scientific computing workloads and input datasets. Extensive evaluations show that EGEMM-TC can achieve on average 3.13× and 11.18× speedup over the cuBLAS kernels and the CUDA-SDK kernels on CUDA Cores, respectively. Our case study on several scientific computing applications further confirms that EGEMM-TC can generalize the usage of Tensor Cores and achieve about 1.8× speedup compared to the hand-tuned, highly-optimized implementations running on CUDA Cores.

3.1 Problem Statement

Recently, many specialized cores and hardware accelerators have been built to speed up the general matrix multiply (GEMM) in deep learning applications. These specialized cores typically exploit low-precision matrix computation (*e.g.*, half-precision) to achieve high performance, based on the fact that deep learning workloads involve many matrix operations and are usually robust to low-precision computation [29, 30, 31]. One example is the Tensor Core on Nvidia Volta GPUs that conduct half-precision matrix-matrix computation, achieving 8× higher throughput over the CUDA Cores [81]. Since GEMM is also one essential building block of many scientific computing applications, we will bring this performance benefit to the scientific computing domain. For example, GEMM operations take 85% and 67% of the total time in popular implementations of kNN [82] and kMeans [83], respectively. We refer to these applications as *GEMM-based scientific computing*. However, many scientific computing applications (*e.g.*, kNN and kMeans in large-scale physical simulations [32] and mathematical computations [33]) are rather sensitive to computation precision to generate valid results. Such a restriction on precision prevents them from exploiting powerful Tensor Cores for performance enhancement.

Several approaches have been proposed for extended-precision computation [87, 88, 86, 89] on limited-precision hardware, which utilizes multiple low-precision computing

Table 3.1: Precision Specifications. Unit: Number of Bits.

Data Type	Sign	Exponent	Mantissa
Half-Precision [84]	1	5	10
Single-Precision [84]	1	8	23
Markidis-Precision [85]	1	5	20
Extended-Precision [86]	**1**	**5**	**21**

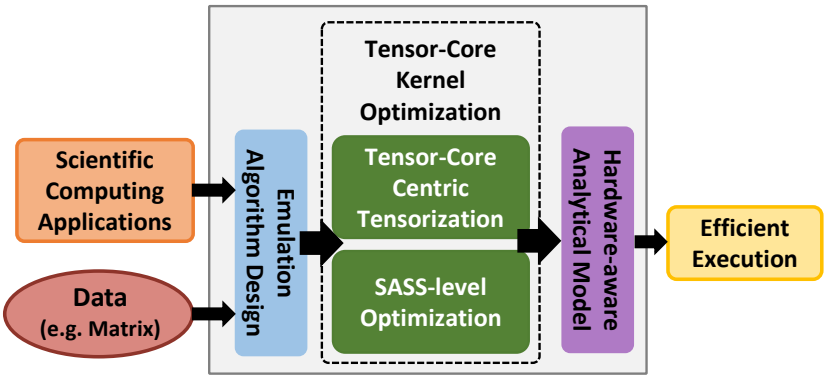


Figure 3.1: Overview of EGEMM-TC.

instructions to emulate a single extended-precision computing instruction. Table 3.1 summarizes these precision types. For example, the popular extended-precision technique, Dekker [86], can utilize 16 half-precision instructions for an extended-precision instruction. One key problem is that these techniques are developed and optimized on CPUs. It requires a significant amount of manual effort to transfer them to Tensor Cores without hurting performance. In particular, Dekker [86] requires serialized execution of the 16 instructions, leading to high overhead. Considering that half-precision computation on Tensor Cores is only 8× faster than single-precision computation on CUDA Cores, this 16× overhead can easily make emulation inappropriate. Markidis [85] proposes a simple algorithm for emulation on Tensor Cores but utilizes a truncate-based strategy with 1-bit precision loss. It fails to achieve extended-precision and shows high overhead.

3.2 Overview of Proposed Solution

In this chapter, we design Emulated GEMM on Tensor Cores (EGEMM-TC) to accelerate GEMM-based scientific computing on Tensor Cores with both high performance and extended-precision computation. We identify several key challenges in the design and the development of EGEMM-TC. First, Tensor Cores require half-precision input matrices, leading to degraded computing precision. Naively borrowing existing emulation algorithms may lead to unsatisfactory performance. Second, the newly designed Tensor Cores bring new computing primitives and memory hierarchies, leading to unexplored optimizations. While Tensor Cores provide high computation performance, memory access speed remains the same as previous CUDA Cores and can easily become the bottleneck. Third, there is a large hyper-parameter design space on mapping scientific computing towards Tensor Cores. Experimenting with new hyper-parameters usually requires manual implementation [90, 91, 92], making the trial-and-error strategy not suitable.

To this end, we propose three techniques to tackle the above challenges, as shown in Figure 3.1. First, EGEMM-TC contains a lightweight emulation algorithm design with only 4 Tensor Core instructions. It achieves both extended-precision and low overhead by exploiting high-precision intermediate computation results. Second, EGEMM-TC utilizes a set of Tensor Core kernel optimizations that efficiently tensorize the emulation workload towards Tensor Cores with low memory overhead. EGEMM-TC also includes SASS-level optimizations for fully exploiting the instruction-level latency hiding opportunities and the register caching capability. Third, EGEMM-TC incorporates a hardware-aware analytic model to automatically explore the design space and reduce manual effort.

We evaluate EGEMM-TC on Tesla T4 and Nvidia RTX 6000. It achieves 3.13× and 11.18× speedup on average over single-precision kernels on CUDA Cores from cuBLAS and CUDA-SDK, respectively. On a set of GEMM-based scientific computing applica-

tions, our approach achieves 1.8× speedup on average compared to hand-tuned code on CUDA Cores.

3.3 Background and Related Work

In this section, we discuss the background and the related work on Tensor Cores and emulation algorithms.

3.3.1 Tensor Cores

Tensor Core Computing and Memory Hierarchy. Different from scalar-scalar computation on CUDA Cores, Tensor Cores provide a matrix-matrix compute primitive. In particular, Tensor Cores support the compute primitive of $D = A \times B + C$, where A and B are required to be half-precision matrices, C and D can be configured to be half-precision or single-precision matrices. Before calling Tensor Cores, all registers in a warp need to collaboratively store these matrices into a new memory hierarchy *Fragment* [29], which allows data sharing across registers. This intra-warp sharing provides opportunities for fragment-based memory optimizations. Existing work [93, 94] reveals that Fragment is implemented as registers, from the perspective of hardware implementation.

Tensor Core Programming Interface. There are two popular programming interfaces for Tensor Cores — CUDA [95] and SASS [96, 97, 93, 94]. CUDA provides C-style APIs and enjoys the widest usage since it is easier to program. However, it provides only limited control over the hardware and cannot exploit the computing and memory capability. SASS provides assembly-style instructions that run natively on NVIDIA GPU hardware [96, 97, 93]. SASS is usually utilized by vendor experts in high-performance libraries (*e.g.*, Nvidia's cuBLAS [98]). In this chapter, we will study the insight of Tensor Core related SASS instructions and propose a set of SASS-level optimizations to support

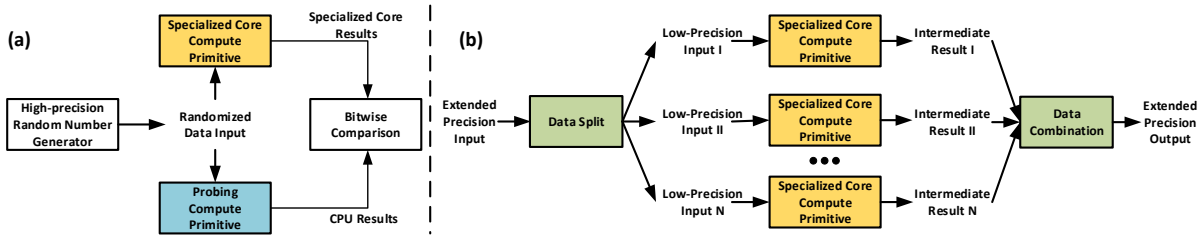


Figure 3.2: Illustration of the generalized emulation design workflow. It first uses (a) *precision profiling* to validate the precision of the intermediate results. Then, it uses (b) *emulation design* to generate a lightweight emulation algorithm based on the profiled precision from (a).

high-performance GEMM-based scientific computing on Tensor Cores.

High-performance Computing on Tensor Cores. Some research efforts have been devoted towards accelerating high-performance computing workloads with Tensor Cores. Yan [99] utilizes Tensor Cores to accelerate half-precision GEMM. Dakkak [100] accelerates half-precision scan on Tensor Cores by transforming *scan* to a GEMM workload. While showing performance improvement by utilizing Tensor Cores, these works focus on half-precision computation and fail to support extended-precision computation. By contrast, EGEMM-TC accelerates GEMM-based scientific computing on Tensor Cores with extended-precision and high performance.

3.3.2 Emulation Algorithms

There have been several emulation algorithms [87, 88, 86, 89, 101] that improve computation precision on low-precision hardware (e.g., IoT devices) and may be extended to Tensor Cores. One traditional emulation algorithm, Dekker [86], utilizes 16 half-precision instructions to emulate an extended-precision instruction. One recent work, Markidis [85], proposes a simple algorithm for emulation on Tensor Cores but utilizes a truncate-based strategy with 1-bit precision loss. By contrast, EGEMM-TC enjoys a lightweight emulation algorithm with 4 instructions, whose overhead is much reduced from Dekker.

EGEMM-TC also employs a round-split algorithm that achieves higher precision by 1 extra mantissa bits, compared to Markidis [85]. In addition, EGEMM-TC achieves high performance by tailoring towards the Tensor Core architecture and incorporating a set of Tensor Core kernel optimizations.

3.4 Emulation Algorithm Design

As discussed in previous sections, existing emulation algorithms usually introduce high computation overhead. These algorithms assume that the hardware has the same input and output precision, thus utilizing a large number of low-precision instructions in the emulation. However, specialized cores usually have higher output precision than the input precision. For example, Tensor Cores require the input precision to be half-precision, while allowing the output precision to be single-precision. Moreover, modern specialized cores usually fuse multiplication and addition (*e.g.*, $D = A \times B + C$), where intermediate results $A \times B$ may also have higher precision than the input precision. Our key insight is that *exploiting high-precision intermediate results from hardware computation can effectively mitigate the emulation overhead.*

To this end, we first propose an extendable workflow to generate a lightweight emulation algorithm. Then, we showcase this workflow on the Tensor Cores and generate a Tensor-Core-specific emulation algorithm. Note that the workflow can be generally applied towards various accelerators and specialized cores. Here, we will focus on improving the precision of small matrices (*i.e.*, 16×16) that directly fit into the Tensor Core compute primitive and leave the performance consideration and large matrix tensorization to the next section.

3.4.1 Generalized Emulation Design Workflow

The emulation design workflow contains a precision profiling and an emulation design, as illustrated in Figure 3.2.

In precision profiling, the main idea is to simulate the computation results on the CPU and compare it bit-wisely with the specialized core results. In particular, we first generate a set of probing compute primitives with diverse intermediate precisions. Then, we evaluate the probing compute primitives on CPUs to get the corresponding results. Since current CPUs usually support a large range of precisions, we can get the ground-truth computation results of the probing compute primitives. Finally, we can get the computation results on the specialized cores as the ground truth, and compare it bitwisely with the CPU results. We repeat this procedure for a large number of randomized high-precision inputs. The "correct" probing compute primitive is identified if its value is bitwisely same with the specialized core results for all the tested inputs.

In emulation design, given the target-precision input, we first utilize a *data split* technique to split the target-precision input into several low-precision inputs following the hardware precision requirement and use each split input for specialized core computation. Then, we utilize a *data combination* technique to combine the intermediate results and generate the target-precision outputs. In the data combination technique, we will utilize the profiled intermediate precision to achieve the minimized overhead. We will provide concrete code snippets and emulation algorithms on Tensor Cores in the following sections.

3.4.2 Emulation Algorithm on Tensor Cores

In this section, we show our emulation algorithm on Tensor Cores. While it exploits the profiling on Nvidia Tensor Cores to mitigate emulation overhead, its correctness

can be easily verified on other specialized cores with our generalized emulation design workflow. When the precision is the same or higher, we can apply the same emulation algorithm as described below to achieve extended-precision computation. When the precision is lower (*e.g.*, half-precision), we may refer to Dekker [86], which assumes the hardware computation precision to be half-precision and emulates extended-precision computation at the cost of low performance.

Precision Profiling on Tensor Cores In this section, we showcase the precision profiling on Tensor Cores. Nvidia officially documents its *specialized core compute primitive* as $A \times B + C$, where the matrix A and B are half-precision, C and D can be either half-precision or single-precision. However, the operation precision during the matrix multiplication $A \times B$ is not officially documented. Without clear profiling, there are multiple *probing compute primitives*. One is that $A \times B$ is conducted in half-precision, which is the same as the data type of A and B. The other is that the half-precision matrices A and B are first converted to single-precision and $A \times B$ is conducted with single-precision or the extended-precision. Operation precision is important for the design and implementation of the emulation algorithm. Assuming both the operation and data are half-precision, Dekker shows that 16 instructions are required to emulate a single-precision instruction, which leads to high overhead.

We use the following code (Figure 3.3) for profiling the operation precision in Tensor Cores. We randomly initialize the Tensor Core input matrices with half-precision data and use the *wmma::mma_sync()* CUDA API to call the specialized core compute primitive for computing d_TC. As reference values, we compute two probing compute primitives d_HALF and d_FLOAT of the above mentioned two possible operation precisions on CUDA Cores. Finally, we compare d_TC with d_HALF and d_FLOAT in a bit-wise manner. We randomly generate $10,000$ groups of data and empirically observe

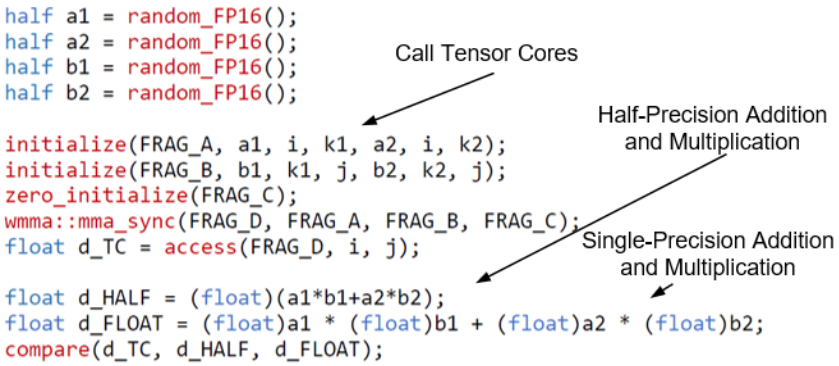


Figure 3.3: Code Snippet for Tensor-Core Precision Profiling.

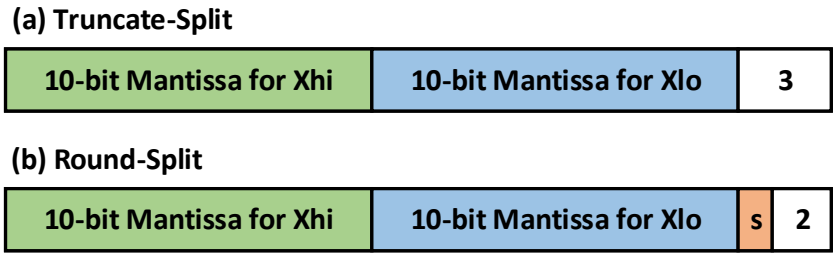


Figure 3.4: Illustration of Round Split Algorithms

that all d_TCs are identical to d_FLOAT bit-wisely up to 21 mantissa bits, which is required by the extended-precision computation. Thus we assume that the operation in Tensor Cores natively supports extended-precision and the only precision loss comes from the half-precision data type of A and B, enables our lightweight emulation algorithm.

Algorithm 1 Lightweight GEMM Emulation Design.

```
1: function EMULATION(D, A, B, C)
2:     Alo, Ahi = Round-Split(A)
3:     Blo, Bhi = Round-Split(B)
4:                                  ▷ Tensor Core natively supports single-precision C and D
5:     D = wmma::mma_sync(Alo, Blo, C)
6:     D = wmma::mma_sync(Alo, Bhi, D)
7:     D = wmma::mma_sync(Ahi, Blo, D)
8:     D = wmma::mma_sync(Ahi, Bhi, D)
9: end function
```

Emulation Design on Tensor Cores In this section, we showcase the emulation design on Tensor Cores, especially the *data split* and the *data combination*. Based on the profiling results, we propose a 4-instruction emulation operation for enabling extended-precision computation on Tensor Cores with 21 mantissa bits. Algorithm 1 summarizes our emulation algorithm. For simplicity, we illustrate with small matrices that match with the Tensor Core computing primitives of shape 16×16 and leave the large-matrix computation to the following sections. Our emulation algorithm takes single-precision matrices A, B, C, and D as the inputs and generates the outputs as $D = A \times B + C$ with extended-precision. The key idea is to first split single-precision matrices A and B into half-precision matrix A_{lo}, A_{hi}, B_{lo} and B_{hi}. Then we can compute on Tensor Cores and accumulate the intermediate results for data combination. Since Tensor Cores natively supports the single-precision C and D, we do not need to conduct data split on these two matrices.

There are multiple approaches for data split. One approach is *truncate-split* from Markidis [85], as illustrated in Figure 3.4(a). It truncates the single-precision data x to be half-precision x_{hi} and uses the x_{lo} to store the remaining value $x - x_{hi}$. While this approach is simple to implement, it supports only 20-bit mantissa from the two 10-bit mantissa of the half-precision data.

Instead, we propose a *round-split* approach as illustrated in Figure 3.4(b). Besides the two 10-bit mantissa, we encode an additional s bit in the sign bit (1-bit) from x_{lo}. For a positive x, the sign-bit of x_{lo} from truncate-split is always 0, but it may be 0 or 1 when round-split is conducted. In particular, for a positive x, we check the 21-th mantissa bit s and, if s is positive, we add 1 to the 10-th mantissa bits for x_{hi} and recompute the x_{lo}. While the round-split method introduces extra overhead, it only needs to be conducted once on every matrix element with time complexity $O(N^2)$. This overhead is significantly less than the matrix multiplication time complexity $O(N^3)$ and introduces

negligible overhead in emulation. To fully exploit GPU capability, EGEMM-TC conducts data split on CUDA Cores and computes the GEMM on Tensor Cores.

Emulation Overhead. Our emulation algorithm introduces 4× computation overhead, which is significantly small than the Dekker [86] method with 16× overhead. A naive implementation may also introduce 4× memory overhead when independently reading the split matrices for each computation instruction. However, this memory overhead can be reduced to 2× when the data reuse is carefully designed. We leave this memory optimization to Section 3.5.

3.5 Tensor-Core-Centric Tensorization

EGEMM-TC has a carefully designed tensorization to efficiently map the GEMM-based scientific computing to Tensor Cores that require specialized matrix inputs. While our tensorization shares some similarities with existing ones, there are two challenges to be addressed before fully exploiting the Tensor Core computing capability. First, existing techniques usually independently assign tasks to individual warps, failing to exploit the collaboration cross warps and within warps. Second, Tensor Cores provide a new memory architecture of fragment (FRAG), which is composed of registers across threads within a warp. This FRAG provides intra-warp caching opportunities that have not been well explored. To this end, we provide two novel optimizations.

Tensorization and Warp Collaboration Different from previous CUDA Cores on the scalar level, Tensor Cores computes at the matrix level, requiring the tensorization design. Matching with the GPU hierarchy, our tensorization recursively divides the matrices into sub-matrices and assign them to GPU blocks, warps, and threads, in a hierarchy-style. Formally, given input matrices A, B, and C, of shape (m, k), (k, n),

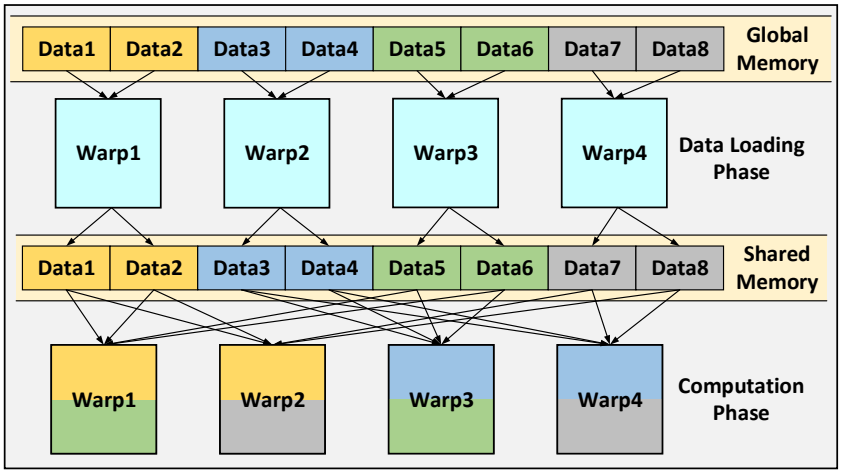


Figure 3.5: Warp Collaboration Illustration. During data loading phase, all warps collaboratively loads all data fragments. During computation phase, a data fragment may be used by multiple warps, indicated by the colors.

(m, n), respectively, we split these matrices into *block matrices* of size (b_m, b_k), (b_k, b_n), and (b_m, b_n). During execution, each GPU block computes a block matrix of C based on the corresponding block matrices of A and B. Here, the sizes of block matrices are typically larger than the Tensor Core compute primitive size, requiring further dividing the block matrices to *warp matrices* with size (w_m, w_k), (w_k, w_n), and (w_m, w_n). These warp matrices will be assigned to individual warps for Tensor Core execution, where warp matrices will be further divided to *TC matrices* for matching the Tensor Core compute primitives with size t_m,t_n and t_k.

EGEMM-TC split the workload into two phases and adopts a warp collaboration strategy as illustrated in Figure 3.5. The main difference from previous CUDA Cores is that Tensor Cores require 32 threads in a warp to collaboratively load matrices into the FRAG memory architecture and jointly compute the matrix multiplication and addition. Catering to the Tensor Core property, we assign different thread organization $(threadDim.x, threadDim.y)$ to the same warp during these two phases. During the computation phase, we utilize the default (32,1) thread layout for collaboratively calling

Table 3.2: Memory access on each GPU warp in GEMM workload. We skip the memory access of Ahi, Blo, and Bhi, since these matrices have similar memory access as the Alo.

Type	**Size**	**w/o FRAG Caching**	**w/ FRAG Caching**
Alo	$2w_m w_k$	$4w_k w_m \cdot w_k/t_k$	$2w_m w_k$
C	$4w_m w_n$	$4w_m w_n \cdot w_k/t_k$	$4w_m w_n$

Tensor Cores, as required by the CUDA programming guide [95]. During the data loading phase, we reorganize the warp threads to 2D layout for assigning non-overlapping memory access workload to each thread. For example, when loading a 16×16 block of data, it is much easier to program with the 16×2 thread configuration than with the default 32×1 one.

Intra-Warp FRAG Caching. Data caching is an effective strategy to reduce the memory overhead in the GEMM-based workload. Existing techniques usually utilize shared memory to cache a portion of the matrices A, B, and C but ignore the FRAG caching opportunity. We name it as the *w/o FRAG caching* strategy. With this strategy, data is still loaded multiple times from the shared memory to the register. Table 3.2 summarizes the memory access on a single GPU warp. When a warp matrix C of shape (w_m, w_n) are assigned to a GPU warp and stored in the shared memory, the memory access between shared memory and FRAG/register is

$$4t_n t_m \cdot \frac{w_k}{t_k} \cdot \frac{w_m}{t_m} \cdot \frac{w_n}{t_n} = 4w_m w_n \cdot \frac{w_k}{t_k} \tag{3.1}$$

where the warp matrix C is divided into $w_m/t_m \cdot w_n/t_n$ TC matrices, w_k/t_k times data loading when iterating over the k-dimension, and each single-precision data requires 4 bytes. Similarly, we can compute the memory access of the warp matrix *Alo* as $2 * 2w_k w_m \cdot w_k/t_k$, where the first 2 comes from the emulation algorithm where *Alo* is used for two times. We observe that the memory access in the *w/o FRAG caching* strategy is significantly larger than the data size of *Alo* and C, leading to extra memory overhead.

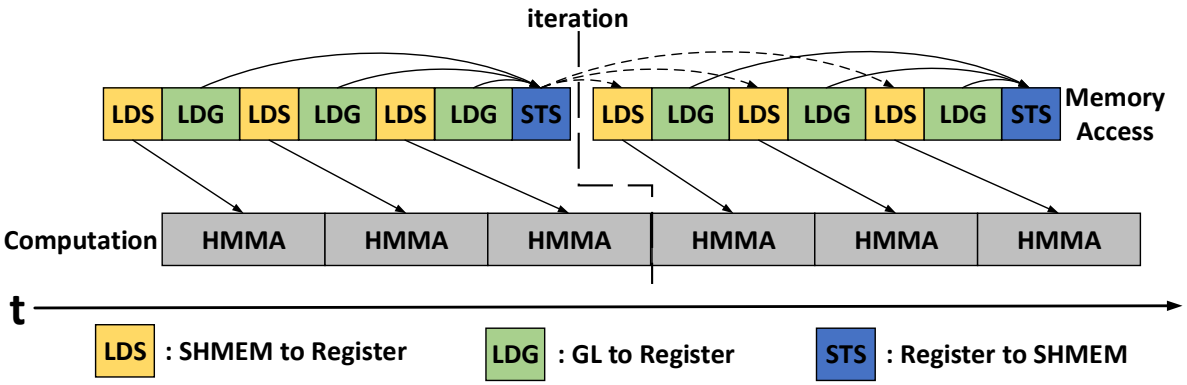


Figure 3.6: Illustration of Register-Enhanced Instruction Scheduling for Latency Hiding.

Instead, we propose an *intra-warp FRAG caching* strategy, that effectively mitigates the memory overhead. The key observations are: 1) FRAG allows registers from multiple threads within a warp to collaboratively access a TC matrix, making it possible to reuse matrix data within warps; 2) FRAG has 256 KB which is much larger than the $64KB$ shared memory. EGEMM-TC will track whether a TC matrix has been stored in the FRAG and skip the data loading when possible. In particular, the TC matrix C is cached in FRAG during the whole computation and the *Alo* is read once to the FRAG. In total, this strategy leads to $4w_m w_n + 4 * 2w_m w_n$ bytes consumption in register/FRAG. While this strategy may increase the register pressure, we will carefully select hyperparameters to avoid register spilling in Section 3.7.

3.6 Instruction-Level Optimizations

In this section, we propose two instruction-level optimizations to fully exploit Tensor Core computing capability and memory hierarchies.

3.6.1 Register-Enhanced Instruction Scheduling for Latency Hiding

The first optimization at the SASS level is a register-enhanced instruction scheduling for latency hiding. While latency hiding has been discussed in existing works [102, 103] and can be implemented at the CUDA-level (*e.g.*, with streams), EGEMM-TC has two distinguishing designs. First, to mitigate the limitation on the shared memory size (*e.g.*, 64 KB per SM on Tesla T4), EGEMM-TC intentionally utilizes registers (*e.g.*, with 256 KB per SM on Tesla T4) to exploiting more latency hiding opportunities. Second, EGEMM-TC supports fine-grained data access latency hiding at the instruction-level by breaking down the KB-level data access into a sequence of Byte-level data access and interleaving with individual Tensor Core computation instructions.

On the SASS instructions, we utilize 4 instructions that are widely used in many generations of Nvidia GPUs [96, 97, 93, 94]. In particular, we use the *LDS* instruction to load data from shared memory to registers, the *LDG* instruction for loading data from global memory to registers, the *STS* instruction for storing data from registers to shared memory, and the *HMMA* instruction for computation on Tensor Cores. Note that existing works [90, 92] demystify that the memory instructions (*e.g.*, *LDS*, *LDG*, and *STS*) are executed sequentially and cannot be further paralleled.

Figure 3.6 illustrates the register-enhanced instruction scheduling for latency hiding. At a high level, EGEMM-TC tensorizes the input matrices into several sub-matrices and processes one sub-matrix at each iteration. Before the first iteration, EGEMM-TC has a "cold-start" that loads the data for the first iteration from global memory to shared memory. On the following iterations, EGEMM-TC simultaneously conducts the computation for the current iteration and the data loading for the next iteration. Assuming that the data for the current iteration has been stored in the shared memory,

EGEMM-TC uses *LDS* to load data from shared memory to registers for computation. Meanwhile, EGEMM-TC loads the data for the next iteration. Noting that Nvidia GPUs usually do not support loading data directly from global memory to shared memory, we first load data from global memory to registers and then store to shared memory with *LDG* and *STS*, respectively. Considering that the shared memory stores the data for the current iteration, we delay *STS* to the end of the current iteration to avoid undesired data overwriting. This design enables caching large matrices in registers and provides more latency hiding opportunities for improving performance.

3.6.2 Register Allocation Design

The second optimization at the SASS level is a manual register allocation to avoid register spilling [104, 105]. To fully exploit the fast register access, we heavily utilize registers for a set of memory-related optimizations. While this register-caching can improve performance theoretically, it also increases the register pressure. Indeed, implementing these optimizations at the CUDA level can easily introduce register spilling, leading to heavy slow down.

While the optimal register allocation has been shown as an NP-problem [106], we propose a heuristic register allocation design for the Tensor-Core centric workload. Our key observation is that these workloads usually contain four stages with different register usage. During the first stage, a large number of registers are utilized on the context information (*e.g.*, threadIdx, blockIdx, and block matrix size) to locate the block matrix for computation. During the following three stages, registers are utilized to load the C matrix from global memory, conducting computation, and saving the C matrix to the global memory. Register allocations across these stages are usually non-overlapping and only utilized in a single stage. Based on this observation, we manually reuse most

registers across stages to reduce the register pressure. In total, we utilize 232 out of 256 registers on each thread for all optimizations mentioned above.

3.7 Hardware-aware Analytic Model

In this section, we propose an analytic model to facilitate the hyper-parameter selection for achieving high performance. There are 6 hyper-parameters $(b_m, b_n, b_k, w_m, w_n, w_k)$ that have a significant influence on the performance. Selecting larger hyper-parameters generally leads to higher data reuse and lower memory overhead. However, larger hyper-parameters also increase the pressure on the shared memory and the register/FRAG. Moreover, when the value exceeds the capability of FRAG, register spilling will happen, leading to degraded performance.

Existing works [90, 90, 91] usually utilize a trial-and-error strategy to select these hyper-parameters. There are two drawbacks of this strategy. First, experimenting with new tiling sizes usually requires extra manual effort, making it a time-consuming task. Second, there is a large design space of 6 parameters, making it infeasible to enumerate all settings. To this end, we propose a hardware-aware analytic model that takes the small set of hardware resource budgets and selects the parameters without trial-and-error.

3.7.1 Resource Consumption

At each iteration, each GPU block needs to do two tasks. First, it reads 2 matrices (Alo, Ahi) of size (b_m, b_k) and 2 matrices (Blo, Bhi) of size (b_k, b_n). This step introduces global memory access of

$$(b_m + b_m + b_n + b_n) \times b_k \times 2 = 4(b_m + b_n)b_k \tag{3.2}$$

Here, the last two comes from half data type (2 bytes). We skip the memory access for block matrices of C since it is only loaded once for every k/b_k times reading of the split matrices and accounts for a negligible portion of memory overhead. Second, EGEMM-TC conducts the computation with FLOPs of

$$2 \times b_m \times b_n \times b_k \times 4 = 8 b_m b_n b_k \tag{3.3}$$

There is a constant 4 since EGEMM takes 4 Tensor Core calls for one extended-precision computation. To this end, the ratio of computation to global memory access is

$$\frac{8 \times b_m \times b_n \times b_k}{4 \times (b_m + b_n) \times b_k} = \frac{2 b_m \times b_n}{b_m + b_n} \tag{3.4}$$

We want to improve this ratio to fully exploit GPU compute capability and achieve compute-bound. Noting that the numerator uses multiplication and the denominator uses addition, we can improve the ratio by choosing a larger b_m and b_n. We surprisingly observe that the ratio is independent of b_k, indicating that we can select a smaller b_k to leave space for storing larger b_m and b_n.

On the memory space, we store a block matrix C of size (b_m, b_n) in the FRAG following the *intra-warp FRAG caching* design. This would consume $b_m \times b_n \times 4 + 2 \times (b_m + b_n) \times b_k \times 2$ bytes in registers. For reducing register pressure, we store the Alo, Ahi, Blo, and Bhi blocks in the shared memory, leading to $2 \times (b_m + b_n) \times b_k \times 2$ bytes shared memory usage.

Inside each warp, we also have the computation and the memory access, determined by the warp tiling size (w_m, w_n, w_k) and the block tiling size (b_m, b_n, b_k). Our design goal is to adjust warp tiling size such that the computation time is larger than the memory access time to achieve the compute-bound. Assuming that each Tensor Core execution takes T_{HMMA} time, the computation time for a block matrix is

$$T_{Comp} = \frac{2 b_m b_n b_k \times 4}{2 \times 16 \times 8 \times 8 \times 4} \times T_{HMMA} \tag{3.5}$$

Table 3.3: Resource Budget on T4 GPU.

Shared Memory Size	64 KB
FRAG/Register Size	256 KB
Peak Computation	2^6 TFLOPS
L2 Cache Speed	750 GB/s

Table 3.4: Design Choice on T4 GPU

(b_m, b_n, b_k)	(128, 128, 32)
(w_m, w_n, w_k)	(64, 32, 8)
Shared memory/block	36 KB
Active Blocks/SM	1
Active Warps / Block	8

where 4 in the numerator represents the 4 Tensor Core calls in the emulation, $2\times16\times8\times8$ is the computation done with a *HMMA.1688.F32* Tensor Core instruction, and each block can call 4 tensor cores simultaneously from the hardware perspective [93, 94]. On the memory access time, there are two steps, as described in the *register-caching-based warp collaboration*. First, all warps collaboratively load data from global memory to the shared memory. Denoting $T_{LDG.128}$ as the time for reading 128-bit data from the global memory and $T_{STS.128}$ the time for writing 128-bit data to the shared memory, the memory access time is

$$T_{Mem1} = \frac{(b_m + b_m + b_n + b_n)b_k \times 2}{32 \times 16} \times (T_{LDG.128} + T_{STS.128}) \tag{3.6}$$

where b_m and b_n are repeated for reading both Alo, Ahi, and Blo, Bhi, 32 is the warp size, and 16 stands for 16 bytes (128 bits). The second step is to load the Alo, Ahi, Blo, and Bhi from shared memory to the FRAG for computing. Denoting $T_{LDS.32}$ as the time for reading 32-bit data from the shared memory, the memory access time is

$$T_{Mem2} = \frac{b_m b_n b_k}{w_m w_n w_k} \times (\frac{w_m}{8} + \frac{w_m}{8} + \frac{w_n}{8} + \frac{w_n}{8}) \times T_{LDS.32} \tag{3.7}$$

3.7.2 Analytic Solver

Our analytic model matches the theoretical resource consumption with the resource budget and transforms the design space exploration to an optimization problem, which can be solved analytically with existing optimization solvers [107]. To support different GPUs, the user only needs to provide a small set of resource budgets. Table 3.3 shows the budget on Tesla T4 GPU.

Formally, we have the following optimization problem

$$\begin{aligned} max \quad & \frac{2b_m \times b_n}{b_m + b_n} \\ s.t. \quad & 4b_m b_n + 4(b_m + b_n)b_k \leq Size_{Register} \\ & 2 \times (b_m + b_n) \times (b_k + 8) \times 2 \leq Size_{SHMEM} \\ & T_{Mem1} + T_{Mem2} \leq T_{Comp} \end{aligned} \tag{3.8}$$

Our goal is to maximize the ratio of computation to global memory access (Equation 3.4) to fully exploit the computing capability. Meanwhile, we need to make sure that the usage of registers and shared memory does not exceed the corresponding resource budget. In addition, we aim to increase (w_m, w_n) for ensuring that each warp spends more time on computation than memory access, leaving space for latency hiding. Table 3.4 details our design choice for Tesla T4.

3.8 Evaluation

In this section, we compare EGEMM-TC with various GEMM kernels and show the benefit of accelerating GEMM-based scientific computing on Tensor Cores.

Table 3.5: Baseline Kernels.

Name	Source	Precision	Description
cuBLAS-CUDA-FP32	cuBLAS	single	*cublasSgemm* on CUDA Cores
cuBLAS-TC-Half	cuBLAS	half	*cublasGemmEx* on Tensor Cores
cuBLAS-TC-Emulation	cuBLAS	extended	implement with *cublasGemmEx* on Tensor Cores
SDK-CUDA-FP32	SDK	single	*matrixMul* on CUDA Cores
Markidis	[85]	extended*	implemented Markidis method on Tensor Cores
kMeans	[82]	single	open-source implementation with cublasSgemm on CUDA Cores
kNN	[83]	single	open-source implementation with cublasSgemm on CUDA Cores

3.8.1 Experiment Setup

Baseline Kernels. We compare EGEMM-TC with a diverse set of GEMM kernels and GEMM-based scientific computing benchmarks shown in Table 3.5. These kernels include cuBLAS kernels running on CUDA Cores and Tensor Cores. We utilize cuBLAS kernel *cublasGemmEx* to implement Algorithm 1 on Tensor Cores, namely *cuBLAS-TC-Emulation*, and compare with EGEMM-TC on the performance benefit of EGEMM-TC optimizations. We also compare the performance with open-source code from CUDA-SDK. Besides, we compare against Markidis [85], the most recent emulation work on Tensor Cores. Note that Markidis has 1-bit lower precision than EGEMM-TC due to the truncate-split, as detailed previously in Table 3.1. We evaluate on diverse matrix sizes from 1024 to 16384 and report the performance averaged over 10 runs, measured with Trillion Floating Point Operations per Second

$$TFLOPS = 2 \times M \times N \times K/(T \times 10^9) \tag{3.9}$$

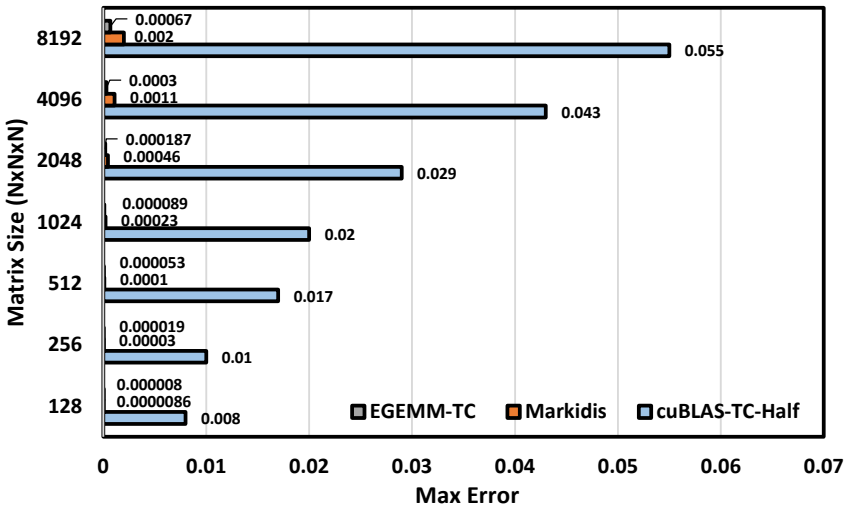
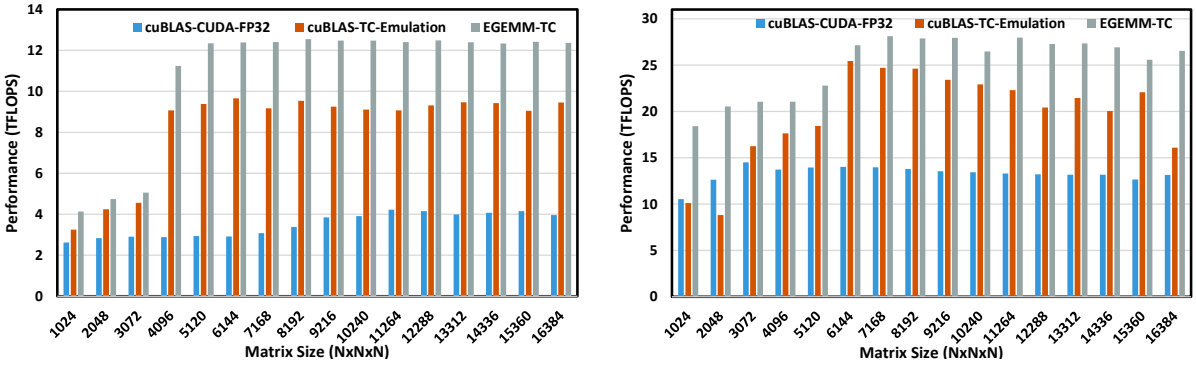


Figure 3.7: Emulation Precision.



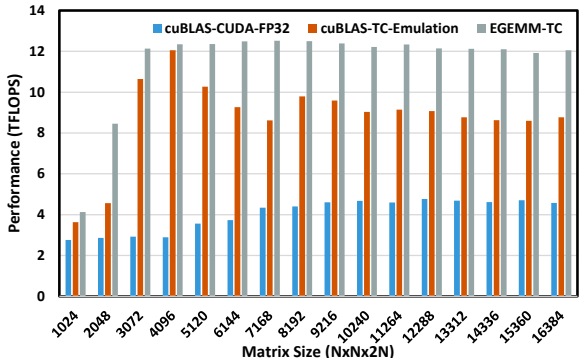
(a) on T4. (b) on RTX6000.

Figure 3.8: Comparison with Vendor Kernels on Square Matrices.

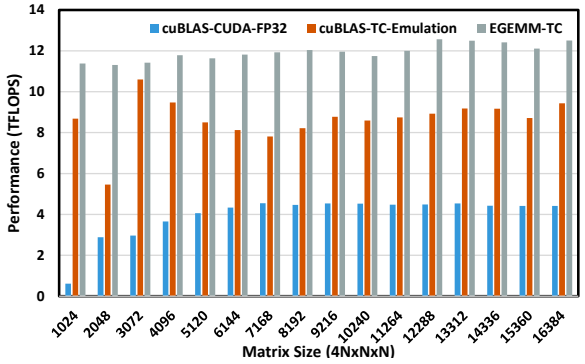
T is the time in milliseconds measured by *cuda event* [108].

We also experiment on two popular scientific computing workloads, kMeans and kNN, that have wide applications in diverse domains (*e.g.*, gene analysis [109], environmental science [110], and astronomy [111]). In particular, we compare with two open-source kernels (kNN [82] and kMeans [83]) on CUDA Cores that implement with *cuBLAS-CUDA-FP32*.

Environments. We evaluate on both Nvidia T4 and Nvidia RTX6000. T4 [112] has 320 Tensor Cores and 16 GB GDDR6 memory. RTX6000 [113] has 576 Tensor Cores and 24



(a) Set matrix shape (M, N, K) as $(N, N, 2N)$.



(b) Set matrix shape (M, N, K) as $(4N, N, N)$.

Figure 3.9: Comparison with Vendor Kernels on Skewed Matrices.

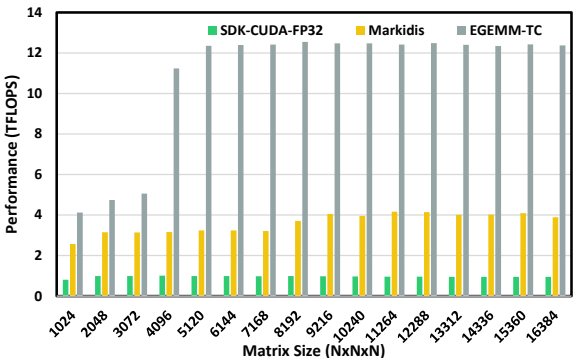


Figure 3.10: Comparison with Open–Source Kernels.

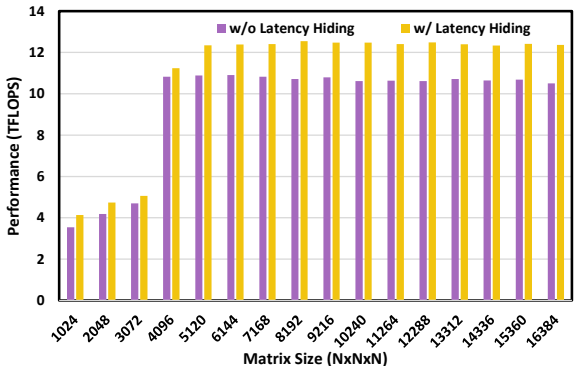


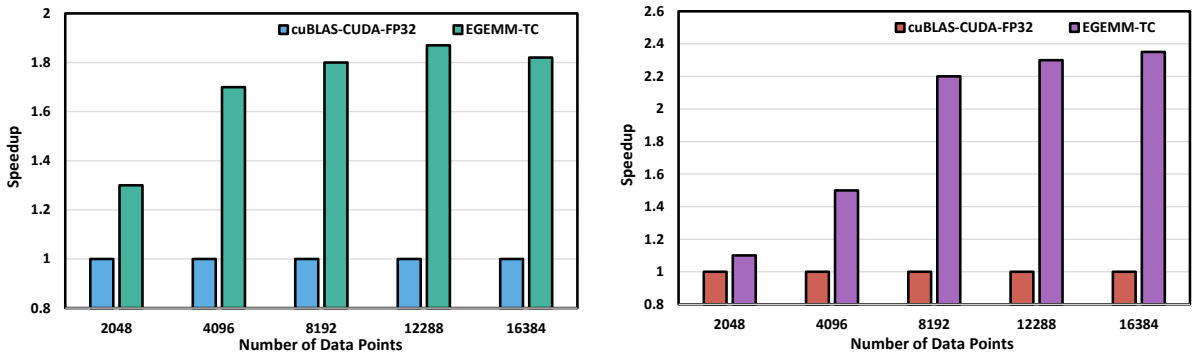
Figure 3.11: Performance Benefit of Latency Hiding.

GB GDDR6 memory. The host server has a 32-core Intel Xeon CPU E5-2620 processor and 126 GB memory and runs Ubuntu 18.04 with CUDA 10.1 and cuBLAS 10.1.

3.8.2 Precision Improvement

Figure 3.7 compares the precision of EGEMM-TC and baseline GEMM kernels. We present the max error relative to the single-precision computation

$$MaxError(p) = |V_p - V_{Single}| \tag{3.10}$$



(a) kMeans (b) kNN

Figure 3.12: GEMM-based Scientific Computing Acceleration with EGEMM-TC.

Here, V_p is the computation results under the precision p, which could be one of the extended-precision, the half-precision, and the single-precision. During the computation, we generate square matrices of size $N \times N \times N$ with values sampled from [-1,+1]. On average, EGEMM-TC effectively reduces the max error by 350× compared to cuBLAS-TC-Half. This result shows the effectiveness of our emulation algorithm in improving the computation precision on Tensor Cores. As the matrix size increases, we observe a slow increase in the max error. The reason is that, a single element in the output matrix involves N additions and N multiplications and the emulation error accumulates as N increases. However, EGEMM-TC still achieves 82× reduction in max error, when computing a large matrix of 8192×8192 with extended-precision. In addition, EGEMM-TC reduces the max error by 2.33× over Markidis, thanks to the round-split algorithm.

3.8.3 Overall Speedup

Comparison with Vendor Kernels. Figure 3.8.1 shows the performance comparison with vendor kernels on Tesla T4. Comparing with *cuBLAS-CUDA-FP32*, EGEMM-TC is faster by 3.13× on average. This result shows that EGEMM-TC on Tensor Cores can effectively outperform the single-precision GEMM on CUDA Cores by a large mar-

gin. This benefit comes from the high-performance half-precision computation on Tensor Cores and our kernel optimizations. Comparing with *cuBLAS-TC-Emulation*, we still observe 1.35× speedup on average. This result shows the effectiveness of our kernel optimizations, considering that cuBLAS provides highly-optimized vendor GEMM kernel. Comparing across matrix sizes, we can see that EGEMM-TC shows a larger speedup at large matrix sizes. The reason is that the GPU capability is not fully utilized at small matrix sizes and the compute-bound has not been achieved. As the matrix size increases, the Tensor Core occupancy also increases and optimizations for reducing memory movement start to show benefit. We show the performance comparison on Nvidia RTX6000 in Figure 3.8.1, where EGEMM-TC has similar benefits as the case on Tesla T4. Since similar patterns show on Telta T4 and RTX6000, we will only show the results on Tesla T4 in the following experiments.

Figure 3.9 shows the performance comparison on skewed matrices, where dimensions K and M are larger than the remaining dimensions by 2× and 4×, respectively. We skip dimension N since it can be viewed as dimension M under matrix transpose. When dimension K is enlarged, we observe that the *cuBLAS-TC-Emulation* exhibits significant slowdown when the matrix size exceeds $4096 \times 4096 \times 8192$. Instead, EGEMM-TC consistently provides high performance across different matrix sizes. In this case, EGEMM-TC provides 1.33× speedup over *cuBLAS-TC-Emulation* and 2.89× speedup over *cuBLAS-CUDA-FP32*. When dimension M is enlarged, *cuBLAS-TC-Emulation* achieves higher performance but is still much slower than EGEMM-TC. Under this setting, our GEMM are 1.40× faster than *cuBLAS-TC-Emulation* and 2.9× faster than *cuBLAS-CUDA-FP32* on average.

Comparison with Open-Source Kernels. Figure 3.10 shows the performance comparison with the open-source kernels. Comparing with *SDK-CUDA-FP32*, EGEMM-TC is faster by 11.18× on average. This result shows the significant performance improve-

ment from EGEMM-TC on Tensor Cores. EGEMM-TC is also faster than *Markidis* by 3.0× on average. We manually tune *Markidis* performance with our optimizations on the hand-written CUDA code, but the performance remains similar. The reason is that the CUDA programming interface provides limited control over the GPU hardware while our implementation-level optimizations with the SASS programming interface can utilize GPU capability to much larger extent (*e.g.*, register-enhanced instruction scheduling).

3.8.4 Benefit of Instruction Scheduling

Figure 3.11 shows the performance benefits of instruction scheduling in latency hiding. In this optimization, we focus on the SASS programming interface and switch orders of computation and memory access instructions for latency hiding. The instruction scheduling can achieve 1.14× speedup on average. Comparing with the latency hiding on the CUDA programming interface, we can achieve more fine-grained latency hiding with the SASS programming interface. For example, loading data from global memory to shared memory is a single instruction with the CUDA programming interface but two instructions with the SASS programming interface (*i.e.*, loading to register from global memory and storing from registers to shared memory). This provides more opportunities to interleave the memory access instructions with the compute instructions.

3.8.5 Scientific Computing Acceleration

Figure 3.12 shows the speedup of scientific computing based on EGEMM-TC over cuBLAS-CUDA-FP32. We observe an average speedup of 1.9× on kMeans and an average speedup of 1.7× on kNN. These speedups show that EGEMM-TC can be effectively utilized to accelerate GEMM-based scientific computing. Comparing across data sizes, EGEMM-TC accelerates kMeans by 1.3× when there are 2048 data points and accelerates

kMeans by 1.82× when there are 16384 data points, as shown in Figure 3.8.2. There are two reasons. First, EGEMM-TC shows a larger speedup than cuBLAS-CUDA-FP32 when the data size increases, as shown previously in Figure 3.8. Second, when data size increases, GEMM accounts for more running time and the acceleration on GEMM shows more benefits. We also observe similar trends on the kNN workload (Figure 3.8.2).

Chapter 4

Palleon: A Runtime System for Efficient Video Processing toward Dynamic Class Skew

In this chapter, we present Palleon which is a runtime system for accelerating video processing toward dynamic class skews. On par with the human classification accuracy, convolutional neural networks (CNNs) have fueled the deployment of many video processing systems on cloud-backed mobile platforms (*e.g.*, cell phones and robotics). Nevertheless, these video processing systems often face a tension between intensive energy consumption from CNNs and limited resources on mobile platforms. To address this tension, we propose to accelerate video processing with a widely-available, but not yet well-explored runtime input-level information, namely *class skew*. Through such runtime-profiled information, it strives to automatically optimize CNNs toward the time-varying video stream. Specifically, we build Palleon, a runtime system that dynamically adapts and selects a CNN model with the least energy consumption based on the automatically detected class skews, while still achieving the desired accuracy. Extensive evaluations on

state-of-the-art CNNs and real-world videos demonstrate that Palleon enables efficient video processing with up to 6.7× energy saving and 7.9× latency reduction.

4.1 Problem Statement

Convolutional neural networks (CNNs) based video processing plays an important role in many emerging applications [114, 115, 116, 117, 118, 119, 120, 121] deployed on cloud-backed mobile platforms. Among them, cognitive assistants and robotic visions are two representative categories. Smart glasses [117, 118, 119], for example, continuously recognize the surrounding environment with CNNs and help the blind person with ordinary tasks (*e.g.,* reading a handwritten note, navigating the grocery store, and even running the Boston Marathon). Robotic visions could automatically search specific animals and document the secret lives of them in the wild [120], as well as detect landmines in various environments [121].

While these applications enjoy both the mobility of the wide deployment in real world and the high accuracy of CNNs, they also face the tension between the limited resource budget on mobile platforms and the high energy consumption and latency of CNNs. A popular CNN, VggNet [122], can easily consume 3.6W and introduce 1.4-second latency, which makes a large smartphone battery (*e.g.*, 2.7-Ah battery in iPhone X [14]) out of power within 2 hours on continuous image classification. To improve the execution efficiency, many techniques have been proposed such as pruning [123, 124, 125] and quantization [126, 127, 128] to reduce the size of CNN models. However, these existing works fail to exploit the special characteristics of video streams; furthermore, the compromised model accuracy also limits the overall pruning or quantization ratio.

Complementing existing model compression techniques, a strong *temporal locality* in video streams is investigated here to enable efficient video processing on mobile platforms.

Considering a video stream collected from a continuous camera feed, it is common that only a small number of classes keep appearing in a large number of consecutive frames. For example, in a film scenario, only a small number of people would come to the master shots frequently, generally lasting for a few minutes, and another group of people will not appear until the scenario has changed. A study on Youtube videos of day-to-day life [129] also shows that more than 90% frames are comprised of less than 10 classes.

We first turn such an abstract concept, *temporal locality*, into something concrete and measurable, *class skew*. Specifically, class skew is formally defined as an unbalanced class distribution that flexibly and effectively extracts the scenario information of both *class cardinality* and *visual separability*. Class cardinality here captures the number of classes in a class skew. By exploiting this class cardinality, we can tailor a general CNN, which is usually trained to recognize thousands of classes, into a specialized model, which only needs to recognize a small number of classes in the current class skew. Meanwhile, we notice that class skews show diverse visual separability under the same class cardinality. For example, a class skew with two classes (*e.g.,* houses and dogs) is easier to recognize compared to that with two more subtle classes (*e.g.*, Husky and Alaskan). By exploiting visual separability, we can use a more compact model for computation and energy saving without loss of accuracy compared with a full model.

We then identify several key challenges that hinder the successful utilization of class skews. First, new class skews may appear and disappear suddenly as time goes, namely *class skew switches*, making it hard to precisely capture the class skew and respond fast to class skew switches. Second, a class skew may last for minutes or even hours between two class skew switches, but this lasting time varies across different videos and even scenarios, thus cannot be decided offline. Third, with the detected class skew, it is still hard to adapt deep models at runtime, since existing model adaptation techniques of retraining fully connected layers are computation-intensive and not affordable on mobile platforms.

Forth, a single model adapted toward various class skews may show a significant difference in accuracy due to the diverse visual separability. This difference in accuracy either allows more lightweight CNNs for more energy saving or requires more computation-intensive CNNs to achieve a satisfactory accuracy. Finally, model selection adaptive to class skews may introduce high overhead, making it infeasible to execute on mobile platforms.

4.2 Overview of Proposed Solution

To address these challenges, we build a runtime system, Palleon, which could not only detect class skews during runtime, but also dynamically adapt and select a CNN model with the least energy consumption accordingly. In contrast, some existing works [130, 131, 132], which share a similar high-level motivation with us, only target some specific application scenarios that are already known offline—If you want to, you could think of these as "static" class skews without dynamic switches. A recent work, FAST [129], has made some progress along this research direction. FAST assumes that the exact set of class skews (and their duration time) in a video stream are foreknown, and FAST trains a set of compact models for each known class skew offline. During runtime, FAST only needs to detect these foreknown class skews with a simple window-based detector and directly apply those pre-trained models accordingly. To this end, Palleon adopts a pure runtime approach and targets a more realistic setting that the class skews in the video stream are not foreknown.

As illustrated in Figure 4.1, the Palleon runtime system continuously takes video frames and efficiently generates video processing predictions with three novel components. First, we propose an agile class skew detector, **ABLE** (Section 4.4), to abstract class skews from video streams. ABLE comes with the *static class-skew profiling* and the *dynamic class-skew switch detection*. The former automatically detects the class skew

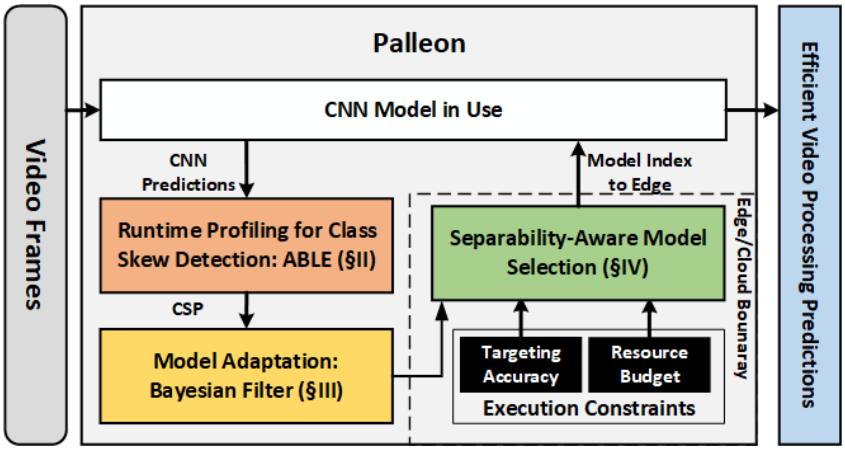


Figure 4.1: Overview of the Palleon Runtime System.

and generates a precise *Class Skew Profile* (CSP) when a class skew is detected. The latter continuously catches class skew switches during runtime without the offline information about the class skew lasting time.

Second, we propose **Bayesian Filter** (Section 4.5) to adapt CNNs toward the detected class skew during runtime. While Bayesian Filter does not directly lead to energy efficiency, it improves the accuracy of compact models with low resource consumption and allows the compact models to replace the complex model. Bayesian Filter is a lightweight module comprised of a *Rescaling* mode that adapts CNNs towards detected CSP without online finetuning and a *Direct Pass* mode that allows the adapted CNNs to still recognize classes out of the current CSP. This lightweight module resolves the complex trade-off between accuracy improvement and adaptation overhead in exploiting class skews.

Third, we design a cloud-backed model selection scheme, namely **Separability-Aware Model Selection** (Section 4.6), to further squeeze system energy consumption. This scheme exploits visual separability with an *efficient online model selection* that identifies the CNN with the least resource consumption while achieving satisfactory accuracy on the detected class skew. Meanwhile, this scheme contains an *edge-cloud duplicated model bank* to mitigate the model selection overhead on mobile platforms and

deliberately schedule the runtime workload between the edge and the cloud.

In summary, we build Palleon, a runtime system that automatically detects input-level information with ABLE and dynamically adatps the given CNNs online with Bayesian Filter. Palleon also controls a set of tuning knobs for balancing the accuracy and the resource efficiency with separability-aware model selection. We build Palleon upon TensorFlow [133] and evaluate it on a cloud-backed mobile platform (with NVIDIA Jetson Nano [134] as the edge device and Dell Workstation T7910 [135] as the cloud server). We evaluate Palleon on various CNN models and different datasets. In particular, for CNN models, we use a variety of the state-of-the-art CNNs from two major domains – object classification (MobileNet [41], VGGNet [122], ResNet [2], and DenseNet [136]) and face recognition (VGGFace [137]). For datasets, we take both synthesized videos and several real-world movies. Extensive experiments confirm the effectiveness of Palleon and show that it could achieve up to 6.7× energy saving and 7.9× latency reduction while achieving an equivalent or better accuracy.

4.3 Related Work

Model Compression. Model compression has been widely explored for accelerating video processing. The popular compression techniques include resolution reduction [138, 139, 41, 140], matrix factorization [141, 142, 143], matrix pruning [144, 123], and distillation [145, 146, 147, 148]. Model compression is orthogonal to our work in exploiting class skews and usually leads to accuracy drop. By contrast, Palleon exploits class skews in video streams and maintains accuracy while reducing energy consumption and processing latency. Meanwhile, Palleon can integrate these compression techniques into our Separability-Aware Model Selection for generating compact models.

Video Processing with Low-Level Temporal Information. Using low-level temporal information can improve accuracy or reduce energy consumption. From the

perspective of system design, existing work exploits low-level temporal information by caching processing results of the most recent frames for future computation reuse [149, 150] or adjusting sampling rate [151, 152]. From the perspective of algorithm design, existing work often augments the traditional 2D-CNN with optical flow [153, 154, 155] for explicitly capturing object motions across frames. A new CNN design, 3D-CNNs [156, 157, 158], has also been proposed to implicitly learn object motions by stacking several 2D-CNNs and processing adjacent video frames in a combined way. These works are orthogonal to our work because we focus on exploiting high-level temporal information across minutes, not on low-level temporal information in a few seconds. Palleon could be integrated with one of these approaches for further performance improvement.

Video Processing with High-Level Temporal Information. Several video processing systems [159, 130, 131, 160, 161, 129] have been proposed to exploit high-level temporal information across minutes, in terms of scenario information. Several early work [131, 130, 160, 161] simplifies processing tasks by targeting a specific scenario and only recognizing a specific object, *e.g.*, buses at a crosswalk. Recent work [129] conducts offline-profiling over a few scenarios and only reduces energy consumption when these offline-profiled scenarios appear, which would be in-effective for more realistic settings that class skews may switch during runtime. By contrast, Palleon abstracts these specific scenarios to a more general class skew of unbalanced distributions and enables online class skew detection and online model adaptation.

4.4 ABLE for Class Skew Detection

We build a class-skew detector, namely *ABLE*, to detect class skews during runtime and enable class-skew based optimizations. Our goal is two-fold: 1) giving a precise class-skew profile (CSP) in static regions between adjacent class-skew switches, and 2)

detecting when the class-skew switches occur. To this end, we break down our class-skew detection into two sub-tasks: **Static Class-Skew Profiling** and **Dynamic Class-Skew Switch Detection**.

4.4.1 Static Class-Skew Profiling

A static CSP generates the distribution of classes in a *static region* where no class skew switch happens. Palleon approximates the CSP in each static region with an empirical distribution [162], which enjoys theoretical properties of converging fast to the ground truth CSP. As illustrated in Figure 4.4.1, given a time window with $r_t = 10$ frames, we collect the predicted labels for each frame and count the frequency of each class. For example, E appears for 4 times out of 10 frames in total, leading to 0.4 for class E in the estimated CSP.

Formally, at time t, in a given frame window with r_t frames (ranging from the $t-r_t+1^{th}$ to the t^{th} frame), the probability of class j in the CSP is computed as

$$p(j|r_t, x_{1:t}) = \frac{1}{r_t} \sum_{i=t-r_t+1}^{t} \mathbb{1}_{x_i=j} \tag{4.1}$$

where x_i is the *predicted label* for the i^{th} frame, $\mathbb{1}_{x_i=j}$ is an indicator function [163] on whether x_i equals j, and $x_{1:t}$ denotes all t predicted labels in the history. The CSP for the given frame window (with r_t frames ending with the t^{th} frame) is computed as a probability vector of all classes:

$$CSP_{t,r_t} = \{p(1|r_t, x_{1:t}), p(2|r_t, x_{1:t}), ..., p(d|r_t, x_{1:t})\} \tag{4.2}$$

where d is the total number of classes.

Early Optimization by Adaptive Waiting Scheme. Palleon splits each static region as two phases (Figure 4.4.1): a *waiting phase* to collect a precise CSP based on the full model and an *optimization phase* to apply class-skew based optimizations. In the optimization phase, we use a compact model adapted toward CSPs, which saves energy and

reduces latency while achieving an equivalent accuracy to the full model. By allocating smaller number of frames in the waiting phase, Palleon can start the optimization phase early and squeeze more optimization opportunities for more frames, leading to better system performance. However, an imprecise CSP may be generated when allocating too few frames to the waiting phase. Hence, we select the frame number carefully to improve system performance and retain precise CSPs.

We develop an *adaptive waiting scheme* to determine whether Palleon has collected a precise CSP and the waiting phase can be terminated. Suppose the waiting phase has already lasted r_t frames and the current class-skew profile is CSP_{t,r_t}, this scheme computes a minimal frame number F_{min} based on CSP_{t,r_t}. If $F_{min} < r_t$, it terminates the waiting phase. Otherwise, it continues and repeatedly applies such check. In principle, F_{min} guarantees a negligible profiling error ϵ between the true probability p_j and the profiled probability $\hat{p}_j$

$$\max_{1 \leq j \leq d} |\hat{p}_j - p_j|/p_j \leq \epsilon \tag{4.3}$$

We next discuss how F_{min} is computed and why it guarantees a negligible profiling error. In addition, we will also propose some practical designs for efficiently computing F_{min}. We start with a theorem, which gives the minimum number of frames to profile a particular class in the class skew.

Theorem 1. (Asymptotic Error Bound). With $n = Z_c/(\epsilon\sqrt{\hat{p}_j})$ samples (frames), the probability of achieving a negligible error $P(|\hat{p}_j - p_j|/p_j < \epsilon) > 1 - c$ for class j holds asymptotically, where Z_c is a Gaussian Distribution Z-score with confidence level $1 - c$ and ϵ is a tolerable error bound.

Proof. Due to the property of multinomial distribution, estimator $\hat{p}_j$ computed with Equation 4.1 is a maximum likelihood estimator (MLE). Based on the asymptotic normality of MLE, we have $\sqrt{n}(\hat{p}_j - p_j) \rightarrow N(0, FI^{-1})$, where FI is the Fisher Infor-

mation matrix $FI_{wh} = E_X[-\frac{\partial^2 ln f_p(X_t)}{\partial p_w \partial p_h}]$. Clearly $FI_{wh} = n/p_w$ if $w = h$; 0, otherwise. Based on the marginalization property of multivariate normality, we can see that $(\hat{p_j} - p_j) \rightarrow N(0, \frac{p_j}{n^2})$. Following this asymptotic distribution, we can derive the required sample number $n = Z_c/(\epsilon\sqrt{\hat{p_j}})$.

Considering that CSP is stable only if estimators for most classes j are stable, we set the minimum number of frames as

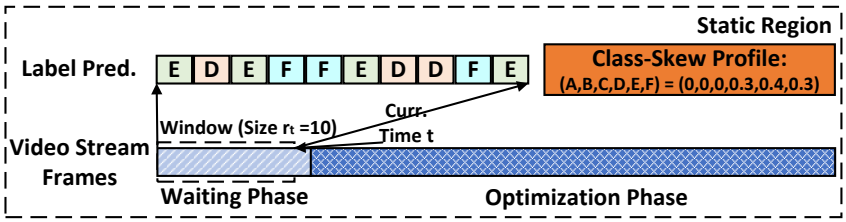
$$F_{min} = \max_{\hat{p_j} > \xi} \; Z_c/(\epsilon\sqrt{\hat{p}(j|r_t, x_{1:t})}) \tag{4.4}$$

Here we only consider classes showing *significant existence* ($\hat{p_j} > \xi$, where $\xi = 1/(2 * d)$ is a probability threshold). The intuition is that *CNNs may make wrong predictions randomly spanning in various classes with significantly low probabilities, leading to an unnecessarily long waiting time.* This strategy can mitigate the effect of prediction errors and focus on the effect of correct predictions.

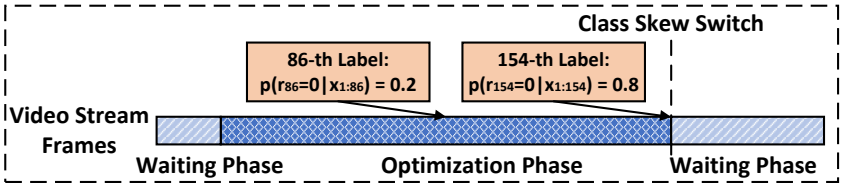
For an arbitrary class number d, computing F_{min} has a low time complexity of $O(d)$, where the main computation resides in iterating through all classes j (Equation 4.4) and estimating the probability $\hat{p}(j|r_t, x_{1:t})$. This estimation can be conducted efficiently in constant time, based on a *computation reuse* technique detailed in the following section.

4.4.2 Dynamic Class-Skew Switch Detection

Dynamic class-skew switch detection identifies class skew switches and provides static regions for the static class skew profiling, as shown in Figure 4.4.1. Specifically, dynamic class-skew switch detection identifies the timestamp t when the previous class skew ends and a new class skew appears. There are two standard techniques to detect class skew switches: Window-based approach [164, 129] and Bayesian-based approach [165, 166, 167, 168, 169, 170]. The former splits video streams into a sequence of windows with a fixed window size k and periodically detects the class skew switch at the boundary. The



(a) Static Class-Skew Profiling.



(b) Dynamic Class-Skew Switch Detection.

Figure 4.2: Illustration of ABLE for Class Skew Detection.

latter detects class skew switch at each timestamp t by tracking all historical windows, where the $k^{th} \in \{1, 2, ..., t\}$ historical window contains the $t - k + 1^{th}$ frame to the t^{th} frame. However, the former only detects the class skew switch when a time window has finished, leading to a detection delay up to the fixed window size. While the latter reacts fast to class skew switches by tracking all historical windows, it introduces high overhead with a quadratic time complexity $O((d + t) * t)$, in the number of frames t and the number of classes d. The latter shows more than 1500-millisecond latency per frame after processing a 3-minute video clip. By contrast, Palleon checks class skew switches at each label prediction x_t (Figure 4.4.1) while introducing low computation complexity of $O(d * k)$, where k is the number of sampled windows ($k \ll t$).

At a high level, we sample a subset of window sizes $r_t \in \{w_1, w_2, ..., w_k\}$ and flag a class skew switch when the probability $p(r_t = 0|x_{1:t})$ is higher than the probability of the other r_t. We estimate the probability $p(r_t|x_{1:t})$ of each window size r_t when a new predicted label x_t comes:

$$p(r_t|x_{1:t}) = p(r_t, x_{1:t}) / \sum_{r_t=0}^{t} p(r_t, x_{1:t}), \tag{4.5}$$

where $p(r_t, x_{1:t})$ is the joint possibility of the lasting time r_t and the predicted labels $x_{1:t}$:

$$\begin{aligned} p(r_t, x_{1:t}) = \sum_{i=1}^{k} p(r_t | r_{t-1} = w_i) \cdot \\ p(x_t | r_{t-1} = w_i, x_{1:t-1}) \cdot p(r_{t-1} = w_i, x_{1:t-1}) \end{aligned} \tag{4.6}$$

Here, $p(r_t | r_{t-1} = w_i)$ is a survival function [171] of the probability that a class skew of length r_{t-1} is still alive at r_t, and $p(x_t | r_{t-1} = w_i, x_{1:t-1})$ is the probability that the predicted label x_t comes from the same distribution as last r_{t-1} labels, computed by Equation 4.1.

Overhead Reduction by Window Sampling. Window sampling selects k windows (*i.e.*, $w_1, w_2, ..., w_k$) that minimize the mean absolute error between tracking the selected k windows and all t windows:

$$\min_{w_1, w_2, ..., w_k} \sum_{r_t=1}^{t} |p(r_t, x_{1:t}) - p_{f(r_t)}|, \tag{4.7}$$

where $f(r_t)$ maps time window r_t to one of the sampled time window $\{w_1, w_2, ..., w_k\}$. A small number of k windows can approximate all t windows since a window size with low possibility $p(r_{t-1}, x_{1:t-1})$ tends to still have low possibility $p(r_t = r_{t-1} + 1, x_{1:t})$ when new data comes:

$$\begin{aligned} p(r_t, x_{1:t}) &= p(r_t | r_{t-1}) p(x_t | r_{t-1}, x_{1:t-1}) p(r_{t-1}, x_{1:t-1}) \\ &\leq p(r_{t-1}, x_{1:t-1}) \end{aligned} \tag{4.8}$$

A straightforward approach is to only track the k most possible windows. However, this approach may not work well when a large number ($>$ k) of windows have equally high probability $p(r_t, x_{1:t})$. To this end, we group all t windows into k clusters and select a representative window out of each cluster. To cluster windows, we first sort the possibility $p(r_t, x_{1:t})$ for all time windows r_t and split into clusters at the top $k-1$ gaps in the sorted sequence. Then, we select the windows with the median probability to represent this cluster and give this window a weight based on the number of windows in

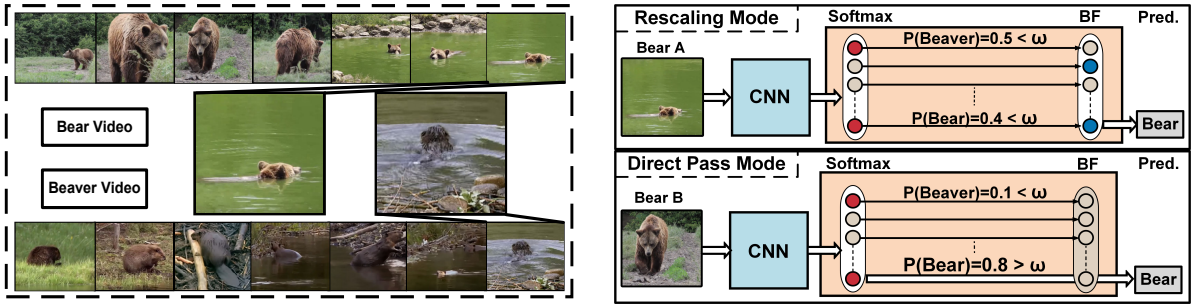
the cluster. Intuitively, we exclude the top $k-1$ gaps by splitting clusters at these gaps to minimize the mean absolute error.

Fast Update by Computation Reuse. Another optimization opportunity is the computation reuse in estimating the conditional distribution $p(x_t|r_{t-1}, x_{1:t-1})$. Since the estimation for each window r_{t-1} at most traverses all data points and all classes, the time complexity of the estimation is linear to the number of data points and classes $O(d+t)$. Since adjacent windows differ only by one input, we can reuse the class frequency in adjacent windows, reducing the time complexity for each window to be $O(d)$. Specifically, the class frequency counts in adjacent windows $r_{t-1} = i$ and $r_{t-1} = i+1$ differ only by one in a single class, determined by the label x_{t-i}. Thus, with the class frequency count $[C_1, C_2, ..., C_d]$ in window $r_{t-1} = i$, we can update the frequency count in window size $r_{t-1} = i+1$ by $C'_j = C_j + 1$, if $j = x_{t-i}$; $= C_j$, o.w.

4.5 Bayesian Filter for Model Adaptation

In this section, we develop a highly flexible module, namely *Bayesian Filter*, to enable class-skew based optimizations. This module consumes a Class Skew Profile (CSP) detected by ABLE and has two functionalities: 1) adapting CNNs toward the detected class skew with low computation overhead and low latency during runtime; 2) allowing the adapted CNNs to recognize classes out of the current CSP for enabling the detection of class skew switches. To this end, we design a **Rescaling** mode (Section 4.5.1) for scenario reference (*i.e.*, model adaptation) and a **Direct Pass** mode (Section 4.5.2) for retaining confident predictions.

Figure 4.5 exhibits two videos with extreme class skews, which are intentionally made simple for understanding. In most video frames, the objects are easy to recognize, during which Palleon extracts a precise CSP based on classes recently detected with high



(a) Two videos with extreme class skews (b) Intuition behind Bayesian Filter.

Figure 4.3: Overview of Bayesian Filter.

confidence. This CSP helps for frames in which objects are hard to recognize (*e.g.*, the animals jump into the water and turn around).

4.5.1 Rescaling

Rescaling mode is a lightweight module for model adaptation with low computation overhead and low latency. This design avoids the heavy computation overhead and latency in existing work [130, 131, 161, 129], which retrains fully connected layers in CNNs during online and may introduce a long latency (up to 14 seconds) [129]. When considering the energy efficiency, this retraining procedure either introduces heavy computation overhead when conducted on the edge, or heavy network communication overhead when retraining is conducted on the cloud and updated weights are transferred to the edge. By contrast, Rescaling mode adapts CNNs by appending an extra layer after the fully connected layer. We stress that Rescaling mode requires neither weight update nor model retraining.

Rescaling mode adapts CNNs toward the detected class skew by initializing the extra layer with the CSP generated by ABLE. Since CSP contains the probability of all d classes, the extra layer is designed to have the same number of d nodes, whose weights are initialized by each probability in CSP. In this way, the magnitude of node weights

indicates the frequency of the corresponding class in the CSP. When a frame comes, the CNN will generate a probability for each class in the softmax layer and these probabilities will be adjusted according to the magnitude of corresponding node weights. Specifically, the extra layer rescales the probability of softmax-layer predictions (red nodes) toward the current CSP (blue nodes) when the highest softmax-layer probability does not pass a pre-defined threshold ω (*i.e.*, not confident enough). Following the spirit of Bayesian statistics [162], Rescaling mode updates the probability for each class by considering both the prior and the posterior information. We tried several different designs, and it turns out that a simple rescaling scheme based on Bayes theorem would already work effectively, as shown in Formula 4.9.

$$P(i|X) = \frac{P(i) \cdot P(X|i)}{P(X)}, \quad i \in \{1, 2, ..., d\} \tag{4.9}$$

where d is the total number of classes.

Formula 4.9 computes the posterior probability of class i for a given image X. The prior probability $P(i)$ is the profiled probability of class i in the current CSP. And the likelihood $P(X|i)$ describes the possibility that an image X comes from class i, according to the softmax-layer probability of class i. $P(X)$ stands for the marginal likelihood for observing the image X, which is same for all classes and does not change the rescaling results. Thus, we can avoid computing $P(X)$ and, instead, use a handy rescaling mechanism as $P(i|X) \propto P(i) \cdot P(X|i)$. To the best of our knowledge, we are the first to design Bayesian rescaling on CNNs for runtime model adaptation toward class skews.

4.5.2 Direct Pass

Direct Pass mode selects the original prediction without rescaling when the predicted probability is higher than a pre-selected threshold ω (Direct Pass mode in Figure 4.5). This design allows detecting class skew switches by identifying classes out of the current

CSP. This design is inspired by observations [172, 173] that *neural networks usually achieve higher accuracy when they predict a high probability.*

Formally, Bayesian Filter with both Rescaling mode and Direct Pass mode can be written as

$$P(i|X) \propto \begin{cases} P(i) \cdot P(X|i) & \text{if } P(X|i) < \omega \\ P(X|i) & \text{if } P(X|i) \geq \omega \end{cases} \tag{4.10}$$

where $P(i)$ is the prior probability of observing class i provided by class skew, $P(X|i)$ is the predicted probability from CNNs. We utilize a hyper-parameter ω as the confidence threshold deciding whether Bayesian Filter should enter the direct pass mode. When a model makes a prediction with high probability ($> \omega$), we believe its prediction is correct and Bayesian Filter will not interfere with the decision. Note that two models with different accuracy—for example, a model A with 70% accuracy and a model B with 95% accuracy—could predict with similar high probability when they are making the correct predictions. Their accuracy difference comes from those frames where the poorer model makes a mistake while the stronger model is still correct, not from those frames where both models are correct. Thus, we select the same threshold ω across different CNNs. In particular, we experiment with diverse ω on an extensive collection of the state-of-the-art CNNs and find that a threshold ω between 75% and 95% exhibits similar performance. By default, we use 90% as the threshold ω in following sections.

4.6 Separability-Aware Model Selection

We propose Separability-Aware Model Selection to enable class-skew based optimizations by exploiting the visual separability. The key observation is that, the same model under different class skew profiles (CSP), even with the same number of classes, may have significantly different accuracy. Figure 4.4 illustrates two CSPs with different visual

Figure 4.4: Class Skews with Different Visual Separability.

separability (*i.e.*, one is easy to classify and the other one is hard). To exploit visual separability, Palleon maintains a set of models with different accuracy and energy consumption, and automatically switches to compact models for saving energy when the detected CSP is easy to classify. We are inspired by the fact that people will relax and spend less energy when objects have significantly different appearance, in contrast to distinguishing similar objects (*e.g.*, cat breeds). To this end, we propose an **Efficient Online Model Selection** (Section 4.6.1) to automatically select models with low resource consumption, and an **Edge-Cloud Duplicated Model Bank** (Section 4.6.2) to reduce model selection overhead and network overhead.

4.6.1 Efficient Online Model Selection

We conduct online model selection on the cloud when we detect class skew switches. There are two baseline strategies. One approach records an average accuracy for each model on all classes, and another approach records the accuracy of one model over all possible CSPs (*i.e.*, multiple accuracy for one model). Both approaches are unsatisfactory. On the former approach [161], the selected model may fail to satisfy the accuracy requirement during runtime since the same model may produce significantly different

	A	B	C	D	E
CSP I	97	91	87	84	81
CSP II	90	88	82	79	77

Figure 4.5: Example of Online Model Selection (Unit:%). Dashed boxes refer to un-profiled models due to binary search.

accuracy on different CSPs. On the latter approach [129], a prohibitive offline profiling overhead and online memory overhead may be introduced due to the huge number of CSPs.

By contrast, we propose a hybrid approach that selects models on the cloud for only class skews detected during runtime. During offline preparation, we profile a single accuracy for each model over all classes and store the model in the order of this accuracy. During online model selection, we use binary search to profile the CNN accuracy on the detected CSP. This binary search leads to logarithm time complexity, compared to the linear time complexity of enumerating all models. Behind the binary search, our key observation is that, while the model accuracy on each CSP may change dramatically, the relative accuracy order of models on all classes stays the same over various class skews. In particular, if one model performs better than another model on all classes, the former one generally performs still better than the latter model on various CSPs. Similar observations have also been made in computer vision area [2, 41] that larger models (e.g., ResNet-50) usually give higher accuracy than smaller ones (e.g., ResNet-18) on the same task. Figure 4.5 illustrates the online model selection. Suppose we have 5 models with decreasing energy consumption and recognition accuracy, and target 90% accuracy. For each CSP, we conduct binary search to find the most compact model with satisfactory accuracy ($> 90\%$). In this case, we will select B for CSP I but A for CSP II.

Cache Service to Avoid Redundant Model Selection. Palleon records the

model selection results along with the CSP and skips model selection for a CSP that have appeared previously. In particular, Palleon maintains a cache service between the CSP and the selected model. When a new CSP comes, Palleon will first retrieve the CSP in the cache. On a cache hit, Palleon immediately returns the selected model. On a cache miss, Palleon conducts online profiling and records the selected model for reuse. In our evaluation, a high cache hit rate is achieved quickly after less than 5 model selections, since the same CSP appears frequently in real videos.

4.6.2 Edge-Cloud Duplicated Model Bank

Palleon's goal of saving energy and improving accuracy is affected by the quality of its model bank. We design a duplicated model bank to store only Pareto-Optimal models and duplicate these models on both the edge and the cloud for reducing network overhead. For each candidate model, Palleon stores the computation graph, the pre-trained weights, and the metadata including energy consumption and latency.

Model Bank Generation with Offline Profiling. For each energy budget, we conduct offline profiling to identify candidate models with the highest accuracy. This offline profiling selects only models on the Pareto-optimal curve to reduce online search space and runtime overhead. Specifically, we first generate a large number of candidate models by applying compression techniques on CNNs. Then, we conduct offline profiling to select models on the Pareto-optimal curve [174], defined as the models that we cannot further reduce energy consumption without worsening the accuracy.

This candidate model generation is *conducted once on all classes*, instead of repeating on different CSPs, since good models on all classes tend to consistently produce good performance over various CSPs. The insight is that *unsalient positions remain similar for all CSPs*. For example, when we repeat a compression technique, Perforation [175], for

Compressed From	Layer Remove Ratio (%)	Filter Remove Ratio (%)	Latency (ms)	Energy (J)
ResNet-50	20	10	65.6	0.63
DenseNet-40	30	10	48.4	0.46
MobileNet-128	30	30	34.5	0.35
MobileNet-128	30	40	20.6	0.18
VggNet-19	60	60	10.1	0.07

Table 4.1: Profiling on selected compact models. Latency and energy are measured on Jetson Nano.

several CSPs on Dense-40 [136], the positions in later blocks will be deleted first while the positions in the leading block remain unchanged until all later blocks have been pruned. More generally, even though the best model (*i.e.*, the one with the highest accuracy under a given reduction in energy consumption) might change between different CSPs, the set of *top-k best* models tends to remain stable over all CSPs. Thus, we can avoid repeating the selection on all CSPs and, under any specific energy-saving requirement, use the best model for all classes to approximate the best model for a specific CSP.

Our full model is a DenseNet-40 with 40 layers. Starting from 4 base models (i.e., MobileNet-128, VGGNet-19, ResNet-50, DenseNet-40), we generate 25 compact models from each of these base models. In particular, we first remove {10%, 20%, 30%, 40%, 60%} of layers from the base model. For the remaining layers, we remove {10%, 20%, 30%, 40%, 60%} of filters. While more sophisticated compression techniques can be applied, we adopt this simple compression technique to validate the effectiveness of model selection. From these pruned models, we select N_{CW} (=5, by default) compact models and put them into our model bank for online use in our evaluation. We have experimented with several numbers and found that 5 compact models can provide a relatively diverse range of accuracy and resource consumption. We show the profiling data on raw latency and raw energy consumption in Table 4.1, measured on Jetson Nano [134]. For each frame, these models have inference latency from 10.1 ms to 65.6 ms and energy consumption

from $0.07J$ to $0.63J$. Here, all pruned models are retrained over all classes during offline model bank generation.

Edge/Cloud Duplication to Reduce Network Overhead. We maintain a duplicated model bank on both the edge and the cloud to avoid weight transportation from the cloud to the edge. The duplicated model banks on the edge and the cloud contain the same deep models and pre-trained weights, while giving each model an index. During online model selection, the cloud will select a model from the duplicated model bank and only send the selected index to the edge. The edge uses the received index to identify the selected model. This design avoids the network overhead of frequently transporting model weights when class skew switches frequently.

4.6.3 System Overhead Analysis

Model Bank Memory Overhead. The model bank introduces negligible memory overhead compared to the simple setting with only large CNNs. In the simple setting, the memory consumption is

$$Mem_{Simple} = Mem_{LW} + Mem_{LF} \tag{4.11}$$

where Mem_{LW} and Mem_{LF} are the memory for storing weights and features of the large CNN, respectively. In our setting with model bank, the memory consumption is

$$\begin{aligned} Mem_{Bank} = Mem_{LW} + max(Mem_{LF}, Mem_{CF}) \\ + N_{CW} \cdot Mem_{CW} \end{aligned} \tag{4.12}$$

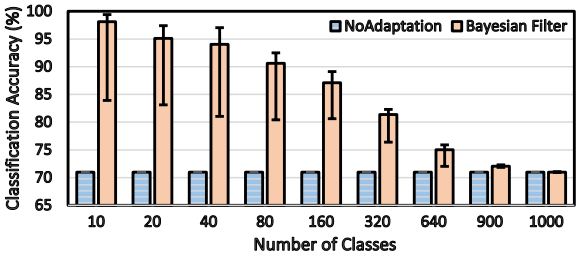
where N_{CW} is the number of compact models, Mem_{CW} and Mem_{CF} are the memory for storing weights and features of compact CNNs. We use $max(\cdot, \cdot)$ on CNN features since each input frame is processed by only one CNN. Comparing Equation 4.11 and 4.12, the model bank only introduces overhead of $N_{CW} \cdot Mem_{CW}$, which is less than 5MB and is negligible compared to the GB-level memory in modern edge devices (GB) (*e.g.*,

1GB in Raspberry Pi 3B+ [176] and 4GB in Jetson Nano [134]). In particular, we use a small N_{CW} (=5, by default), since Bayesian Filter can adapt compact models toward the detected CSPs during runtime with low overhead (Section 4.5). The Mem_{CW} is usually less than 1MB on compact models generated with compression techniques, especially when the base model also consumes negligible memory (*e.g.*, 0.5MB in SqueezeNet [177] and 2MB in MobileNet [41]).

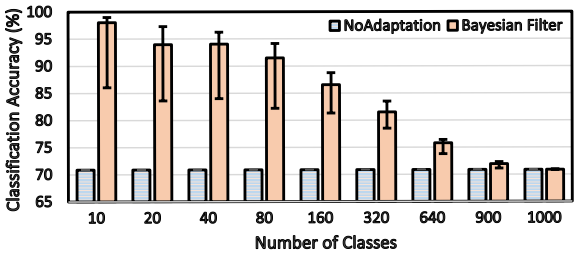
Runtime Overhead. Palleon's runtime overhead comes from three sources. The first is the model selection overhead. While this overhead is relatively large, model selection is conducted on the cloud which is powerful and can evaluate several models concurrently. Also, we have sorted the models and proposed a binary search for accelerating the model selection. This procedure introduces negligible runtime overhead (<1%). The second is the data transfer overhead. Existing work usually transfers frames (around 100 KB per frame) to the cloud, which introduces heavy network overhead. Instead, we summarize the surrounding environment into the CSP (a short string within 1KB) and only need to transport the CSP from the edge to the cloud through a wireless network. This network overhead is low since we only transport CSP instead of data or CNN weights. Besides these two overhead from model selection, the third comes from class skew detection on the edge, which is negligible due to optimizations for low-overhead detection in Section 4.4.2.

4.7 Evaluation

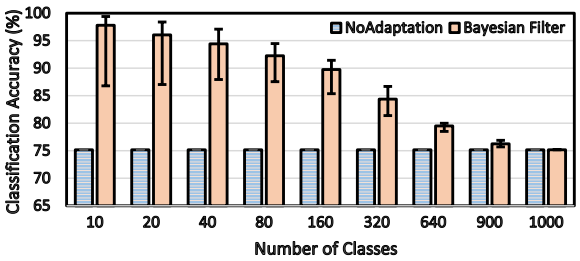
To show the effectiveness of Palleon, we perform extensive experiments on both synthesized videos and real videos. We first evaluate Palleon on **synthesized videos** (Section 4.7.1) to study the performance in diverse settings, including varying class numbers, class types, and lasting time of each class skew. We then conduct **real video experi-**



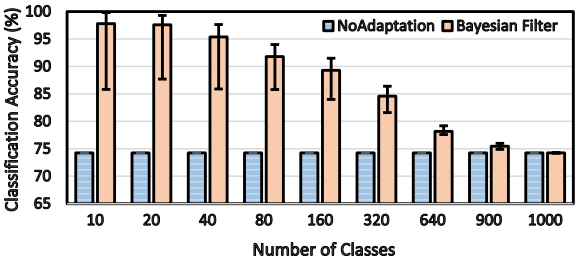
(a) MobileNet with Bayesian Filter.



(b) VGGNet with Bayesian Filter.



(c) ResNet with Bayesian Filter.



(d) DenseNet with Bayesian Filter.

Figure 4.6: Accuracy on Fixed Class Skews. Error bars represent the accuracy range for each number of classes.

ments (Section 4.7.2) to further validate the performance of Palleon for detecting and exploiting class skews during runtime.

Experiment Platform. We have implemented Palleon in Tensorflow [133] for our CNNs. For the edge device, we use NVIDIA Jetson Nano [134] which is a popular mobile GPU platform with wide deployment in robotics [178], AI glasses [179], and doorbell cameras [180]. Jetson Nano runs Ubuntu 18.04 with built-in support for Tensorflow. For the cloud server, we use a Dell Workstation T7910 with an NVIDIA 1080Ti GPU (with 11 GB dedicated memory and a nominal peak performance of 11.3 TFLOPS), a 6-core Intel Xeon CPU E5-2603 processor with 32 GB memory running Ubuntu 18.04. All energy measurements mentioned are directly measured unless otherwise specified, using an Extech EX330 Compact Digital Multimeter [181].

4.7.1 Synthesized Video Experiments

In this section, we extensively evaluate Palleon on synthesized videos in diverse settings. We generate synthesized videos based on ImageNet dataset [182] with diverse class skews. The ImageNet dataset consists of $1,200,000$ images categorized into $1,000$ classes. We generate class skews with varying numbers of classes and diverse lasting time. These synthesized class skews create challenging scenarios and showcase the robustness of Palleon in challenging settings. We train CNNs on ImageNet with all classes and conduct offline profiling to generate a single accuracy over all classes and collect their latency/energy consumption on mobile devices (e.g., Jetson Nano). This offline profiling is conducted only once. During online, we use Bayesian Filter for online model adaptation and do not use online finetuning (*i.e.*, retraining CNNs on the detected CSPs). On dynamic class skews, we also have online profiling about the model accuracy on the detected CSP, which is conducted on the cloud and introduces negligible overhead.

Bayesian Filter on Fixed Class Skews Figure 4.6 shows the accuracy improvement from Bayesian Filter in an ideal case that the true class skew is known and fixed. Under this setting, there is no detection delay and Bayesian Filter can adapt CNNs toward the true class skew. To show the generality of Bayesian Filter, we use four state-of-the-art CNNs as base models (*i.e.*, MobileNet [41], VGGNet [122], ResNet [2], and DenseNet [136]). When synthesizing fixed class skews, for each $N \in \{10, 20, ..., 1000\}$ classes, we generate 100 CSPs. Each CSP contains 1000N images by randomly selecting N classes and 1000 images from each class following a uniform distribution. For each number of classes, we run the adapted model and present the average, minimum, and maximum accuracy. We note that we use Bayesian Filter to adapt the model and do not use online finetuning.

Bayesian Filter provides on average 25% accuracy improvement when there are 10

classes. This accuracy improvement shows the effectiveness of Bayesian Filter in adapting models. As the number of classes increases, this accuracy improvement diminishes gradually until all $1,000$ classes appear in the class skew (*i.e.*, no scenario information to exploit). The reason is that, as the number of classes increases, opportunities to rule out classes decreases. For example, Bayesian Filter rules out 990 classes when the CSP contains 10 (out of 1000) classes, but only rules out 100 classes when the CSP contains 900 (out of 1000) classes. Surprisingly, an accuracy improvement around 2% still exists when there are 900 classes in the class skew, considering that underlying models can only recognize $1,000$ classes. This result shows that Bayesian Filter can consistently improve accuracy on challenging class skews with a large number of classes.

A large accuracy difference exists for each model and each number of classes, demonstrating the existence of visual separability. In particular, an accuracy difference of 15% can be observed for MobileNet when there are 10 classes. This accuracy difference becomes less significant as the number of classes increases, since a smaller number of classes indicates larger variation in the constituent classes. Comparing across models (*e.g.*, ResNet v.s. MobileNet), we see that a model tends to perform better than another model on a specific class skew, if the former model has higher accuracy on all classes than the latter model. This observation indicates that the relative order of model accuracy is invariant over various class skews and supports our binary profiling.

ABLE on Dynamic Class Skews In this section, we show the accuracy improvement when the true class skew is unknown and may switch abruptly, namely *dynamic class skew*. Under this setting, the class skew detector decides the CSP quality and the detection delay, which has a significant impact on the classification accuracy of the adapted models. When synthesizing dynamic class skews, we randomly generate 30 CSPs. For each CSP, we first randomly select a small number (ranging from 10 to 20) of

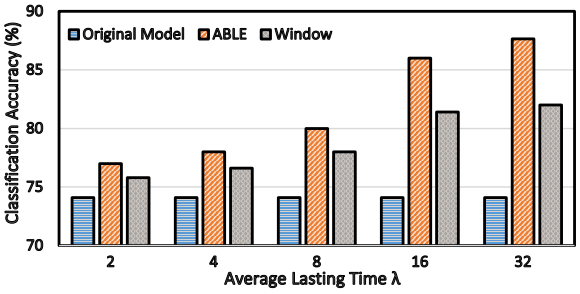


Figure 4.7: Accuracy with Various Detection Methods.

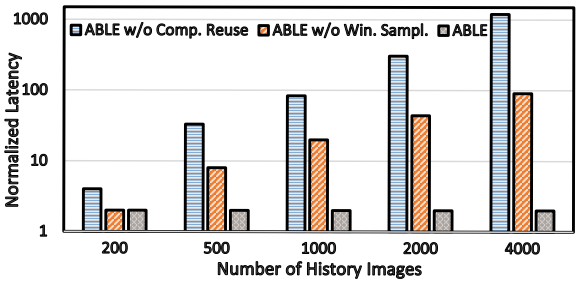


Figure 4.8: ABLE Latency relative to Window Detector.

classes. Among all testing images from a given set of classes, a CSP uniformly samples $lastingTime = 60 * T$ images. T is a random variable following the Poisson(λ) distribution and $\lambda \in \{2, 4, 8, 16, 32\}$ controls the average number of images. We choose the Poisson distribution, as it outputs positive integers. We use DenseNet as the original model and elide the results on other CNNs due to the similar behavior.

Average Accuracy. Figure 4.7 shows the classification accuracy on dynamic class skews when combining Bayesian Filter with two class skew detectors (*i.e.*, Window and ABLE). In the Window detector, we sample a sequence of window sizes for each synthesized video and present the best accuracy for a strong baseline. Comparing across λ, we can see a clear trend that the classification accuracy increases as λ increases. In particular, "ABLE" can increase accuracy by 13.65% when the average lasting time λ reaches 32. This trend indicates that a class skew lasting longer provides more optimization opportunities to exploit. Comparing across detectors, we can see that ABLE achieves higher accuracy improvement around 5% than the window-based detection. The reason is that the lasting time for a specific class skew varies even for a fixed average lasting time λ, such that a fixed window size can hardly hit the balance between CSP quality and detection delay.

ABLE Detection Latency. Figure 4.8 shows the ABLE detection latency reduc-

tion. This detection latency measures the computation overhead of incurring ABLE on an incoming CNN prediction. *Number of history images* is the total number of images in a synthesized video representing the video length, ranging from 200 to 4000 images. “ABLE" represents ABLE with both computation reuse and window sampling where $k = 30$ windows are sampled. “ABLE w/o Win. Sampl." disables window sampling and “ABLE w/o Comp. Reuse" further disables computation reuse. “ABLE w/o Comp. Reuse" shows a quadratic increase in the latency over time, which becomes costly when the number of inputs increases gradually. By adding computation reuse, “ABLE w/o Win. Sampl." decreases this quadratic time complexity to linear time complexity, leading to a much lower computation overhead. When adding window sampling, we can see that “ABLE" further reduces the linear time complexity to a constant complexity, which is similar to the Window detector.

Separability-Aware Model Selection In this section, we show the energy-saving, runtime speedup, and the memory overhead from Separability-Aware Model Selection. To study the impact of lasting time, we adopt the same setting as dynamic class skew (Section 4.7.1). In each dynamic class skew, we randomly generate 30 class skews and report the average energy saving and runtime speedup.

Energy Saving. Figure 4.9 shows energy saving when targeting the same accuracy as the baseline model. Palleon can save energy consumption up to 6.2× while maintaining the accuracy. This benefit comes from automatically replacing the original large model with small models by separability-aware model selection. In addition, we can observe that the energy saving increases as lasting time increases, since a longer lasting time indicates less class skew switches and less system overhead for detecting and exploiting new class skews.

Runtime Speedup. Figure 4.10 shows the overall runtime speedup when targeting

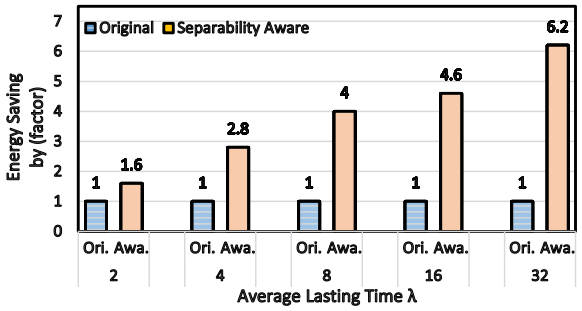


Figure 4.9: Energy Saving with Model Selection.

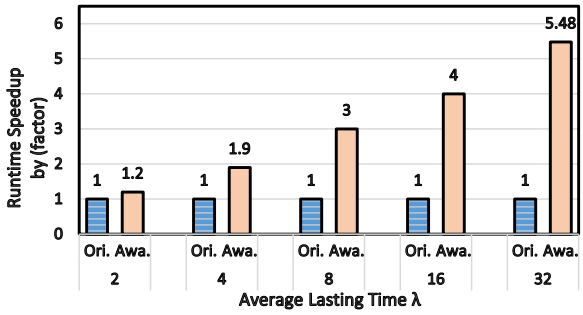


Figure 4.10: Runtime Speedup with Model Selection.

the same accuracy as the baseline model. Palleon can achieve up to 5.48× speedup, considering both the model execution speed and the system latency, including the model adaptation latency, the model selection latency on the cloud, and the network latency. Note that the model selection on cloud only introduces latency and has no impact on the energy consumption on the edge device. Similar to the observation in energy saving, we can observe an increase in runtime speedup as the lasting time increases, due to the reduced system overhead.

Memory and data transfer overhead. We observe negligible memory overhead in our current system with 5 compact models. In particular, these compact models usually consume less than 1MB memory and Jetson Nano has 4GB memory. On the data transfer overhead, we only transfer a short CSP (within 1KB) to the cloud when ABLE detects class skew switches. This is significantly smaller than alternative system designs that transfer frames (around 100 KB per frame) or CNN weights with the cloud.

4.7.2 Real Video Experiments

We evaluate Palleon on real videos to show the end-to-end accuracy improvement, runtime speedup, and energy saving, including the overhead from model adaptation and class skew detection. We compare Palleon with a state-of-the-art energy-efficient video

processing system, FAST [129]. *FAST* approach studies the benefit of class skews in an ideal case, assuming that all class skews that may appear at runtime are known offline. In particular, FAST adapts a large number of compact CNNs towards each class skew during offline preparation and identifies these foreknown class skews with a window detector. As such, we denote it as *FAST (offline)*. For a fair comparison of our online framework, we further extend FAST (offline) to an online version, namely FAST (online). The only difference between these two versions is that FAST (online) does not have the pre-trained CNN models for different class skews. Instead, it adopts a standard retraining method [183], which retrains the last few CNN layers toward the online detected class skews, for online model adaption. To strike a good balance between accuracy and performance, we manually tune the number of layers for retraining and find that two is a good number and use it in our experiments. Different from FAST, we do not foreknow the class skews offline. We conduct online class skew detection with ABLE, online model adaptation with Bayesian Filter, and online model selection.

Real Video Datasets. We evaluate Palleon and FAST on four real videos [129] depicted in Table 4.2. These videos come from several movies for face recognition and have diverse length ranging from 6 minutes to 24 minutes. "#Switch" indicates the number of class skew switches in each video and "#Class" represents the average number of classes (faces) in each class skew between adjacent class skew switches. For example, "Friends" is a 24-minute video with 45 class skew switches and each class skew contains 2.8 classes on average. While the total number of classes in these real videos is large (> 20), each class skew contains only a small portion of classes (2 to 3.5). This is a common case in films and Youtube videos, as we have discussed in introduction. The lasting time for each class skew varies on videos from 10 seconds to 4 minutes (about 1.3 minutes on average), computed by "Len.(min) / #Switch". This diversity makes it a challenging setting to detect and exploite class skews during runtime.

Table 4.2: Real videos for evaluating Palleon. "#Switch" indicates the number of class skews switches and "#Class" shows the average number of classes in each class skew.

Video Name	Len. (min)	#Switch	#Class
Friends	24	45	2.8
Good Will Hunting	14	4	3.5
The Departed	9	8	2.4
Ocean's Eleven / Twelve	6	25	2.0

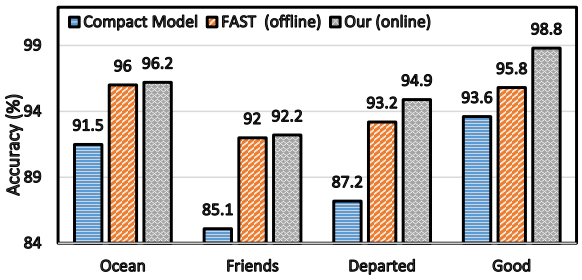


Figure 4.11: Accuracy Improvement on Various Videos.

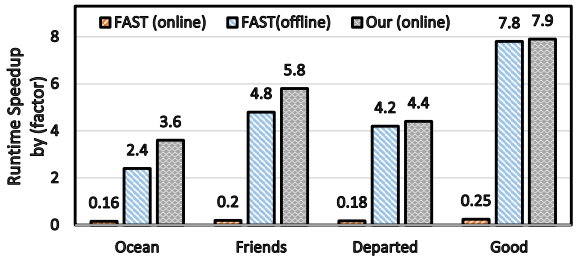


Figure 4.12: Runtime Speedup on Various Videos.

During the detection of faces, we follow the standard two-phase pipeline in object detection [184, 185, 186] and face recognition [187]. We first use a Viola Jones detector [188] to locate faces in video frames, which is agnostic to class skews. Then we crop faces and feed into CNNs for face recognition [137]. Note that this procedure can be easily applied to other object detection tasks by retraining the face recognition CNNs.

Base Model for Real Video Datasets. For a fair comparison with FAST, we choose the state-of-the-art deep model, VGGFace [137], as our full model for face recognition. We generate 5 compact models with diverse resource consumption and accuracy, following the model bank generation in Section 4.6.2. We use the same compact models for FAST. We train these models from scratch following the hyper-parameter setting in FAST, and achieve comparable accuracy as reported. This offline training is conducted once on LFW dataset. During online, we use Bayesian Filter for model adaptation and do not use online finetuning.

Accuracy Improvement. Figure 4.11 shows the overall classification accuracy on real videos of the most *compact VGGFace* model, FAST (offline), and *our online approach*. We skip the accuracy of FAST (online) since it consistently provides lower accuracy than the Fast (offline) by around 1%, since we retrain only the last two layers in Fast (online) to hit a good balance between accuracy and performance. Palleon provides 6.2% accuracy improvement on average, compared to utilizing the compact model without adaptation. This accuracy improvement comes from Bayesian Filter that dynamically adapts models to class skews detected in real videos, containing only 2 to 4 people on average ('#Class' in Table 4.2). This reduced number of faces greatly eases the task compared to recognizing thousands of faces in un-adapted models.

Comparing to FAST (offline) relying on offline adapted models, Palleon provides 1.3% accuracy improvement due to the faster class skew switch detection in ABLE. When a class skew switches, Window detector in FAST (offline) leads to a detection delay up to 10 seconds, during which the accuracy suffers from a dramatic drop. Moreover, ABLE can effectively detect 98% class skew switches while Window detector can only detect 86% class skew switches, since Window detector fails to detect class skews that exist for only a few seconds.

Runtime Speedup. Figure 4.12 shows the end-to-end runtime speedup on real videos when targeting the same accuracy as the full model. Palleon achieves on average 5.43× speedup (up to 7.9× speedup on Good) compared to the full model. This speedup comes from automated model selection by replacing the full model with a compact model. When both assuming that the class skews are not foreknown and adapting models at runtime, Palleon achieves 26.9× speedup over the FAST (online) approach. Indeed, FAST (online) shows a 5× slow down due to heavy overhead from online model adaptation. This comparison demonstrates the efficiency of Palleon in online class skew detection and online model adaptation. For FAST (offline) with strong assumption that true class

skews are foreknown and models are adapted during offline preparation, Palleon can still achieve a higher speedup, due to the early optimization strategy in ABLE. Comparing across videos, the speedup becomes more significant when class skews have longer lasting time (Good Will Hunting), showing the same pattern as evaluations on synthesized videos (Section 4.7.1). We note that Palleon takes 14.2 ms latency on average to process one frame, which is significantly faster than the real-time requirement of 30 ms per frame.

We also observe negligible overhead from model switches ($<$1%) due to several reasons. First, the number of model switches is much smaller than the number of class skew switches, since class skew switches can usually be handled by the Bayesian Filter without model switches. In particular, only 20% class skew switches lead to model switches. Second, the model selection is conducted on the cloud and we cache all models in the memory to avoid the repeatedly loading models, which introduces negligible memory overhead.

Energy Saving. Figure 4.13 shows the end-to-end energy saving on real videos when targeting the same accuracy as the full model. Palleon achieves on average 4.9× energy saving (up to 6.7× on Good) compared to the full model. Palleon achieves a higher energy saving compared to "FAST (offline)" (*i.e.*, without counting the energy consumption in retraining) due to the early optimization strategy in ABLE. FAST (online) conducts model adaptation on the cloud and transfers the adapted model weights (in megabytes) to the edge through the network, leading to extra energy consumption from network communication. This network overhead becomes intensive when class skew switches frequently (Ocean and Friends), leading to more energy consumption in FAST (online) compared to the full model. This overhead reduces when class skews have longer lasting time (Departed and Good), leading to energy saving. By contrast, Palleon shows a consistent benefit on all four videos due to Palleon's low overhead.

Workload Distribution over Models. We observe that most frames are processed

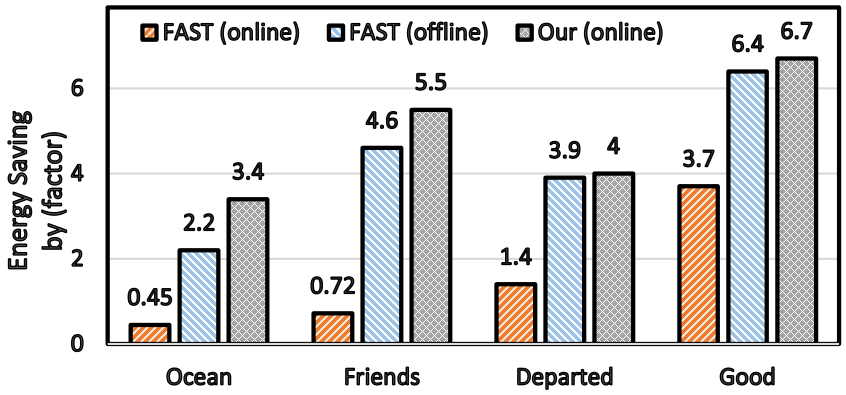


Figure 4.13: Energy Saving on Various Videos.

by the most compact model. For eaxmple, on "Friends" dataset, the most compact model processes 85% frames. While workload distribution varies for different CSPs, we observe similar trends across datasets. The reason is that CSPs usually only contain a few classes (Table 4.2) and the most compact model with Bayesian Filter can provide high accuracy.

4.8 Discussion

Comparison with alternative design that trains a model for each CSP. One alternative design to exploit class skews is to train many small models to recognize only a small set of classes for individual class skews. This alternative design has two intrinsic drawbacks. First, we usually do not foreknow the class skew in an online video such that we can hardly train compact models for each class skew offline. Second, even if we assume that all class skews are foreknown (as the case in FAST), we may need to train a large number of models due to the large number of class skews. By contrast, Palleon does not assume that class skews are foreknown and trains the model on all classes offline. During online video analytics, we use ABLE to conduct online class skew detection, Bayesian Filter for efficient online model adaptation, and separability-aware model selection to automatically select CNNs for balancing the accuracy and resource efficiency.

Generality to other CNNs. Palleon can accelerate a large number of workloads on mobile devices with the temporal locality that a small number of classes keep appearing in a large number of consecutive frames. We have shown the performance benefits of Palleon on object classification and face recognition. Palleon can be generalized to 2D object detection [189] and 3D point cloud analytics [190] which share a similar pipeline as face recognition. We also note that Palleon can benefit from more compact models and pruning techniques designed for mobile systems. In particular, these compact models can be incorporated during model bank generation to provide Pareto-optimal boundary with reduced resource consumption and equivalent accuracy.

Chapter 5

ZEN: Efficient Zero-Knowledge Proofs for Neural Networks

In this chapter, we present ZEN, the first optimizing compiler that generates efficient verifiable, zero-knowledge neural network inference schemes. ZEN generates two schemes: ZEN_{acc} and ZEN_{infer}. ZEN_{acc} proves the accuracy of a committed neural network model; ZEN_{infer} proves a specific inference result. Used in combination, these verifiable computation schemes ensure both the privacy of the sensitive user data as well as the confidentiality of the neural network models. However, directly using these schemes on zkSNARKs requires prohibitive computational cost. As an optimizing compiler, ZEN introduces two kinds of optimizations to address this issue: first, ZEN incorporates a new neural network quantization algorithm that incorporate two R1CS friendly optimizations which makes the model to be expressed in zkSNARKs with less constraints and minimal accuracy loss; second, ZEN introduces a SIMD style optimization, namely stranded encoding, that can encode multiple 8bit integers in large finite field elements without overwhelming extraction cost. Combining these optimizations, ZEN produces verifiable neural network inference schemes with $\mathbf{5.43} \sim \mathbf{22.19}\times$ (15.35× on average) less R1CS

constraints.

5.1 Problem Statement

From health care AI to machine translation, our civilization relies more and more on neural networks. With the increasing adoptions of neural networks, privacy leakage is ever-growing [191, 192, 193]. In particular, we need to protect two kinds of privacy: the privacy of the end user's sensitive data; and the intellectual properties of the neural network that many companies spent millions of dollars in training [194, 195].

In recent years, zero-knowledge proof systems, commonly known as zkSNARKs (zero-knowledge Succinct ARgument of Knowledge) [196, 197, 198, 199, 200, 201, 202, 203, 204], become increasingly efficient. One natural question to ask is:

> *Can we build an optimizing compiler that leverages these powerful zkSNARKs to construct efficient privacy-preserving, verifiable neural network schemes?*

This question is important since a zkSNARK based privacy-preserving and verifiable neural network scheme has many desired and unique features, compared with other popular solutions in this domain, such as secure multi-party computation (MPC) and homomorphic encryption (HE) [205, 206, 207, 208, 209, 210, 211, 212]. To name a few, (1) the computation result will be publicly verifiable; (2) the whole process could be non-interactive.

Many potential applications require these two properties. For example, a patient may want to selectively disclose the diagnosis result by an AI doctor without leaking sensitive personal information. Blockchain oracles may want to put public verifiable results of neural network based inference such as facial recognition, natural language processing on the blockchain to interact with smart contracts, yet not leak the input information.

To support these promising applications, we first propose verifiable neural network inference schemes that protect both the privacy of input and the confidentiality of neural network model, namely ZEN_{acc} and ZEN_{infer}. They can work in combination: during the setup phase (ZEN_{acc}), the neural network inference service provider first commits the model, then takes a random challenge from the user to prove the accuracy of the committed model; then, the service provider can provide verifiable inference using the committed model without leaking the input data (ZEN_{infer}).

However, naively implementing these schemes on zkSNARKs incurs prohibitive cost. The neural network that is useful in the real world is usually defined on floating-point numbers, which is not the first-class citizen in zkSNARKs. Despite previous attempts to improve the efficiency of verifiable floating-point computation [213], there is still a huge performance gap between the floating-point computation compared with "native" arithmetic computations in finite field. Additionally, inference tasks using modern neural network models require a significant amount of computation, which makes constraints size very large and prohibitive performance.

5.2 Overview of Proposed Solution

We develop the first end-to-end optimizer that compiles a floating-point PyTorch model to R1CS constraints, a common intermediate representation (IR) that is supported by a variety of zkSNARK back-ends. First, we observed that neural network quantization has been extensively studied [30, 214, 215, 216, 217, 218, 219] and widely deployed [220, 221, 222, 223]. As a result, instead of adapting zkSNARKs to floating-point computation, we adapt state-of-the-art neural network quantization technique to convert floating-point neural network models to quantized models which only contains low precision integers. To optimize the number of the constraints in R1CS of the quantized

models, we propose two "lossless" R1CS friendly optimizations: sign-bit grouping and remainder-based verification, which reduces the constraint sizes without any accuracy loss compared with models generated by the state-of-the-art quantization technique.

Since quantized neural network models work well on low precision integers (8bit or 16bit) and the underlying zkSNARKs usually works on large finite field element (e.g. 254bit), an intuitive idea is to encode many integers in a single finite field element, similar to SIMD (Single Instruction Multiple Data). However, simply stacking low precision integers in finite field elements would not work since extracting these integers requires expensive bit decompositions, which out-weights the savings. To solve this problem, we propose a novel encoding technique, namely *stranded encoding*, that could bring a SIMD style optimization to batched dot product. We also develop an analytical cost model for stranded encoding so that the encoding scheme can always choose the best parameter based on neural network kernel sizes. Combining R1CS friendly optimizations and stranded encoding, ZEN brings up to 22.19× savings in constraint size compared with a vanilla implementation of neural networks in zkSNARK.

5.2.1 Our Contributions

In this chapter, we present ZEN: an optimizing compiler for verifiable neural networks with zero-knowledge. To our best knowledge, ZEN is the first work of its kind. In short, ZEN makes the following contributions:

- *ZEN schemes:* We propose two privacy-preserving, verifiable inference schemes for neural networks, namely ZEN_{acc} and ZEN_{infer}. These schemes can be used in many promising applications where a publicly verifiable inference result is essential. Additionally, they provide cryptographic privacy guarantees (zero-knowledge) for both **sensitive inputs** and **neural network models**.

- *R1CS friendly quantization:* The first compilation challenge that ZEN faces is converting a floating-point neural network to a *fully quantized* model, so that it could be expressed in Rank-1 Constraint Systems (R1CS), a common intermediate representation used by many proving systems. We proposed a new quantization algorithm, which incorporates two R1CS friendly optimizations, namely sign-bit grouping and remainder-based verification. As a result, compared with the state-of-the-art quantization schemes, ZEN brings up to 73.9× **savings** in R1CS constraints for convolution kernel and up to 8.4× **reduction** for fully connected kernel **without any additional accuracy loss**.

- *Stranded encoding of R1CS Constraints:* To further improve the number of R1CS constraints in the compilation result, ZEN incorporates a new *stranded encoding* technique, which optimally encodes multiple low-precision integers (8bit), that are common to quantized neural networks, with a single finite field element (usually 254bit). A caveat here is that encoding methods always come with some extra cost for extraction. Simply stacking the low-precision integers in the finite field does not work in our scenario, since the extraction coast is prohibitively high. Our stranded encoding mechanism comes with an efficient extraction. Additionally, our mechanism is adaptive, in that it always employs optimal encoding parameters, for any given input. Empirically evaluation shows that stranded encoding leads to up to 2.2× improvement in R1CS constraints for convolution kernel and 3.4× improvement for fully connected kernel.

- *ZEN toolchain:* We build an open-sourced toolchain [224] (Figure 5.1) that takes a floating-point PyTorch model and converts it to ZEN schemes with all the above optimizations. Our evaluation shows that, without incurring any additional accuracy loss, ZEN brings 5.43 $\sim$ 22.19× (15.35× on average) savings, in the number of

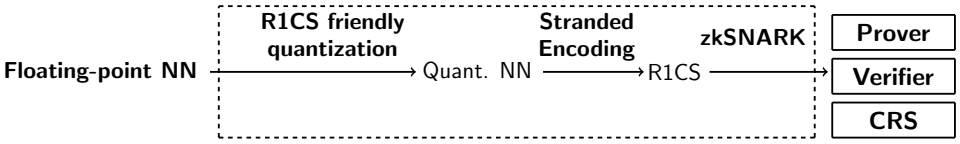


Figure 5.1: Overview of ZEN tool chain.

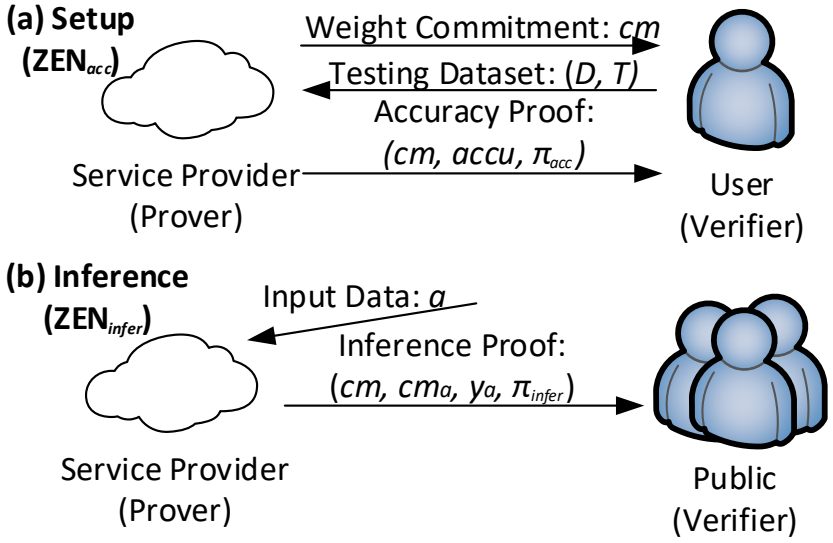


Figure 5.2: Privacy-preserving, verifiable inference workflow.

constraints, compared with a vanilla implementation of the neural network models in zkSNARK.

In a nutshell, while many existing works focus on improving the zero-knowledge proof system backends, ZEN demonstrates a powerful new approach: co-designing the domain-specific algorithms (neural network inference) and constraints compilation process. This new approach brings orders of magnitude improvement on the performance.

5.2.2 Our Techniques

Privacy-preserving, verifiable inference schemes Neural networks are eating the world. We focus on how to let service providers, such as medical AI service, identity service, blockchain oracles [225], to provide verifiable proof of neural network inference

Table 5.1: Overall performance of ZEN_{infer}.

Model-Dataset	Constraints (K)	Linear Combinations (K)	Setup (s)	Comm. (s)	Prove (s)	Verify (s)	CRS Size (GB)
ShallowNet-MNIST	1,751	7,728	37.67	0.310	32.24	1.06	0.506
LeNet-small-CIFAR	2,053	11,870	46.21	0.255	41.71	0.79	0.575
LeNet-medium-CIFAR	12,796	88,485	446.43	1.069	430.97	4.59	3.711
LeNet-Face-small-ORL	9,862	54,200	305.04	1.226	292.07	4.54	3.058
LeNet-Face-medium-ORL	44,204	324,960	2319.02	3.779	2264.18	16.87	13.169
LeNet-Face-large-ORL	162,289	1,225,466	14407.32	7.949	14337.31	66.95	49.577

Table 5.2: Overall performance of ZEN_{acc} scheme.

Model-Dataset	Constraints (K)	Linear Combinations (K)	Setup (s)	Comm. (s)	Prove (s)	Verify (s)	CRS Size (GB)
ShallowNet-MNIST	8,305	75,210	37.99	0.366	32.59	1.05	0.506
LeNet-small-CIFAR	78,481	675,365	46.10	0.265	42.24	0.80	0.564
LeNet-medium-CIFAR	544,921	5,762,645	447.91	1.069	433.31	4.59	3.701
LeNet-Face-small-ORL	265,779	2,395,061	303.75	1.081	288.08	4.51	3.049
LeNet-Face-medium-ORL	1,762,250	21,266,440	2311.41	3.737	2260.45	16.85	13.160
LeNet-Face-large-ORL	5,628,278	77,689,757	14553.10	8.092	14533.57	66.37	49.568

result, without leaking either the user's privacy or the service providers neural network models.

To enable these privacy-preserving verifiable neural network inference applications, we propose two schemes:

- ZEN_{acc}, a verifiable NN accuracy scheme for classification and recognition workloads;
- ZEN_{infer}, a verifiable NN inference scheme for classification and recognition workloads.

We show overall performance results in Table 5.1[1] and Table 5.2. More details will be given in the corresponding section.

[1]In Table 5.1 and Table 5.2, we record the number of linear combination before inlining. It influences the time spent on inlining all linear combinations. We use the constraints of model commitment, inference on **100** images and the final accuracy commitment check circuit as the ZEN_{acc} total constraints. We record the time spent on model commitment, inference on **one** image plus the final accuracy commitment check circuit as the ZEN_{acc} execution time and the CRS size due to parallelizing inference step on testing dataset.

Table 5.3: Overall saving on the number of constraints of ZEN vanilla and ZEN$_{infer}$. Constraints from Poseidon commitment are also shown above. The unit of number of constraints is thousand (K).

Model	**Commitment (K)**	**ZEN vanilla (K)**	**ZEN$_{infer}$ (K)**	**Saving (×)**
ShallowNet-MNIST	1,685	364	67	5.43
LeNet-small-CIFAR	1,281	16,809	772	21.77
LeNet-medium-CIFAR	7,421	85,126	5,375	15.84
LeNet-Face-small-ORL	7,279	57,352	2,585	22.19
LeNet-Face-medium-ORL	26,850	274,490	17,354	15.82
LeNet-Face-large-ORL	107,078	610,797	55,212	11.06

These two schemes are compatible with each other (Figure 5.2). In a typical example, the service provider firstly demonstrates the effectiveness of her model during the setup phase using ZEN$_{acc}$. This is an interactive protocol: she needs to first commit the neural network (NN) into a commitment cm. Then, a user or a trusted third party sends her a random challenge: a dataset and truth label pair (D, T). Next, the service provider returns a zero-knowledge proof π_{acc}, proving that the committed NN maintains an accuracy $accu$ on the input D.

Once this (one time) setup is completed, the service provider begins her service using ZEN$_{infer}$: the service provider collects the input data a from the end user, then generates a zero-knowledge proof π_{infer} to attest the result of inference using the previously committed model.

Both ZEN$_{acc}$ and ZEN$_{infer}$ leak no information of either the neural network model or the sensitive user input by the zero-knowledge property of zkSNARK.

R1CS friendly quantization A crucial part of ZEN is converting neural network models with floating-points to arithmetic circuit (R1CS constraints) on a given finite field. This part is named *quantization*. Although neural network quantization has been extensively studied [30, 214, 215, 216, 217, 218, 219] and widely deployed [220, 221, 222, 223], these existing works do not apply to zkSNARKs, for the following reasons. First,

most schemes are *partial quantization*, in which part of the quantized models remain on signed floating numbers; second, most quantization schemes require operations such as divisions, which are non-atomic in an R1CS; last, directly applying the state-of-art full quantization algorithm [30] will produce a prohibitive number of R1CS constraints.

We address these challenges by proposing an improved, dedicated full quantization algorithm that minimizes the R1CS constraints. In this quantization algorithm, we incorporate two R1CS friendly optimizations (subsection 5.5.2), namely, *sign-bit grouping*, and *remainder-based verification*, to the [30] algorithm. The core idea is to use *algebraic equalities* to bypass expensive bit-decompositions caused by non-atomic operations (e.g., comparisons and divisions):

- With the sign-bit grouping, we completely eliminate the bit-decompositions from element-wise zero-comparisons for *all* kernels.
- With the remainder-based verification, we reduce the bit-decompositions caused by divisions (due to the scale factor) up to 8.4× and 73.9× for fully connected and convolution kernels, respectively.

Both optimizations bring significant savings in terms of the number of constraints in the generated circuits. In addition, since these optimizations only use *algebraic equalities*, the resulted quantized neural network models do not incur any additional accuracy losses, compared with the original quantization algorithm [30].

Stranded encoding of R1CS Constraints This technique comes from a fundamental observation: most quantized neural networks work with low precision (e.g. 8-bit) unsigned integers; on the other hand, most zkSNARKs use ellipitic curves (e.g. BLS12-381 [226]) with an underlying finite field of order $\approx 2^{254}$. It makes sense that if we encode multiple matrix entries into a single finite field element, we should be able to reduce the

number of constraints. This is analogous to the SIMD (Single Instruction Multiple Data) technique that is widely used in modern CPUs and GPUs [227].

As alluded earlier, extraction remains problematic. Indeed, simply stacking many 8-bit unsigned integers into a finite field element would not work since the extraction cost will out-weight the benefits. Instead, we propose a novel encoding scheme, namely *stranded encoding*, which encodes *batched vector dot products* with fewer field operations (section 5.6). Given $\{A_j, B_j\}_{j=1}^{s}$ as inputs, where $|A_j| = |B_j| = n$, to compute s dot products simultaneously, i.e.,

$$(A_1 \cdot B_1), (A_2 \cdot B_2), \ldots, (A_s \cdot B_s)$$

a naïve encoding requires $2ns$ field elements. Our stranded encoding encodes these dot product operations with $2n$ field elements:

$$x_i = \sum_{j=1}^{s} a_{j,i}\delta^{\phi(j)},\ y_i = \sum_{j=1}^{s} b_{j,i}\delta^{\phi(j)}$$

where $x_i, y_i \in \mathbb{F}_p, i \in \{1, \ldots, n\}$. For appropriate parameters δ and $\phi(\cdot)$ (see Definition 5.1), these s dot products could be extracted from the following quantity

$$\sum_{i=1}^{n} x_i y_i = (A_1 \cdot B_1)\delta^{2\phi(1)} + \ldots + (A_2 \cdot B_2)\delta^{2\phi(2)} +$$

$$\ldots + (A_s \cdot B_s)\delta^{2\phi(s)},$$

and the cost is not much different from a single extraction.

A caveat here is to pack as many matrix coefficients into a field element, while still allowing a proper $\phi(\cdot)$ to extract $A_1 \cdot B_1, \ldots, A_s \cdot B_s$ correctly. We formulate this as a discrete optimization problem; and develop a cost model for stranded encoding, so that our implementation automatically chooses the optimal batch size s for any instance.

Implementation and evaluation To evaluate the effectiveness of ZEN, we implemented both ZEN-vanilla, a straight-forward implementation of existing full quantization scheme [30], and the fully optimized ZEN toolchain. We summarize our benchmark

Table 5.4: Case study: benefits of optimizations on the number of constraints for the four kernels in LeNet-5-Medium CIFAR-10. Opt. Lv1 only includes sign-bit grouping; Opt. Lv2 adds remainder-based verification; Opt. Lv3 adds stranded encoding. We note that our current optimizations did not focus on ReLU, since it only contributes to a very small percentage (0.1%) of constraints.

Kernels	**ZEN**-vanilla	**Opt. Lv1**	**Opt. Lv2**	**Opt. Lv3**
Conv	69,692,928	28,630,272	8,082,688	5,195,008
FC	394,000	219,706	132,490	54,906
AvgPool	14,925,312	4,982,976	7,872	7,872
ReLU	114,227	97,920	97,920	97,920

results in Table 5.3. We see an improvement of 5.43∼22.19× (15.35× on average), depending on the inference model. Further discussions will be given in the corresponding sections.

As one shall see, while the verification speed is more or less stable, the proving cost increases drastically with the increase of the number of constraints. We make various optimizations that reduce the number of constraints. See Table 5.4 for a highlight of optimizations on LeNet-5-Small for CIFAR-10.

Our code is open-sourced on GitHub [224] .

Our underlying zero-knowledge proof scheme is from the celebrated work of Groth [228]. We remark that the selection of underlying zero-knowledge proof systems is largely orthogonal to our ZEN design. In particular, our optimization is independent of the underlying proving system, and we expect to see similar gains from our optimizations for other proving systems. With [228], our proof size is always a constant, i.e., 192 bytes for our choice of parameters. Verifying a proof can be done in a few seconds in all cases. This feature may be particularly appealing in practical use cases such as blockchains, where decisions (whether authentication passes or not) need to be made almost instantly.

5.2.3 Related work

Verifiable machine learning. We first compare ZEN with existing verifiable machine learning systems. Ginger [213] is an argument system that supports floating-point computation. Despite the improvement, Ginger's concrete efficiency is still far behind verifying arithmetic computation that can be directly expressed in finite fields. As a result, ZEN's approach introduces quantization techniques to avoid expensive zero-knowledge proof systems for floating-point values.

SafetyNet [229] implements a specialized interactive proof protocol for verifiable execution of a class of deep neural networks. Compared with SafetyNet, ZEN has three major advantages: first, ZEN's inference schemes is non-interactive; second, SafetyNet still leaks the neural network model, such as the weights, which are usually treated as trading secrets of the service providers; last, SafetyNet only supports quadratic activation functions, which are usually considered less effective than the widely used ReLU activation function [230].

zkDT [231] implements verifiable inference and accuracy schemes on decision trees. zkDT proposes protocols with tree-specialized commitment design. Verifiable inference and accuracy schemes for neural networks pose a whole different set of technical challenges. We also note that neural network inference itself has more complicated application scenarios, such as computer vision, natural language processing, and computational biology, compared with a limited scope of decision tree models.

vCNN [232] proposes a verifiable inference scheme for neural networks with zero-knowledge. ZEN differentiates from vCNN in the following aspects. First, vCNN requires a specific built underlying zero-knowledge system as a mixing of QAP [233], QPP [234] and CP-SNARKs [235]. Instead, ZEN generates optimized R1CS constraint which is used by many different zero-knowledge systems and shows large generality. Second,

vCNN only optimizes convolution, while ZEN optimizes various kinds of NN kernels. As a result, ZEN's optimization would still be valid in newer generations of NNs (such as transformers [5], which do not use convolution) while vCNN cannot. Last, ZEN includes an accuracy statement, so that the effectiveness of neural networks from the service provider is verifiable, rather than being blindly trusted.

zkCNN [236] explores a different trade-off point in zk NN design, namely trading non-interactivity for better performance. We note that zkCNN is specialized for scenarios where an interactive protocol is permitted. However, in many potential use cases of ZEN, such as blockchain oracles, non-interactive is crucial.

To the best of our knowledge, ZEN is the first end-to-end optimizing compiler that compiles neural network models to R1CS constraints.

Other privacy models for machine learning. Many research efforts [237, 238, 239, 240, 241, 242, 231] have been devoted to the security and privacy in machine learning recently. These works largely fall into three categories. The first approach [207, 208, 205] utilizes homomorphic encryption to execute machine learning models on encrypted data. The second approach [243, 12, 244, 206] builds upon the multi-party computations (MPC), enabling multiple parties with local datasets to learn the same machine learning model on the aggregated datasets, while preserving privacy for individual's data. The third approach [245, 246, 247] adopts differential privacy (DP) to ensure that the individual data points in a large dataset will not be leaked even if they have been utilized to train a machine learning model. The privacy models of these works are largely orthogonal to ZEN since none of these work can provide a *publicly verifiable* inference result with both user input data and the service provider model private.

Zero-knowledge proof systems A large body of zero-knowledge proof systems [197, 198, 199, 200, 201, 202, 203, 204] have been proposed to facilitate various use cases. These systems usually come with diverse setups, verification time, and proof size, leading to

trade-offs in these dimensions. In particular, we focus on the scheme developed by Groth [228] and the Arkworks implementation [248, 204] to provide constant size proofs and millisecond-level verifications. We stress again that the selection of zero-knowledge proof systems is largely orthogonal to our ZEN design; our design is applicable to other R1CS-based proof systems and will deliver various performance preference, suiting dedicated use cases.

Neural network quantization Quantization has been widely studied [30, 214, 215, 216, 217, 218, 219] to accelerate neural networks by replacing `float32` data with low-precision data (*e.g.*, `float16`). Apple and XORNet.ai [220, 221] have deployed quantization of neural network to accelerate neural network on devices with resource constraints. Facebook [222] and Google [223] have integrated quantization as standard feature of PyTorch and TensorFlow due to its wide usage. However, existing quantization techniques are usually not R1CS friendly due to two reasons. First, most quantization techniques [214, 215, 216] involve floating-point data. However, applying neural network models in zkSNARKs requires full quantization: converting a floating-point neural network model to a neural network model consisting of only integer arithmetic. Second, for a few quantization techniques [30] with full quantization, it still involves negative values and division operations, which cannot be efficiently supported in zkSNARKs. We use the full quantization scheme in [30] as in our baseline system ZEN-`vanilla`, since it establishes state-of-the-art accuracy on `uint8` quantization with minimal accuracy loss, for a variety of real-world NNs.

5.3 Background

We briefly recall zk-SNARK in subsection 5.3.1. Then, we explain the necessary background on neural networks to understand our optimizations in subsection 5.3.2.

5.3.1 Cryptography

The major cryptographic building block used in this chapter is zk-SNARK, formally *(publicly-verifiable, preprocessing,) zero-knowledge Succinct Non-interactive ARgument of Knowledge*. A zk-SNARK is defined in the context of arithmetic circuit satisfiability. A more formal definition can be found in [249].

We denote a finite field of order p as $\mathbb{F}_p$. An $\mathbb{F}_p$-arithmetic circuit is a circuit whose inputs and outputs are from $\mathbb{F}_p$. We consider circuits that have an input $x \in \mathbb{F}_p^n$ and a witness $w \in \mathbb{F}_p^h$. We restrict the circuits to the ones with only *bilinear gates*, *i.e.* addition, multiplication, negation, and constant gates with input $y_1, \ldots, y_m$ is bilinear if the output is $\langle \vec{a}, (1, y_1, \ldots, y_m) \rangle \cdot \langle \vec{b}, (1, y_1, \ldots, y_m) \rangle$ for $\vec{a}, \vec{b} \in \mathbb{F}_p^{m+1}$.

It is not hard to convert boolean circuits to arithmetic circuits via bit decomposition. We define arithmetic circuit satisfiability as follows:

Definition 5.3.1 *The arithmetic circuit satisfiability of an $\mathbb{F}_p$-arithmetic circuit $C : \mathbb{F}_p^n \times \mathbb{F}_p^h \to \mathbb{F}_p^l$ can be defined by the relation $\mathcal{R}_C = \{(x, w) \in \mathbb{F}_p^n \times \mathbb{F}_p^h : C(x, w) = 0^l\}$ and the language $\mathcal{L}_C = \{x \in \mathbb{F}_p^n : \exists\ a \in \mathbb{F}^h\ s.t.\ C(x, w) = 0^l\}$.*

A zk-SNARK for $\mathbb{F}_p$-arithmetic circuit satisfiability is a triple of polynomial time algorithms, namely $(\mathsf{Gen}, \mathsf{Prove}, \mathsf{Verify})$:

- $\mathsf{Gen}(1^\lambda, C) \to (\mathsf{pk}, \mathsf{vk})$. Using a security parameter λ and an $\mathbb{F}_p$-arithmetic circuit C as inputs, the key generator Gen randomly samples a *proving key* pk and a *verification key* vk. These keys are considered as public parameters $\mathsf{pp} := (\mathsf{pk}, \mathsf{vk})$, and can be used any number of times to prove/verify the membership in $\mathcal{L}_C$.

- $\mathsf{Prove}(\mathsf{pk}, x, w) \to \pi$. Taking a proving key pk, and any $(x, w) \in R_C$ as inputs, the Prove algorithm generates a non-interactive proof π for the statement $x \in \mathcal{L}_C$.

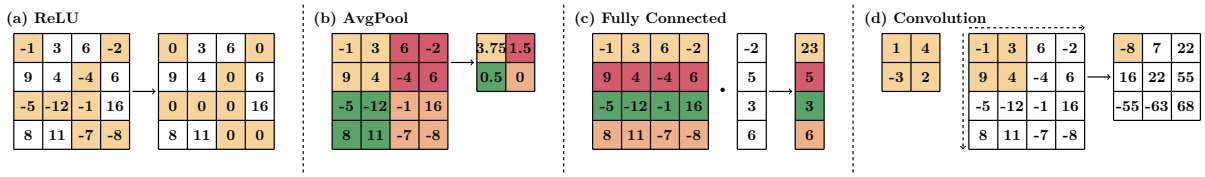


Figure 5.3: Popular neural network kernels

- $\mathsf{Verify}(\mathsf{vk}, x, \pi) \to \{0, 1\}$. Taking the verification key vk, public input x, and proof π, the Verify algorithm output 1 is the verification success, *i.e.* the verifier is convinced that $x \in \mathcal{L}_C$.

Remark 5.3.2 *In practice, the prover may segment the proving processing into multiple stages. For example, it may commit to an input and publish the commitment first, and at a later stage generates the proof. It may even commit different parts of the inputs independently and separately. For simplicity, for the rest of the paper, we will model this whole process as a single function. It is straightforward to see that our security features remain intact in this simplified model.*

A zk-SNARK has the following properties:

- **Completeness.** For any security parameter λ, any $\mathbb{F}_p$ arithmetic circuit C, and $(x, w) \in \mathcal{R}_C$, an honest prover can convince the verifier, namely that the verifier will output 1 with probability $1 - negl(\lambda)$ in the following experiment: $(\mathsf{vk}, \mathsf{pk}) \leftarrow (1^\lambda, C)$; $\pi \leftarrow \mathsf{Prove}(\mathsf{pk}, x, w)$; $1 \leftarrow \mathsf{Verify}(\mathsf{vk}, x, \pi)$.

- **Succinctness.** An honestly-generated proof π has $O_\lambda(1)$ bits and $\mathsf{Verify}(\mathsf{vk}, x, \pi)$ runs in time $O_\lambda(|x|)$ [2].

[1] The concrete numbers are from [203].
[2] $O_\lambda(\cdot)$ hides a fixed polynomial factor in λ.

- **Proof of Knowledge.** If the verifier accepts a proof output by a computationally bounded prover, the prover must know a witness for a given instance. This is also called *soundness against bounded provers*. More precisely, for every $poly(\lambda)$-size adversary $\mathcal{A}$, there is a $poly(\lambda)$-size extractor $\mathcal{E}$ such that $\mathsf{Verify}(\mathsf{vk}, x, \pi) = 1$ and $(x, w) \notin \mathcal{R}_C$ with probability $negl(\lambda)$ in the following experiment: $(\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{KeyGen}(1^\lambda, C)$; $(x, \pi) \leftarrow \mathcal{A}(\mathsf{pk}, \mathsf{vk})$; $w \leftarrow \mathcal{E}(\mathsf{pk}, \mathsf{vk})$.

- **Zero Knowledge.** An honestly generated proof is zero knowledge. Specifically, there is a $poly(\lambda)$-size simulator Sim such that for all stateful $poly(\lambda)$-size distinguishers $\mathcal{D}$, the probability of $\mathcal{D}(\pi) = 1$ on an honest proof and on a simulated proof is indistinguishable.

zk-SNARK's security can be reduced to knowledge-of-exponent and variants of Diffie-Hellman assumptions in bilinear groups [250, 251, 252]. Although the knowledge-of-exponent assumption is considered fairly strong, Gentry and Wichs showed that assumptions from this class are likely to be inherent for efficient, non-interactive arguments for NP relations [253].

There are a number of zero-knowledge proof systems proposed in recent years [197, 198, 199, 200, 201, 202, 203, 204]. In this chapter, we use Arkworks implementation [248] that was part of [204]. We will use the scheme by Groth [228], commonly referred to as Groth16, to generate and verify proofs. This scheme is the state-of-the-art in terms of proof size and verifier efficiency.

In a typical Groth16-type of proving system, the statements that are to be proved are translated into a so-called Rank-1 Constraint System (R1CS). To prove that the prover knows some secret inputs that satisfy the given statements, is then converted into the satisfiability of the R1CS over any points over the field. The proof is therefore a demonstration that the R1CS is satisfied at a random point, which, as a prior, is agreed

upon a trusted setup (result into the so-called *common reference string*) and remains unknown to both prover and verifier. Without going into further details, we note that the overall cost of the proving system is dominated by the number of constraints in the R1CS.

Apart from zk-SNARK, we use a cryptographic commitment scheme as a building block, formally, $\mathsf{COMM} : \{0,1\}^{O(\lambda)} \times \{0,1\}^* \to \{0,1\}^{O(\lambda)}$. We require the scheme to be both binding and hiding. Looking ahead, we will be using Pedersen commit which is statistically hiding and computationally binding under well-accepted assumptions.

5.3.2 Neural network based classification and recognition

Neural network (NN) brings significant advancement in computer vision [2, 254], natural language processing [255, 256] and computational biology [257, 258]. Abstractly, An NN can be viewed as a function $Y = f(X_0)$ that takes an input $X_0 \in \mathbb{R}^{c \times h \times w}$ and generates an output $Y \in \mathbb{R}^m$: $f = f_l \circ f_{l-1} \circ \cdots \circ f_1$ is usually composed by a sequence of kernels. Each kernel f_i (also called a single layer in NN) is usually one of the following 4 "elementry" matrix operations (illustrated in Figure 5.3):

1. *ReLU* [230] kernel applies a simple non-linear activation function (*i.e.*, $\mathrm{ReLU}(X) = max(X, 0)$) element-wisely to the input matrix. This kernel enables NNs to learn non-linearity patterns of the input data.

2. *Average pool* kernel splits the input data spatially into a set of $r \times r$ grid (r=2 in Figure 5.3(b)) and computes the mean over all values in each grid. This kernel spatially summarizes local image features and extract high-level features to facilitate pattern recognition.

3. *Fully connected* kernel takes two inputs (*i.e.*, a weight matrix $W \in \mathbb{R}^{m \times n}$ and a data matrix $X \in \mathbb{R}^n$) and computes an output matrix $W \cdot X \in \mathbb{R}^m$. This fully

connected kernel mixes signals from individual pixels and extra high-level features based on the learnable weight matrix W.

4. *Convolution* kernel slides a weight matrix W along the height and width of the input matrix X, retrieves a slice of input matrix with the same spatial size (2×2 in Figure 5.3(d)), and computes the dot product to generate a real number. This kernel mixes local signals (*e.g.*, local edges and angles) in contrast to fully connected kernel that mixes global signals over all pixels. Note that convolution kernel can be transformed into matrix-matrix multiplications [259], which can be computed similar as fully connected kernel.

Two prominent use cases of NNs are classification [2, 254] and recognition [260, 261]. The classification task puts an input into one of m candidate classes (*e.g.*, cat, dog, and house). In this case, the final layer f_l is a softmax function that outputs $Y = [y_1, y_2, ..., y_m]$ which satisfies $0 \leq y_j \leq 1$ and $y_1 + \cdots y_m = 1$. As a result, y_j can be treated as the probability that the input image X_0 has class j. The classification result is the class with the highest predicted probability: $\hat{y} = \text{argmax}_j\, y_j$.

The recognition task compares an input X to a reference input X_R and decides whether the input and the reference input are a same object. Different from classification, the final layer f_l takes neural embeddings of both the input and the reference, and then computes their distance d in a metric space (*e.g.*, Eculidean space). These two objects are decided as a same object, if their distance d is smaller than a pre-defined threshold τ (e.g. 0.5).

5.4 ZKP for accuracy and inference

In this section, we present our constructions for accuracy and inference. We first introduce a verifiable neural network accuracy scheme that proves the accuracy of neural network models without revealing the weights of the models (subsection 5.4.1). We then introduce a verifiable neural network inference scheme for classification and recognition (subsection 5.4.2). Here, we consider a quantized neural network $\mathcal{Q} : \mathbb{F}_p^{d_1} \to \mathbb{F}_p^{d_2}$ which is a mapping from a size d_1 vector over $\mathbb{F}_p$ to a size d_2 vector on the same field. While neural networks are usually floating-point models, we defer our R1CS friendly quantization to section 5.5.

We remark that we will use Groth16 method to generate proofs. This implies that the prover and the verifier need to agree on certain common reference string (CRS) that is generated through either a trusted third party, or a multiparty computation protocol. For a dedicated verifier use case, it may also be sufficient for the verifier to generate the CRS.

5.4.1 $\mathbf{ZEN}_{acc}$

We present ZEN_{acc}, a *zero-knowledge, verifiable neural network accuracy scheme*, that works for the following typical scenario: the prover firstly commits to a private neural network, then proves the prediction accuracy of the committed neural network, on either a public or a verifier-chosen testing dataset. The verifier learns nothing about the model except its prediction accuracy.

Formally, we consider a testing dataset $\mathcal{D} = \{a_1, a_2, \ldots, a_n\}$ and the corresponding truth labels $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$ for the testing dataset. In classification task, $t_i \in \{1, 2, ..., m\}$ indicates whether the image a_i contains object t_i. In recognition task, $t_i \in \{0, 1\}$ indicates whether the current image a_i contains the same person as a reference

image a_{ref}. Given a neural network $\mathcal{Q}$, we define a *classification computation routine* as: a) running the neural network Q on the testing dataset $\mathcal{D}$ provided by the verifier to get prediction results $\mathcal{Y} \leftarrow \mathcal{Q}(\mathcal{D})$; b) comparing $\mathcal{Y}$ with their truth label $\mathcal{T}$ to obtain the accuracy, denoted by *accu*, of the neural network prediction.

For recognition task, we additionally define a distance metric $\mathcal{L} : \mathbb{F}_p^{d_2} \times \mathbb{F}_p^{d_2} \rightarrow \mathbb{F}_p$ over the embedding space, $\tau \in \mathbb{F}_p$ be the agreed threshold, and a ground truth embedding y_g. Given a neural network $\mathcal{Q}$, we define a *recognition computation routine* as: a) running the neural network $\mathcal{Q}$ on $\mathcal{D}$ to obtain the embedding results $\mathcal{Y} = \{y_1, y_2, \ldots, y_n\}$; b) for each y_i in $\mathcal{Y}$, calculate $\mathcal{L}(y_i, y_g) \leq \tau$ to obtain the recognition prediction results $\mathcal{R}$; c) comparing $\mathcal{R}$ and $\mathcal{T}$ and obtaining the recognition accuracy results *accu* on testing dataset $\mathcal{D}$.

Based on these two computation routines, we define a zero-knowledge, verifiable neural network accuracy scheme (ZEN_{acc}) as the following algorithms:

- $(\mathsf{pk}, \mathsf{vk}) \leftarrow \text{ZEN}_{acc}.\mathsf{Gen}(1^\lambda, \mathcal{Q})$: given a security parameter λ and a quantized neural network model $\mathcal{Q}$ for classification or recognition tasks, randomly generate a proving key pk and a verification key vk.

- $cm \leftarrow \text{ZEN}_{acc}.\mathsf{Commit}(\mathcal{Q}, r)$: given a random opening r, the prover commits to the neural network $\mathcal{Q}$ with r, i.e., $cm \leftarrow \mathsf{COMM}(r, \mathcal{Q})$.

- $(accu, \pi_{acc}) \leftarrow \text{ZEN}_{acc}.\mathsf{Prove}(\mathsf{pk}, \mathcal{D}, \mathcal{T}, \mathcal{Q})$: Upon receiving the commitment, the verifier sends a testing dataset $\mathcal{D}$ to the prover. Then the prover runs either classification or recognition computation routine for each data sample in $\mathcal{D}$, compares with the predictions with their truth labels in $\mathcal{T}$ and outputs *accu*, the prediction accuracy of the neural network $\mathcal{Q}$. Finally, the prover generates a proof π_{acc} for this computation routine.

- $\{0,1\} \leftarrow ZEN_{acc}.\mathsf{Verify}(\mathsf{vk}, cm, \mathcal{D}, \mathcal{T}, accu, \pi_{acc})$: given the verification key vk and proof π_{acc}, verifies if the following statements are correct: the number of correct predictions on dataset $\mathcal{D}$ with model $\mathcal{Q}$ is $accu$; cm is a commitment to $\mathcal{Q}$.

5.4.2 $\mathbf{ZEN}_{infer}$

Our zero-knowledge, verifiable neural network inference scheme (ZEN_{infer}) assumes the prover keeps the neural network private, but will publicly commit to it (binding). At a later time, the prover generates a proof for the result of the classification or recognition. Upon receiving the model commitment and the proof, the verifier checks if the proof is valid or not. During the whole process, the prover's neural network is kept secret (hiding).

Formally, we consider a public input $a \in \mathbb{F}_p^d$ and a neural network $\mathcal{Q}$. We define the *classification inference routine* as running the neural network $\mathcal{Q}$ on the input a to get $y_a \leftarrow \mathcal{Q}(a)$. For the recognition task, we additionally have the distance metric $\mathcal{L}$, agreed threshold τ, and the ground truth embedding y_g. We define the *recognition inference routine* as running the neural network $\mathcal{Q}$ on a to obtain the embedding y_a and calculating $\mathcal{L}(y_a, y_g) \leq \tau$ to obtain recognition predictions.

Based on these two inference routines, we define a zero-knowledge, verifiable neural network based inference scheme (ZEN_{infer}) as the following algorithms:

- $(\mathsf{pk}, \mathsf{vk}) \leftarrow \text{ZEN}_{infer}.\mathsf{Gen}(1^\lambda, \mathcal{Q})$: given a security parameter λ and a quantized neural network model $\mathcal{Q}$ for classification, randomly generate a proving key pk and a verification key vk.

- $(cm, cm_a, y_a, \pi_{inf}) \leftarrow \text{ZEN}_{infer}.\mathsf{Prove}(\mathsf{pk}, a, r, s, \mathcal{Q})$: given an input $a \in \mathbb{F}_p^d$ and a random opening r and s, the prover commits to the input with r and the neural network model with s, i.e., $cm_a \leftarrow \mathsf{COMM}(r, a)$ and $cm \leftarrow \mathsf{COMM}(r, \mathcal{Q})$, respectively. Then the prover runs the classification or the recognition inference routine

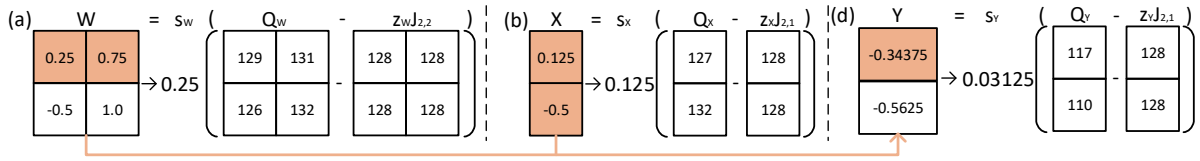


Figure 5.4: Illustration of Partial Quantization when computing $Y = WX$.

to get a result. Finally, the prover generates a proof π_{inf} for the above process.

- $\{0, 1\} \leftarrow ZEN_{infer}.\mathsf{Verify}(\mathsf{vk}, cm, cm_a, y_a, \pi)$: validate input a's inference result on model $\mathcal{Q}$ given the verification key vk, a's commitment, $\mathcal{Q}$'s commitment, a's inference result, and a zero-knowledge proof π_{inf}.

5.5 R1CS Friendly Quantizaiton

In this section, we introduce our R1CS friendly quantization. Recall that popular neural networks require arithmetic computation on floating-point numbers (*e.g.*, float32). However, zero-knowledge systems work over finite fields where data is represented by large unsigned integers (*e.g.*, uint256). This poses special challenges on the operation and data, namely, converting floating points to non-negative integers, and handling divisions. To bridge this gap of numerical data types, we first integrate the full quantization scheme in [30] to the existing zkSNARK library and achieve our baseline system, ZEN-vanilla. Then, we introduce two R1CS friendly optimizations on top of our baseline system, to significantly reduce the number of constraints while maintaining an equivalent accuracy. Last, we demonstrate these R1CS friendly quantizations can be applied to 4 major neural network kernels: fully connected, convolution, average pool, and ReLU.

5.5.1 Baseline: ZEN-vanilla

As discussed in subsection 5.3.2, given a floating-point weight matrix W and a floating-point input matrix X, a neural network kernel computes the output matrix Y as follows:

$$Y = WX, \quad W \in \mathbb{R}^{m \times n}, X \in \mathbb{R}^{n}, Y \in \mathbb{R}^{m} \tag{5.1}$$

The first step of quantization is to generate floating-point scale parameters ($s_Y, s_W, s_X \in \mathbb{R}$) and the lifted zero points ($z_Y, z_W, z_X \in uint$) for each matrix, as illustrated in Figure 5.4. As a result, we have the quantized representation:

$$Y = s_Y(Q_Y - z_Y\mathbf{J}_{m,1}) \quad W = s_W(Q_W - z_W\mathbf{J}_{m,n})$$
$$X = s_X(Q_X - z_X\mathbf{J}_{n,1})$$

Here, $\mathbf{J}_{k,l}$ represents a $k \times l$ matrix of ones.

During neural network computation, we can substitute Y, X, and W in Equation 5.1 with their quantized representation:

$$s_Y(Q_Y - z_Y\mathbf{J}_{m,1}) = s_W s_X(Q_W - z_W\mathbf{J}_{m,n})(Q_X - z_X\mathbf{J}_{n,1})$$

The second step is to replace the floating-point scale parameters with unsigned integers and enable the full quantization computation:

$$M = \lfloor 2^k \frac{s_W s_X}{s_Y} \rfloor$$
$$Q_Y - z_Y\mathbf{J}_{m,1} = M(Q_W - z_W\mathbf{J}_{m,n})(Q_X - z_X\mathbf{J}_{n,1})/2^k$$

By multiplying with 2^k for a large k (=22 by default), we preserve the precision of the floating-point scale parameters in an unsigned integer.

Efficiency Challenges in ZEN-vanilla. While ZEN-vanilla produces effective privacy-preserving and verifiable neural network models, it is not efficient due to its large number of constraints. There are two main reasons coming from [30]. First, [30] is developed for signed `int8` on integer-only hardware, allowing negative elements in $(Q_W - z_W\mathbf{J}_{m,n})$ and

$(Q_X - z_X \mathbf{J}_{n,1})$ of Equation 5.2. However, zk-SNARK supports only finite field arithmetic and requires expensive sign checks for signed integers.As a result, for $mn+n$ elements in $(Q_W - z_W \mathbf{J}_{m,n})$ and $(Q_X - z_X \mathbf{J}_{n,1})$, one may still need $O(mn)$ sign-checks in the generated constraints, where each sign-check needs expensive bit-decomposition. Considering that Equation 5.2 accounts for most computations in neural networks in terms of fully connected kernels and convolution kernels, this would lead to significant overhead. Second, [30] involves division operations while the division operation is non-atomic in zk-SNARK systems. To naïvely support this division operation, we need to first conduct the expensive bit-decomposition. Then, we need to drop the n least significant bits and pack the rest back to enforce equality in Equation 5.2. While this strategy allows verifying Equation 5.2, it would introduce heavy overhead from the bit decomposition.

5.5.2 R1CS Friendly quantizations

In this section, we introduce two R1CS friendly optimizations on top of the baseline quantization scheme, namely, sign-bit grouping and remainder-based verification. Here, we focus on fully connected kernel and convolution kernel. Both can be viewed as matrix multiplications (Equation 5.1). We defer the discussion of ReLU kernel and average pool kernel to the next section. Both optimizations use algebraic equalities to reduce the number of expensive bit-decomposition operations in zkSNARK, while maintaining the semantics of the quantization. As a result, our techniques incur similar accuracy loss as [30]. Nonetheless, we note that [30] itself introduces accuracy loss.

Sign-bit grouping. In ZEN-vanilla, the constraints for a forward step on each layer is generated by Equation 5.2. Our *sign-bit grouping* first reformulates Equation 5.2 to:

$$Q_Y = z_Y \mathbf{J}_{m,1} + M(Q_W - z_W \mathbf{J}_{m,n})(Q_X - z_X \mathbf{J}_{n,1})/2^k$$

Here, both sides are guaranteed to be positive. While $(Q_W - z_W)$ and $(Q_X - z_X)$ may have negative elements, we use the associativity of matrix multiplication to group operands of the same sign:

$$G_1 = Q_W Q_X,\ G_2 = z_X Q_W,\ G_3 = z_W Q_X,\ M' = \lfloor \frac{z_Y 2^k}{M} \rfloor$$
$$Q_Y = M(G_1 + n z_W z_X \mathbf{J}_{m,1} + M' \mathbf{J}_{m,1} - G_2 - G_3)/2^k$$

Note that we add z_Y before subtraction such that all intermediate elements in the reformulated system are guaranteed to be positive. Now we can directly encode Equation 5.2 on the finite field without the need of any sign check, and thus completely remove bit-decompositions. This optimization saves $O(mn \log p)$ constraints for $Q_W \in \mathbb{R}^{m \times n}$, where p is the order of the finite field used by zk-SNARK.

Remainder-based verification. NN computation involves abundant division computations in average pool kernels (*e.g.*, division by 4) and quantization (*e.g.*, division by 2^k). These division computations usually lead to non-integers and make it costly to verify since zk-SNARK supports only integers. A naïve approach is to use expensive bit-decomposition operations in zk-SNARK. To efficiently verify division operation in zk-SNARKs, we propose a *remainder-based verification optimization* to avoid this high overhead. We first use an extra matrix R to store the division remainder. During verification, we utilize R to avoid division in zk-SNARK systems. Formally, instead of Equation 5.2, we prove Equation 5.2:

$$Q_Y 2^k + R = M(G_1 + n z_W z_X \mathbf{J}_{m,1} + M' \mathbf{J}_{m,1} - G_2 - G_3) \tag{5.2}$$

As a result, we can verify the computation without the need of any division operations. This optimization saves $O(m \log p)$ constraints ($Y \in \mathbb{R}^m$, p is the order of the finite field used by zkSNARK).

5.5.3 Adapting R1CS friendly quantization for average pool and ReLU kernels

Now, we show how to adapt the previously introduced R1CS friendly quantization techniques to average pool and ReLu kernels as well. Here, we utilize average pool instead of max pool following popular setting in zkNN design [229] to reduce constraint sizes. We include a detailed discussion on its impact over constraint size and accuracy in subsection 5.8.4.

Average pool kernel. The average pool kernel computes the average values among a set of integers. It is useful to summarize neural network features across spatial dimensions, as illustrated in Figure 5.3(b). Formally, given a matrix in quantized representation (Q, s, z) ($Q \in \mathbb{N}^{c_{in} \times m \times n}$), and a pooling parameter r, the average pooling operator splits the data into a set of $r \times r$ grid and computes the average in each grid

$$\bar{q}_{c,i,j} = \sum_{p=0}^{r-1} \sum_{t=0}^{r-1} q_{c,ri+p,rj+t}/r^2, c \in \{1, 2, ..., c_{in}\}$$
$$i \in \{1, 2, ..., \lfloor \frac{m}{r} \rfloor\}, j \in \{1, 2, ..., \lfloor \frac{n}{r} \rfloor\}$$

Let's briefly summarize the obstacles. First, the average pooling operator contains division operation, which is not generally supported in zk-SNARK systems. Second, even if division for certain pooling parameters (*e.g.*, $r = 2$) can be conducted with bit operations, it may still lead to non-integer outputs after division.

To this end, we incorporate the aforementioned remainder-based verification strategy to the average pool kernels. In particular, we first use an extra scalar γ to store the division remainder and use the following verification for the average pooling kernel

$$\bar{q}_{c,i,j} r^2 + \gamma = \sum_{p=0}^{r-1} \sum_{t=0}^{r-1} q_{c,ri+p,rj+t}, c \in \{1, 2, ..., c_{in}\}$$
$$i \in \{1, 2, ..., \lfloor \frac{m}{r} \rfloor\}, j \in \{1, 2, ..., \lfloor \frac{n}{r} \rfloor\}$$

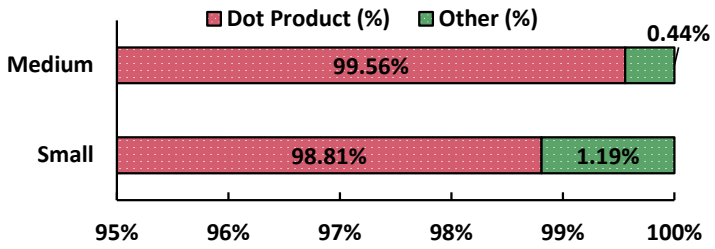


Figure 5.5: Dot product/other computation Ratio in LeNet-5-Medium and LeNet-5-Small.

ReLU kernel. The ReLU kernel contains only maximum operations, and is used to extract nonlinear neural network features, as illustrated in Figure 5.3(a). Formally, given a quantized matrix represented in a triple (Q, s, z), where Q is the quantized matrix ($Q \in \mathbb{N}^{c_{in} \times m \times n}$), s is the scale parameter ($s \in \mathbb{R}$), and z is the zero point $z \in \mathbb{N}$. We compute the ReLU kernel by element-wisely applying the maximum comparison

$$Q_{\text{ReLU}} = \max(Q, z\mathbf{J}_{c_{in},m,n}))$$

The key insight is that z is an integer value corresponding to the lifted zero in the floating-point data. Note that this design involves only integer arithmetics, and avoids the conversion between floating-point and integer completely.

5.6 Optimizing Matrix Operation Circuits using Stranded Encoding

In this section, we propose the *stranded encoding*, a general methodology of optimizing matrix operation circuits for zk-SNARKs. Our profiling on neural networks shows that matrix operation, especially dot products, consumes most computation in neural networks, as shown in Figure 5.5.

One important observation that we make is: neural network models can be effectively quantized to models consisting of small integers, such as uint8 or uint16, while the un-

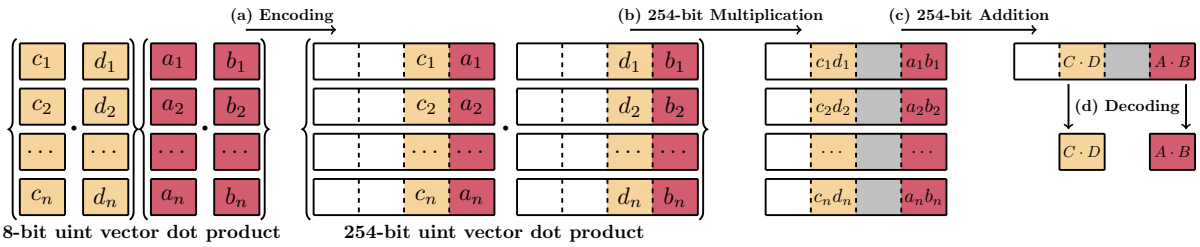


Figure 5.6: Stranded encoding with batch size $s = 2$

derlying finite field is usually much larger (e.g. $\approx 2^{254}$ in case of BLS12-381 [226]). From this observation, we propose a new encoding scheme, namely stranded encoding, that could encode multiple low precision integers into a single finite field element.

Naïve encoding. One intuitive solution is, to encode $A \cdot B$ where $A = [a_1, a_2], B = [b_1, b_2], a_i, b_i \in \mathsf{uint8}$, we can use finite field elements x and y ($x, y \in \mathbb{F}_p, p \geq 2^{16}$) to encode A and B:

$$x = a_1 + a_2\delta, \qquad y = b_1 + b_2\delta.$$

where $\delta \geq 2^{16}$. This encoding is already additive homomorphic, i.e., $A + B = [a_1 + b_1, a_2 + b_2]$ since $x + y = (a_1 + b_1) + (a_2 + b_2)\delta$, from which $a_1 + b_1$ and $a_2 + b_2$ can be easily extracted. This naïve encoding is not multiplicative homomorphic though. Take dot product computation $A \cdot B = a_1b_1 + a_2b_2$ as an example. We know that

$$xy = a_1b_1 + (a_1b_2 + a_2b_1)\delta + a_2b_2\delta^2$$

To get $A \cdot B$, we need to extract each a_ib_i separately from xy. This costs $O(n), n = |A| = |B|$, which defeats the purpose of the encoding.

Stranded encoding. To address this problem, we propose a stranded encoding of low precision integers in finite field elements. The core idea of stranded encoding is to encode multiple matrix operations at the same time. For example, to better encode $A \cdot B = a_1b_1 + a_2b_2$ and $C \cdot D = c_1d_1 + c_2d_2$, we could first encode low-precision integers

in finite fields:

$$x_1 = a_1 + c_1\delta \quad x_2 = a_2 + c_2\delta$$

$$y_1 = b_1 + d_1\delta \quad y_2 = b_2 + d_2\delta$$

with sufficiently large δ ($\delta \geq 2^{17}$). Now, $A \cdot B$ and $C \cdot D$ can be all easily extracted from $\sum x_i y_i$, since:

$$x_1 y_1 = a_1 b_1 + (a_1 d_1 + c_1 d_1)\delta + c_1 d_1 \delta^2$$

$$x_2 y_2 = a_2 b_2 + (a_2 d_2 + c_2 d_2)\delta + c_2 d_2 \delta^2$$

$$\begin{aligned} x_1 y_1 + x_2 y_2 = (a_1 b_1 + a_2 b_2) + (\ldots)\delta \\ + (c_1 d_1 + c_2 d_2)\delta^2 \end{aligned}$$

We can extract $A \cdot B$ and $C \cdot D$ from $x_1 y_1 + x_2 y_2$ by mod δ and extracting the lowest 9 bits.

It is not hard to see that this stranded encoding can be easily extended to the case that vector length $|A| = |B| = |C| = |D| = n > 2$, as illustrated in Figure 5.6:

$$A \cdot B = \sum_{i=1}^{n} a_i b_i, \quad C \cdot D = \sum_{i=1}^{n} c_i d_i$$

We encode x_i and y_i as:

$$x_i = a_i + c_i\delta, \quad y_i = b_i + d_i\delta, \quad i \in \{1, 2, \ldots, n\}$$

Here, we set $\delta = 2^k, k = 2w_{in} + \log n$, where w_{in} is the bit width of the low precision unsigned integer and n is the size of the vector. We need to add n to catch possible overflow of accumulating n w_{in}-bit unsigned integers. Now we have:

$$\begin{aligned} \sum_{i=1}^{n} x_i y_i &= \sum_{i=1}^{n} a_i b_i + (\ldots)\delta + \left(\sum_{i=1}^{n} c_i d_i\right)\delta^2 \\ &= A \cdot B + (\ldots)\delta + (C \cdot D)\delta^2 \end{aligned} \tag{5.3}$$

Finally, we can decode the dot products $A \cdot B$ and $C \cdot D$ with bit operations

$$A \cdot B = \left(\sum_{i=1}^{n} x_i y_i\right) \bmod \delta, \quad C \cdot D = \left(\sum_{i=1}^{n} x_i y_i\right) \gg 2k$$

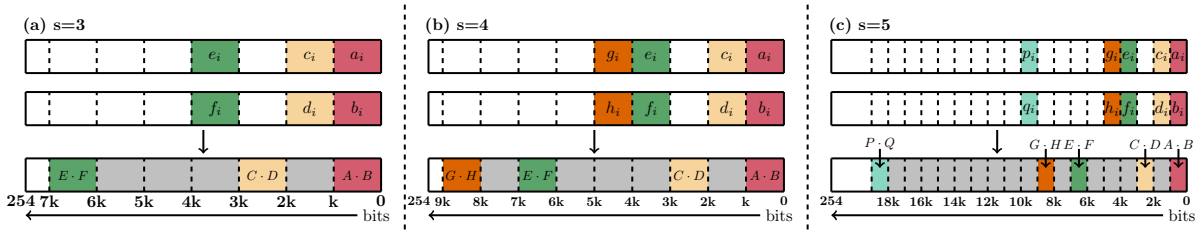


Figure 5.7: Data layout for variable batch size s. Each block has k bits and can store a value up to 2^k. k is a hyperparameter that is generally larger than 8 to accumulate the dot product and avoid overflow.

where *mod* is the module operation and $\gg 2k$ indicates right-shift by $2k$ bits. While the decoding adds more overhead, this overhead is amortized as we increase the number of batched operations in stranded encoding.

In the above example, we batch two dot product operations: $A \cdot B$, $C \cdot D$. We will discuss how to extend stranded encoding with a larger batch size next.

Stranded encoding with arbitrary batch sizes. Let batch size s be the number of batched dot product operations. To achieve further saving in the constraint size, we would like to extend s from 2 to larger batch sizes.

However, a naïve extension of the stranded encoding when batch size $s = 2$ would not work. For example, for $s = 3$, we encode $A \cdot B$, $C \cdot D$, $E \cdot F$ ($A = [a_1, \ldots, a_n]$ etc.) as follows:

$$x_i = a_i + c_i\delta + e_i\delta^2, \quad y_i = b_i + d_i\delta + f_i\delta^2.$$

Then, the multiplication becomes:

$$\begin{aligned} x_i y_i =& a_i b_i + (a_i d_i + b_i c_i)\delta + (c_i d_i + a_i f_i + b_i e_i)\delta^2 \\ &+ (c_i f_i + d_i e_i)\delta^3 + e_i f_i \delta^4 \end{aligned}$$

It becomes very difficult to extract $c_i d_i$ from $x_i y_i$ since $c_i d_i$ is "mixed" in the coefficient of δ^2. To solve this problem, we use the following encoding instead for $i \in \{1, ..., n\}$:

$$x_i = a_i + c_i\delta + e_i\delta^3, \quad y_i = b_i + d_i\delta + f_i\delta^3.$$

Now it is not hard to see:

$$x_i y_i = a_i b_i + (\ldots)\delta + (c_i d_i)\delta^2 + \ldots + (e_i f_i)\delta^6 \tag{5.4}$$

As a result:

$$\sum_{i=1}^{n} x_i y_i = A \cdot B + (...)\delta + (C \cdot D)\delta^2 + \ldots + (E \cdot F)\delta^6$$

In fact, stranded encoding can be generalized to an arbitrary batch size s (with constraints). Formally, we define stranded encoding as follows:

Definition 5.1 (Stranded encoding scheme) For a series of dot product $A_1 \cdot B_1, \ldots, A_s \cdot B_s$, where $A_j = [a_{j,1}, \ldots, a_{j,n}], B_j = [b_{j,1}, \ldots, b_{j,n}] (j \in \{1, \ldots, s\})$ and $a_{j,i}, b_{j,i} \in [2^{w_{in}}]$, stranded encoding encodes these dot product operations in finite field elements $x_i, y_i \in \mathbb{F}_p, p \geq 2^{w_{out}}, i \in \{1, \ldots, n\}$ as follows:

$$x_i = \sum_{j=1}^{s} a_{j,i}\delta^{\phi(j)}, \quad y_i = \sum_{j=1}^{s} b_{j,i}\delta^{\phi(j)}$$

where $\delta = 2^{2w_{in}+\log n}$ and $\phi(\cdot) : \{1, \ldots, s\} \to \mathbb{N}$ can be defined by the following optimization problem:

$$\begin{aligned} \min \quad & \phi(s) \\ s.t. \quad & \Omega_1 = \{\phi(1) + \phi(s),\ ...,\ \phi(s-1) + \phi(s)\} \\ & \Omega_2 = \{2\phi(1),\ 2\phi(2),\ ...,\ 2\phi(s-1), 2\phi(s)\} \\ & \Omega_1 \cap \Omega_2 = \emptyset \end{aligned} \tag{5.5}$$

In addition, n needs to satisfy the following constraint:

$$(2\phi(s) + 1)(2w_{in} + \log n) \leq w_{out} \tag{5.6}$$

As a result:

$$\sum_{i=1}^{n} x_i y_i = (A_1 \cdot B_1)\delta^{2\phi(1)} + \ldots + (A_2 \cdot B_2)\delta^{2\phi(2)} +$$

$$\ldots + (A_s \cdot B_s)\delta^{2\phi(s)}$$

The core of this definition is to formulate stranded encoding as an optimization problem in Equation 5.5. Intuitively, as shown in Equation 5.7, Ω_2 is the set of exponents of δs in the terms of $x_i y_i$ that ends up to be "useful". Ω_1 represents the set of exponents of δs in the terms of $x_i y_i$ that are going to be discarded. For example, in case of $s = 2$, $\phi(1) = 0, \phi(2) = 1, \Omega_1 = \{1\}, \Omega_2 = \{0, 2\}$. This can be verified in Equation 5.3. In case of $s = 3$, $\phi(1) = 0, \phi(2) = 1, \phi(3) = 3, \Omega_1 = \{1, 3, 4, 5\}, \Omega_2 = \{0, 2, 6\}$. This can be verified in Equation 5.4.

In addition, the constraint shown in Equation 5.6 prevents the stranded encoding scheme from blowing up the finite fields. Since $\delta = 2^{2w_{in}+\log n} > max\{A_i \cdot B_i\}$, each term in Equation 5.7 is non-overlapping in the final encoded bits (as shown in Figure 5.7). Now, we only need to worry about the last term not exceeding the size of the finite field, which is captured by Equation 5.6. We list $\phi(s)$ for different s and their n_{max} in Table 5.5.

Cost based optimization. Now, we can analyze the benefits brought by stranded encoding in terms of the number of constraints. Since the encoding part is "free": the addition would not cost extra constraints. The major cost is decoding, which requires bit decomposition (generating $O(w_{out})$ constraints). For example, in the zk-SNARK implementation we use, bit decomposition of a finite field element in BLS12-381 generates 632 constraints. Then, the amortized cost of each element-wise multiplication in dot product is:

$$cost(s, n) = \frac{O(w_{out})}{sn} \tag{5.7}$$

where s is the batch size and n is size of the vectors to be dot producted. For the cost function listed in Equation 5.7, we always choose the best batch size s for the given input. For example, in our setting, $w_{in} = 8, w_{out} = 254$, the fixed cost of bit decomposition is 632. For $n < 632/3$, we shall not do stranded encoding since the amortized cost is greater than 1; for $632/3 \leq n \leq 4096$, we choose a batch size of 4; and for $n > 4096$, we choose a

Table 5.5: Largest supported vector size n_{max} in stranded encoding for different batch size s ($\omega_{in} = 8$ and $\omega_{out} = 254$). '-' indicates not supported.

s	2	3	4	5
$\phi(s)$	1	3	4	9
$2\phi(s)+1$	3	7	9	19
n_{max}	2^{68}	2^{20}	2^{12}	-

batch size of 3. We believe this cost function is useful when determining both the integer precision during the neural network quantization and the underlying field field used by a zk-SNARK.

5.7 Evaluation

We first describe ZEN's implementation details and evaluation settings in subsection 5.7.1. Next, we demonstrate the effectiveness of our proposed optimizations on reducing constraints in subsection 5.7.2. Then, we show the benefits of reducing linear combinations in subsection 5.7.3. Last, we show the end-to-end results on the number of constraints as well as accuracy in subsection 5.7.4.

5.7.1 Implementation and Evaluation Settings

Implementation. ZEN implementation consists of three major parts: a quantization engine, circuit generators, and a scheme aggregator. The quantization engine takes a pretrained floating-point PyTorch model, applies our zk-SNARK friendly quantizations, and produces a quantized neural network model. We include a detailed discussion on quantization engine in subsection 5.8.3. Circuit generators generate individual components of circuit. We implemented circuit generators for FC, Conv, average pooling, and ReLU kernels. Our system also includes a commitment circuit generator from the underlying zk-SNARK system we used. We implement stranded encoding as part of FC

and Conv circuit generators. The scheme aggregator assembles all component circuits together, and produces the final zero-knowledge proof systems according to the specified ZEN_{infer} scheme. Our system uses arkworks' implementation [248] of Groth16 scheme [228] as the underlying zk-SNARK. We choose the BLS12-381[226] as the underlying curve for in Groth16. We use Poseidon sponge scheme [262] implemented in Arkworks library.

Datasets. We select three popular datasets (MNIST, CIFAR-10, and ORL) used by many secure machine learning projects [207, 246, 205, 206, 208]. Among these datasets, MNIST and CIFAR-10 are used for the classification task and ORL is used for the recognition task (*e.g.*, face recognition).

- MNIST is a large dataset for handwritten digit classification with $60,000$ training images and $10,000$ testing images. Images in MNIST are gray-scale of shape $28 \times 28 \times 1$.

- CIFAR-10 is a classification dataset with 10 classes (e.g., cat and dog). It contains $50,000$ training images and $10,000$ testing images of shape $32 \times 32 \times 3$.

- ORL dataset contains face images from 40 distinct subjects with diverse lighting, facial expression, and facial details. Since ORL dataset does not specify the training and testing dataset split, we randomly select 90% images as the training dataset and use the remaining 10% images as the testing dataset.

All images are stored with uint8 data type and values are between 0 and 255.

Models. We use ShallowNet, a lightweight neural network model and a series of LeNet variants [254], as summarized in Table 5.6: ShallowNet contains two fully connected kernels and one ReLU kernel. LeNet has three convolutional kernels, two fully connected kernels, and four ReLU kernels. These variants have different kernel sizes for

Table 5.6: Neural Networks used in our evaluation.

Network	Number of Layers				# FLOPs (K)
	Conv	FC	Act	Pool	
ShallowNet	0	2	1	0	102
LeNet-5-small	3	2	4	2	530
LeNet-5-medium	3	2	4	2	7,170
LeNet-Face-small	3	2	4	2	2,880
LeNet-Face-medium	3	2	4	2	32,791
LeNet-Face-large	3	2	4	2	127,466

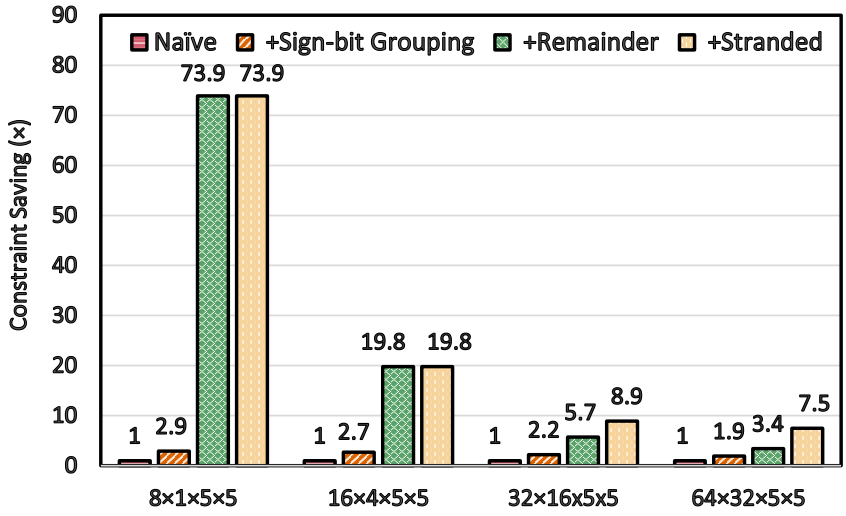


Figure 5.8: Constraint saving on conv kernels. Shape is [# of in channels]×[# of out channels]×[kernel width]×[kernel height].

adapting to different sizes of inputs. The evaluation on these six variants demonstrates the performance of ZEN under diverse model sizes.

Experiment Configuration. All the evaluations run on a Microsoft Azure M32ms instance with 32 core Intel Xeon Platinum 8280M vCPU @ 2.70GHz and 875 GiB DRAM. We compile ZEN code using Rust 1.51.0 in release mode. The Arkworks library version is 0.3.0.

5.7.2 Benefits on reducing constraints

R1CS friendly quantization. We demonstrate the optimization benefits from R1CS friendly quantization on convolution (Conv) kernels and fully connected (FC) kernels with diverse kernel sizes. We choose Conv and FC kernels since they take up to 60% share in the total number of constraints. We report the benefits from sign-bit grouping and remainder-based verification for each kernel.

Figure 5.8 shows the constraint saving on Conv kernels. We observe that the number of constraints can be reduced by 3.4× to 73.9× with R1CS friendly quantization, including the sign-bit grouping and remainder-based verification. Looking at individual kernels, we find that sign-bit grouping and remainder-based verification significantly bring benefits on diverse Conv kernels, especially on small Conv kernels of shape $8 \times 1 \times 5 \times 5$ and $16 \times 4 \times 5 \times 5$. Figure 5.9 shows similar constraint saving (from 1.1× to 8.4×) for FC kernels from R1CS-friendly quantization.

From Table 5.4 we know that AvgPool obtains 3.0× and 633× constraint reduction from the sign-bit grouping and remainder-based verification in R1CS-friendly quantization. ReLU obtains 1.2× constraint reduction from sign-bit grouping. This improvement is regardless of AvgPool and ReLU kernel size, because the constraint reduction for each division in AvgPool is fixed and independent of the number of division operations, and similarly for ReLU.

Stranded encoding. We first show the effectiveness of stranded encoding with the optimal batch s determined by the optimizer. Since stranded encoding can only be applied to FC kernel and Conv. kernel, we show constraint savings cause by stranded encoding in Figure 5.8 and Figure 5.9. When applied to Conv. kernel, stranded encoding didn't get triggered when the kernel sizes are small ($8 \times 1 \times 5 \times 5$ and $16 \times 4 \times 5 \times 5$). This is because the optimizer decides that the gain of stranded encoding will be negative. For

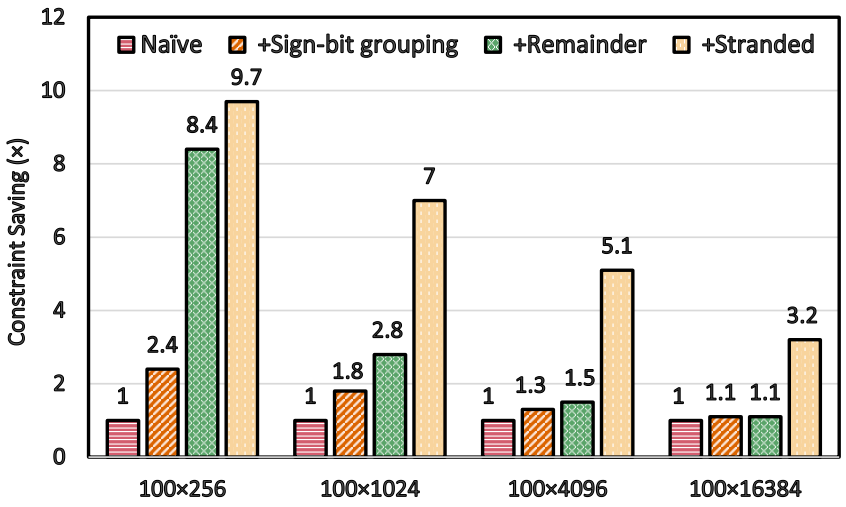


Figure 5.9: Constraint saving FC kernels. Shape is [# of in channels]×[# of out channels].

larger size kernels, stranded encoding brings 1.6× to 2.2× constraint size savings on top of R1CS friendly quantization. We observe a similar trend when applying stranded encoding to FC kernel in Figure 5.9: it brings additional 1.2× to 2.9× constraint reduction for FC kernels of different sizes. The larger FC kernel is, the more benefit from stranded encoding it gets. This is because the amortized cost of stranded encoding decreases proportionally as the kernel size increases.

Selecting the optimal batch size s in stranded encoding. We further demonstrate the benefits of stranded encoding under different batch sizes s (Figure 5.10). Comparing across different batch sizes s, we notice that a larger batch size usually leads to higher savings. This result shows that we should choose a larger batch size s when the vector satisfies the corresponding length requirement. We also observe that the constraint saving increases as the FC size grows. In particular, when the kernel shape is 100×16384, $s = 3$ can lead to a 2.8× constraint saving, which almost reaches the theoretical upper bound on constraint saving. Meanwhile, stranded encoding brings little benefits on small FC kernels (*e.g.*, 100×256). The reason is that the encoding and decoding overheads are constant across FC sizes and become relatively small on large FC sizes. This observation

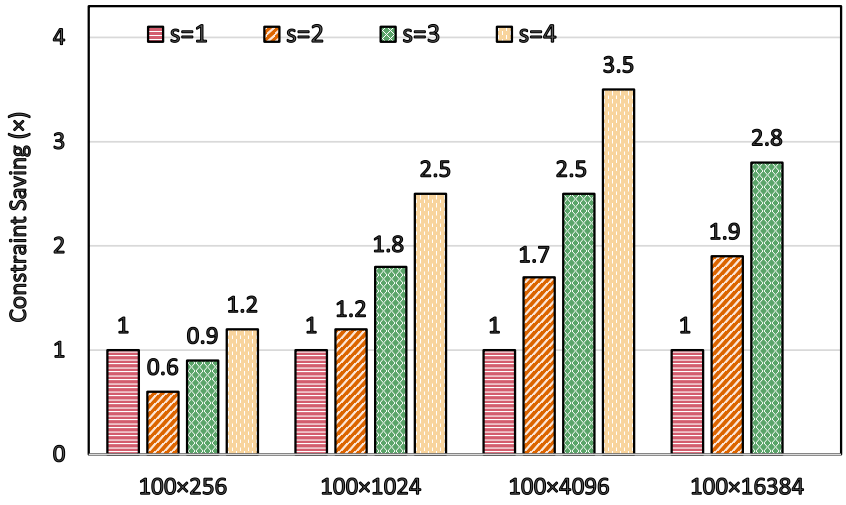


Figure 5.10: Constraint saving on FC kernels with different batching size s. Shape is [# of in channels] × [# of out channels].

Table 5.7: Accuracy comparison between floating-point (FP) models and R1CS friendly quantized models.

Model	FP Acc. (%)	Quant. Acc. (%)	ΔAcc. (%)
ShallowNet-MNIST	95.13	94.91	−0.22
LeNet-small-CIFAR	55.76	55.35	−0.41
LeNet-medium-CIFAR	64.23	63.68	−0.55
LeNet-Face-small-ORL	84.3	84.0	−0.3
LeNet-Face-medium-ORL	88.6	88.2	−0.4
LeNet-Face-large-ORL	91.6	92.1	0.5

motivates the usage of $s = 1$ for these small FC kernels. Our optimizer maximizes the benefits from stranded encoding by automatically selecting the best batch size s.

Case study: end-to-end results on LeNet-5-Medium. We show end-to-end results of different levels of optimizations on inference using LeNet-5-Medium on CIFAR-10 in Table 5.4. Starting from ZEN-vanilla, we add individual optimizations one by one and show the total number of constraints from all NN layers. In ZEN-vanilla, LeNet-5-Medium has almost 85 million constraints. This large number of constraints comes from

[2]We skip $s = 4$ for the FC kernel of shape 100×16384 since our stranded encoding with $s = 4$ requires a vector of length less than $2^{12} = 4096$, as described in Table 5.5

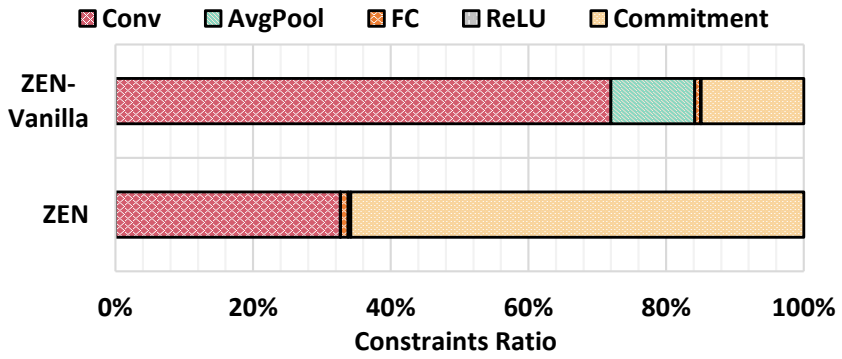


Figure 5.11: Breakdown of constraints in LeNet-Face-large ORL dataset from both ZEN-vanilla and ZEN.

the intensive computation and bit decomposition in NNs. When all three optimizations are applied, ZEN significantly reduces the total number of constraints by 15.45×. This is similar to the constraint saving on popular NN kernels in Figure 5.8 and Figure 5.9. These results show that our optimization techniques can significantly mitigate the intensive number of constraints on NN workloads and enable a more efficient deployment of ZEN.

Constraint size breakdown by kernel type. We breakdown the number of constraints in LeNet-Face-Large on CIFAR-10 circuits generated by ZEN-vanilla and fully optimized ZEN in Figure 5.11. We split the constraints into those from the commitment scheme and those from 4 different kinds of kernels. Overall, we observe that convolution kernels and fully connected kernels are the dominant sources of constraints in the ZEN-vanilla (59.2%) implementation. Since these two kinds of kernels heavily rely on dot products, this justifies our efforts on improving dot product circuit size by using stranded encoding. It is worth noting that commitment accounts for only 30.1% constraints in the ZEN-vanilla, but this ratio significantly rises to 82.6% in ZEN. Note that the absolute number of constraints from commitment remains the same in both ZEN-vanilla and ZEN. This ratio change comes from optimizations in ZEN that significantly reduces the number of constraints from neural network inference part. Further improving the commitment size to make ZEN even more efficient is our future work.

5.7.3 Benefits on reducing linear combinations

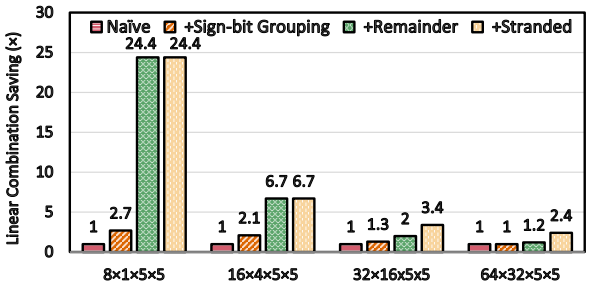


Figure 5.12: LC saving on conv kernels. Shape is [# of in channels]×[# of out channels]×[kernel width]×[kernel height].

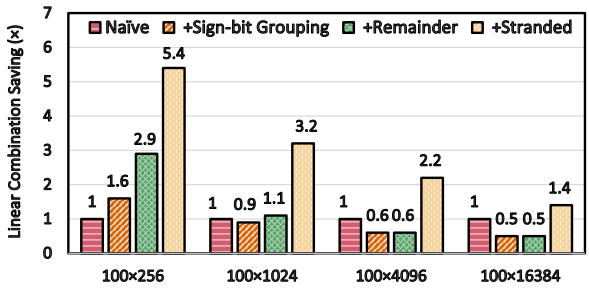


Figure 5.13: LC saving on FC kernels. Shape is [# of in channels]×[# of out channels].

Despite the traditional wisdom that the prover time is mostly decided by the number of constraints, we find this is not the case for ZEN schemes on arkworks implementation of Groth16 protocol [248], especially on the bigger neural network models (which translates to a larger number of constraints). In Table 5.8, we breakdown the run time of ZEN$_{infer}$'s proving time into four parts: synthesizing constraints from circuits (Const. syn.), inlining linear combinations (Inline LCs), transforming R1CS to QAP (R1CS to QAP), and computation in groups (Comp. in Gs). Surprisingly, as the number of constraints goes up, the time spent on inlining linear combinations goes up and dominates the run time.

Fortunately, our optimizations reduce the number of linear combinations (LCs) as well. We demonstrate the effectiveness of our optimizations in terms of reducing the number of LCs in Figure 5.12 and Figure 5.13. We can observe a result that is similar to our savings of constraint sizes shown in subsection 5.7.2. Overall, our optimizations save the number of LCs from 1.4× to 24.4× in ZEN$_{infer}$. We also observe that stranded encoding consistently saves the number of LCs across various kernel sizes. Meanwhile, sign-bit grouping and remainder-based verification may slightly increase the number of LCs on certain NN kernels (*e.g.*, 100×4096 and 100×16384 in Figure 5.13).

Table 5.8: Prover time breakdown of ZEN_{infer} using Arkworks implementation of Groth16

Model-Dataset	Const. syn.	Inline LCs	R1CS to QAP	Comp. in Gs
ShallowNet-MNIST(4M)	21.6%	11.6%	17.3%	49.5%
LeNet-small-CIFAR(4M)	27.7%	16.2%	12.1%	44.0%
LeNet-medium-CIFAR(23M)	21.0%	28.9%	13.2%	36.9%
LeNet-small-ORL(20M)	23.1%	20.8%	14.9%	41.2%
LeNet-medium-ORL(83M)	17.9%	39.4%	12.5%	30.2%
LeNet-large-ORL(318M)	15.1%	67.4%	8.8%	15.7%

5.7.4 End-to-end performance

In this section, we evaluate the end-to-end performance of ZEN in terms of accuracy, savings on total number of constraints, savings on total number of LCs, setup/proving time, and CRS sizes.

ZEN is accurate. In Table 5.7, we list the accuracy of our quantized model compared with the original floating-point PyTorch models. One can see the accuracy drop is minimal and some of them are within the error bound of different implementation of the same machine learning models. Additionally, compared with the state of the art full quantization scheme [30], ZEN has exactly the same accuracy thanks to our semantic preserving optimizations proposed in section 5.5. As a result, we can conclude that ZEN maintains the accuracy of neural network models in practice.

Overall saving on the number of constraints. Table 5.3 shows the overall saving on the number of constraints with our optimizations. Overall, we can significantly reduce the number of constraints by 15.35× on average. This shows a similar saving on the number of constraints as our case study in the micro-benchmarks. On large models such as LeNet-Face-Large, we achieve a saving of 11.06×. On small models of LeNet-5-small and LeNet-Face-small, we achieve more than 22.19× saving. This is mainly due to the fact that kernels in these small models are dot products of vectors which can be drastically improved with our sign-bit grouping and remainder-based verification. This

result is similar to our microbenchmark in Figure 5.8 and Figure 5.9.

Overall saving on the number of LCs. Table 5.9 shows that the number of LCs in ZEN_{infer} varies significantly from 509K to 766,147K. Our optimizations save the number of LCs from 2.34× to 8.03× (5.34× on average).

Table 5.9: Overall saving on the # of linear combinations in ZEN_{infer}.

Model-Dataset	Commitment LCs(K)	ZEN vanilla LCs(K)	ZEN_{infer} LCs(K)	Saving (×)
ShallowNet-MNIST	7,173	1,193	509	2.34
LeNet-small-CIFAR	5,276	51,231	6,375	8.03
LeNet-medium-CIFAR	31,618	263,779	56,649	4.66
LeNet-small-ORL	31,032	175,066	22,982	7.62
LeNet-medium-ORL	115,000	856,834	209,775	4.08
LeNet-large-ORL	459,155	1,944,147	766,125	2.54

Setup time, prover time and CRS sizes. Table 5.1 shows the overall performance of ZEN_{infer} on 6 neural networks with diverse number of constraints, ranging from 1,751 thousands to 162,289 thousands. We observe that a model with a higher number of constraints comes with higher prover and setup time consumption and a larger Common Reference String (CRS) size. For a small model ShallowNet on MNIST with 1,751 K constraints, we have a short time of 37 seconds and 32 seconds for setup and proving, respectively; the CRS consists of 0.5 GB of data. The overall cost increases as the number of constraints increases. For large models such as LeNet-Face-large on ORL with over 162,289 thousand constraints, ZEN_{infer} requires nearly 14400 seconds for setup and proving respectively. Its CRS size is around 50 GB.

Due to the large memory and time consumption by ZEN_{acc}, we can parallelize the testing dataset inference step across multiple machines and add a prediction accuracy commitment sum check circuit. Table 5.2 shows the overall performance of ZEN_{acc} on 6 neural networks with a testing dataset size of 100. The constraints of ZEN_{acc} here consist of the commitment to the neural network model, the inferences on 100 images, and a

commit to the final accuracy. The number of constraints of ZEN_{acc} ranges between $8,305$ K to $5,628,278$ K on 6 NNs.

Overall speedup in practice. ZEN shows significant speedup in practice thanks to our saving in the R1CS constraint in inference part. We reduce inference part prove time from 252s to 23s (10.96×) on LeNet-small-cifar, and from 1267s to 311s (4.07×) on LeNet-medium-cifar.

5.8 Discussion

5.8.1 zkNN Scalability

ZEN is the first work that compiles NN to optimized R1CS constraints with 5.43× to 22.19× constraint size reduction. ZEN shows it merit in developing the technique and illustrating the importance of "front-end" optimization. Since ZEN outputs R1CS, an intermediate representation used by many different zkSNARKs, the innovations on the backend proof systems can be combined with ZEN to bring even better scalability, such as leveraging different level of parallelism in zkSNARK designs (e.g., DIZK [263] and pipeZK [264]). While ZEN did not solve all scalability challenges, we are optimistic about future ZK NN scalability.

5.8.2 zkNN Accuracy

The concrete model accuracy is orthogonal to ZEN's main contributions. Many new ML techniques could be applied to ZEN to get better accuracies, such as neural architecture search [265], neural network distillation [266], and pruning [267, 268, 269]. For example, [270] can remove 84.82% computation while only reducing 0.57% top-1 accuracy on VGG-16. We consider such accuracy improvement to be currently out-of-scope and

leave it as future work.

5.8.3 Quantization Engine: Selection of quantization parameters

Our R1CS friendly quantization shares the same quantization parameters as existing quantization algorithms [30] and implementations [222]. The key novelty of R1CS friendly quantization is to use algebraic equalities to reduce the number of expensive bit-decomposition operations in zkSNARK while maintaining the semantics of the quantization. In particular, we follow [30] and PyTorch quantization implementation [222] that first generate a floating-point NN and then compute a scaling parameter s and a lifted zero point z for each layer. For example, when quantizing to uint8, we profile the range of a NN layer as (a, b) and compute $s = (b - a)/256$. We also fix the zero point z to 128 such that the quantized value Q is always non-negative (ranging from 0 to 255).

5.8.4 Comparing Average Pooling and Max Pooling

While ZEN currently focuses on average pooling, it could be easily extended to max/min-pooling at the cost of increased constraint size. For example, in LeNet-small-cifar, the first max pool layer introduces 4704 comparisons. Based on our profiling based on Arkworks' default comparison operator, this translates to 8.9 million constraints. In contrast, average pooling only needs 1176 constraints.

We also empirically observe that NNs with max pooling and average pooling shows similar accuracy. In particular, we observe only 0.46% accuracy improvement by replacing average pooling layer with max pooling layer on LeNet-small-CIFAR and LeNet-medium-CIFAR.

To this end, we stick to average pooling for saving constraints and improving scalability.

Chapter 6

A Type-based Optimization Framework for Zero Knowledge Neural Network Inference

In this chapter, we present ZENO, a type-based optimization framework for zero-knolwedge neural network inference. Zero knowledge Neural Networks draw increasing attention for guaranteeing computation integrity and privacy of neural networks (NNs) based on **z**ero-**k**nowledge **S**uccinct **N**on-interactive **AR**gument of **K**nowledge (zkSNARK) security scheme. However, the performance of zkSNARK NNs is far from optimal due to the million-scale arithmetic circuit computation with heavy scalar-level dependency. In this chapter, we propose a type-based optimizing framework for efficient zero-knowledge NN inference, namely ZENO (**ZE**ro knowledge **N**eural network **O**ptimizer). We first introduce ZENO language construct to maintain high-level semantics and the type information (*e.g.*, privacy and tensor) for allowing more aggressive optimizations. We then propose privacy-type driven and tensor-type driven optimizations to further optimize the generated zkSNARK circuit. Finally, we design a set of NN-centric system optimizations

to further accelerate zkSNARK NNs. Experimental results show that ZENO achieves up to 8.5× end-to-end speedup than state-of-the-art zkSNARK NNs. We reduce proof time for ResNet50 from 1.5 hours to 11 minutes, which makes constructing practical zkSNARK NNs possible.

6.1 Problem Statement

Zero Knowledge Neural Networks [35, 229, 232, 271, 236] draw increasing attention in solving the privacy issues of neural networks. Leveraging zero-knowledge Succinct Non-interactive ARgument of Knowledge (zkSNARK) security scheme [196, 197, 198, 199, 200, 201, 202, 203, 204], zkSNARK NNs guarantee two important security properties. First, zkSNARK NNs can validate the accuracy of a private NN without releasing the NN weights. This is important for verifying the accuracy of many commercialized NNs, considering that these NN weights are usually treated as important intellectual properties and may contain sensitive information from private user data. Second, zkSNARK NNs can verify that the same private NN is consistently used to serve user requests. In particular, a company may use smaller NNs with lower accuracy and lower energy consumption to save costs, while claiming that a larger NN is provided. zkSNARK NNs enable users to eliminate such integrity concerns by verifying that the same private NN is used across user requests.

zkSNARK [196, 197, 198, 199, 200, 201, 202, 203, 204, 228] is a security scheme where, given an arithmetic function $F(x)$ and a target output y, the prover shows that the prover knows a specific value x such that $F(x) = y$ while not revealing such value x. In zkSNARK NNs [35, 229, 232, 271, 236], arithmetic function is a plaintext neural network with multiplication and addition. When protecting privacy of NN weights, x is the NN weights and y is the NN predictions on a public dataset. Early zero-knowledge

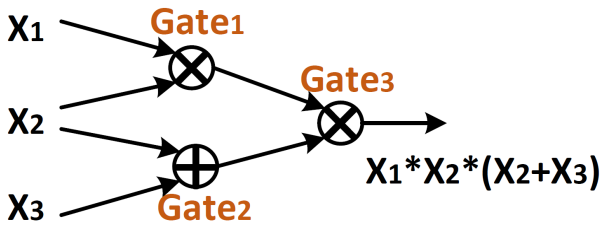


Figure 6.1: Arithmetic circuit in zero-knowledge proof for an arithmetic function $X_1 * X_2 * (X_2 + X_3)$. Here, all values are stored in finite-fields (*e.g.*, 254-bit integer [272]) for privacy.

proofs [273] usually generate a large proof whose size is proportional to the number of computation in arithmetic function. This leads to large communication overhead during proof generation and limits the size of arithmetic function. Recent zero-knowledge proofs [248] generate a fixed-size proof (*e.g.*, 192 bytes [228]). The key behind such succinct property is to decompose an arbitrary large and complex arithmetic function into a sequence of small and uniform-format constraints. Some magic security schemes [228] can then be applied to these constraints to generate a fixed-size proof.

Existing works on zkSNARK NNs usually treat a plaintext neural network as the arithmetic function. This arithmetic function is first decomposed into scalar addition and multiplication operations, where each operation is mapped to a gate (addition or multiplication) in *arithmetic circuit*. We show an example of *arithmetic circuit* for a simple arithmetic function in Figure 6.1. Then, this arithmetic circuit is condensed into the uniform-format constraints. Finally, the constraints are further condensed into a fixed-size proof. The last two steps are *circuit computation* and *security computation*, which are major bottlenecks in zkSNARK NNs. While zkSNARK NNs provide privacy properties, existing works usually cannot scale to large neural networks. For example, based on a popular zkSNARK framework Arkworks [248], it takes hundreds of seconds to prove zkSNARK LeNet on a single face image while non-zkSNARK LeNet usually requires less than 100 ms on the same hardware. We summarize three key challenges

that hinder deeper system optimizations for zkSNARK NNs.

Failing to maintain high-level semantics during proof generation. Existing zkSNARK systems [248, 204, 35, 274, 213] map an arbitrary arithmetic function into a low-level arithmetic circuit. During this procedure, NN semantics such as privacy and tensor are not preserved and hard to recover. For example, reconstructing tensor semantic by scanning and parsing the assembly-style circuit would introduce heavy runtime overhead. Therefore, zkSNARK systems can only consider individual gates in circuits and fail to exploit high-level NN-specialized optimization opportunities.

Lack of semantic-aware optimizations during compiling zkSNARK NNs. Most zkSNARK optimizations [275, 276, 248]) focus on individual scalar gates and support only local circuit optimization at small scale. These scalar gates usually show heavy dependency and prevent parallel computation. For example, in Figure 6.1, parent gates (*e.g.*, $Gate_3$) cannot be computed until all children gates (*e.g.*, $Gate_1$ and $Gate_2$) have been computed. However, most NN computations are conducted at the tensor level (*e.g.*, convolution layers and fully connected layers) and provide abundant parallelization opportunities. Moreover, NN computation usually requires floating-point values (*e.g.*, single-precision or half-precision) or small integers [47, 45, 46, 127, 214], such as `int8` or even `int1`, while zkSNARK operates on finite field (*e.g.*, $\approx 2^{254}$ in case of BLS12-381 [277]) to provide security guarantees. Naively representing these small values from NNs with finite field elements may lead to extra memory and computation overhead.

Lack of NN-centric system optimizations. Neural networks usually contain abundant computation reuse opportunities. For example, a zkSNARK NN shares the same circuit when proving on different images. Existing works usually focus on proving individual images and repeatedly generate redundant constraints. Moreover, fusing NN layers can usually save the number of addition and multiplication computation. This can potentially save the number of constraints in zkSNARK NNs. However, kernel fusion

from existing plaintext NN systems usually cannot directly bring benefits to zkSNARK NNs. For example, ReLU layer is usually fused with convolution layer in plaintext NNs but cannot be fused in zkSNARK NNs.

6.2 Overview of Proposed Solution

In this chapter, we propose a type-based optimizing framework for efficient zero knowledge neural network inference, namely **ZENO** (**ZE**ro knowledge **N**eural network **O**ptimizer). We show the overview of ZENO in Figure 6.2. ***First***, we introduce a *ZENO language construct* to maintain high-level semantics (*e.g.*, privacy and tensor) during zkSNARK proof generation. Our key insight is that, instead of parsing an assembly-style arithmetic circuit, we maintain the privacy type and structured tensor computation to guide efficient zkSNARK proof generation. We further propose a set of compute primitives to effectively express zkSNARK NNs.

Second, we design an *optimized circuit generation* that reduces both computation complexity and the number of computation by exploiting high-level semantics. Our optimized circuit generation includes a a privacy-type driven optimization and a tensor-type driven optimization. The privacy-type driven optimization reduces the number of constraints while maintaining zkSNARK NN semantics. We propose a knit encoding to efficiently represent multiple `uint8` NN computation with a single finite field (*e.g.*, 254 bits [277]) to reduce number of zkSNARK computation. The tensor-type driven optimization exploits tensor computation semantics in zkSNARK NNs to generate a ZENO circuit with minimized dependency. We use ZENO circuit as an in-place replacement for arithmetic circuit to reduce dependency.

Third, we propose *NN-centric system optimizations* to further accelerate zkSNARK NNs. We first propose *NN-inspired computation reuse* to identify the computation reuse

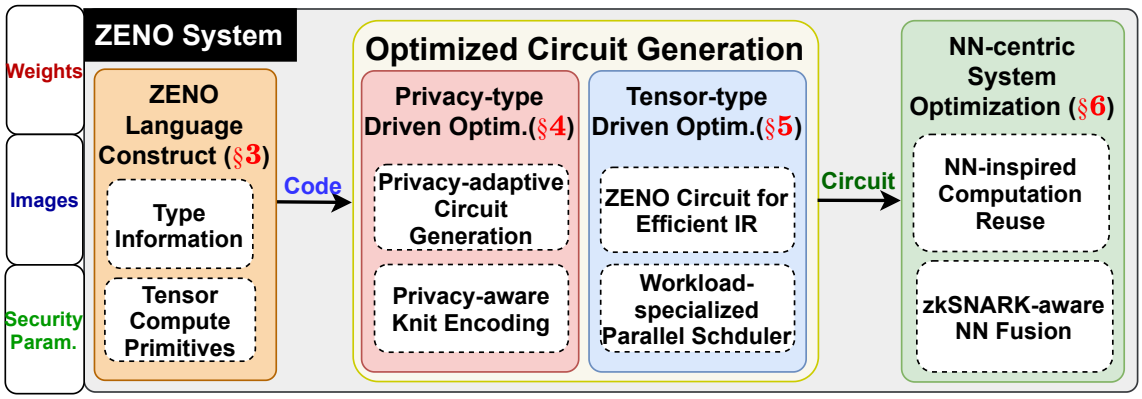


Figure 6.2: Overview of ZENO.

opportunities within images and cross images by exploiting NN semantics. Then, we propose a *zkSNARK-aware NN fusion* to fuse NN layers while considering both NN and zkSNARK properties. Our zkSNARK-aware NN fusion can save the number of constraints for reducing zkSNARK NN latency.

We extensively evaluate ZENO using six zkSNARK NNs on multiple datasets. We achieve 8.5× end-to-end speedup over state-of-the-art systems.

6.3 Related Work and Motivation

In this section, we will first give an in-depth discussion on background and related work of zkSNARK Neural Networks (NNs). Then, we will demonstrate the unique optimization opportunities for zkSNARK NNs.

6.3.1 zkSNARK

Zero-Knowledge Succinct Non-interactive Argument of Knowledge (zkSNARK) [196, 197, 198, 199, 200, 201, 202, 203, 204, 228] is a security scheme where, given a function $F(x)$ and a target output y, the prover shows that the prover knows a specific value

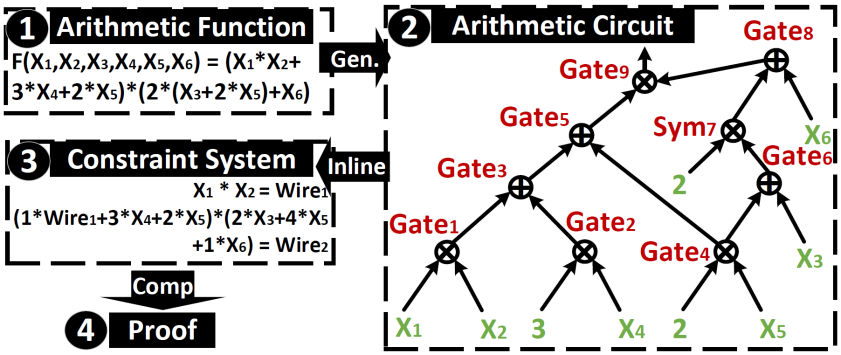


Figure 6.3: Workflow of generating zero-knowledge proof. All values are stored in ciphertext for privacy.

x such that $F(x) = y$ while not revealing such value x. Here, the function $F(\cdot)$ can describe an arbitrary arithmetic computation. One specific example is that, given a function $F(x_1, x_2, x_3, x_4\ , x_5, x_6) = (x_1x_2+3x_3)(2(x_4+2x_5)+x_6)$, the prover can generate a proof that the prover knows a set of secrete values $(x_1, x_2, x_3, x_4, x_5, x_6)$ such that $F(x_1, x_2, x_3, x_4\ , x_5, x_6) = y$ with a public value y while not revealing the exact values of $(x_1, x_2, x_3, x_4, x_5, x_6)$. As detailed in subsection 6.3.2, a neural network with millions of computations can also be interpreted as an arithmetic computation.

We illustrate zero-knowledge proof generation in Figure 6.3. Given an arithmetic function with an arbitrary number of inputs and computation, the proof generation encodes the computation in ciphertext and condense the computation into a fixed-size proof (*e.g.*, 192 bytes [228]) for efficient and public verification. There are three steps in proof generation. The first step is **Generate**, which takes a given *arithmetic function F(x)* ❶ and generates an arithmetic circuit ❷. In this step, each scalar addition and multiplication in arithmetic function is mapped to a addition gate (*e.g.*, $Gate_3$) and a multiplication gate (*e.g.*, $Gate_1$) in the arithmetic circuit, respectively. For a large arithmetic function with millions of computation (*e.g.*, zkSNARK NNs [35, 229, 232, 271, 236]), the arithmetic circuit contains millions of gates. The latency of generating zero-knowledge proof is proportional to this number of gates in the arithmetic circuit.

These million-level gates can easily lead to hour-level latency.

The second step is **Circuit Computation** that condenses the *arithmetic circuit* ❷ into a constraint system ❸. The constraint system contains a set of *constraints*, which is a specialized mathematical format:

$$(\sum_{i=1}^{n} a_{j,i} X_i) * (\sum_{i=1}^{n} b_{j,i} X_i) = Wire_j, \quad j \in \{1, 2, ..., m\} \tag{6.1}$$

Here, X_i are private input values (*e.g.*, private NN weights) and $Wire_j$ are private output values which can be used in following constraints. n is the number of private values including both private input values X_i and private output $Wire_j$. m is the number of multiplication between private values (*e.g.*, X_1 and X_2) or linear combination (LC) of private values (*e.g.*, $1 * Wire_1 + 3 * X_4 + 2 * X_5$). The zkSNARK proof generation latency is proportional to the number of private values n and the number of constraints m. For a realistic arithmetic function (*e.g.*, a neural network), both m and n could be million-level. We note several properties in the constraints. First, privacy plays an important role where multiplying a public value and a private value (*e.g.*, $3 * X_4$) does not lead to constraints. Second, the addition is "free" in zkSNARK in terms of not introducing constraints, since a large number of additions can be expressed in a single linear combination (*e.g.*, adding $1 * Wire_1$, $3 * X_4$, and $2 * X_5$) by incorporating into the linear combination of private values. Third, in the circuit computation, children gates (*e.g.*, $Gate_1$ to $Gate_4$) need to be computed before parent gates (*e.g.*, $Gate_5$). This leads to heavy dependency in circuits and is major bottleneck in zkSNARK NNs (see Figure 6.4).

The third step is **Security Computation**. Given a constraint system with a large number of n private values and m constraints, *security computation* generates a small fixed-size proof for independent and efficient verification. The latency of this step depends on the number of n private values and m constraints. On the m-dimension, this step

compresses m constraints into a single constraint

$$(\sum_{i=1}^{n} A_i(s)X_i) * (\sum_{i=1}^{n} B_i(s)X_i) = \sum_{i=1}^{n} C_i(s)X_i \tag{6.2}$$

Here, $Wire_i$ is treated as a special case of X_i for notation simplicity. Intuitively, $a_{j,i}$ for all $j \in \{1, 2, ..., m\}$ can be encoded with a single polynomial function $A_i(s)$ with degree $m-1$ by requiring $A_i(j) = a_{j,i}$. Similar property holds for $B_i(s)$ and $C_i(s)$. On the n-dimension, this step sums the cipertext (*e.g.*, $A_i(s)X_i$ and $B_i(s)X_i$) and random numbers δ to generate a small fixed-size proof. This fixed-size proof can be publicly distributed and efficiently verified by a verifier in a few milliseconds [228].

6.3.2 zkSNARK Neural Networks

Neural network (NN) [255, 256, 2, 254] is a function $F(X) = Y$ that maps an input image $X \in uint8^{H \times W \times 3}$ to a prediction $Y \in \mathbb{R}^n$, where n is the number of labels that the NN needs to distinguish (*e.g.*, $n = 2$ when only distinguish cat and dog). NN is usually defined as the composition of a sequence of NN layers $F(X) = F_1 \circ F_2 \cdots \circ F_n(X)$. Popular layers include convolution, fully connected, pooling, and ReLU, where each layer computes at tensor-level. For example, the convolution layer and fully connected layer take two inputs: the activation $X^{(k)}$ (the output of the preceding k^{th} layer) and the weight W for the current layer. Then, these two layers compute the output activation $X^{(k+1)} = W \cdot X^{(k)} + b$.

zkSNARK NNs [35, 229, 232, 271, 236] draw increasing attention in recent years to improve privacy and integrity of neural networks. These zkSNARK NNs treat a NN as a function $F(X)$ and generate proof following the workflow in Figure 6.3. To facilitate the development of zkSNARK programs, several frameworks have been proposed such as Arkworks [204, 248], Bellman [274] and Ginger [213]. However, existing zkSNARK

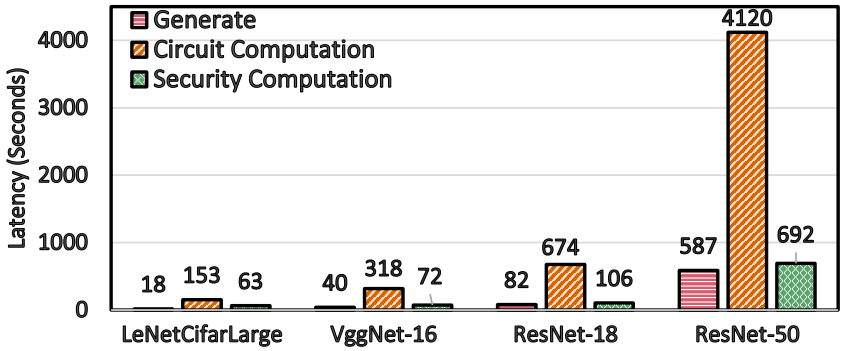


Figure 6.4: Proof latency: *private* images and *public* weights.

frameworks usually focus on scalar computation and ignore optimization opportunities from tensor-level computations which lead to prohibitive latency. In this chapter, we propose ZENO to exploit tensor-level computation and privacy type for efficient zkSNARK NN inference.

Why Non-interactive zkSNARK NN? There are two types of zero-knowledge NNs – interactive ones and non-interactive ones. The key difference is whether interactive communication is required during proof generation. It is well-accepted in cryptography that non-interactive zero-knowledge proof has more widely use cases than interactive ones [228] since non-interactive zero-knowledge proof eliminates communication overhead. For example, Mystique [278] accelerates interactive zero-knowledge NNs but still require GB-level communication for proving a single image. Instead, non-interactive NNs [35, 232] do not require such interactive communication during proof generation and generate a fixed size proof (*e.g.*, 192 bytes with [228]) for efficient and public verification. To this end, ZENO focuses on accelerating non-interactive NNs.

6.3.3 Opportunities and Challenges

In this section, we introduce optimization opportunities and challenges in enabling efficient zkSNARK NNs.

Table 6.1: Tensor Compute Primitives.

Instruction	**Semantics**
addConst(c_1,c_2) mulConst(c_1,c_2)	Add or multiply two Consts. The output is a Const
addVariableConst(v, c) mulVariableConst(v, c) compare(x, y)	Add or multiply variable v with Const c. The output is a LC. Adding or multiplying Gate, Wire, and LC with Const are also supported (skipped for notation simplicity). Compare x and y and return the larger value. x and y can be Const, Variable, Gate, Wire, or LC.
addVariableVariable(v_1, v_2)	Add variable v_1 with variable v_2 and generate a LC. Adding Gate, Wire, and LC with another Gate, Wire, and LC are also supported (skipped for notation simplicity).
mulVariableVariable(v_1, v_2)	Multiply variable v_1 with variable v_2 and generate a wire. Multiplying Gate, Wire, and LC with another Gate, wire, and LC are also supported (skipped for notation simplicity).
dotProduct(T_1, T_2)	Dot product a 1-dimensional zkTensor T_1 with another 1-dimensional zkTensor T_2. The output is a zkTensor.
fullyConnected(T_1, T_2) convolution(T_1, T_2)	Computing fully-connected and convolution layers on two input zkTensors T_1 and T_2. The output is a zkTensor.
pool(T, p)	Pooling operation on a zkTensor T and a plain scalar p. The output is a zkTensor.
ReLU(T)	ReLU operation on a zkTensor T. The output is a zkTensor.
addTensor(T_1, T_2) mulTensor(T_1, T_2)	Elementwisely add or multiply a zkTensor T_1 with another zkTensor T_2. The output is a zkTensor.

We show the latency of individual proof generation steps in Figure 6.4 for private images and public NN weights. We have similar observations on other privacy settings (*e.g.*, private weights and private images, or private weights and public images). We profile this latency based on state-of-the-art zkSNARK framework, Arkworks [248], on a single image. Note that these three steps need to be executed sequentially and the total time is the sum of individual steps. We have three major observations. First, the total time of zkSNARK NN can easily exceed 5000 seconds, while the corresponding non-zkSNARK NNs usually takes less than 1 second to compute. Second, the latency of *circuit computation* increases significantly as NN sizes increase. Third, latency of *security computation* also increases on large z as NN sizes increase.

Opportunities. There are two major opportunities to accelerate zkSNARK NNs. The first opportunity is to exploit privacy types (*e.g.*, private weights or public weights, as

Table 6.2: ZENO Type Information.

	Type	Description
Standard	**Const**	Public constant value in λ-bit finite field.
	Variable	Private scalar value in circuit for input.
	Gate	Private scalar value in circuit for intermediate results.
	Wire	Private scalar value in constraint system.
	LC	Linear Combination of wires in constraint system.
ZENO	**Privacy**	'private' or 'public'
	Tensor	A tensor of finite field data.
	zkTensor	Tuple (T, P) where "T" is a Tensor and "P" specifies privacy.

discussed in section 6.4) of zkSNARK NNs. Our investigation shows significant impact from privacy types, which motivates privacy-driven optimizations. The second opportunity is to exploit tensor computation primitives in NNs for optimizing circuits and exploiting parallelism. This opportunity has not been explored in existing zkSNARK frameworks which focus on scalar operations.

Challenges. Although these ideas sound promising, the efforts to capitalize on their benefits are non-trivial due to several challenges. First, while tensor operations may provide optimization opportunities, it is highly non-trivial to identify and reconstruct such high-level semantics from assembly-style arithmetic circuits. We need a language construct to maintain these high-level semantics and facilitate optimizations for zkSNARK NNs. Second, the zkSNARK computation procedure usually shows complex dependency across Gates and synergy between privacy types. For example, arithmetic circuits (as discussed in Figure 6.3) are inherently sequential since earlier computation results (*e.g.*, $Gate_1$) may be used by later computation (*e.g.*, $Gate_3$). We need specialized optimizations based on type information in zkSNARK NNs to reduce the computation workload

and mitigate the dependency.

6.4 ZENO Language Construct

In this section, we introduce ZENO language construct to facilitate the zkSNARK NN development and maintain the semantic information during zkSNARK computation.

Type Information with Tensor and Privacy. The goal of ZENO type information is to express the two important information in zkSNARK NN – *tensor* and *privacy*. We summarize ZENO type information in Table 6.2. There are two complexities in zkSNARK systems. First, previous zkSNARK systems contain only scalar-level data types, which makes it complicated to implement zkSNARK NNs with intensive tensor computations. Second, individual scalar data types have different privacy properties. This makes it challenging to manually set the privacy type for scalar values in zkSNARK NNs.

To tackle these challenges, we introduce tensor-level data types to directly express zkSNARK NN tensor computation and hide the complexity of privacy selection. *zkTensor* is the basic data unit in zkSNARK NNs, which can represent weight tensors and feature tensors in NNs. When "P" is public, "T" is a tensor of Const scalars for public constant values. When "P" is private, "T" is a tensor of *Variable*, *Gate*, *Wire*, and *LC*, where the specific type can be inferred automatically. Our type information abstracts details of zkSNARK implementations and enables users to focus on complex NN structures.

Tensor Compute Primitives. We propose a set of tensor-level compute primitives respecting the privacy and tensor types, as shown in Table 6.1. The goal of tensor compute primitives is to maintain the high-level semantics of zkSNARK NN computation and maps directly to gate-level circuits. In particular, the tensor compute primitives hide the complexity of scalar-level operations and expose tensor computation capability, which is the building block of many zkSNARK NNs. The tensor compute primitives

Listing 6.1: Example of a 2-layer NN model in ZENO.

```
import ZENO
# Create a zkSNARK NN class.
class Shallow_Net():
   def __init__(self, X, Y, W1, W2):
        # Specify the zkSNARK NN
        # First Fully Connected Layer
        X = ZENO.fullyConnected(W1, X)
        # ReLU Layer
        X = ZENO.ReLU(X)
        # Second Fully Connected Layer
        X = ZENO.fullyConnected(W2, X)
        Check_Equality(X,Y)

# Define a two-layer zkSNARK NN.
X_data, Y_data = ZENO.Tensor([...]), ZENO.Tensor([...])
W1_data, W2_data = ZENO.Tensor([...]), ZENO.Tensor([...])
X = ZENO.zkTensor(X_data, 'public')
Y = ZENO.zkTensor(Y_data, 'public')
W1 = ZENO.zkTensor(W1_data, 'private')
W2 = ZENO.zkTensor(W2_data, 'private')
model = Shallow_Net(X, Y, W1, W2)
# Generate circuit
circuit = ZENO.generate(model)
# Generate Constraint System (CS)
CS = ZENO.circuitComputation(model)
# Proof generation
proof = ZENO.securityComputation(X_data, W1_data, W2_data)
# Share proof to verifier for verification ...
```

also allow users to easily specify the privacy type of images and weights, which mitigates the manual efforts in specifying the privacy of each scalar. The tensor compute primitives directly support `dotProduct` which consumes most computation in neural networks. This high-level `dotProduct` can be directly mapped to gate-level circuits with optimized circuit generation (discussed in section 6.5 and section 6.6). We then introduce `fullyConnected`, `convolution`, `pool`, and `ReLU` to support popular layers in NNs. We also provide `addTensor` and `mulTensor` to facilitate user-defined NN operations such as residual connection [2].

Case Study of the ShallowNet. We show ShallowNet in ZENO language construct, as shown in Listing 6.1. Note that the weights W1 and W2 are labeled as "private" to protect the privacy of private NNs. The input image X is labeled as "public" following

the accuracy zkSNARK NN scheme that proves the accuracy of a private NN on a public dataset. The input image X can also be labeled as "private" when also protecting the privacy of the input image.

6.5 Privacy-type Driven Optimization

In this section, we propose privacy-type driven optimizations. Our key insight is that fully exploiting privacy of input data can significantly reduce the number of constraints, which leads to proportional performance improvement for zkSNARK NNs. To this end, we first propose a *privacy-adaptive circuit generation* to exploit privacy type for reducing the number of constraints. Then, we propose *privacy-aware knit encoding* to further squeeze the number of constraints when only NN weight or image is private.

6.5.1 Privacy-adaptive Circuit Generation

We propose privacy-adaptive circuit generation to reduce the number of constraints in zkSNARK NNs. We observe that many zkSNARK NNs algorithmic designs [35, 229, 232, 271, 236] only require one of image or weights to be private. For example, ZEN [35] only keeps privacy of NN weights and use a public dataset to prove the NN accuracy. A naive implementation usually ignores privacy type of input data and generate constraints for each multiplication in zkSNARK NN, which leads to a large number of constraints and high latency. Our key insight is that, in zkSNARK, only multiplying two private scalars generates constraints while multiplying a public scalar (*e.g.*, Const) with a private scalar does not generate constraint. To this end, ZENO automatically incorporates user-specified privacy requirement into circuit generation to reduce the number of constraints.

We present our privacy-adaptive circuit generation for dot products which can be easily applied to many zkSNARK NN layers (*e.g.*, fully-connected, convolution, and

average pooling). Formally, we consider a weight vector $W = [w_1, w_2, ..., w_n]$ with privacy p_w and a feature vector $X = [x_1, x_2, ..., x_n]$ with privacy p_X where p_W and p_X are user-specified privacy type ("private" or "public"). zkSNARK first computes a *reference value* ref in plaintext according to dot product. Then, zkSNARK proves in constrains that $ref = \sum_{i=1}^{n} w_i * x_i$. In the last layer of zkSNARK NN, ref is the NN prediction such as a "cat" or "dot". We show the mapping from high-level dot product computation $\sum_{i=1}^{n} x_i * w_i$ to low-level constraints $(\sum_{i=1}^{n} a_{j,i} X_i) * (\sum_{i=1}^{n} b_{j,i} X_i) = Wire_j, \;\; j \in \{1, 2, ..., m\}$ where X_i and $Wire_j$ are private values, and $a_{j,i}$ and $b_{j,i}$ are public coefficients (see background in Equation 6.1).

Both private feature and private weights. When both feature and weights are private, we have n multiplications between private scalars w_i and x_i and $n-1$ addition to sum the multiplication output. Since both w_i and x_i are private values, we generate one constraint for each multiplication $w_i * x_i = Wire_i$. Formally, each multiplication can be written as constraints $(1 * w_i) * (1 * x_i) = Wire_i$. This leads to n constraints for multiplying private scalars. Then, we generate a linear combination $LC = \sum_{i=1}^{n} 1 * Wire_i$ to represent the computation result in zkSNARK and check the equality between LC and a reference value ref for dot product $W \cdot X$. Intuitively, this circuit checks that the dot product of private input W and X equals to ref without releasing the value of W and X. Checking equality leads to an extra constraint. Formally, we have $n+1$ constraints:

$$\begin{aligned} &(1 * w_i) * (1 * x_i) = Wire_i, \quad i \in \{1, 2, ..., n\} \\ &(\sum_{i=1}^{n} 1 * Wire_i + (-1) * ref) * (1 * D_1) = D_0 \end{aligned} \tag{6.3}$$

where $D_1 = 1$, $D_0 = 0$, and -1 is conducted on finite field.

Either private feature or private weights. We consider public weight W and private feature X since the design can be easily applied to the case with public weight and private feature. When weight W is private and feature X is public, we have n multipli-

cations between private weight scalar w_i and public feature scalar x_i and $n-1$ additions to sum the multiplication output. One naive design is to still generate one constraint for each multiplication. However, our key insight is that the public weight scalar w_i can be treated as public coefficients in Equation 6.1 which eliminates unnecessary constraints. To this end, we can directly generate a linear combination $LC = \sum_{i=1}^{n} w_i * x_i$ with public scalars w_i as coefficients and check equality with ref. This design requires only 1 constraint

$$(\sum_{i=1}^{n} w_i * x_i + (-1) * ref) * (1 * D_1) = D_0 \tag{6.4}$$

This is significantly smaller than $n+1$ constraints required for both private feature and private weights.

6.5.2 Privacy-aware Knit Encoding

We propose privacy-aware knit encoding to further reduce the number of constraints when only image or weight is private. Behind knit encoding, there are two major observations. First, zkSNARK NNs usually build upon quantized neural networks (*e.g.*, with `uint8`) while zkSNARK computes over finite fields (*e.g.*, 254-bit integers). Naively encoding `uint8` with finite fields leads to unnecessary memory and computation overhead. Second, when only image or weight is private, multiplying a public scalar with a private scalar does not introduce new constraints. In this case, constraints are only introduced when checking the equality, as discussed in subsection 6.5.1. One natural question is *whether we can reduce the number of constraints by exploiting "free" (in the sense of the number of constraints) multiplications between public scalars and private scalars.*

Naive encoding. Consider a fully connected layer with a public weight $W = [W_1, W_2] \in \texttt{uint8}^{2 \times c_{in}}$, a private feature $X = [x_1, x_2, ..., x_{c_{in}}] \in \texttt{uint8}^{c_{in}}$, and the output $Y = [y_1, y_2] \in \texttt{uint8}^2$. The fully-connected layer can be treated as two dot products

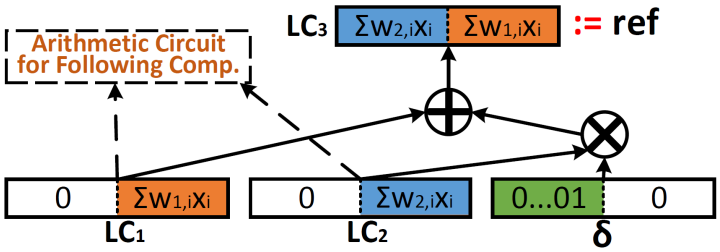


Figure 6.5: Knit encoding with batch size $s = 2$. LC_1 and LC_2 are two finite fields with leading bits as 0. $\delta = 2^{2*b_{in}+\lceil log(c_{in})\rceil}$ is a finite field such that multiplying δ is equivalent to bit shifting. ":=" indicates equality check.

$y_i = W_i \cdot X, i \in \{1, 2\}$. One naive approach is to independently encode individual dot products following Equation 6.4. This approach leads to 1 constraint for each dot product and require 2 constraints for the fully connected layer. However, this approach encodes low-bit quantized neural network values (*e.g.*, `uint8`) with high-bit finite fields (*e.g.*, 254-bit), which leads to extra constraints and higher latency.

Knit encoding with batch size $s = 2$. We propose to batch multiple low-bit values (*e.g.*, `uint8`) into one high-bit finite field (*e.g.*, 254-bit) to reduce the number of constraints, as illustrated in Figure 6.5. We first generate two LCs

$$LC_1 = \sum_{i=1}^{c_{in}} w_{1,i} * x_i, \quad LC_2 = \sum_{i=1}^{c_{in}} w_{2,i} * x_i$$

Generating LC_1 and LC_2 does not introduce constraints since we are multiplying public scalars with private scalars. Here, both LC_1 and LC_2 are finite fields. We note that only $2 * b_{in} + \lceil log(c_{in})\rceil$) bit of each LC are non-zero values where $b_{in}(=8)$ is the bit width of weights and features.

Instead of naively introducing constraints for checking equality between LC_i and y_i, we further encode these two LCs into one LC:

$$\begin{aligned} LC_3 &= LC_1 + LC_2 * \delta \\ &= \sum_{i=1}^{c_{in}} w_{1,i} * x_i + \sum_{i=1}^{c_{in}} (w_{2,i} * \delta) * x_i \end{aligned}$$

Here, $\delta = 2^{2*b_{in}+\lceil log(c_{in})\rceil}$ is sufficiently large to ensure the correctness of encoding. δ is

also a public scalar such that generating LC_3 does not introduce constraints.

Finally, we compute the encoded output value $ref = y_1 + y_2 * \delta$ and introduce 1 constraint to check equality of these two dot products simultaneously

$$(\sum_{i=1}^{c_{in}} w_{1,i} * x_i + \sum_{i=1}^{c_{in}} (w_{2,i} * \delta) * x_i + (-1) * ref) * (1 * D_1) = D_0$$

This constraint bitwisely checks equality such that $W_1 \cdot X = y_1$ and $W_2 \cdot X = y_2$ when δ is sufficiently large.

Knit encoding for arbitrary batch size s. Knit encoding can be generalized to arbitrary batch size s. Formally, knit encoding takes a public weight $W = [W_1, W_2, ..., W_s] \in \texttt{uint8}^{s \times c_{in}}$, a private feature $X = [x_1, x_2, ..., x_{c_{in}}] \in \texttt{uint8}^{c_{cin}}$, and the output $Y = [y_1, y_2, ..., y_s] \in \texttt{uint8}^s$. We first generates s LCs for dot products

$$LC_j = \sum_{i=1}^{c_{in}} w_{j,i} * x_i, \quad j \in \{0, 1, ..., s-1\}$$

Then, we encode s LCs into one LC

$$LC_s = \sum_{j=0}^{s-1} \sum_{i=1}^{c_{in}} (w_{j,i} * \delta^j) * x_i$$

Since we only require multiplication between public scalars and private scalars, we do not introduce constraints when generating these LCs. Finally, we can compute the encoded output value $ref = \sum_{j=0}^{s-1} y_j * \delta^j$ and use 1 constraint to bitwisely check the euqality of s dot products:

$$(\sum_{j=0}^{s-1} \sum_{i=1}^{c_{in}} (w_{j,i} * \delta^j) * x_i + (-1) * ref) * (1 * D_1) = D_0$$

Theoretical analysis on vector length n and batch size s. Knit encoding automatically selects batch size s to maximize the performance benefits while avoiding bit overflow. Formally, given the vector length n, input data bitwidth b_{in}, and finite field bitwidth b_{out}, each dot product requires $2 * b_{in} + \lceil \log n \rceil$ bits and all s dot products requires $s * (2 * b_{in} + \lceil \log n \rceil)$ bits. To avoid bit overflow and maximize benefits, we select

Table 6.3: Comparison between knit encoding and stranded encoding. We consider 8 bits for features and weights and 254 bits for finite fields.

	Knit Encoding	**Stranded Encoding [35]**
Max Constraint Saving	8×	4×
Encoding Overhead	0 Constraint	0 Constraint
Decoding Overhead	0 Constraint	632 Constraints
Privacy	One private	Both Private

batch size as the largest integer satisfying $s \leq b_{out}/(2 * b_{in} + \lceil \log n \rceil)$. For example, on dot product with $b_{in} = 8$-bit data, $b_{in} = 8$-bit weight, $b_{out} = 254$-bit finite field, and length $n = 1024$, we select $s = 9$ to maximize benefits while avoiding bit overflow.

Comparing with Stranded Encoding. Existing work [35] proposed stranded encoding which shares similar high-level motivation as our knit encoding. It focuses on the case with private weights and private features by reducing the number of multiplications. However, stranded encoding and knit encoding are significantly different in multiple perspectives, as summarized in Table 6.3. Stranded encoding can be applied when both features and weights are private while knit encoding can be applied when only features or weights is private. By exploiting privacy type, knit encoding can save more constraints with significantly reduced decoding overhead.

6.6 Tensor-type Driven Optimization

In this section, we propose tensor-driven optimizations. We first propose *ZENO circuit* as an efficient intermediate representation (IR) between high-level NN layers and low-level constraints. Then, we propose *workload-specialized parallel scheduler* to identify parallel computation opportunities in ZENO circuit especially across NN layers.

6.6.1 ZENO Circuit for Efficient IR

We present our ZENO circuit as an efficient intermediate representation (IR) from high-level zkSNARK NN arithmetic function to low-level constraints. Since low-level constraints require a specialized mathematical format $(\sum_{i=1}^{n} a_{j,i}X_i) * (\sum_{i=1}^{n} b_{j,i}X_i) = Wire_j$ (see Equation 6.1), it is challenging to manually write constraints for an arbitrary arithmetic function. Existing work [248] utilizes arithmetic circuit as an intermediate representation to automatically map arithmetic functions into constraints. However, it is designed for scalar computations and ignores intrinsic tensor types in zkSNARN NNs which leads to unsatisfactory performance. We first analyze the bottleneck in arithmetic circuit and then propose ZENO circuit as an efficient intermediate representation.

Arithmetic circuit. Arithmetic circuit first breaks an arbitrary arithmetic function into a sequence of scalar multiplication and scalar addition operations. Then, it maps each operation to a corresponding multiplication gate and addition gate, as discussed in subsection 6.3.1. We show an example of arithmetic circuit for dot product in Figure 6.6(a).

Consider a weight vector $W = [w_1, w_2, w_3, w_4]$, a feature vector $X = [x_1, x_2, x_3, x_4]$, and an arithmetic function

$$F(W, X) = w_1 * x_1 + w_2 * x_2 + w_3 * x_3 + w_4 * x_4$$

Here, we have 4 scalar multiplications and 3 scalar additions. Arithmetic circuit first maps each multiplication to a multiplication gate (*e.g.*, $Gate_1$ and $Gate_2$) and maps each addition to an addition gate (*e.g.*, $Gate_3$ and $Gate_5$):

$$Gate_i = w_i * x_i, \quad i \in \{1, 2, 4, 6\}$$

$$Gate_i = Gate_{i-2} + Gate_{i-1}, \quad i \in \{3, 5, 7\}$$

In total, there are 4 multiplication gates and 3 addition gates. Here, all computation are

symbolic since the arithmetic circuit describes computation in the arithmetic function regardless of specific values in input vector W and X.

Given this arithmetic circuit, we need to conduct *circuit computation* which converts individual gates into constraints with a specialized mathematical format (Equation 6.1). Without loss of generality, we consider public weight and private feature here. We can first check privacy of each scalar and generate a tuple where public input w_i is coefficient (*i.e.*, $a_{j,i}$ and $b_{j,i}$ in constraints Equation 6.1) and private input x_i is variables:

$$(1, Gate_i) = (w_i, x_i), \quad i \in \{1, 2, 4, 6\}$$

$$(1, Gate_i) = (1, Gate_{i-2}) + (1, Gate_{i-1}), \quad i \in \{3, 5, 7\}$$

For addition gates, we have 1 as the coefficient. We note that this coefficient can be arbitrary integer in general. Then, we need to recursively expand children gates for addition gate, if one of its children gates is till an addition gate. For example, $Gate_5$ has two children gates $Gate_3$ and $Gate_4$ where $Gate_3$ is still an addition gate. In this case, we need to symbolically expand $Gate_5$ by multiplying the coefficient of $Gate_5$ with the coefficients of individual children and grandchildren gates (*i.e.*, $Gate_1$, $Gate_2$, and $Gate_4$). This expansion introduces m finite field multiplications due to iterating through m grandchildren gates. For dot product, m increases from 2 to $n-1$(=3 in Figure 6.6). Overall, circuit computation on arithmetic circuit costs $O(n^2)$ where n is the vector length. This leads to prohibitive latency for zkSNARK NNs with millions of gates in arithmetic circuit.

ZENO Circuit for Dot Product. To address this problem, we propose a ZENO circuit to minimize the number of gates and reduce the computation complexity to $O(n)$ where n is the vector length. The core idea is to exploit the commutative property of addition gates in zkSNARK. In particular, the order of addition gates can be exchanged while the order between two multiplication gates and the order between a multiplication gate and an addition gate need to be maintained. We show ZENO circuit for dot product

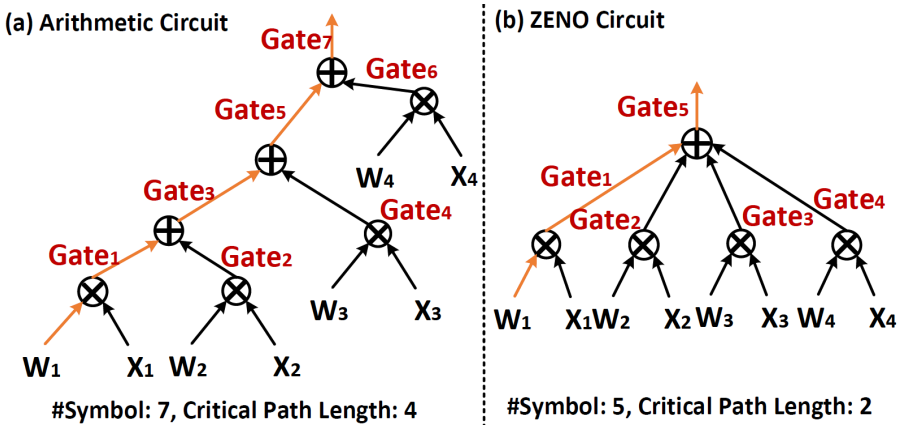


Figure 6.6: Illustration of ZENO IR for dot product of $\mathbf{W} \cdot \mathbf{X} = [W_1, W_2, W_3, W_4] \cdot [X_1, X_2, X_3, X_4]$.

Table 6.4: NN layer complexity comparison between conventional arithmetic circuit and proposed ZENO circuit.

IR	layer	Input Shape	# Gate	# Wire	# LC	len(CriticalPath)	Computation
Arithmetic Circuit	Dot Product	(n, n)	$2n-1$	n	$n-1$	n	$O(n^2)$
	Fully Connected	$(m \times n, n)$	$m(2n-1)$	mn	$m(n-1)$	n	$O(mn^2)$
	Convolution	$(m \times n, n \times k)$	$mk(2n-1)$	mkn	$mk(n-1)$	n	$O(mkn^2)$
	Pool	$(m \times n)$, s	$\frac{mn}{s^2}(s^2-1)$	0	$\frac{mn}{s^2}(s^2-1)$	s^2-1	$O(mns^2)$
ZENO Circuit	Dot Product	(n, n)	$n+1$	n	1	2	$O(n)$
	Fully Connected	$(m \times n, n)$	$m(n+1)$	mn	m	2	$O(mn)$
	Convolution	$(m \times n, n \times k)$	$mk(n+1)$	mkn	mk	2	$O(mkn)$
	Pool	$(m \times n)$, s	$\frac{mn}{s^2}$	0	$\frac{mn}{s^2}$	1	$O(mn)$

of length 4 in Figure 6.6(b). We introduce 4 multiplication gatess ($Gate_1$, $Gate_2$, $Gate_3$, and $Gate_4$) and only one addition gate ($Gate_5$). Note that ZENO circuit has the same number of multiplication gates but a significantly smaller number of addition gates. This reduced number of addition gates significantly saves the number of computations during *circuit computation*. On ZENO circuit, we can skip *circuit computation* operation for addition gates and directly generate constraints. In particular, we only need 5 operations for converting ZENO circuit while requiring 12 operations for converting arithmetic circuit. We also note that ZENO circuit shows short critical path length (=2) than arithmetic circuit with length (=4).

Formally, given two vectors $[w_1, w_2, ..., w_n]$ and $[x_1, x_2, ..., x_n]$ of length n, ZENO circuit contains binary multiplication gates and multi-child addition gates. The binary

multiplication gate takes two input gates. To support dot product on two vectors of length n, we need n multiplication gates for each $w_i * x_i$. The multi-child addition gate takes n inputs where n can be arbitrarily large number. This gate efficiently supports summation over a large number of scalars in dot product and significantly reduces the number of addition gates. In comparison, arithmetic circuit for dot product requires $n-1$ binary addition gates. In total, ZENO circuit for dot product generates $n+1$ gates while arithmetic circuit generates $2n-1$ gates. We also stress that both ZENO circuit and arithmetic circuit generate the same constraint systems. Thus ZENO circuit maintains the semantic and can be used as an in-place replacement of arithmetic circuit.

ZENO circuit for fully connected, convolution, and pooling layers. We propose ZENO circuit for fully connected, convolution, and pooling layers as an extension to ZENO circuit for dot product. Fully-connected layer takes two input tensors $\mathbf{W} \in \mathbb{R}^{m \times n}$ and $\mathbf{X} \in \mathbb{R}^n$ and generates one output tensor $\mathbf{Y} = \mathbf{WX} \in \mathbb{R}^m$. With the help of *img2col* algorithm [259], convolution layer can also be transformed into a matrix-matrix multiplication. It takes two input tensors $\mathbf{W} \in \mathbb{R}^{m \times n}$ and $\mathbf{X} \in \mathbb{R}^{n \times k}$ and computes an output tensor $\mathbf{Y} = \mathbf{WX} \in \mathbb{R}^{m \times k}$. Since fully connected and convolution layer can be viewed as m and mk independent dot products, we simply duplicate dot product circuits for m and mk times as ZENO circuit for fully-connected and convolution layers, respectively. For the pooling layer, we focus on average pool following state-of-the-art zkSNARK NN security scheme [35]. Given an input tensor of shape $m \times n$ and a constant s, average pool splits the tensor into small grids of shape $s \times s$ and computes the average value in each grid. Thus average pool can be viewed as a dot product between a *one* vector $\mathbf{1}$ of length s^2 (*i.e.*, all elements are 1's) and a vector of all values in a grid. On the ReLU layer, ZENO shares the same circuit as scalar-level zkSNARK frameworks since ReLU contains only elementwise comparison.

Theoretical benefit analysis. We summarize theoretical benefits of ZENO circuit

in Table 6.4. One significant result is that ZENO circuit requires $O(n)$ computation for dot product while arithmetic circuit requires $O(n^2)$ computation. This generalizes to fully connected, convolution, and pool layers with significantly reduced complexity. This saving leads to significant performance improvement on zkSNARK NNs with millions of gates. ZENO circuit also introduces a constant critical path length of 2, in contrast to the length n in arithmetic circuit. This exposes parallel opportunities that can hardly be identified in arithmetic circuit due to complex dependency.

6.6.2 Workload-specialized Parallel Scheduler

Workload-specialized parallel scheduler identifies the parallel computation opportunities in circuits and exploits these opportunities for speedup. While NNs have parallel opportunities in the same NN layer (*e.g.*, fully connected layer), NNs are also intrinsically sequential across layers where leading layer needs to be computed before following layers. This cross-layer dependency still hurdles paralleling zkSNARK NN computation even with ZENO circuit that improves parallelism within NN layer. Naively parsing the circuit at NN level still leads to heavy overhead.

We propose a lightweight *dependency-aware workload scheduler* to identify cross-layer dependency in circuit and map parallel workloads to individual threads. We have two major observations. First, gates in the same zkSNARK NN layer usually can be computed independently while gates in later layers depend on gates in leading layers. Second, the number of gates for a NN layer is proportional to the number of computation in this layer. To this end, we propose a three-step design. First, based on the plaintext NN with specific layer shapes, we first count the number of addition and multiplication in each layer. For example, given a fully connected layer with shape $M \times N$, there are $M \times N$ multiplications and $M \times (N-1)$ additions. Then, based on this number of computation,

we directly identify the gates for each NN layer since each addition and multiplication is mapped to exactly one gate in the circuit. Finally, we evenly assign gates in the same layer to each thread for acceleration.

6.7 NN-centric System Optimization

In this section, we propose *NN-centric system optimization* to further accelerate zkSNARK NN computation.

6.7.1 NN-inspired Computation Reuse

ZENO identifies redundant computation in zkSNARK NNs and removes such redundancy for improving performance. In particular, we identify two types of computation reuse opportunities – *frequency-based cache service* for mitigating redundancy when computing a single image and *batch-specialized constraint system sharing* for mitigating redundancy when computing a batch of images.

Frequency-based Cache Service. We build a lightweight cache service to cache the computation results of frequent weight and data pairs. We have two insights behind our cache service. First, zkSNARK NNs usually computes with `uint8` values since zkSNARK supports only computation on finite fields (*e.g.*, 254-bit integers). Since there are at most 256 values for `uint8`, the same value appears frequently. Second, NN weights and features usually follow Normal distribution where many weights and features are around zero, as widely observed in the NN algorithmic area [1, 67]. This distribution makes many values around zero appear frequently. To this end, our cache service can improve the performance by reducing the number of expensive computations on λ-bit finite fields ($\lambda \geq 254$).

To mitigate the runtime overhead, we adopt a two-phase design. During the offline

profiling phase, we evaluate the plaintext NN on a small set (=100) of images and profile the frequency of weight and data pairs. We rank all pairs by frequency and keep the top-k(=5) values and the computation results in a hash table. This offline profiling introduces negligible overhead since it is only conducted once on a plaintext NN. During the online computation phase, for each weight and data pair, we first search the pair in the hash table and reuse the results in the hash table. In this way, we can mitigate expensive security computation for a large number of weight and data pairs that appear frequently.

Batch-specialized Constraint System Sharing. We share the constraint system across images when using the same zkSNARK NN to process a batch of images. Our key insight is that the constraint system is a description of the zkSNARK NN computation. Since we usually use the same zkSNARK NNs to process a batch of images, the same computation applies to each image such that the constraint system can be shared. One specific example is the accuracy scheme in ZEN [35], where the same zkSNARK NN is used to process $n(= 100)$ images for proving the accuracy of the zkSNARK NN. To this end, ZENO provides a batch mode that takes a zkSNARK NN and a batch of images. The *circuit computation* step is only conducted once and the constraint system is reused for different images. In particular, in the same constraint system, we assign different values to input variables according to images.

6.7.2 zkSNARK-aware NN Fusion

We propose *zkSNARK-aware NN fusion* to further reduce the number of constraints for performance improvement. Our key insight is that the number of constraints is proportional to the number of computation in zkSNARK NNs. While fusion has been utilized to accelerate non-zkSNARK NNs [279, 280, 27], there are several intrinsic differences in tensor fusion for zkSNARK NNs. *First*, fusion in non-zkSNARK NNs usually

Table 6.5: Neural Networks for Evaluation.

Network	Abbr.	Dataset	#FLOPs (K)	Acc.(%)
ShallowNet	*SHAL*	MNIST	102	94.91
LeNetCifarSmall	*LCS*	Cifar-10	530	55.35
LeNetCifarLarge	*LCL*	Cifar-10	7,170	63.68
VggNet-16	*VGG16*	Cifar-10	19,917	84.19
ResNet-18	*RES18*	Cifar-10	32,355	85.45
ResNet-50	*RES50*	Cifar-10	69,191	87.05

target reducing memory access by avoiding saving intermediate results in memory. In zkSNARK NNs, we target reducing the number of computations which decides the number of constraints and the latency of generating zero-knowledge proofs. *Second*, fusion in non-zkSNARK NNs usually fuses all element-wise computation (*e.g.*, relu) with convolution layers. However, many element-wise computation cannot be fused in zkSNARK NNs. For example, relu layer cannot be fused since relu requires expensive comparison operator with hundreds of constraints in zkSNARK.

To this end, we propose *pre-computation-based fusion* to reduce computation in zkSNARK NNs. Many NN layers involve injective computation such as one-to-one scale and addition. We can fuse such injective layers with convolution and fully-connected layers. For example, consider a fully connected layer $Y = WX$ and a batch normalization layer $BN(Y) = \gamma * Y + \beta$ which is an injective layer. Naive approach is to independently prove the computation of these two layers which leads to extra constraints. Instead, we can precompute the fused weight value $\gamma * W$ and directly prove the computation of $(\gamma * W)X + \beta$ to save constraints.

6.8 Evaluation

In this section, we comprehensively evaluate ZENO over various datasets and popular NNs.

Baselines. We compare ZENO with Arkworks [204, 248], which is the state-of-the-art zkSNARK framework and widely used in industry zkSNARK products [281, 282, 283]. We also compare with two other representative zkSNARK frameworks, Bellman [274] and Ginger [213] for comprehensive comparison.

Datasets. We evaluate with two popular datasets (MNIST and CIFAR-10) in secure deep learning field [207, 246, 205, 206, 208, 35]. **MNIST** [284] is a large dataset for handwritten digits classification with $60,000$ training images and $10,000$ testing images in gray-scale with the shape of $28 \times 28 \times 1$. **CIFAR-10** [285] is a classification dataset with 10 classes (e.g., cat and dog). It contains $50,000$ training images and $10,000$ testing images of shape $32 \times 32 \times 3$.

Models. We evaluate six neural networks, as summarized in Table 6.5. The evaluation of these six variants demonstrates the performance of ZENO under diverse model sizes. In particular, ShallowNet [35] contains two fully connected layers and one ReLU layer. *LeNetCifarSmall* and *LeNetCifarLarge* are two variants of LeNet [254] with 5 layers but different number of computation. *VggNet-16* [286], *ResNet-18* [2], *ResNet-50* [2] have 16, 18, and 50 NN layers, respectively.

Experiment Configuration. All the evaluations run on a server with Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz and 503 GB DRAM.

6.8.1 End-to-End Evaluation

In this section, we show the end-to-end performance improvement from ZENO on various privacy settings that cover diverse use cases of zkSNARK NNs. For example, the privacy setting of private image and public weights can be used when we only protect the user image privacy (*e.g.*, face image) and prove the user's identity on a public NN (*e.g.*, a face recognition based door lock system). The privacy setting of private weights and

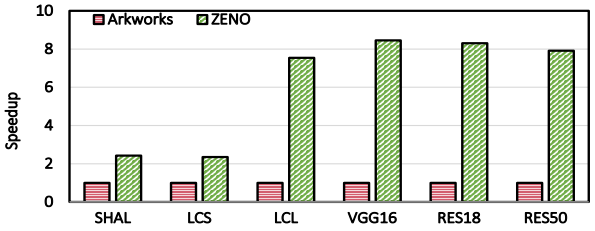


Figure 6.7: Overall speedup: *private* images & *public* weights.

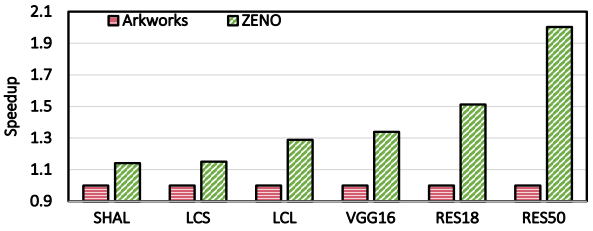


Figure 6.8: Overall speedup: *private* images & *private* weights.

private images can be used when we aim to protect both privacy-sensitive images (*e.g.*, medical images) and weights (*e.g.*, private NNs as we discussed in section 6.1). We skip the privacy setting of private weights and public images since it shows similar results as private images and public weights.

We first show the overall speedup when proving private images and public weights in Figure 6.7. Overall, ZENO achieves upto 8.5× speedup than Arkworks. This result shows that ZENO can significantly improve the performance of zkSNARK NNs. We also observe that ZENO achieves higher speedup on large NNs (*e.g.*, 8.5× on VGG16) than small NNs (*e.g.*, 2.4× on SHAL). The reason is that tensor-type driven optimization (section 6.6) reduces the quadratic computation complexity to linear complexity for many NN layers (*e.g.*, fully connected, convolution, and pool). We highlight that we reduce the latency of ResNet-50 from 5154 seconds (around 1.5 hours) to 680 seconds (around 11 minutes), which makes it promising to construct practical zkSNARK NNs.

We show overall speedup when proving private NN weights and private images in Figure 6.8. We achieve up to 2.01× speedup, which shows the effectiveness of ZENO optimizations. We also observe a similar trend as Figure 6.7 that ZENO achieves higher speedup on larger zkSNARK NNs. This validates the benefits from our tensor-driven optimizations on reducing the computation complexity. Comparing with Figure 6.7, we observe smaller speedup. The reason is that our type-sensitive circuit generation provides

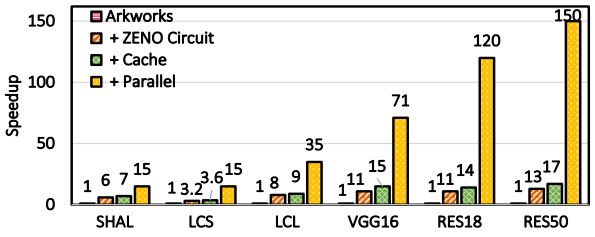


Figure 6.9: Circuit comput. speedup: *private* images & *public* weights.

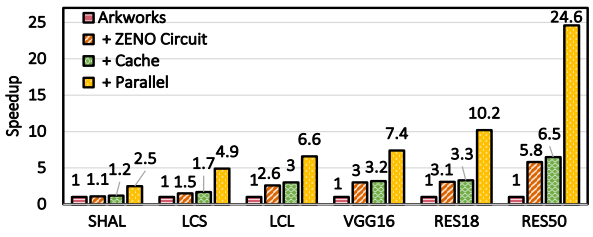


Figure 6.10: Circuit comput. speedup: *private* images & *private* weights.

more aggressive optimization for the setting with private weights and public images. This shows the importance of considering privacy information (section 6.5) when optimizing zkSNARK NNs.

6.8.2 Optimization Analysis

In this section, we show speedup from individual ZENO optimizations. We demonstrate the benefits on *circuit computation* and *security computation* steps in zkSNARK which account for most latency in zkSNARK NNs (see Figure 6.4). We first show speedup on *circuit computation* step at NN level and individual NN layer level. Then, we show the benefits on *security computation* from knit encoding. Finally, we show benefits when proving a large number of images.

Performance benefits on *circuit computation* step for entire NNs. We show speedup on *circuit computation* step for private images and public weights in Figure 6.9. Overall, we achieve speedup of 67.7× on average (from 15× to 150×) for *circuit computation* step. This speedup increases as zkSNARK NN size increases due to our ZENO circuit (subsection 6.6.1) that reduces quadratic complexity to linear complexity. On individual optimizations, we observe 8.7× speedup from ZENO Circuit (subsection 6.6.1), 1.5× speedup from frequency-based cache service (subsection 6.7.1), and 6.2× speedup from workload-specialized parallel scheduler (subsection 6.6.2). These results show ben-

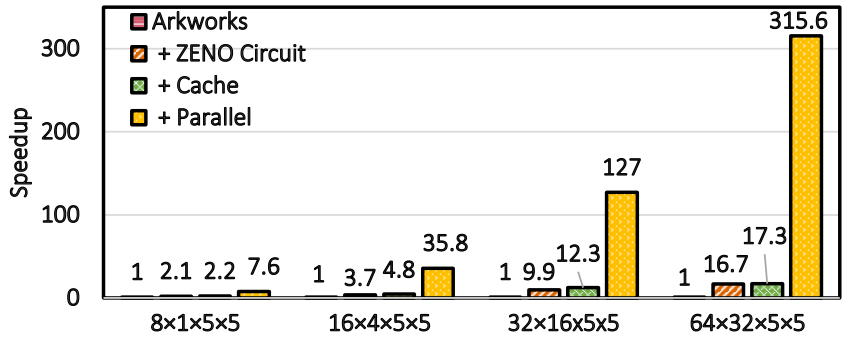


Figure 6.11: Circuit computation speedup: convolution. Shape: [#*out_channels*, #*in_channels*, *kernel_width*, *kernel_height*].

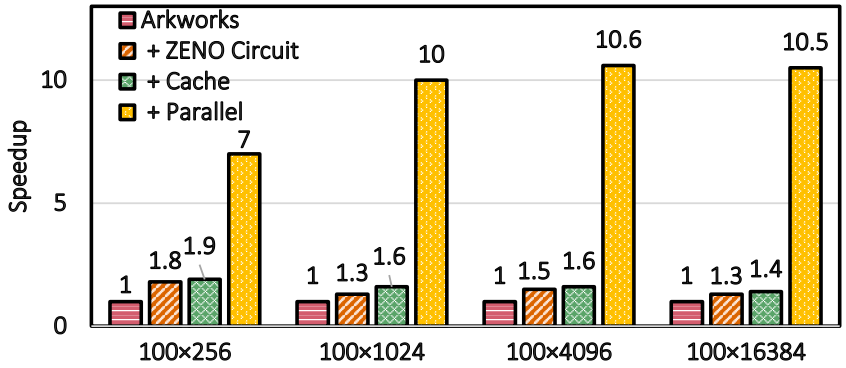


Figure 6.12: Circuit computation speedup: fully-connected layer. Shape: [#*in_channels*, #*out_channels*].

efits of individual optimizations on reducing zkSNARK NN latency.

We show speedup on *circuit computation* step for private images and private weights in Figure 6.10. We have similar observations as the case in private image and public weights. In particular, we observe 9.4× speedup on average (from 2.5× to 24.6×). On individual optimizations, we observe 2.9× speedup from ZENO circuit, 1.1× speedup from frequency-based cache service, and 2.9× speedup from workload-specialized parallel scheduler. Similar to the case in subsection 6.8.1, this speedup is smaller than the case for private weights and public images. This shows importance of privacy-type driven optimizations (section 6.5) that customize the circuit generation and encoding methods according to privacy types.

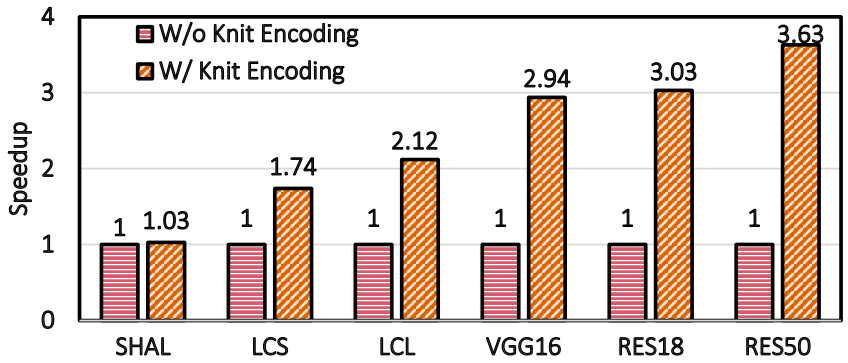


Figure 6.13: Speedup on *security computation* from knit encoding.

Performance benefits on *circuit computation* step at NN layer level. We further show the circuit computation speedup at NN layer level in Figure 6.11 and Figure 6.12. We focus on the two most time consuming layers – convolution and fully connected layers, under the privacy setting of private images and public weights. We omit the privacy setting of private images and private weights due to page limits. We achieve up to 315.6× speedup on convolution layers and 10.5× speedup on fully connected layers. This result matches up to 150× circuit computation speedup at NN level in Figure 6.9. We achieve higher speedup on convolution layers which gain more benefit from ZENO circuit due to the larger number of dot products. We also observe an increasing speedup on both convolution layers and fully connected layers as the layer size increases, thanks to the tensor-driven optimization that reduces computation complexity of *circuit computation* step.

Speedup on *security computation* from knit encoding. We show the benefits from knit encoding on accelerating *security computation* step in Figure 6.13. We show the result for private weights and public images, as discussed in subsection 6.5.2. Overall, we achieve up to 3.63× speedup. The reason is that knit encoding can effectively reduce the number of constraints, which decides the latency in *security computation* step. We observe that speedup increases from 1.03× to 3.63× as NN size increases. The reason is

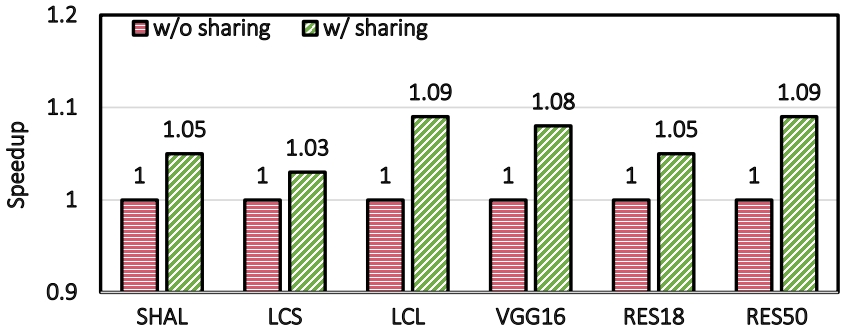


Figure 6.14: Overall performance: proving n (=100) images.

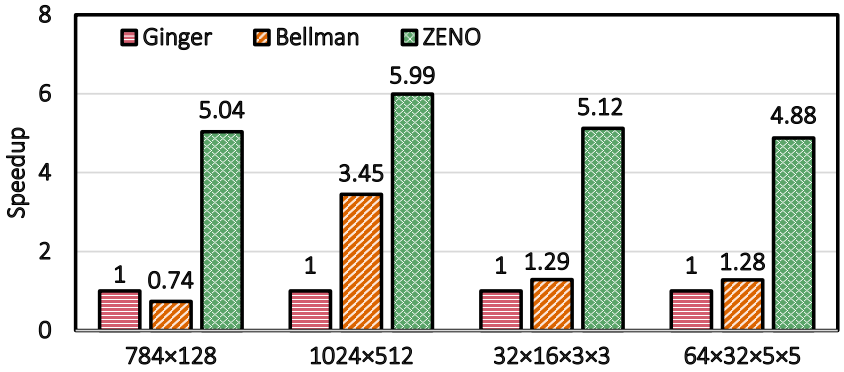


Figure 6.15: Speedup over Bellman and Ginger.

that, in larger zkSNARK NNs, fully-connected, convolution, and pooling layers account for larger portion of *security computation* latency such that knit encoding can bring more benefits.

Benefits from sharing when proving n (=100) images. We show the speedup from batch-specialized constraint system sharing (subsection 6.7.1) in Figure 6.14. While the latency of *circuit computation* step has been significantly reduced, we can still observe 6.5% speedup from this optimization. The reason is that the constraint system represents the computation procedure of a zkSNARK NN with constraints which can be assigned different values for different images.

6.8.3 Compared with other Frameworks

In this section, we further compare ZENO with two other representative general zkSNARK frameworks – Bellman [274] and Ginger [213]. These two frameworks are general zkSNARK framework and do not provide direct support for zkSNARK NNs. They require constraints (Equation 6.1) as inputs and cannot automatically compile arbitrary arithmetic function to constraints. We manually port compiled constraints from ZENO into Bellman and Ginger and compare *security computation* latency. We show results in Figure 6.15. We demonstrate the performance on two fully-connected layers with shape [$\#in\_channels$, $\#out\_channels$] and two convolution layers with shape [$\#out\_channels$, $\#in\_channels$, $kernel\_width$, $kernel\_height$]. Overall, we observe that ZENO achieves 4.09× speedup over Bellman and 5.26× speedup over Ginger. These benefits come from our NN-tailored optimizations such as privacy-aware knit encoding. Comparing across layers, we observe 1.7× to 6.8× speedup over Bellman and 4.9× to 6× speedup over Ginger. This result demonstrates the consistent benefits from ZENO on various layers.

Chapter 7

Faith: An Efficient Framework for Transformer Verification on GPUs

Transformer verification draws increasing attention in machine learning research and industry. It formally verifies the robustness of transformers against adversarial attacks such as exchanging words in a sentence with synonyms. However, the performance of transformer verification is still not satisfactory due to bound-centric computation which is significantly different from standard neural networks.

In this chapter, we propose **Faith**, an efficient framework for transformer verification on GPUs. We first propose a semantic-aware computation graph transformation to identify semantic information such as bound computation in transformer verification. We exploit such semantic information to enable efficient kernel fusion at the computation graph level. Second, we propose a verification-specialized kernel crafter to efficiently map transformer verification to modern GPUs. This crafter exploits a set of GPU hardware supports to accelerate verification-specialized operations which are usually memory-intensive. Third, we propose an expert-guided autotuning to incorporate expert knowledge on GPU backends to facilitate large search space exploration. Exten-

sive evaluations show that Faith achieves 2.1× to 3.4× (2.6× on average) speedup over state-of-the-art frameworks.

7.1 Problem Statement

Transformers [5, 78, 287, 288, 289, 290, 291] is an important category of neural networks (NNs) in machine learning research and industry. Transformers are first designed for natural language processing (NLP) and have achieved state-of-the-art accuracy across many NLP tasks such as neural machine translation [292, 293, 294] and sentiment analysis [295, 296, 297]. Due to its success, transformers have been widely used in many industrial products such as Facebook for hate speech detection [7] and Alexa for question answering [8]. Recently, transformers also show extraordinary accuracy for many computer vision tasks [298, 299, 300, 301, 302] and become the new trending model. However, similar to prior NNs, transformers are also vulnerable to adversarial attacks that add imperceptible perturbations to input data for maliciously changing transformer predictions [303, 304, 305, 306, 307]. One specific example of adversarial attack is to exchange words (*e.g.*, cold) in a sentence with carefully selected synonyms (*e.g.*, frigid). This vulnerability may result in security concerns for real-world applications. For example, an intentionally crafted hate speech may spread widely on social network.

Transformer verification has been proposed to formally verify the robustness of a transformer against adversarial attacks [11, 9, 10, 36]. Given an input data x and a transformer $F(x)$, transformer verification identifies a maximal bound ϵ, such that all inputs x' that are "close" to the input data (*i.e.*, $|x' - x| \leq \epsilon$) cannot "mislead" the transformer (*i.e.*, $F(x) = F(x')$). A larger ϵ indicates better robustness. Early verification approaches [11] enumerate all possible inputs x' that satisfy $|x' - x| \leq \epsilon$ and conduct inference on each input to check predictions. These approaches show prohibitive latency

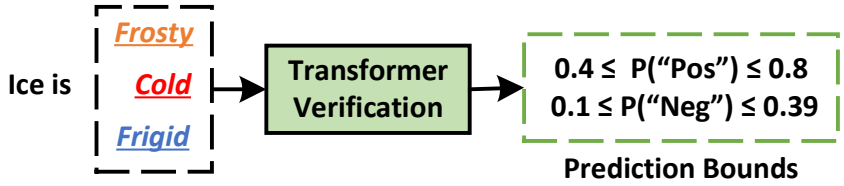


Figure 7.1: Illustration of transformer verification. Here, all perturbed inputs share the same prediction "positive" since the lower bound probability for "positive" (0.4) is higher than the upper bound probability for "negative" (0.39).

due to the large number of inputs x'. Recent transformer verification [9, 10] avoids such enumeration by providing a single pair of lower and upper bounds for transformer predictions over all these inputs, as illustrated in Figure 7.1. We can verify the robustness of a transformer if the lower bound of the correct prediction is higher than the upper bound of other predictions. The key computing pattern is a *bound-centric computation*, which computes a pair of inequality bounds for individual neurons. It first represents the input perturbations with inequality bounds over input neurons (*e.g.*, $x - \epsilon \leq x' \leq x + \epsilon$) and then propagates these bounds across layers to generate the bounds for transformer predictions.

While transformer verification can formally verify the robustness of transformers, it also introduces high latency and limits its applications. In particular, transformer verification usually leads to second-level latency [9] in contrast to millisecond-level latency of standard transformers. We identify three challenges behind efficient transformer verification.

Lack of performance optimization over transformer verification computing patterns. Existing transformer verifications usually utilize the existing deep learning (DL) frameworks, such as PyTorch [58], which are designed for standard NNs. However, transformer verification shows significantly different computing patterns from standard NNs due to the nature of bound-centric computation. For example, when computing the upper bound of an output neuron, transformer verification needs to use the upper

bound of the input neuron if the weight is positive; and the lower bound of the input neuron if negative. Straightforwardly deploying transformer verification to the existing DL frameworks usually leads to poor performance.

Lack of framework support for verifying diverse NN layers. Transformer verification shows large diversity in the bound computation for different types of NN layers such as projection layer with only perturbed features and self-attention layer with both perturbed weights and features. Even for the same type of NN layers, diverse upper bounds and lower bounds may be designed which requires different implementations. For example, Crown [308] utilizes two ReLU bound designs for generating more precise bounds for verification, where these bounds are selected dynamically according to the range of input neurons. This diversity makes it challenging to hand optimize GPU kernels in transformer verification.

Lack of verification-specialized adaptability towards modern GPUs. Transformer verification involves abundant memory-intensive operations such as reduction and broadcast. These memory-intensive operations can usually be significantly accelerated with rich architecture supports (*e.g.*, warp-level synchronized reduction) in modern GPUs. However, existing DL frameworks usually only focus on computation-intensive operations (*e.g.*, convolution) and ignore abundant optimization opportunities for memory-intensive operations. This leads to significant overhead in transformer verification with a large number of memory-intensive operations.

7.2 Overview of Proposed Solution

In this chapter, we build **Faith**, the first framework for efficient transformer verification on GPUs. We show an overview of the Faith framework in Figure 7.2. First, we propose ***semantic-aware computation graph transformation*** to fully exploit fusion

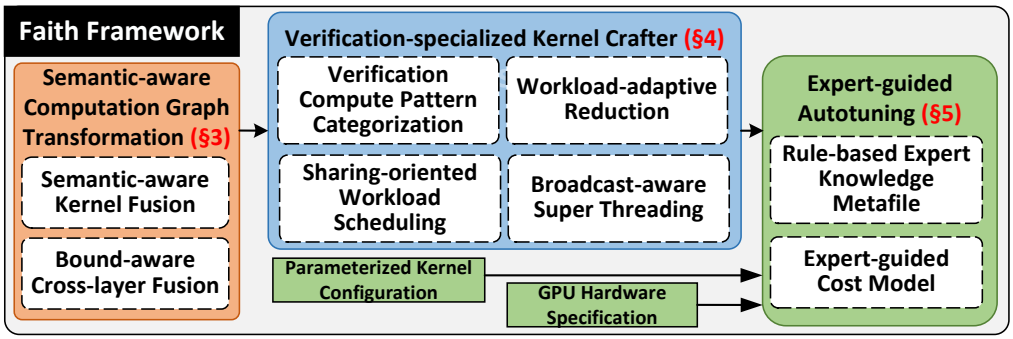


Figure 7.2: Overview of Faith Framework

opportunities in transformer verification at the computation graph level. Our key insight is that transformer verification shows significantly different computing patterns (*e.g.*, two kernels for computing lower and upper bounds involve similar input data) from standard NNs. These computing patterns usually exhibit abundant data reuse opportunities. By exploiting such semantic information, Faith can fully harvest performance potential in transformer verification and achieve significant speedup over existing DL frameworks.

Second, we propose a ***verification-specialized kernel crafter*** to optimize transformer verification towards modern GPUs. Transformer verification contains abundant memory-intensive operations, such as elementwise computation, reduction, and broadcast. These operations may have complex dependencies and lead to performance bottlenecks. To this end, Faith automatically exploits a set of GPU architecture supports to improve the parallelism of such operations. Moreover, Faith introduces a set of optimizations to effectively mitigate memory access and improve performance by exploiting GPU memory hierarchies.

Third, we propose ***expert-guided autotuning*** to efficiently search optimized implementations in the large search space. Existing DL frameworks [279, 280] usually conduct autotuning in a hardware-agnostic approach where an ML-based cost model is deployed to implicitly learn hardware impact over performance from scratch. Instead, we pro-

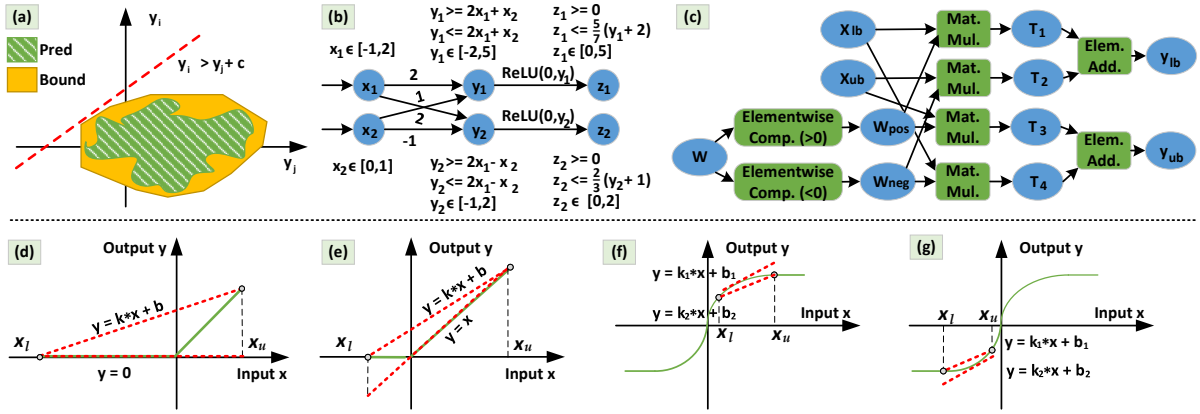


Figure 7.3: Illustration of transformer verification. (a) model prediction and verification bound; (b) an example of verifying a model with a fully connected layer and a ReLU layer; (c) computation graph of projection layer in transformer verification; (d)-(e) two types of bounds for ReLU layer; (f)-(g) two types of bounds for the *Tanh* layer.

pose a rule-based expert knowledge metafile to explicitly provide a small set of hardware characterizations and an expert-guided cost model to incorporate the expert knowledge. Faith exploits these two components to achieve efficient schedule exploration in the large design space of transformer verification.

Extensive experiments show that Faith achieves up to 3.4× speedup (2.6× on average) over state-of-the-art frameworks.

7.3 Related Work and Motivation

In this section, we first introduce the background of transformer verification (subsection 7.3.1). Then, we discuss related work on DL frameworks (subsection 7.3.2). Finally, we present opportunities and challenges for efficient transformer verification on GPUs (subsection 7.3.3).

7.3.1 Transformer Verification

Standard Transformers. Transformer [5, 78, 287, 288] takes a sentence as input and predicts a label for this sentence. Formally, the input X is a tensor of shape $Batch\_size \times Length \times Dim\_in$, where $Batch\_size$ is the number of sentences in a batch, $Length$ is the number of words in a sentence, and Dim_in is the embedding size of each word. When sentences in the same batch have different lengths, $Length$ is set to the maximal length in practice. A transformer has three types of operators. The first type is the *elementwise operator*, such as ReLU and Tanh. The second type is the *matrix multiplication operator* that takes an input tensor X, a weight matrix W, and generates an output tensor $Y = XW$. We note that these two types are similar to operators in prior neural networks. The third type is the *dot product operator*, which is the key idea behind the transformer model. Informally speaking, it takes two input tensors Q and K of the same shape $Batch\_size \times Length \times Dim\_in$. Then, it computes an output tensor $Y = Q^T K$ of shape $Batch\_size \times Length \times Length$ to measure the pairwise similarity between individual words in a sentence. This similarity can significantly improve the learning capacity of the model and the prediction accuracy.

Adversarial Attack on Transformers. Adversarial attack [309, 303, 304, 305, 306, 307] identifies small perturbations to input data X that can change the transformer prediction. Formally, consider a transformer $f(\cdot)$, an input sentence X, and a tolerable input perturbation bound ϵ, where the transformer correctly classifies X as a label i (*e.g.*, hate speech). In other words, the sentence has label i and $y_i > y_j$ for any $j \neq i$ where y_i is the predicted probability. Adversarial attack identifies a slightly perturbed sentence $X' = X + \eta$ such that $\eta \in B(0, \epsilon)$ and there exists a label j (*e.g.*, benign speech) such that $y_i < y_j$. This perturbed sentence X' is an *adversarial example*.

Transformer Verification. Transformer verification [11, 9, 10, 36] computes a

maximum bound ϵ and mathematically proves that there does not exist an adversarial example X' within the ϵ-ball of X (*i.e.*, $(X' - X) \in B(0, \epsilon)$). Verifying transformers is challenging since transformers are essentially non-convex functions. The key idea of transformer verification is to utilize linear bounds as an approximation to NN predictions. We illustrate transformer verification at the model prediction layer in Figure 7.3(a). Given these linear bounds, transformer verification can simply check if the predictions insides the bounds satisfy certain linear requirements, such as $y_i > y_j + c$, where c is a positive number. As illustrated in Figure 7.3(a), this bound-based approach is sound since the linear bound covers the non-convex area of NN predictions.

We show an example of bound-centric computation of transformer verification in Figure 7.3(b). Consider a fully connected layer $Y[j] = \sum_{i=1}^{n} W[j,i] \cdot X[i]$ where $Y[j]$, $W[j,i]$, and $X[i]$ are scalars. Here, we skip the index for batch size and length for notation simplicity. A formal summary of notations can be found in Table 7.1. For each neuron $X[i]$, there is a lower and a upper bound

$$X[i] \geq X_{lb}[i] + X_{lw}[i] * \vec{\epsilon}, \quad X[i] \leq X_{ub}[i] + X_{uw}[i] * \vec{\epsilon}$$

where $X_{lb}[i]$ and $X_{ub}[i]$ are scalars, $X_{lw}[i]$, $X_{uw}[i]$, and $\vec{\epsilon}$ are vectors. For the input neurons, we have $X_{lb}[i] = X_{ub}[i] = X[i]$, $X_{lw}[i]$ and $X_{uw}[i]$ are one-hot vectors with 1 at the index i and 0 at other indices. Given this linear bound, we can compute *concretized* bounds for each neuron as

$$X_l[i] = X_{lb}[i] - \epsilon * ||X_{lw}[i]||, \quad X_u[i] = X_{ub}[i] + \epsilon * ||X_{uw}[i]|| \tag{7.1}$$

where $||\cdot||$ computes the norm with reduction operations.

When computing the bounds for output neuron $Y[j]$, we note that bound computation

Table 7.1: Notations in transformer verification.

W	Transformer weights. Shape: $Dim\_in \times Dim\_out$
X	Input feature tensor. Shape: $Batch\_size \times Length \times Dim\_in$
X_{lb}, X_{ub}	The tensor of lower and upper bound bias of input features. Shape: $Batch\_size \times Length \times Dim\_in$
X_{lw}, X_{uw}	The tensor of lower and upper bound weights of input features. Shape: $Batch\_size \times Length \times Dim\_in \times Dim\_out$
X_l, X_u	The tensor of concretized lower and upper bounds of input features. Shape: $Batch\_size \times Length \times Dim\_in$

depends on the sign of weights $W[j,i]$. In particular, we have upper bounds $Y_{ub}[j]$ as

$$\begin{aligned} Y[j] \leq & Y_{ub}[j] + Y_{uw}[j] * \vec{\epsilon} \\ = & (\sum_{W[j,i]\geq 0} W[j,i] \cdot X_{ub}[i] + \sum_{W[j,i]<0} W[j,i] \cdot X_{lb}[i]) \\ & + (\sum_{W[j,i]\geq 0} W[j,i] \cdot X_{uw}[i] + \sum_{W[j,i]<0} W[j,i] \cdot X_{lw}[i])) * \vec{\epsilon} \end{aligned} \tag{7.2}$$

The lower bounds can be computed in a similar way. This bound computation (Equation 7.2) is significantly different from standard NN computation since it explicitly considers the sign of weights. Previous transformer verification directly exploits the standard DL frameworks to build a computation graph (Figure 7.3(c)) for computing bounds, which leads to inefficient memory access and computation overhead. We will discuss the opportunities and challenges of efficient transformer verification in subsection 7.3.3.

For the same NN layer, diverse bound computation designs may still be developed to provide tighter bounds on NN predictions. We illustrate two types of bounds for the ReLU layer in section 7.3(d)-(e) and two types of bounds for the Tanh layer in section 7.3(f)-(g). A tighter bound (*i.e.*, less space between linear bounds and ReLU function) is preferred to provide a better linear bound approximation to NN prediction. For example, consider the concretized lower bound $X_l[i]$ and upper bound $X_u[i]$ for an input neuron $X[i]$, when we have $abs(X_l[i]) > abs(X_u[i])$, linear bound in Figure 7.3(d) is preferred over the linear bound in Figure 7.3(e) since the former one provides a tighter

approximation. This diversity in bound design adds more complexity to developing frameworks for transformer verification.

7.3.2 Deep Learning Frameworks

Many DL frameworks [58, 279, 280] have been developed recently to efficiently support NN workload. Early works such as PyTorch [58] take user-specified computation graphs for neural networks and maps towards hand-tuned kernels on backend platforms (*e.g.*, GPUs). However, this approach usually builds upon kernels developed for standard NNs and cannot efficiently support transformer verification computation. Recent works, such as TVM [279] and Ansor [280], can automatically generate such backend kernels based on a set of heuristic rules on fusion and operator optimizations. However, these heuristic rules are developed specifically for standard NNs. Naively incorporating these rules into transformer verification may lead to unsatisfactory performance due to the significant difference in computing patterns. For example, Figure 7.3(c) shows the computation graph for utilizing the kernels of standard NNs on transformer verification. This approach leads to heavy sparsity and redundant memory access. In particular, only half of the elements in W_{pos} and W_{neg} are non-zero values, leading to 50% sparsity. To this end, we build Faith, the first framework for efficient transformer verification on GPUs.

7.3.3 Opportunities and Challenges

In this section, we introduce optimization opportunities and challenges in enabling efficient transformer verification.

We show the latency of verifying individual transformer operators in Figure 7.4. We profile this latency breakdown based on the state-of-the-art transformer verification implemented with PyTorch [58]. We have three major observations. First, dot product

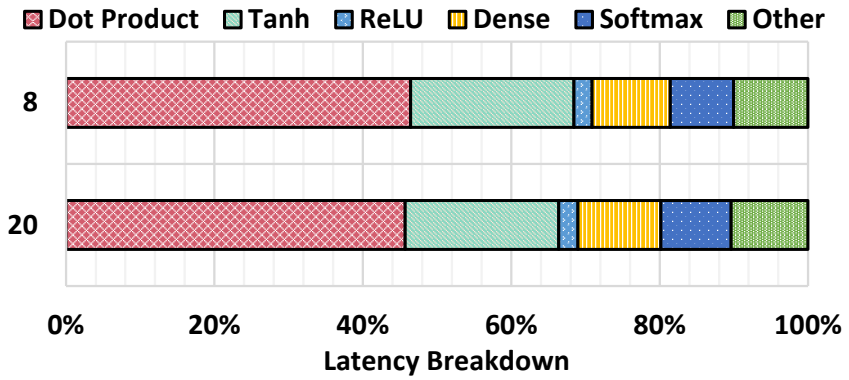


Figure 7.4: Latency breakdown of transformer verification on sentences with length 8 and 20. Here, we show the latency of verifying individual operators such as dot product and Tanh.

accounts for around 45% latency. Dot product takes two input tensors Q and K where both inputs may be perturbed during adversarial attack, which is significantly different from matrix multiplication that only one input (*i.e.*, feature X) may be perturbed. This adds complexity to the verification of dot product operators [9] and longer latency. Second, elementwise operators such as Tanh and ReLU account for a large portion of latency in transformer verification. This is significantly different from standard NNs where elementwise operators can usually be fused with remaining operators and show low latency. Third, we observe that matrix multiplication and softmax accounts for certain latency.

Opportunities: There are two major opportunities to accelerate transformer verification. The first opportunity is to exploit the semantics of transformer verification to minimize redundant memory access and computation. Our investigation shows that transformer verification has rich semantic information (*e.g.*, 50% sparsity in W_{pos} and W_{neg}), which can be exploited to accelerate transformer verification. The second opportunity is to exploit the modern GPU architectures to efficiently support diverse computing patterns in transformer verification. One example is to accelerate abundant reduction computation in Equation 7.1.

Challenges: Although these ideas sound promising, the efforts to realize the ben-

efits are non-trivial due to several challenges. First, transformer verification shows significantly different computing patterns from standard NNs. Straightforwardly borrowing optimizations for standard NNs such as kernel fusion can hardly bring similar benefits. Second, while exploiting GPU architecture supports may bring benefits, we still need specialized designs as a synergy between architecture and specialized computing patterns. Moreover, exploiting advanced GPU architecture supports will add more complexity to the search space of optimized kernels which motivates novel autotuning optimizations.

7.4 Semantic-aware Computation Graph Transformation

In this section, we propose *semantic-aware computation graph transformation* for efficient transformer verification. We first propose **semantic-aware kernel fusion** to fuse kernels within a transformer layer. It contains two novel types of fusions – *weight-paring based fusion* and *double bound based fusion*. Then, we propose **bound-aware cross-layer fusion** to efficiently fuse kernels across transformer layers.

7.4.1 Semantic-aware Kernel Fusion

The semantic-aware kernel fusion fuses operators in a single transformer layer to minimize memory access. Different from standard transformers, a single layer in transformer verification usually involves multiple kernels to compute the bounds adaptively to the sign of weights, as discussed in subsection 7.3.1. Existing transformer verification [9, 10] usually uses a set of GPU kernels developed for standard transformers to serve the need for transformer verification. We illustrate the memory access pattern of this baseline approach in Figure 7.5(a). These kernels need to independently read data from the global

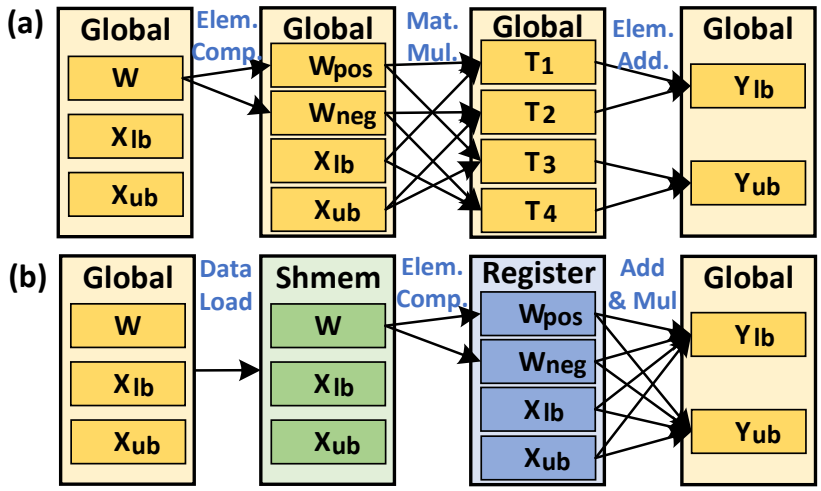


Figure 7.5: Illustration of Semantic-aware Kernel Fusion. We show the memory access pattern before and after applying semantic-aware kernel fusion in (a) and (b), respectively.

memory of GPUs and lead to heavy memory overhead. Moreover, these kernels fail to exploit semantic information in transformer verification and show heavy redundancy during memory access. For example, baseline approaches usually first split the weight matrix W into two weight matrices W_{pos} and W_{neg} according to weight signs and then use each matrix for computing lower and upper bounds. Here, these two split matrices W_{pos} and W_{neg} have the same shape of $M \times N$ as the weight matrix W. However, reading these matrices independently requires loading $2MN$ scalars, which leads to redundant memory access.

We propose semantic-aware kernel fusion to minimize such memory overhead by exploiting transformer verification semantics and GPU memory hierarchies (*i.e.*, global memory, shared memory, and registers). We illustrate our semantic-aware kernel fusion in Figure 7.5(b). Our key insight is to first load data collaboratively from global memory and only distinguish data semantics (*e.g.*, W_{pos} and W_{neg}) at the register level to mitigate redundant memory access. In particular, we identify *weight-paring based fusion* and *double bound based fusion* as the two most important semantics in transformer

verification.

Weight-pairing based fusion. We first propose weight-paring-based fusion to mitigate redundant memory access when reading W_{pos} and W_{neg}. Our key observation is that the zero values in W_{pos} are exactly the position of non-zero values in W_{neg}. Formally, we have $W_{pos} + W_{neg} = W$. To this end, instead of using an operator to split weight matrix W into W_{pos} and W_{neg}, we first load the matrix W from global memory to shared memory without distinguishing the sign of individual scalars. Then, we split the weight matrix W into W_{pos} and W_{neg} when loading data from shared memory to registers, as illustrated in Figure 7.5(b). In our design, we only need to load MN scalars from global memory, which leads to significantly reduced memory access compared with loading $2MN$ scalars in baseline approaches.

Double bound based fusion. Our second optimization is a double-bound-based fusion. One important semantics in transformer verification is to multiply the same weight matrix with lower and upper input bounds (*e.g.*, X_{lb} and X_{ub}) to compute the output bounds (*e.g.*, Y_{lb} and Y_{ub} in Figure 7.5(b)). Meanwhile, when computing the bound for output neurons, we usually need to read both lower and upper bounds for computation. For example, when computing the upper bound of output neurons, we need to read upper bound when weight is positive and read lower bound when weight is negative. Suppose the input bounds X_{lb} and X_{ub} have shape $N \times K$, we need to load $4NK$ scalars during transformer verification.

Instead, we propose to fuse the computation of lower and upper bounds such that the lower and upper bounds only need to be loaded once to save memory access. In particular, we first use threads across GPU blocks to collaboratively load tiles of input matrices from global memory to shared memory, which can be accessed by different GPU threads. Here, we use shared memory to enable data sharing across GPU threads since different threads may multiply the same input bound scalar with different weight scalars (*e.g.*,

multiplying the first row in X_{lb} and X_{ub} with various columns in W). Then, each thread loads independent data from shared memory to registers and directly accumulates output bounds Y_{lb} and Y_{ub} in registers. We note that this design further improves performance by eliminating the redundant global memory access during generating Y_{lb} and Y_{ub}.

7.4.2 Bound-aware Cross-layer Kernel Fusion

Bound-aware cross-layer kernel fusion fuses the verification of kernels across multiple transformer layers to further minimize memory access. Existing frameworks for accelerating standard NNs usually rely on a set of rules to fuse kernels. One popular example is to fuse convolution kernel with the following elementwise kernels (*e.g.*, ReLU kernel for elementwise comparison with 0). However, these rules usually cannot be applied to fuse kernels for transformer verification. For example, verifying the ReLU kernel requires first a concretization operation with a global reduction to compute the concretized bounds for a neuron and then applies different computation according to the concretized bounds (see subsection 7.3.1).

To this end, we propose a set of rules for cross-layer kernel fusion in transformer verification. In particular, we recognize three types of operators. The first type is *input-reduction-compute* that conducts reduction or concretization operation on the input data before computation. One example is verifying nonlinear activation functions such as *ReLU* and *Tanh* that requires concretized bounds to apply different computation. Another example is the *softmax* operator that computes a global summation for normalization. The second type is *strict-elementwise* that contains only elementwise computation and does not require concretization or global summation. The third type is *dense-computation* such as matrix-matrix multiplication kernels. In our cross-layer kernel fusion design, we can always fuse a *dense* operator with its following *strict-elementwise*

operator. However, we cannot fuse *dense* operator with *input-reduction-compute* due to the concretization or reduction operation. In addition, we can fuse *input-reduction-compute* with its following *strict-elementwise* operator. Finally, we can fuse multiple *strict-elementwise* operators (*e.g.*, elementwise addition and multiplication).

7.5 Verification-specialized Kernel Crafter

In this section, we propose a verification-specialized kernel crafter to efficiently map transformer verification towards modern GPUs. We exploit intrinsic properties (*e.g.*, abundant reduction operations) of transformer verification which are significantly different from standard transformer operators. One major challenge in building the kernel crafter is the large diversity in verification designs across operators (see Figure 7.3(d)-(g)). To tackle this challenge, we first propose a *verification pattern categorization* to abstract such diversity and provide a small set of computing patterns over verification of diverse operators. Then, we propose three optimizations to efficiently support these computing patterns of transformer verification.

7.5.1 Verification Pattern Categorization

While there are diverse bound designs across different operators, we characterize transformer verification into four typical computing patterns. Based on this characterization, Faith can abstract the diversity in bound designs into a combination of computing patterns and exploit optimizations towards individual computing patterns for improving performance. Similar to standard NNs, one important computing pattern is *generalized matrix multiplication (GEMM)* when verifying projection layers and fully connected layers. Matrix multiplication is the major bottleneck in standard NNs and has been well-optimized by existing DL frameworks. Besides GEMM, transformer verification

introduces three other time-consuming computing patterns, which are highlighted as follows:

The first computing pattern is *generalized vector reduction*. One typical source of generalized vector reduction is concretization that computes the norm and generates the concretized lower and upper bounds for individual neurons (see Equation 7.1). Formally, consider a matrix $X = [\vec{x}_1, \vec{x}_2, \cdots, \vec{x}_m] \in \mathbb{R}^{m \times n}$ where $\vec{x}_i = [x_{i,1}, x_{i,2}, \cdots, x_{i,n}]$ are vectors of length n. The generalized vector reduction computes an output $Y = [y_1, y_2, \cdots, y_n] \in \mathbb{R}^n$ that satisfies

$$y_i = reduction(\vec{x}_i) = \sum_{j=1}^{n} f(x_{i,j}), \quad i \in \{1, 2, \cdots, m\} \tag{7.3}$$

Here, $f(x)$ is an elementwise function that takes a scalar input and generates a scalar output. One example for $f(x)$ is x^2 when computing the L_2 norm for input vectors.

The second computing pattern is *generalized elementwise multiplication* which appears frequently when verifying elementwise operators such as ReLU and Tanh. Formally, consider a concretized lower bound $l \in \mathbb{R}^{m \times n}$ and an upper bound $u \in \mathbb{R}^{m \times n}$ where $l_{i,j}$ and $u_{i,j}$ are concretized lower and upper bounds for the neuron at position (i, j). Let $X \in \mathbb{R}^{m \times n}$ be the input values. The generalized elementwise multiplication computes an output $Y \in \mathbb{R}^{m \times n}$ that satisfies

$$y_{i,j} = f(l_{i,j}, u_{i,j}) * x_{i,j}, \quad i \in \{1, 2, \cdots m\}, j \in \{1, 2, \cdots, n\} \tag{7.4}$$

Here, transformer verification introduces a function $f(\cdot, \cdot)$ that takes the lower and upper bounds for an input neuron and computes a scaling parameter which is multiplied with the input value of this neuron. One example is the tangent line between the concretized lower and upper bounds when verifying Tanh layer, which accounts for more than 20% latency as we profiled in Figure 7.4. Another example is $f(l_{i,j}, u_{i,j}) = 1$ when verifying ReLU layer and $l_{i,j}$ is non-negative. While $f(\cdot, \cdot)$ shows large diversity across operators,

we stress that the same computing pattern is shared across these operators such that a uniform framework can be applied to improve performance.

The third computing pattern is *generalized scalar-vector multiplication*. This computing pattern exists widely when verifying dot products in the self-attention layer of transformers. This computing pattern accounts for more than 40% latency in transformer verification, as discussed in Figure 7.4. Formally, consider a vector $S = [s_1, s_2, \cdots, s_m] \in \mathbb{R}^m$ and a matrix $X = [\vec{x}_1, \vec{x}_2, \cdots, \vec{x}_m] \in \mathbb{R}^{m \times n}$, where s_i are scalars and $\vec{x}_i = [x_{i,1}, x_{i,2}, \cdots, x_{i,n}]$ are vectors of length n. The generalized scalar-vector multiplication computes an output $Y = [\vec{y}_1, \vec{y}_2, \cdots, \vec{y}_n] \in \mathbb{R}^{n \times n}$ that satisfies

$$\vec{y}_i = f(s_i) * \vec{x}_i = [f(s_i) * x_{i,1}, f(s_i) * x_{i,2}, \cdots, f(s_i) * x_{i,n}], \quad i \in \{1, 2, \cdots, m\} \tag{7.5}$$

Here, $f(\cdot)$ is a function that takes a scalar input and generates a scalar output.

In the following sections, we first demonstrate a *workload-adaptive reduction* to improve the performance of generalized vector reduction (Equation 7.3). We then propose a *sharing-oriented workload scheduling* to improve the performance of generalized element-wise multiplication (Equation 7.4). Finally, we demonstrate *broadcast-aware super threading* to efficiently support the generalized scalar-vector multiplication (Equation 7.5).

7.5.2 Workload-adaptive Reduction

Transformer verification contains abundant reduction operations where a sequence of scalars are summed up into one scalar. One common reduction operation is the concretization operation that computes the concretized lower and upper bounds for individual neurons, as discussed in section 7.3. Another common reduction operation is the softmax operation that is applied in each self-attention layer for measuring the relationship between individual words. These reduction operations pose challenges between

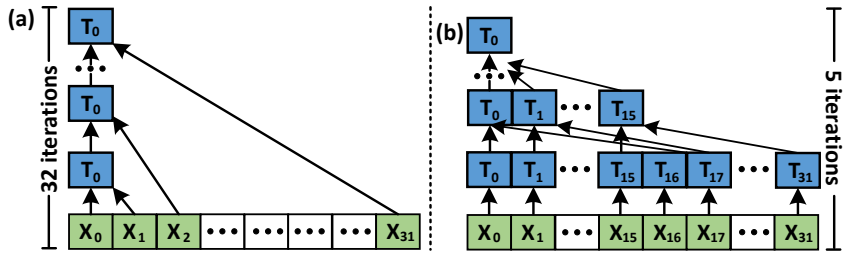


Figure 7.6: Illustration of Workload-adaptive Reduction. (a) Sequential Mode; (b) Parallel Mode. Here, x_i and T_i are the i-th data and thread, respectively.

parallelism and data locality. One baseline approach is to use a single thread to read and accumulate a sequence of scalars as illustrated in Figure 7.6(a). However, this approach usually leads to low parallelism and fails to exploit abundant threads in GPUs. For example, we need 32 iterations to accumulate 32 scalars. Another baseline approach is to first split this sequence of scalars into multiple chunks and allocate one thread to each chunk for accumulation. Then, each thread writes the accumulated results for each chunk to global memory and uses an additional thread to finally accumulate the sum of each chunk. While this approach improves parallelism, it requires expensive global memory access and high overhead.

Workload-adaptive Reduction with length $n = 32$. We propose a *workload-adaptive reduction* to fully exploit GPU memory hierarchies and the inter-register communication functionalities. We illustrate our design in Figure 7.6(b). Our design achieves high parallelism by enabling multiple threads for reduction simultaneously. Meanwhile, we avoid the expensive data communication through global memory and exploit only efficient registers. In particular, we use 32 threads (*i.e.*, a warp) to read these 32 scalars simultaneously from global memory. Considering these 32 scalars are consecutive in global memory, we can efficiently load them with 32 threads through coalesced memory access. Then, we exploit the specialized instruction `_shfl_down_sync` to directly communicate data in registers across individual threads. As illustrated in the parallel mode of

Figure 7.6(b), our design involves only five iterations of cross-thread data communication to generate the final accumulated result, rather than the 32 iterations in the sequential mode of Figure 7.6(a).

Workload-adaptive Reduction with Arbitrary Length n. For an arbitrary length n, one naive approach is to repeatedly use 32 threads to reduce 32 scalars and then use 1 thread to accumulate the final results. However, this approach may lead to unnecessary communication across threads. Suppose we are accumulating a vector of length $n = 32k$, we need 5 iterations for reducing every 32 scalars, leading to $5k$ iterations in total for accumulating the vector. Instead, we propose a *hybrid mode* to minimize the number of iterations while still achieving high parallelism. In particular, we first split the input sequence into chunks where each chunk contains 32 scalars. Then, we use 32 threads to read one chunk simultaneously from global memory and accumulate individual chunks iteratively. For example, the 1-st thread accumulates the 1-st scalar in each chunk. Here, the accumulation is conducted in registers and does not require communication across threads. Finally, we apply a single 5-iteration reduction across 32 threads. In total, our design has only $k + 5$ iterations which are significantly less than $6k$ iterations in the naive approach.

7.5.3 Sharing-oriented Workload Scheduling

We propose *sharing-oriented workload scheduling* to efficiently verify elementwise operators. Different from standard transformers, verifying elementwise operators, especially non-linear ones (*e.g.*, ReLU and Tanh), accounts for a large portion of latency in transformer verification as we discussed in Figure 7.4. Verifying these operators usually first requires computing a concretized lower bound X_l and upper bound X_u for each input neuron and then computes the bounds for the output neuron. Different signs of concretized

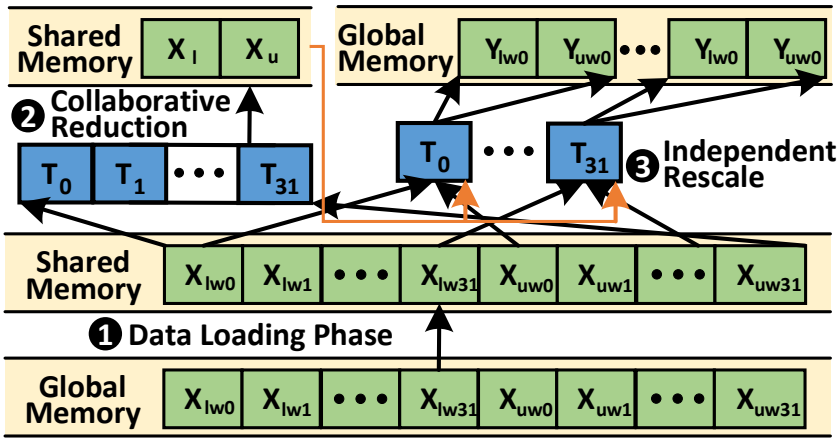


Figure 7.7: Illustration of sharing-oriented workload scheduling

input bounds usually lead to different computations for output bounds, which could easily lead to warp divergence and unsatisfactory performance. Moreover, when computing the output bound weights (*i.e.*, Y_{lw} and Y_{uw}) for a neuron, we need to repeatedly use the same input bounds which leads to extra memory overhead.

To efficiently verify elementwise operators, we propose *sharing oriented workload scheduling* to minimize memory access and improve performance. Our key observation is that the same set of input bound weights X_{lw} and X_{uw} are used to compute the concretized input bounds X_l and X_u, while these input weights are also used for computing the output bound weights Y_{lw} and Y_{uw}. Instead of repeatedly loading X_{lw} and X_{uw}, we can exploit the GPU memory hierarchies to cache X_{lw} and X_{uw} and minimize the global memory access to improve the overall performance.

As illustrated in Figure 7.7, we use a set of $T(=32)$ threads to first (Step ❶) load input bound weights X_{lw} and X_{uw} from global memory to shared memory. Here, T is a hyper-parameter to balance the parallelism and compute intensity, which will be selected in section 7.6. Then (Step ❷), these T threads load input bound weights from shared memory and collaboratively compute the concretized lower and upper bounds X_l and X_u,

following our design in subsection 7.5.2. These concretized lower and upper bounds are stored in shared memory which can be accessed by individual threads. Finally (Step ❸), each thread independently loads individual X_{lw} and X_{uw} scalars from shared memory and rescales according to the concretized bounds X_l and X_u. Here, all threads in a warp are computing the output bound weights for the same neuron and the concretized input bounds are the same across threads in a warp. Thus, all threads in a warp can apply the same rescaling computation and avoid warp divergence. We also note that input bound weights are only loaded once from global memory which mitigates redundant global memory access.

7.5.4 Broadcast-aware Super Threading

We propose *broadcast-aware super threading* to efficiently support generalized scalar-vector multiplication, as discussed in Equation 7.5. One naive approach is to use one thread to read a scalar s_i and a vector $\vec{x}_i$ and computes the generalized scalar vector multiplication $f(s_i)\vec{x}_i$. However, this approach fails to exploit the parallelism opportunities in generalized scalar vector multiplication. Another approach is to split the vector $\vec{x}_i$ into multiple chunks and use one thread for each chunk. However, this approach requires threads to repeatedly read the same scalar s_i from global memory and shows redundant memory access.

Instead, we propose a broadcast-aware super threading to achieve high parallelism while minimizing memory access. We consider two types of super threading for generalized scalar vector multiplication. The first type is a group of 32 threads (*i.e.*, a warp for one vector). When using 32 threads to compute the multiplication between a scalar s_i and a vector $\vec{x}_i$, these 32 threads can read the scalar s_i once, broadcast across threads with modern GPU memory, and compute $f(s_i)$ simultaneously. Based on this broadcast,

we can mitigate the redundant memory access that each thread repeatedly read the same scalar s_i. The second type is a group of $32t$ threads (*i.e.*, t warps for one vector). In this case, we use one warp to read the scalar s_i and use shared memory to broadcast s_i across warps.

7.6 Expert-guided Autotuning Optimization

Considering the large design space of optimization towards GPUs, one natural question arises: *Can we effectively incorporate hardware knowledge to find optimal operator implementation?* Existing works such as TVM [279] and Ansor [280] usually autotune operator implementations in a hardware-agnostic way. In particular, these works extract implementation-specific parameters such as tiling size and use a cost model to implicitly learn the relationship between these parameters and performance. However, there are two drawbacks in this hardware-agnostic approach. First, there is a complex interaction between implementation and the hardware properties, which could be hard to be implicitly learned by the cost model. For example, existing works [310, 311, 312, 313] on hand-tuning large matrix-matrix multiplication operators usually maximize the number of registers in use to improve cache performance. However, this optimization is also limited by the number of registers for each GPU thread since exceeding such limitation may lead to register spilling [314] and a significant performance drop. A careful reasoning on the interaction between the implementation-specific parameters (*e.g.*, the number of registers for caching data) and the hardware properties (*e.g.*, the number of registers per thread) is usually necessary to maximize the performance. To tackle this challenge, we propose an *expert-guided autotuning optimization* to automatically reason both implementation-specific parameters and hardware properties. In particular, we have the following two designs.

Rule-based Expert Knowledge Metafile. We propose a *rule-based expert knowledge metafile* to capture hardware properties. This metafile only needs to be set once for each type of GPUs and requires limited manual efforts. In particular, we consider two types of rules. The first type is *hard rules* which represents hardware limitation such as the maximal shared memory size and the maximal number of registers per thread. Violating these rules may lead to significant performance drop such as register spilling. The second type is *soft rules* which represents intrinsic trade-offs related to the hardware properties such as the number of streaming multiprocessors (SM) and the number of threads per SM. One typical design choice is the number of threads per block which will be mapped to threads on the same SM. Allocating more threads per block usually leads to better parallelism for the sub-task assigned to a block. However, allocating more threads per block may also hinder executing multiple blocks on the same GPU SM hardware and lead to worse overall parallelism.

Expert-guided Cost Model. We propose an *expert-guided cost model* to automatically tackle the complex interaction between implementation-specific parameters and hardware properties. Given a set of candidate operator implementations, we have two phases to select the optimal implementation. In the first phase, we generate an estimation of shared memory and register usage for each operator implementation. We compare the estimated usage with the hard rules and rule out operator implementations that violate hard rules. In the second phase, we utilize a regression tree based cost model to automatically explore remaining candidate implementations and identify optimal implementations. In particular, we feed both the implementation-specific parameters (*e.g.*, tiling sizes) and the hardware properties to the regression tree and predict the top-k candidate implementations. We profile these top-k candidate implementations on GPUs and use the profiling to finetune our cost model. We repeat this procedure until the performance of top-k candidates becomes stable.

Table 7.2: Dataset statistics

Dataset	#Train	#Val	#Test	Length		
				min	mean	max
SST	67,349	872	1,821	4	25	62
YELP	560,000	0	38,000	5	98	128

7.7 Evaluation

In this section, we comprehensively evaluate Faith over various datasets and GPU backends. We first present our experiment setup in subsection 7.7.1. Then, we show the overall speedup on end-to-end transformer verification in subsection 7.7.2. Finally, we provide more optimization analysis on individual transformer layers in subsection 7.7.3.

7.7.1 Experiment Setup

Baselines. We compare Faith with the state-of-the-art transformer verification [9] based on PyTorch. We further compare with TVM [279] and Ansor [280], as stronger baselines. TVM and Ansor are two state-of-the-art deep learning compilers for standard neural networks. We feed the pytorch model into TVM and Ansor through relay frontend [315] which will automatically optimize transformer verification performance. While TVM and Ansor take minutes to compile an operator implementation, we do not incorporate this compilation latency and record only inference latency for a fair comparison.

Datasets. We evaluate two popular datasets, Yelp [316] and SST [317], following the setting in state-of-the-art transformer verification [9]. These two datasets are widely used in the natural language processing for analyzing sentiment in languages. We summarize the statistics of these two datasets in Table 7.2. SST dataset contains 67,349 training sentences, 872 validation sentences, and 1,821 testing sentences. In SST dataset, there are 4 to 62 tokens in each sentence and the average number of tokens in a sentence is 25. YELP dataset contains 560,000 sentences as training data and 38,000 sentences as

testing data. In YELP dataset, there are 5 to 128 tokens in each sentence and the average number of tokens in a sentence is 98.

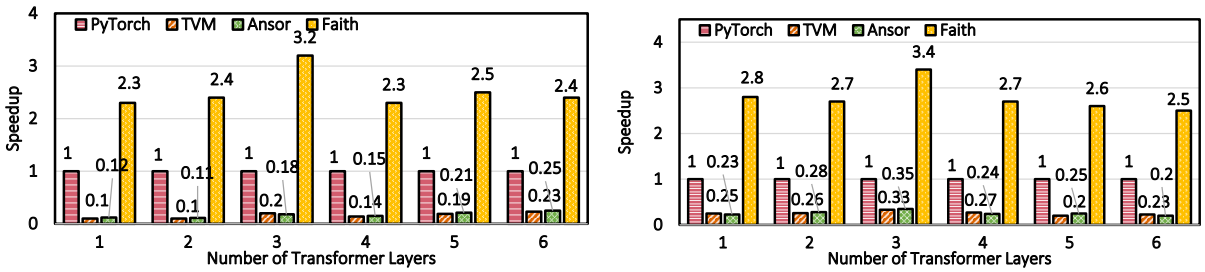
Transformer Networks. We evaluate Faith on transformer networks with 1 to 6 layers to demonstrate the performance on large models. Following popular transformer settings, each transformer layer has 4 attention heads and an embedding size of 128. Furthermore, we study the Faith performance under diverse embedding sizes in subsection 7.7.3.

Experiment Configuration. We evaluate with an NVIDIA A100 GPU and an NVIDIA V100 GPU to show Faith performance on various GPU backends. The host server with A100 GPUs is an AMD EPYC 7742 64-Core Processor and runs Ubuntu 20.04 with CUDA 11.3. The host server with V100 GPUs has 32 cores of Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz and runs Ubuntu 16.04 with CUDA 10.1.

7.7.2 Overall Performance

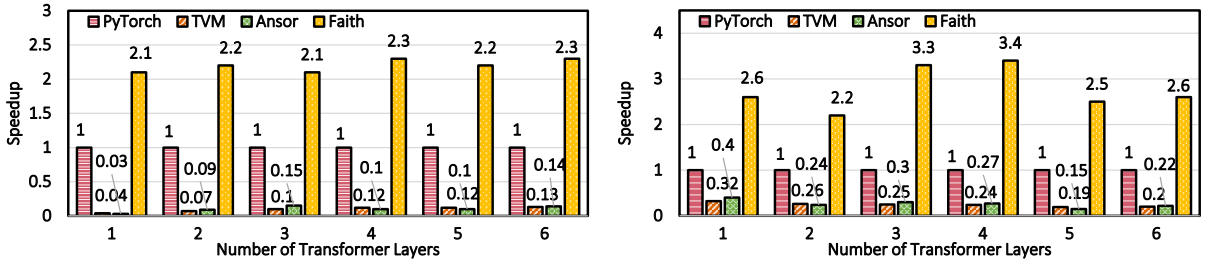
We show the overall speedup on SST dataset and Yelp dataset in Figure 7.8 and Figure 7.9, respectively. We show the performance improvement over transformers with diverse numbers of layers from 1 to 6, which covers popular settings in the natural language processing domain. While the length of input sentences may have an impact on the performance improvement, we show the averaged speedup over all testing sentences in this section and study the impact of sentence length in subsection 7.7.3. We compare Faith with the PyTorch baseline following existing transformer verification open-source implementations [9]. We further compare Faith with two state-of-the-art deep learning frameworks (*i.e.*, TVM and Ansor) to provide a comprehensive comparison, as we discussed in subsection 7.7.1.

We show the overall speedup on SST dataset and A100 GPU in Figure 7.8(a). Com-



(a) On A100 GPU. (b) On V100 GPU.

Figure 7.8: Overall speedup on SST dataset.



(a) On A100 GPU. (b) On V100 GPU.

Figure 7.9: Overall speedup on Yelp dataset.

pared with PyTorch, we observe 2.3× to 3.2× speedup (2.5× on average). We contribute this performance improvement to our semantic-aware computation graph transformation (section 7.4) and verification-specialized kernel crafter (section 7.5). We further observe 17.2× and 15.9× speedup over TVM and Ansor, respectively. The main reason is that TVM and Ansor focus on optimizing standard neural networks and fail to efficiently support verification-specific computing patterns, as discussed in subsection 7.3.2. While Faith and these three baselines show different performance, we stress that the same verification bounds are generated, and the only difference resides in system optimizations. Comparing across different numbers of transformer layers from 1 to 6, the performance improvement remains similar around 2.5×. This result shows that Faith can efficiently support transformer verification with diverse numbers of transformer layers. We show the overall speedup on SST dataset and V100 GPU in Figure 7.8(b). We have similar

observation about the results on A100 GPU which shows that Faith can effectively adapt to diverse GPU backends, thanks to expert-guided autotuning optimization (section 7.6).

We show overall speedup on Yelp dataset and A100 GPU in Figure 7.9(a). Sentences in YELP dataset has 5 to 128 tokens (98 on average), which is longer than sentences in SST dataset with 4 to 62 tokens (25 on average). This provides an opportunity to show Faith performance on long sentences. Overall, we observe 2.1× to 2.3× speedup (2.2× on average) when comparing with the PyTorch baseline. We also observe 26.7× and 28.3× speedup on average over TVM and Ansor, respectively. This speedup is similar to the performance improvement on SST dataset and shows the good generality of Faith over diverse input data. We also have similar observations on Yelp dataset and V100 GPU in Figure 7.9(b).

7.7.3 Optimization Analysis

In this section, we show speedup from individual Faith optimizations. We first show speedup on *verification of matrix multiplication* over the diverse lengths and diverse embedding sizes. Verification of matrix multiplication plays an important role in verifying projection layers and fully connected layers in transformers. Then, we show the benefits on verification of ReLU, verification of dot product, and verification of Tanh, which in total accounts for around 70% latency in transformer verification. Since we observe similar performance on A100 GPU and V100 GPU, we focus on A100 GPU and omit results on V100 GPU in this section due to page limits.

Performance benefits on verification of matrix multiplication. We show speedup on verification of matrix multiplication over the diverse lengths in Figure 7.10. We study the speedup over diverse lengths from 2 to 128, following the setting in the popular natural language processing datasets as summarized in Table 7.2. Overall, we

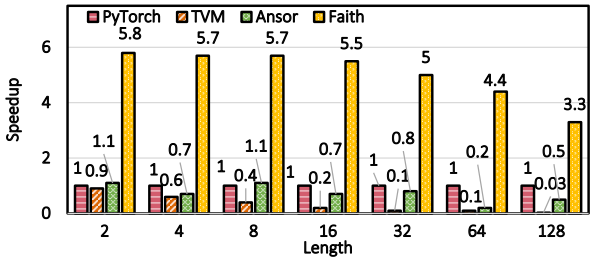


Figure 7.10: Speedup on verification of matrix multiplication over the diverse lengths. Embedding Size: 128.

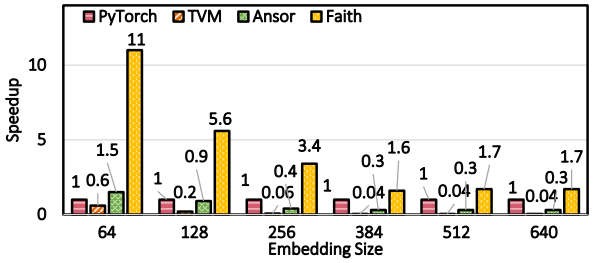


Figure 7.11: Speedup on verification of matrix multiplication over the diverse embedding sizes. Length: 16.

observe 5.1× speedup on average over the PyTorch baseline. This result shows significant performance benefits from utilizing Faith on accelerating transformer verification. Comparing across lengths, we observe a higher speedup of 5.54× over the PyTorch baseline on shorter sentences with 2 to 32 words. The reason is that our autotuning optimization (section 7.6) automatically adjusts the number of threads and memory layout to improve the parallelism. We achieve a smaller speedup of 3.85× on longer sentences with 64 and 128 words. For these longer sentences, we have achieved high occupancy on GPUs and the speedup is limited by the hardware capability.

Surprisingly, we observe that TVM and Ansor achieve 0.33× and 0.73× speedup, which is significantly slower than PyTorch baselines on verification of matrix multiplication. The main reason is that TVM and Ansor focus on accelerating standard NNs and cannot efficiently support computing patterns in the verification of matrix multiplication (Figure 7.3(c)). Instead, Faith exploits a semantic-aware kernel fusion (subsection 7.4.1) to efficiently support such computing patterns in verification.

We show speedup on verification of matrix multiplication over the diverse embedding sizes in Figure 7.11. We study embedding size from 64 to 640 following popular transformer settings. We note that transformer in natural language processing usually adopts a relatively small embedding size (*e.g.*, 64 to 256), which is different from con-

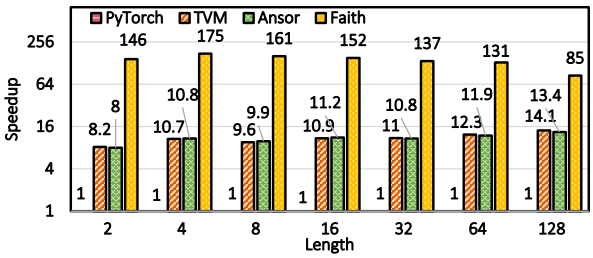


Figure 7.12: Speedup on verification of ReLU over the diverse lengths. Embedding Size: 128.

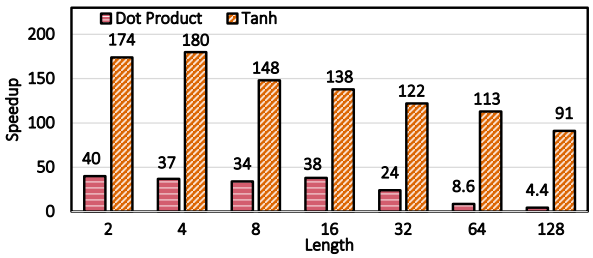


Figure 7.13: Speedup on verification of Tanh and dot product over the diverse lengths. Embedding Size: 128.

volutional neural networks in computer vision that adopts a large embedding size (*e.g.*, 1024). Overall, Faith achieves 4.2× speedup on average over the PyTorch baseline. This result shows that Faith can improve performance over diverse embedding sizes. We also observe that Faith achieves larger speedup for smaller embedding sizes, which is similar to the case when verifying matrix multiplication over diverse lengths.

Performance benefits on verification of ReLU. We show speedup on verification of ReLU over diverse lengths in Figure 7.12. As we discussed earlier in subsection 7.5.1, verification of ReLU represents an important computing pattern of verifying elementwise operators. Due to similar behaviors between diverse lengths and embedding sizes, we focus on verification over diverse lengths and keep embedding size as 128, which is a popular setting in transformers. Overall, Faith achieves 141× speedup over PyTorch baseline. This large speedup shows it promising to accelerate verification of elementwise operators. Besides, Faith achieves 13.4× and 13.5× speedup over TVM and Ansor. The reason is that our *workload-adaptive reduction* (subsection 7.5.2) can significantly improve parallelism during reduction and *sharing-oriented workload sharing* can minimize memory access with GPU memory hierarchy.

Performance benefits on verifying Tanh and dot product layers. We show the speedup from Faith over the PyTorch baseline on verification of Tanh and verification

of dot product in Figure 7.13. We skip the results of TVM and Ansor since these two frameworks do not support computing patterns in verification of Tanh and verification of dot product. Here, we show results of verification of Tanh since it is a popular elementwise operator in transformer verification. We also show results of verification of dot product since it accounts for around 45% latency in transformer verification. Overall, we observe that Faith achieves 138× speedup on average for verification of Tanh. This result is similar to the performance improvement for verification of ReLU, since both Tanh and ReLU are elementwise operators and share benefits from the same set of optimizations. We also observe that Faith achieves 26.5× speedup on average for verification of dot product. This result shows the performance benefits from semantic-aware kernel fusion (subsection 7.4.1) and broadcast-aware super threading (subsection 7.5.4) that mitigate redundant memory access.

Chapter 8

Conclusions and Future Work

In this chapter, I summarize the dissertation and discuss future directions.

8.1 Conclusions

In thesis, I present my research on building systems for efficient big data analytics. We first demonstrate hardware-aware kernel tuning in chapter 2 and chapter 3 on generalizing limited hardware compute primitives to efficiently support workloads with diverse precision requirements. Then, we present a runtime system in chapter 4 that automatically identifies and exploits runtime information in big data analytics to reduce latency and energy consumption. Finally, we discuss three secure deep learning frameworks in chapter 5, chapter 6, and chapter 7 that efficiently support diverse NN operators with specialized computing patterns and mitigate manual efforts.

Overall, this study shows that APNN-TC (chapter 2) can accelerate arbitrary-precision neural networks on Ampere GPU Tensor Cores. Specifically, APNN-TC contains an int1-based emulation design on Tensor Cores to enable arbitrary-precision computation, an efficient AP-Layer design for efficiently mapping NN layers towards Tensor Cores, and an

APNN design to minimize the memory access across NN layers. Extensive evaluations on two Ampere GPUs show that APNN-TC can achieve significant speedup over CUTLASS kernels and various mainstream NN models, such as ResNet and VGG.

We also show that EGEMM-TC (chapter 3) can accelerate general-purpose scientific computing on Tensor Cores with extended-precision. Specifically, EGEMM-TC contains a lightweight emulation algorithm on Tensor Cores to achieve the extended-precision computation, a set of Tensor Core kernel optimizations to efficiently map these workloads to Tensor Cores, and a hardware-aware analytic model to facilitate the selection of performance-related hyper-parameters. Overall, EGEMM-TC achieves 3.13× and 11.18× speedup on average over the single-precision kernels on CUDA Cores from cuBLAS and CUDA-SDK, respectively. EGEMM-TC also achieves 1.8× speedup on a set of popular GEMM-based scientific computing workloads and diverse input sizes.

We further show that exploiting runtime information can significantly accelerate big data analytics. We present Palleon (chapter 4), a runtime system for efficient video processing, by detecting and exploiting class skews in video streams. We propose ABLE to detect class skews in video streams. Based on these detected class skews, Palleon uses Bayesian Filter for online model adaptation and Separability-Aware Model Selection to select the most energy efficient model during runtime. Evaluations on both synthesized videos and real videos demonstrate that Palleon achieves up to 6.7× energy saving and up to 7.9× latency reduction. We conclude that Palleon is a highly practical and effective approach for efficiently processing video streams.

Then, we present our secure deep learning frameworks, ZEN (chapter 5) and ZENO (chapter 6), that efficiently efficiently support zero-knowledge neural networks with specialized computing patterns. In chapter 5, we focus on reducing the theoretical constraint size and present ZEN as an optimizing compiler that effectively map deep learning workloads to zero-knowledge proof security schemes. In particular, ZEN takes an existing

neural network as the input and generates a privacy-preserving verifiable neural network scheme. To improve efficiency and minimize accuracy loss, we propose a zkSNARK friendly quantization and a novel encoding scheme, namely stranded encoding. Our evaluation demonstrates 5.43 $\sim$ 22.19$\times$ (15.35$\times$ on average) savings in the number of constraints compared with a vanilla implementation of neural network on zkSNARK.

We present ZENO (chapter 6) to realize the latency reduction in practice. Specifically, we design a set of ZENO language constructs to maintain high-level semantics and type information while accommodating a more aggressive compilation from a zkSNARK NN to a gate-level circuit. We then propose several privacy-type driven and tensor-type driven optimizations to further optimize the generated zk-SNARK circuit. Finally, we propose NN-centric system optimizations to further accelerate zkSNARK NNs. Extensive experimental results show that ZENO outperforms the state-of-the-art zkSNARK framework across diverse applications.

Finally, we propose a Faith framework (chapter 7) for efficient transformer verification. Specifically, we first design a set of semantic-aware computation graph transformations to fully exploit fusion opportunities in transformer verification at the computation graph level. Then, we propose a verifier-specialized kernel crafter to efficiently map fused verification kernels towards modern GPUs with minimized memory overhead and improved parallelism. Finally, we propose an expert-guided autotuning to dynamically optimize kernels according to the transformer verification workload and GPU backend characteristics. Comprehensive experimental evaluation shows that Faith significantly improves the performance of transformer verification over state-of-the-art frameworks.

In addition to above papers, my research also leads to QGTC [318], DSXplore [44], STPAcc [319], TiAcc [320], MPInfer [321] and KPynq [322] for hardware-aware kernel tuning, GNNAdvisor [323] for efficient deep learning frameworks, UAG [324], SAGA [325], and SAG [326] for adversarial attacks, SGQuant [327], 3DRF [328], and [329] for

efficient deep learning algorithms, and deep learning applications in manufacture [330] and bioinformatics [331].

8.2 Future Work

System optimizations are the key to efficiently support diverse big data analytics in terms of precision, latency, and energy consumption. In my future work, I plan to contribute more to the development of systems for big data analytics and provide practical solutions to contemporary usages. I would like to highlight two important research directions where building systems for big data analytics worth exploring and can potentially bring large real-world impact.

Hardware-aware deep learning compilers. Recently, many deep learning compilers have been built to automatically generate efficient implementations given the mathematical expression. However, existing deep learning compilers usually still require heavy manual efforts on specifying a sequence of implementation optimizations. Moreover, existing deep learning compilers usually provide only limited supports and cannot fully exploit hardware properties (*e.g.*, PTX features on NVIDIA GPUs) to squeeze the performance. We envision that hardware-aware deep learning compilers can fill this gap between mathematical expression and hardware backend by building a hollistic hardware analytic model and incorporating an enhanced hardware-aware autotuning design.

Secure deep learning framework on GPUs. Over the recent few years, the security and privacy concerns of widely deployed deep neural networks draw significant attention from both academic and industry. However, these secure deep learning workload usually cannot fully benefit from GPUs due to the significantly different computing patterns. Existing works usually either utilize GPU frameworks developed for standard neural

network workloads or rely on CPU-based approaches. We envision that building the next generation secure deep learning framework on GPUs can significantly accelerate secure deep learning workloads and make secure deep learning practical.

Bibliography

[1] K. He, X. Zhang, S. Ren, and J. Sun, *Delving deep into rectifiers: Surpassing human-level performance on imagenet classification*, in *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

[2] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, in *CVPR*, pp. 770–778, IEEE Computer Society, 2016.

[3] M. Rangaiah, "How waymo using ai autonomous driving." `https://www.analyticssteps.com/blogs/how-waymo-using-ai-autonomous-driving`.

[4] TuSimple, "Self-driving technology designed for trucks." `https://www.tusimple.com/technology/`.

[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, *Attention is all you need*, in *NIPS*, pp. 5998–6008, 2017.

[6] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, *Language models are few-shot learners*, *CoRR* **abs/2005.14165** (2020).

[7] Facebook, "How facebook uses super-efficient ai models to detect hate speech." `https://ai.facebook.com/blog/how-facebook-uses-super\/-efficient-ai-models-to-detect-hate-speech/`.

[8] S. Garg, T. Vu, and A. Moschitti, "Tanda: Transfer and adapt pre-trained transformer models for answer sentence selection." `https://www.amazon.science/publications/tanda-transfer-and-adapt-pre-trained-\/transformer-models-for-answer-\/sentence-selection`.

[9] Z. Shi, H. Zhang, K. Chang, M. Huang, and C. Hsieh, *Robustness verification for transformers*, in *ICLR*, OpenReview.net, 2020.

[10] K. Xu, Z. Shi, H. Zhang, Y. Wang, K. Chang, M. Huang, B. Kailkhura, X. Lin, and C. Hsieh, *Automatic perturbation analysis for scalable certified robustness and beyond*, in *NeurIPS*, 2020.

[11] G. Katz, C. W. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, *Reluplex: An efficient SMT solver for verifying deep neural networks*, in *CAV (1)*, vol. 10426 of *Lecture Notes in Computer Science*, pp. 97–117, Springer, 2017.

[12] P. Mohassel and P. Rindal, *Aby^3: A mixed protocol framework for machine learning*, in *CCS*, pp. 35–52, ACM, 2018.

[13] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. E. Lauter, M. Naehrig, and J. Wernsing, *Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy*, in *ICML*, vol. 48 of *JMLR Workshop and Conference Proceedings*, pp. 201–210, JMLR.org, 2016.

[14] "iphone x specification." https://www.gsmarena.com/apple_iphone_x-8858.php. Accessed: 2019-12-3.

[15] E. T. Barron and R. M. Glorioso, *A micro controlled peripheral processor*, in *Conference Record of the 6th Annual Workshop on Microprogramming*, MICRO 6, (New York, NY, USA), p. 122–128, Association for Computing Machinery, 1973.

[16] D. Steinkrau, P. Y. Simard, and I. Buck, *Using gpus for machine learning algorithms*, in *ICDAR*, pp. 1115–1119, IEEE Computer Society, 2005.

[17] K. Chellapilla, S. Puri, and P. Simard, *High Performance Convolutional Neural Networks for Document Processing*, in *Tenth International Workshop on Frontiers in Handwriting Recognition* (G. Lorette, ed.), (La Baule (France)), Université de Rennes 1, Suvisoft, Oct., 2006. http://www.suvisoft.com.

[18] R. Raina, A. Madhavan, and A. Y. Ng, *Large-scale deep unsupervised learning using graphics processors*, in *ICML*, vol. 382 of *ACM International Conference Proceeding Series*, pp. 873–880, ACM, 2009.

[19] D. C. Ciresan, U. Meier, and J. Schmidhuber, *Multi-column deep neural networks for image classification*, in *CVPR*, pp. 3642–3649, IEEE Computer Society, 2012.

[20] NVIDIA, "Tensor cores." `https://www.nvidia.com/en-us/data-center/tensor-cores/`.

[21] NVIDIA, "Jetson nano." `https://developer.nvidia.com/embedded/jetson-nano`.

[22] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou, *Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients*, *CoRR* **abs/1606.06160** (2016).

[23] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, *HAQ: hardware-aware automated quantization with mixed precision*, in *CVPR*, pp. 8612–8620, Computer Vision Foundation / IEEE, 2019.

[24] D. Zhang, J. Yang, D. Ye, and G. Hua, *Lq-nets: Learned quantization for highly accurate and compact deep neural networks*, in *Proceedings of the European conference on computer vision (ECCV)*, pp. 365–382, 2018.

[25] G. Gupta, "Shapely bell curve: Nvidia volta tensor core gpus power 5 of 6 gordon bell finalists." `https://blogs.nvidia.com/blog/2018/09/17/nvidia-volta-tensor-core-gpus-gordon-bell-finalists/`.

[26] PyTorch, "Cuda semantics." `https://pytorch.org/docs/stable/notes/cuda.html`.

[27] B. Feng, Y. Wang, T. Geng, A. Li, and Y. Ding, *Apnn-tc: Accelerating arbitrary precision neural networks on ampere gpu tensor cores*, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, (New York, NY, USA), Association for Computing Machinery, 2021.

[28] B. Feng, Y. Wang, G. Chen, W. Zhang, Y. Xie, and Y. Ding, *EGEMM-TC: accelerating scientific computing on tensor cores with extended precision*, in *PPoPP*, pp. 278–291, ACM, 2021.

[29] NVIDIA, "Programming tensor cores in cuda 9." https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/, 2017.

[30] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. G. Howard, H. Adam, and D. Kalenichenko, *Quantization and training of neural networks for efficient integer-arithmetic-only inference*, in *CVPR*, pp. 2704–2713, 2018.

[31] R. Banner, Y. Nahshan, and D. Soudry, *Post training 4-bit quantization of convolutional networks for rapid-deployment*, in *Advances in Neural Information Processing Systems* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), vol. 32, (Vancouver), pp. 7950–7958, Curran Associates, Inc., 2019.

[32] F. D. Dinechin and G. Villard, *High precision numerical accuracy in physics research*, *Nuclear Instruments and Methods in Physics Research Section A-accelerators Spectrometers Detectors and Associated Equipment* **559** (2006) 207–210.

[33] D. H. Bailey, *High-precision floating-point arithmetic in scientific computation*, *Computing in Science Engineering* **7** (2005), no. 3 54–61.

[34] B. Feng, Y. Wang, G. Li, Y. Xie, and Y. Ding, *Palleon: A runtime system for efficient video processing toward dynamic class skew*, in *USENIX Annual Technical Conference*, pp. 427–441, USENIX Association, 2021.

[35] B. Feng, L. Qin, Z. Zhang, Y. Ding, and S. Chu, *ZEN: efficient zero-knowledge proofs for neural networks*, *IACR Cryptol. ePrint Arch.* (2021) 87.

[36] G. Bonaert, D. I. Dimitrov, M. Baader, and M. T. Vechev, *Fast and precise certification of transformers*, in *PLDI*, pp. 466–481, ACM, 2021.

[37] N. Liu, X. Ma, Z. Xu, Y. Wang, J. Tang, and J. Ye, *Autocompress: An automatic dnn structured pruning framework for ultra-high compression rates*, in *Proceedings of the AAAI Conference on Artificial Intelligence*, pp. 4876–4883, 2020.

[38] W. Niu, P. Zhao, Z. Zhan, X. Lin, Y. Wang, and B. Ren, *Towards real-time dnn inference on mobile platforms with model pruning and compiler optimization*, *IJCAI* (2020).

[39] X. Ma, F.-M. Guo, W. Niu, X. Lin, J. Tang, K. Ma, B. Ren, and Y. Wang, *Pconv: The missing but desirable sparsity in dnn weight pruning for real-time execution on mobile devices*, in *Proceedings of the AAAI Conference on Artificial Intelligence*, pp. 5117–5124, 2020.

[40] L. Liu, L. Deng, Z. Chen, Y. Wang, S. Li, J. Zhang, Y. Yang, Z. Gu, Y. Ding, and Y. Xie., *Boosting deep neural network efficiency with dual-module inference.*, in *International Conference on Machine Learning. (ICML'20)*, 2020.

[41] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*, *arXiv e-prints* (Apr., 2017).

[42] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, *Mobilenetv2: Inverted residuals and linear bottlenecks*, in *CVPR*, 2018.

[43] F. Chollet, *Xception: Deep learning with depthwise separable convolutions*, in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2017.

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

[45] B. Zhuang, L. Liu, M. Tan, C. Shen, and I. Reid, *Training quantized neural networks with a full-precision auxiliary module*, in *CVPR'20*, 2020.

[46] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, *Quantized convolutional neural networks for mobile devices*, in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4820–4828, 2016.

[47] Z. Yang, Y. Wang, K. Han, C. Xu, C. Xu, D. Tao, and C. Xu, *Searching for low-bit weights in quantized neural networks*, in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual* (H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, eds.), 2020.

[48] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et. al.*, *In-datacenter performance analysis of a tensor processing unit*, in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12, ACM, 2017.

[49] B. Hickmann, J. Chen, M. Rotzin, A. Yang, M. Urbanski, and S. Avancha, *Intel nervana neural network processor-t (nnp-t) fused floating point many-term dot product*, in *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*, pp. 133–136, IEEE, 2020.

[50] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky, *Nvidia a100 tensor core gpu: Performance and innovation*, *IEEE Micro* **41** (2021), no. 2 29–35.

[51] J. Choquette, O. Giroux, and D. Foley, *Volta: Performance and programmability*, *Ieee Micro* **38** (2018), no. 2 42–52.

[52] A. Li and S. Su, *Accelerating binarized neural networks via bit-tensor-cores in turing gpus*, *IEEE Transactions on Parallel and Distributed Systems* **32** (2020), no. 7 1878–1891.

[53] S. Han, H. Mao, and W. J. Dally, *Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding*, in *ICLR*, 2016.

[54] M. Courbariaux, Y. Bengio, and J. David, *Binaryconnect: Training deep neural networks with binary weights during propagations*, in *NIPS*, pp. 3123–3131, 2015.

[55] E. Park, D. Kim, and S. Yoo, *Energy-efficient neural network accelerator based on outlier-aware low-precision computation*, in *ISCA*, pp. 688–698, IEEE Computer Society, 2018.

[56] A. Li, T. Geng, T. Wang, M. Herbordt, S. L. Song, and K. Barker, *Bstc: A novel binarized-soft-tensor-core design for accelerating bit-based approximated neural nets*, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019.

[57] T. Geng, A. Li, T. Wang, C. Wu, Y. Li, R. Shi, W. Wu, and M. Herbordt, *O3bnn-r: An out-of-order architecture for high-performance and regularized bnn inference*, *TPDS'20* (2020).

[58] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Z. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, *Pytorch: An imperative style, high-performance deep learning library*, in *NeurIPS*, pp. 8024–8035, 2019.

[59] Nvidia, "Nvidia tesla v100 gpu architecture." `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`.

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

[61] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, *Dissecting the nvidia volta gpu architecture via microbenchmarking*, *arXiv preprint arXiv:1804.06826* (2018).

[62] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, *Dissecting the nvidia turing t4 gpu via microbenchmarking*, *arXiv* (2019).

[63] Nvidia, "Nvidia a100 tensor core gpu architecture." `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf`.

[64] NVIDIA, "Cuda programming guide: Sub-byte operations." https://docs.nvidia.com/cuda/cuda-c-programming-guide/#wmma-subbyte, 2021.

[65] A. Li, Y. Tay, A. Kumar, and H. Corporaal, *Transit: A visual analytical model for multithreaded machines*, in *Proceedings of the 24th international symposium on high-performance parallel and distributed computing*, pp. 101–106, 2015.

[66] A. Li, S. L. Song, E. Brugel, A. Kumar, D. Chavarria-Miranda, and H. Corporaal, *X: A comprehensive analytic model for parallel machines*, in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 242–252, IEEE, 2016.

[67] S. Ioffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, in *ICML*, vol. 37 of *JMLR Workshop and Conference Proceedings*, pp. 448–456, JMLR.org, 2015.

[68] A. Li, S. L. Song, A. Kumar, E. Z. Zhang, D. Chavarría-Miranda, and H. Corporaal, *Critical points based register-concurrency autotuning for gpus*, in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1273–1278, IEEE, 2016.

[69] A. Krizhevsky, I. Sutskever, and G. E. Hinton, *Imagenet classification with deep convolutional neural networks*, *Advances in neural information processing systems* **25** (2012) 1097–1105.

[70] Z. Cai, X. He, J. Sun, and N. Vasconcelos, *Deep learning with low precision by half-wave gaussian quantization*, in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 5918–5926, 2017.

[71] NVIDIA, "Cuda template library for dense linear algebra at all levels and scales (cutlass)."

[72] A. G. Research, "Sesor technologies and data analytics." https://www.smartgrid.gov/files/Sensor_Technologies_MYPP_12_19_18_final.pdf.

[73] J. Kung, D. C. Zhang, G. S. van der Wal, S. M. Chai, and S. Mukhopadhyay, *Efficient object detection using embedded binarized neural networks*, *J. Signal Process. Syst.* **90** (2018), no. 6 877–890.

[74] B. McDanel, S. Teerapittayanon, and H. T. Kung, *Embedded binarized neural networks*, in *EWSN*, Junction Publishing, Canada / ACM, 2017.

[75] N. Fasfous, M. R. Vemparala, A. Frickenstein, L. Frickenstein, and W. Stechele, *Binarycop: Binary neural network-based COVID-19 face-mask wear and positioning predictor on edge devices*, *CoRR* **abs/2102.03456** (2021).

[76] A. Garofalo, G. Tagliavini, F. Conti, D. Rossi, and L. Benini, *Xpulpnn: Accelerating quantized neural networks on RISC-V processors through ISA extensions*, in *DATE*, pp. 186–191, IEEE, 2020.

[77] S. Zhang, B. Frey, and M. Bansal, *Chren: Cherokee-english machine translation for endangered language revitalization*, in *EMNLP'20*, 2020.

[78] J. Devlin, M. Chang, K. Lee, and K. Toutanova, *BERT: pre-training of deep bidirectional transformers for language understanding*, in *NAACL-HLT (1)*, pp. 4171–4186, Association for Computational Linguistics, 2019.

[79] AMD, "Amd accelerated parallel processing opencl programming guide." http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf, 2013.

[80] Intel, "Intel xeon phi coprocessor instruction set architecture reference manual." https://software.intel.com /content/dam/develop/external/us/en/documents/327364001en.pdf, 2012.

[81] NVIDIA, "Tensor core performance." https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/, 2017.

[82] V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud, *K-nearest neighbor search: Fast gpu-based implementations and application to high-dimensional feature matching*, in *Proceedings of the International Conference on Image Processing 2010*, (Hong Kong, China), pp. 3757–3760, IEEE, 2010.

[83] angelhof, "Hipeac gpus k-means." https://github.com/angelhof/ gpus-kmeans.git, 2017.

[84] I. of Electrical and E. Engineers, *Ieee standard for binary floating point arithmetic*, 1985.

[85] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, *Nvidia tensor core programmability, performance precision*, in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, (Vancouver, British Columbia, CANADA), pp. 522–531, IEEE Computer Society, 2018.

[86] T. Dekker, *A floating-point technique for extending the available precision.*, *Numerische Mathematik* **18** (1971/72) 224–242.

[87] J. R. Shewchuk, *Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates*, *Discrete & Computational Geometry* **18** (Oct., 1997) 305–363.

[88] D. M. Priest, *On properties of floating point arithmetics: Numerical stability and the cost of accurate computations*, tech. rep., University of California, Berkeley, 1992.

[89] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Boston, third ed., 1997.

[90] X. Zhang, G. Tan, S. Xue, J. Li, K. Zhou, and M. Chen, *Understanding the gpu microarchitecture to achieve bare-metal performance tuning*, in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, (New York, NY, USA), p. 31–43, Association for Computing Machinery, 2017.

[91] X. Li, Y. Liang, S. Yan, L. Jia, and Y. Li, *A coordinated tiling and batching framework for efficient gemm on gpus*, in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, (New York, NY, USA), p. 229–241, Association for Computing Machinery, 2019.

[92] J. Lai and A. Seznec, *Performance upper bound analysis and optimization of SGEMM on fermi and kepler gpus*, in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization*, (Shenzhen, China), pp. 4:1–4:10, IEEE Computer Society, 2013.

[93] Z. Jia, M. Maggioni, B. Staiger, and D. Scarpazza, *Dissecting the nvidia volta gpu architecture via microbenchmarking*, 04, 2018.

[94] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, *Dissecting the nvidia turing T4 GPU via microbenchmarking*, 2019.

[95] NVIDIA, "Cuda c++ programming guide." https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2020.

[96] NVIDIA, "Cuda binary utilities." https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#instruction-set-ref, 2020.

[97] NVIDIA, "Ptx and sass assembly debugging." https://docs.nvidia.com/gameworks/content/developertools/desktop/ptx_sass_assembly_debugging.htm, 2020.

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

[99] D. Yan, W. Wang, and X. Chu, *Demystifying tensor cores to optimize half-precision matrix multiply*, in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, (New Orleans, LA, USA), pp. 634–643, IEEE, 2020.

[100] A. Dakkak, C. Li, J. Xiong, I. Gelado, and W.-m. Hwu, *Accelerating reduction and scan using tensor core units*, in *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, (New York, NY, USA), p. 46–57, Association for Computing Machinery, 2019.

[101] S. Gopinath, N. Ghanathe, V. Seshadri, and R. Sharma, *Compiling kb-sized machine learning models to tiny iot devices*, in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, (New York, NY, USA), p. 79–95, Association for Computing Machinery, 2019.

[102] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, *Tvm: An automated end-to-end optimizing compiler for deep learning*, in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, (USA), p. 579–594, USENIX Association, 2018.

[103] S. Lee and C. Wu, *Characterizing the latency hiding ability of gpus*, in *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014*, (Monterey, CA, USA), pp. 145–146, IEEE Computer Society, 2014.

[104] S. S. Pinter, *Register allocation with instruction scheduling*, in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, (New York, NY, USA), p. 248–257, Association for Computing Machinery, 1993.

[105] F. M. Quintão Pereira and J. Palsberg, *Register allocation by puzzle solving*, in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, (New York, NY, USA), p. 216–226, Association for Computing Machinery, 2008.

[106] F. M. Q. a. Pereira and J. Palsberg, *Register allocation after classical ssa elimination is np-complete*, in *Proceedings of the 9th European Joint Conference on Foundations of Software Science and Computation Structures*, FOSSACS'06, (Berlin, Heidelberg), p. 79–93, Springer-Verlag, 2006.

[107] M. S. Andersen, J. Dahl, and L. Vandenberghe, "Convex optimization solver." https://cvxopt.org/, 2020.

[108] NVIDIA, "Cuda event." https://devblogs.nvidia.com/how-implement-performance-metrics-cuda-cc/, 2020.

[109] e. a. Parry, R.M., *k-nearest neighbor models for microarray gene expression analysis and clinical outcome prediction*, *The Pharmacogenomics Journal* **10** (2010), no. 4 292.

[110] K. Lin, L. Jing, M. Wang, M. Qiu, and Z. Ji, *A novel long-term air quality forecasting algorithm based on knn and narx*, in *2017 12th International Conference on Computer Science and Education (ICCSE)*, (Beijing, China), pp. 343–348, IEEE, 2017.

[111] Y. Z. LiLi Li and Y. Zhao, *k-nearest neighbors for automated classification of celestial objects*, in *Sci. China Ser. G-Phys. Mech. Astron.*, vol. 51, (China), pp. 916–922, Springer, 2008.

[112] NVIDIA, "Nvidia t4." https://www.nvidia.com/en-us/data-center/tesla-t4/, 2018.

[113] NVIDIA, "Nvidia rtx 6000." https://www.nvidia.com/en-us/design-visualization/quadro/rtx-6000/, 2018.

[114] A. Ahmad, M. Anisetti, E. Damiani, and G. Jeon, *Special issue on real-time image and video processing in mobile embedded systems*, *Journal of Real-Time Image Processing* **16** (Feb, 2019) 1–4.

[115] K. M. bin Saipullah, A. Anuar, N. A. binti Ismail, and Y. Soo, *Real-time video processing using native programming on android platform*, in *2012 IEEE 8th International Colloquium on Signal Processing and its Applications*, pp. 276–281, March, 2012.

[116] "Advancing ai for video.." https://phys.org/news/2019-06-advancing-ai-video-startup-powerful.html. Accessed: 2019-11-29.

[117] Aria, "The wearables giving computer vision to the blind." https://www.wired.com/story/wearables-for-the-blind/, 2018. Accessed: 2018-07-16.

[118] D. Dakopoulos and N. G. Bourbakis, *Wearable obstacle avoidance electronic travel aids for blind: a survey*, *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* (2009).

[119] M. Hersh and M. A. Johnson, *Assistive technology for visually impaired and blind people*. Springer Science Business Media, 2010.

[120] BBC, "Bbc series uses robot creatures to document secret lives of animals." https://www.theguardian.com/media/2016/dec/31/bbc-robot-creatures-spy-secret-lives-animals.-wildlife-series, 2018. Accessed: 2018-07-16.

[121] S. IEEE, "Robot takes on landmine detection while humans stay very very far away." https://spectrum.ieee.org/automaton/robotics/military-robots/husky-robot-takes-on-landmine-detection-while.-humans-stay-very-very-far-away, 2018. Accessed: 2018-07-16.

[122] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, in *ICLR*, 2015.

[123] S. Han, J. Pool, J. Tran, and W. Dally, *Learning both weights and connections for efficient neural network*, in *Advances in neural information processing systems*, pp. 1135–1143, 2015.

[124] J.-H. Luo, J. Wu, and W. Lin, *Thinet: A filter level pruning method for deep neural network compression*, in *Proceedings of the IEEE international conference on computer vision*, pp. 5058–5066, 2017.

[125] J. Lin, Y. Rao, J. Lu, and J. Zhou, *Runtime neural pruning*, in *Advances in Neural Information Processing Systems*, pp. 2181–2191, 2017.

[126] E. Park, J. Ahn, and S. Yoo, *Weighted-entropy-based quantization for deep neural networks*, in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5456–5464, 2017.

[127] R. Zhao, Y. Hu, J. Dotzel, C. De Sa, and Z. Zhang, *Improving neural network quantization without retraining using outlier channel splitting*, in *International Conference on Machine Learning*, pp. 7543–7552, 2019.

[128] Y. Xu, Y. Wang, A. Zhou, W. Lin, and H. Xiong, *Deep neural network compression with single and multiple level quantization*, in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[129] H. Shen, S. Han, M. Philipose, and A. Krishnamurthy, *Fast video classification via adaptive cascading of deep models*, in *CVPR*, 2017.

[130] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia, *Noscope: optimizing neural network queries over video at scale*, *VLDB* (2017).

[131] K. Hsieh, G. Ananthanarayanan, P. Bodik, S. Venkataraman, P. Bahl, M. Philipose, P. B. Gibbons, and O. Mutlu, *Focus: Querying large video datasets with low latency and low cost*, in *OSDI*, 2018.

[132] J. Jiang, G. Ananthanarayanan, P. Bodik, S. Sen, and I. Stoica, *Chameleon: Scalable adaptation of video analytics*, in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, (New York, NY, USA), pp. 253–266, ACM, 2018.

[133] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, and M. Isard, *Tensorflow: A system for large-scale machine learning.*, in *OSDI*, 2016.

[134] "Jetson nano specification." https://developer.nvidia.com/embedded/buy/jetson-nano-devkit. Accessed: 2018-04-20.

[135] "Dell workstation t7910." https://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/Dell_Precision_Tower_7910_Spec_Sheet.pdf. Accessed: 2018-04-20.

[136] G. Huang, Z. Liu, K. Q. Weinberger, and L. van der Maaten, *Densely connected convolutional networks*, in *CVPR*, vol. 1, p. 3, 2017.

[137] O. M. Parkhi, A. Vedaldi, and A. Zisserman, *Deep face recognition.*, in *BMVC*, 2015.

[138] A. Krizhevsky and G. Hinton, *Learning multiple layers of features from tiny images*, *Technical report, University of Toronto* **1** (01, 2009).

[139] J. Fu, H. Zheng, and T. Mei, *Look closer to see better: Recurrent attention convolutional neural network for fine-grained image recognition*, in *CVPR*, 2017.

[140] M. Philipose, *Efficient object detection via adaptive online selection of sensor-array elements.*, in *AAAI*, 2014.

[141] M. Jaderberg, A. Vedaldi, and A. Zisserman, *Speeding up convolutional neural networks with low rank expansions*, in *BMVC*, 2014.

[142] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio, *Fitnets: Hints for thin deep nets*, in *ICLR*, 2015.

[143] J. Xue, J. Li, D. Yu, M. Seltzer, and Y. Gong, *Singular value decomposition based low-footprint speaker adaptation and personalization for deep neural network*, in *ICASSP*, 2014.

[144] W. Chen, J. Wilson, S. Tyree, K. Weinberger, and Y. Chen, *Compressing neural networks with the hashing trick*, in *ICML*, 2015.

[145] J. Ba and R. Caruana, *Do deep nets really need to be deep?*, in *NeurIPS*, 2014.

[146] G. Chen, W. Choi, X. Yu, T. Han, and M. Chandraker, *Learning efficient object detection models with knowledge distillation*, in *NeurIPS*, 2017.

[147] D. Lopez-Paz, B. Schölkopf, L. Bottou, and V. Vapnik, *Unifying distillation and privileged information*, in *ICLR*, 2016.

[148] C. Buciluǎ, R. Caruana, and A. Niculescu-Mizil, *Model compression*, in *SIGKDD*, 2006.

[149] M. Xu, M. Zhu, Y. Liu, F. X. Lin, and X. Liu, *Deepcache: Principled cache for mobile deep vision*, in *MobiCom*, 2018.

[150] P. Guo and W. Hu, *Potluck: Cross-application approximate deduplication for computation-intensive mobile applications*, in *ASPLOS*, pp. 271–284, 2018.

[151] A. H. Jiang, D. L.-K. Wong, C. Canel, L. Tang, I. Misra, M. Kaminsky, M. A. Kozuch, P. Pillai, D. G. Andersen, and G. R. Ganger, *Mainstream: Dynamic stem-sharing for multi-tenant video processing*, in *ATC*, 2018.

[152] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman, *Live video analytics at scale with approximation and delay-tolerance*, in *NSDI*, 2017.

[153] L. Wang, Y. Xiong, Z. Wang, Y. Qiao, D. Lin, X. Tang, and L. Val Gool, *Temporal segment networks: Towards good practices for deep action recognition*, in *ECCV*, 2016.

[154] C.-Y. Wu, M. Zaheer, H. Hu, R. Manmatha, A. J. Smola, and P. Krähenbühl, *Compressed video action recognition*, in *CVPR*, 2018.

[155] Z. Shou, X. Lin, Y. Kalantidis, L. Sevilla-Lara, M. Rohrbach, S.-F. Chang, and Z. Yan, *Dmc-net: Generating discriminative motion cues for fast compressed video action recognition*, in *CVPR*, 2019.

[156] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri, *Learning spatiotemporal features with 3d convolutional networks*, in *ICCV*, 2015.

[157] J. Carreira and A. Zisserman, *Quo vadis, action recognition? a new model and the kinetics dataset*, in *CVPR*, 2017.

[158] L. Ge, H. Liang, J. Yuan, and D. Thalmann, *3d convolutional neural networks for efficient and robust hand pose estimation from single depth images*, in *CVPR*, 2017.

[159] M. Xu, J. Liu, Y. Liu, F. X. Lin, Y. Liu, and X. Liu, *A first look at deep learning apps on smartphones*, in *WWW*, 2019.

[160] T. Xu, L. M. Botelho, and F. X. Lin, *Vstore: A data store for analytics on large videos*, in *EuroSys*, 2019.

[161] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, *Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints*, in *MobiSys*, 2016.

[162] J. Shao, *Mathematical Statistics*. Springer Texts in Statistics. Springer, 2003.

[163] W. Rudin, *Principles of Mathematical Analysis*. International series in pure and applied mathematics. McGraw-Hill, 1976.

[164] S. Aminikhanghahi and D. J. Cook, *A survey of methods for time series change point detection*, *Knowledge and information systems* (2017).

[165] R. P. Adams and D. J. MacKay, *Bayesian online changepoint detection*, *arXiv preprint arXiv:0710.3742* (2007).

[166] P. Fearnhead and Z. Liu, *On-line inference for multiple changepoint problems*, *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* (2007).

[167] Y. Saatçi, R. D. Turner, and C. E. Rasmussen, *Gaussian process change point models*, in *ICML*, 2010.

[168] T. D. Jeremias Knoblauch, *Spatio-temporal Bayesian on-line changepoint detection with model selection*, in *ICML*, 2018.

[169] O. Mazhar, C. Rojas, C. Fischione, and edit Mohammad Reza Hesamzadeh, *Bayesian model selection for change point detection and clustering*, in *ICML*, 2018.

[170] A. Zhang and J. Paisley, *Deep Bayesian nonparametric tracking*, in *ICML*, 2018.

[171] D. G. Kleinbaum and M. Klein, *Survival analysis*, vol. 3. Springer, 2010.

[172] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger, *On calibration of modern neural networks*, in *ICML*, 2017.

[173] A. Niculescu-Mizil and R. Caruana, *Predicting good probabilities with supervised learning*, in *ICML*, 2005.

[174] "Wikipedia: Pareto efficiency.." https://en.wikipedia.org/wiki/Pareto _efficiency. Accessed: 2019-06-12.

[175] M. Figurnov, A. Ibraimova, D. P. Vetrov, and P. Kohli, *Perforatedcnns: Acceleration through elimination of redundant convolutions*, in *NeurIPS*, 2016.

[176] "Raspberry pi 3b+ specification." https://www.raspberrypi.org/products /raspberry-pi-3-model-b-plus/. Accessed: 2018-04-20.

[177] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, *Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size*, 2016.

[178] "Jetbot, a $250 diy autonomous robot based on jetson nano impresses at gtc." https://blogs.nvidia.com/blog/2019/03/26/jetbot-diy-autonomous-robot/. Accessed: 2019-06-12.

[179] "Home automation at a glance using ai glasses." https://hackaday.com/2019/08/15/home-automation-at-a-glance-using-ai-glasses/. Accessed: 2019-06-12.

[180] "Build a hardware-based face recognition system for $150 with the nvidia jetson nano and python." https://medium.com/@ageitgey/build-a-hardware-based-face-recognition-system-for-150-with-the-nvidia-jetson-nano-and-python-a25cb8c891fd. Accessed: 2019-06-12.

[181] "Extech multimeter model ex330 manual." http://www.extech.com/resources/EX330_UM.pdf. Accessed: 2018-04-20.

[182] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, *Imagenet: A large-scale hierarchical image database*, in *CVPR*, 2009.

[183] C. Tan, F. Sun, T. Kong, W. Zhang, C. Yang, and C. Liu, *A survey on deep transfer learning*, 2018.

[184] R. Girshick, *Fast r-cnn*, in *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ICCV '15, (USA), p. 1440–1448, IEEE Computer Society, 2015.

[185] R. Girshick, J. Donahue, T. Darrell, and J. Malik, *Rich feature hierarchies for accurate object detection and semantic segmentation*, in *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '14, (USA), p. 580–587, IEEE Computer Society, 2014.

[186] S. Ren, K. He, R. Girshick, and J. Sun, *Faster r-cnn: Towards real-time object detection with region proposal networks*, in *Advances in Neural Information Processing Systems 28* (C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, eds.), pp. 91–99. Curran Associates, Inc., 2015.

[187] B. Amos, B. Ludwiczuk, and M. Satyanarayanan, *Openface: A general-purpose face recognition library with mobile applications*, tech. rep., CMU-CS-16-118, CMU School of Computer Science, 2016.

[188] P. Viola and M. Jones, *Rapid object detection using a boosted cascade of simple features*, *CVPR* (2001).

[189] J. Dai, Y. Li, K. He, and J. Sun, *R-FCN: object detection via region-based fully convolutional networks*, in *NIPS*, pp. 379–387, 2016.

[190] C. He, H. Zeng, J. Huang, X. Hua, and L. Zhang, *Structure aware single-stage 3d object detection from point cloud*, in *CVPR*, pp. 11870–11879, IEEE, 2020.

[191] M. Gault, "DHS Admits Facial Recognition Photos Were Hacked, Released on Dark Web." `https://www.vice.com/en/article/m7jzbb/dhs-admits-facial-recognition-photos-were-\hacked-released-on-dark-web`.

[192] K. O'Flaherty, "China facial recognition database leak sparks fears over mass data collection." `shorturl.at/OUW59`.

[193] X. Shen, "Facial recognition data leaks are rampant in china as covid-19 pushes wider use of the technology." `shorturl.at/kFHY1`.

[194] K. Wiggers, "Openai's massive gpt-3 model is impressive, but size isn't everything." `https://venturebeat.com/2020/06/01/ai-machine-learning-openai-gpt-3-size-isnt\-everything/`.

[195] T. Peng and M. Sarazen, "The staggering cost of training sota ai models." `https://syncedreview.com/2019/06/27/the-staggering-cost-of-training-sota-ai-\models/`.

[196] B. Parno, J. Howell, C. Gentry, and M. Raykova, *Pinocchio: Nearly practical verifiable computation*, in *IEEE Symposium on Security and Privacy*, pp. 238–252, IEEE Computer Society, 2013.

[197] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, *Succinct non-interactive zero knowledge for a von neumann architecture*, in *USENIX Security Symposium*, pp. 781–796, USENIX Association, 2014.

[198] S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam, *Ligero: Lightweight sublinear arguments without a trusted setup*, in *CCS*, pp. 2087–2104, ACM, 2017.

[199] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, *Bulletproofs: Short proofs for confidential transactions and more*, in *IEEE Symposium on Security and Privacy*, pp. 315–334, IEEE Computer Society, 2018.

[200] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish, *Doubly-efficient zksnarks without trusted setup*, in *IEEE Symposium on Security and Privacy*, pp. 926–943, IEEE Computer Society, 2018.

[201] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, *Scalable, transparent, and post-quantum secure computational integrity*, *IACR Cryptol. ePrint Arch.* **2018** (2018) 46.

[202] E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, and N. P. Ward, *Aurora: Transparent succinct arguments for R1CS*, in *EUROCRYPT (1)*, vol. 11476 of *Lecture Notes in Computer Science*, pp. 103–128, Springer, 2019.

[203] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song, *Libra: Succinct zero-knowledge proofs with optimal prover computation*, in *CRYPTO (3)*, vol. 11694 of *Lecture Notes in Computer Science*, pp. 733–764, Springer, 2019.

[204] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu, *ZEXE: enabling decentralized private computation*, in *IEEE Symposium on Security and Privacy*, pp. 947–964, IEEE, 2020.

[205] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. E. Lauter, M. Naehrig, and J. Wernsing, *Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy*, in *ICML*, vol. 48, pp. 201–210, 2016.

[206] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, *GAZELLE: A low latency framework for secure neural network inference*, in *USENIX Security Symposium*, pp. 1651–1669, USENIX Association, 2018.

[207] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. E. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, *CHET: an optimizing compiler for fully-homomorphic neural-network inferencing*, in *PLDI*, pp. 142–156, ACM, 2019.

[208] X. Jiang, M. Kim, K. E. Lauter, and Y. Song, *Secure outsourced matrix computation and application to neural networks*, in *CCS*, pp. 1209–1222, ACM, 2018.

[209] P. Mohassel and Y. Zhang, *Secureml: A system for scalable privacy-preserving machine learning*, in *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 19–38, IEEE, 2017.

[210] J. Liu, M. Juuti, Y. Lu, and N. Asokan, *Oblivious neural network predictions via minionn transformations*, in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, (New York, NY, USA), p. 619–631, Association for Computing Machinery, 2017.

[211] B. D. Rouhani, M. S. Riazi, and F. Koushanfar, *Deepsecure: Scalable provably-secure deep learning*, in *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, (New York, NY, USA), Association for Computing Machinery, 2018.

[212] R. Shokri and V. Shmatikov, *Privacy-preserving deep learning*, in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, (New York, NY, USA), p. 1310–1321, Association for Computing Machinery, 2015.

[213] S. T. V. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish, *Taking proof-based verified computation a few steps closer to practicality*, in *USENIX Security Symposium*, pp. 253–268, USENIX Association, 2012.

[214] D. Das, N. Mellempudi, D. Mudigere, D. Kalamkar, S. Avancha, K. Banerjee, S. Sridharan, K. Vaidyanathan, B. Kaul, E. Georganas, A. Heinecke, P. Dubey, J. Corbal, N. Shustrov, R. Dubtsov, E. Fomenko, and V. Pirogov, *Mixed precision training of convolutional neural networks using integer operations*, in *ICLR*, 2018.

[215] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, *Mixed precision training*, in *ICLR*, 2018.

[216] X. Sun, J. Choi, C. Chen, N. Wang, S. Venkataramani, V. Srinivasan, X. Cui, W. Zhang, and K. Gopalakrishnan, *Hybrid 8-bit floating point (HFP8) training and inference for deep neural networks*, in *NeurIPS*, pp. 4901–4910, 2019.

[217] R. Li, F. Liang, H. Qin, Y. Wang, R. Fan, and J. Yan, *Fully quantized network for object detection*, in *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019*, 2019.

[218] Y. Wei, X. Pan, H. Qin, W. Ouyang, and J. Yan, *Quantization mimic: Towards very tiny cnn for object detection*, in *15th European Conference on Computer Vision, ECCV 2018*, 2018.

[219] P. Stock, A. Joulin, R. Gribonval, B. Graham, and H. Jégou, *And the bit goes down: Revisiting the quantization of neural networks*, in *International Conference on Learning Representations*, 2020.

[220] A. Boyle, T. Soper, and T. Bishop, "Exclusive: Apple acquires xnor.ai, edge ai spin-out from paul allen's ai2, for price in $200m range." `https://www.geekwire.com/2020/exclusive-apple-acquires-xnor-ai-edge-ai\-spin-paul-allens-ai2-price-200m-range/`.

[221] A. Alford, "Apple acquires edge-focused ai startup xnor.ai." `https://www.infoq.com/news/2020/01/apple-acquires-xnor-ai/`.

[222] PyTorch, "Pytorch quantization." `https://pytorch.org/docs/stable/quantization.html`.

[223] TensorFlow, "Post-training quantization." `https://www.tensorflow.org/lite/performance/post_training_quantization`.

[224] B. Feng, L. Qin, Z. Zhang, Y. Ding, and S. Chu, "Zen." `https://github.com/UCSB-TDS/ZEN`.

[225] S. Ellis, A. Juels, and S. Nazarov, "Chainlink: A decentralized oracle network." `https://link.smartcontract.com/whitepaper`.

[226] P. S. L. M. Barreto, B. Lynn, and M. Scott, *Constructing elliptic curves with prescribed embedding degrees*, in *Security in Communication Networks, Third International Conference, SCN 2002, Amalfi, Italy, September 11-13, 2002. Revised Papers* (S. Cimato, C. Galdi, and G. Persiano, eds.), vol. 2576 of *Lecture Notes in Computer Science*, pp. 257–267, Springer, 2002.

[227] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design - The Hardware / Software Interface (Revised 4th Edition)*. The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press, 2012.

[228] J. Groth, *On the size of pairing-based non-interactive arguments*, in *EUROCRYPT (2)*, vol. 9666 of *Lecture Notes in Computer Science*, pp. 305–326, Springer, 2016.

[229] Z. Ghodsi, T. Gu, and S. Garg, *Safetynets: Verifiable execution of deep neural networks on an untrusted cloud*, in *NIPS*, pp. 4672–4681, 2017.

[230] A. L. Maas, A. Y. Hannun, and A. Y. Ng, *Rectifier nonlinearities improve neural network acoustic models*, in *Proc ICML*, 2013.

[231] J. Zhang, Z. Fang, Y. Zhang, and D. Song, *Zero knowledge proofs for decision tree predictions and accuracy*, in *CCS*, pp. 2039–2053, 2020.

[232] S. Lee, H. Ko, J. Kim, and H. Oh, *vcnn: Verifiable convolutional neural network*, *IACR Cryptol. ePrint Arch.* **2020** (2020) 584.

[233] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, *Quadratic span programs and succinct nizks without pcps*, in *EUROCRYPT*, vol. 7881 of *Lecture Notes in Computer Science*, pp. 626–645, Springer, 2013.

[234] A. E. Kosba, D. Papadopoulos, C. Papamanthou, M. F. Sayed, E. Shi, and N. Triandopoulos, *TRUESET: Faster verifiable set computations*, in *23rd USENIX Security Symposium (USENIX Security 14)*, (San Diego, CA), pp. 765–780, USENIX Association, Aug., 2014.

[235] M. Campanelli, D. Fiore, and A. Querol, *Legosnark: Modular design and composition of succinct zero-knowledge proofs*, in *CCS*, pp. 2075–2092, ACM, 2019.

[236] T. Liu, X. Xie, and Y. Zhang, *zkcnn: Zero knowledge proofs for convolutional neural network predictions and accuracy*, *IACR Cryptol. ePrint Arch.* (2021) 673.

[237] N. Carlini, M. Jagielski, and I. Mironov, *Cryptanalytic extraction of neural network models*, in *Crypto*, pp. 189–218, 2020.

[238] Y. Huang, Z. Song, K. Li, and S. Arora, *Instahide: Instance-hiding schemes for private distributed learning*, in *ICML*, 2020.

[239] Y. Huang, Z. Song, D. Chen, K. Li, and S. Arora, *Texthide: Tackling data privacy for language understanding tasks*, in *EMNLP* (T. Cohn, Y. He, and Y. Liu, eds.), pp. 1368–1382, 2020.

[240] R. Jia and P. Liang, *Adversarial examples for evaluating reading comprehension systems*, in *EMNLP*, pp. 2021–2031, Association for Computational Linguistics, 2017.

[241] H. Chabanne, J. Keuffer, and R. Molva, *Embedded proofs for verifiable neural networks*, *IACR Cryptol. ePrint Arch.* **2017** (2017) 1038.

[242] S. Goldwasser, G. N. Rothblum, J. Shafer, and A. Yehudayoff, *Interactive proofs for verifying machine learning*, *Electron. Colloquium Comput. Complex.* **27** (2020) 58.

[243] X. Ma, F. Zhang, X. Chen, and J. Shen, *Privacy preserving multi-party computation delegation for deep learning in cloud computing*, in *Information Science*, 2018.

[244] D. Demmler, T. Schneider, and M. Zohner, *ABY - A framework for efficient mixed-protocol secure two-party computation*, in *NDSS*, The Internet Society, 2015.

[245] M. Abadi, A. Chu, I. J. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang, *Deep learning with differential privacy*, in *CCS*, pp. 308–318, ACM, 2016.

[246] Y. Zhu, X. Yu, M. Chandraker, and Y.-X. Wang, *Private-knn: Practical differential privacy for computer vision*, in *CVPR*, June, 2020.

[247] M. Abadi, A. Chu, I. J. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang, *Deep learning with differential privacy*, in *CCS*, pp. 308–318, ACM, 2016.

[248] arkworks, "arks-snark." `https://github.com/arkworks-rs/snark`.

[249] N. Bitansky, A. Chiesa, Y. Ishai, R. Ostrovsky, and O. Paneth, *Succinct non-interactive arguments via linear interactive proofs*, in *TCC*, vol. 7785 of *Lecture Notes in Computer Science*, pp. 315–333, Springer, 2013.

[250] J. Groth, *Short pairing-based non-interactive zero-knowledge arguments*, in *ASIACRYPT*, vol. 6477 of *Lecture Notes in Computer Science*, pp. 321–340, Springer, 2010.

[251] D. Boneh and X. Boyen, *Secure identity based encryption without random oracles*, in *CRYPTO*, vol. 3152 of *Lecture Notes in Computer Science*, pp. 443–459, Springer, 2004.

[252] R. Gennaro, *Multi-trapdoor commitments and their applications to proofs of knowledge secure under concurrent man-in-the-middle attacks*, in *CRYPTO*, vol. 3152 of *Lecture Notes in Computer Science*, pp. 220–236, Springer, 2004.

[253] C. Gentry and D. Wichs, *Separating succinct non-interactive arguments from all falsifiable assumptions*, in *STOC*, pp. 99–108, ACM, 2011.

[254] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, *Gradient-based learning applied to document recognition*, in *Proceedings of the IEEE*, pp. 2278–2324, 1998.

[255] F. Davis and M. V. Schijndel, *Recurrent neural network language models always learn english-like relative clause attachment*, in *ACL*, pp. 1979–1990, Association for Computational Linguistics, 2020.

[256] X. E. Wang, V. Jain, E. Ie, W. Y. Wang, Z. Kozareva, and S. Ravi, *Environment-agnostic multitask learning for natural language grounded navigation*, in *ECCV*, 2020.

[257] A. W. Senior, R. Evans, J. Jumper, J. Kirkpatrick, L. Sifre, T. Green, C. Qin, A. Žídek, A. W. R. Nelson, A. Bridgland, H. Penedones, S. Petersen, K. Simonyan, S. Crossan, P. Kohli, D. T. Jones, D. Silver, K. Kavukcuoglu, and

D. Hassabis, *Protein structure prediction using multiple deep neural networks in the 13th critical assessment of protein structure prediction (casp13)*, *Proteins: Structure, Function, and Bioinformatics* **87** (2019), no. 12 1141–1148.

[258] A. W. Senior, R. Evans, J. Jumper, J. Kirkpatrick, L. Sifre, T. Green, C. Qin, A. Žídek, A. W. R. Nelson, A. Bridgland, H. Penedones, S. Petersen, K. Simonyan, S. Crossan, P. Kohli, D. T. Jones, D. Silver, K. Kavukcuoglu, and D. Hassabis, *Improved protein structure prediction using potentials from deep learning*, *Nature* (2020).

[259] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, *cudnn: Efficient primitives for deep learning*, *CoRR* **abs/1410.0759** (2014) [arXiv:1410.0759].

[260] Y. Shi, X. Yu, K. Sohn, M. Chandraker, and A. K. Jain, *Towards universal representation learning for deep face recognition*, in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June, 2020.

[261] Q. Wang, T. Wu, H. Zheng, and G. Guo, *Hierarchical pyramid diverse attention networks for face recognition*, in *CVPR*, June, 2020.

[262] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger, *Poseidon: A new hash function for zero-knowledge proof systems*, in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[263] H. Wu, W. Zheng, A. Chiesa, R. A. Popa, and I. Stoica, *DIZK: A distributed zero knowledge proof system*, in *USENIX Security Symposium*, pp. 675–692, USENIX Association, 2018.

[264] Y. Zhang, S. Wang, X. Zhang, J. Dong, X. Mao, F. Long, C. Wang, D. Zhou, M. Gao, and G. Sun, *Pipezk: Accelerating zero-knowledge proof with a pipelined architecture*, in *ISCA*, pp. 416–428, IEEE, 2021.

[265] B. Zoph and Q. V. Le, *Neural architecture search with reinforcement learning*, in *ICLR*, OpenReview.net, 2017.

[266] G. E. Hinton, O. Vinyals, and J. Dean, *Distilling the knowledge in a neural network*, *CoRR* **abs/1503.02531** (2015).

[267] L. Orseau, M. Hutter, and O. Rivasplata, *Logarithmic pruning is all you need*, in *NeurIPS*, 2020.

[268] J. Kim, K. Yoo, and N. Kwak, *Position-based scaled gradient for model quantization and pruning*, in *NeurIPS*, 2020.

[269] M. Lin, R. Ji, Y. Zhang, B. Zhang, Y. Wu, and Y. Tian, *Channel pruning via automatic structure search*, in *IJCAI*, pp. 673–679, ijcai.org, 2020.

[270] B. Peng, W. Tan, Z. Li, S. Zhang, D. Xie, and S. Pu, *Extreme network compression via filter group approximation*, in *ECCV (8)*, vol. 11212 of *Lecture Notes in Computer Science*, pp. 307–323, Springer, 2018.

[271] B. Dong, B. Zhang, and H. Wang, *Veridl: Integrity verification of outsourced deep learning services*, *ECML/PKDD* (2021).

[272] R. Lidl, H. Niederreiter, P. Cohn, G. Rota, B. Doran, C. U. Press, P. Flajolet, M. Ismail, T. Lam, and E. Lutwak, *Finite Fields*. No. v. 20, pt. 1 in EBL-Schweitzer. Cambridge University Press, 1997.

[273] R. Cramer and I. Damgård, *Linear zero-knowledge - A note on efficient zero-knowledge proofs and arguments*, in *STOC*, pp. 436–445, ACM, 1997.

[274] zkcrypto, "Bellman." `https://github.com/zkcrypto/bellman`.

[275] P. Valiant, *Incrementally verifiable computation or proofs of knowledge imply time/space efficiency*, in *Theory of Cryptography* (R. Canetti, ed.), (Berlin, Heidelberg), pp. 1–18, Springer Berlin Heidelberg, 2008.

[276] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer, *Recursive composition and bootstrapping for snarks and proof-carrying data*, in *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '13, (New York, NY, USA), p. 111–120, Association for Computing Machinery, 2013.

[277] D. Boneh, B. Lynn, and H. Shacham, *Short signatures from the weil pairing*, in *ASIACRYPT*, vol. 2248, pp. 514–532, 2001.

[278] C. Weng, K. Yang, X. Xie, J. Katz, and X. Wang, *Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning*, in *USENIX Security Symposium*, pp. 501–518, USENIX Association, 2021.

[279] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Q. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, *TVM: an automated end-to-end optimizing compiler for deep learning*, in *OSDI*, pp. 578–594, USENIX Association, 2018.

[280] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, J. E. Gonzalez, and I. Stoica, *Ansor: Generating high-performance tensor programs for deep learning*, in *OSDI*, pp. 863–879, USENIX Association, 2020.

[281] O. Labs, "Mina cryptocurrency." `https://minaprotocol.com/`, 2017.

[282] Celo, "Celo cryptocurrency." `https://celo.org/`, 2018.

[283] Anoma, "Anoma network." `https://anoma.network/`, 2020.

[284] Y. LeCun and C. Cortes, *MNIST handwritten digit database*, .

[285] A. Krizhevsky, V. Nair, and G. Hinton, *Cifar-10 (canadian institute for advanced research)*, .

[286] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, in *ICLR*, 2015.

[287] Z. Yang, Z. Dai, Y. Yang, J. G. Carbonell, R. Salakhutdinov, and Q. V. Le, *Xlnet: Generalized autoregressive pretraining for language understanding*, in *NeurIPS*, pp. 5754–5764, 2019.

[288] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, *Roberta: A robustly optimized BERT pretraining approach*, *CoRR* **abs/1907.11692** (2019).

[289] A. Radford and K. Narasimhan, *Improving language understanding by generative pre-training*, 2018.

[290] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, *Exploring the limits of transfer learning with a unified text-to-text transformer*, *J. Mach. Learn. Res.* **21** (2020) 140:1–140:67.

[291] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen, *Gshard: Scaling giant models with conditional computation and automatic sharding*, in *ICLR*, OpenReview.net, 2021.

[292] Y. Lu, J. Zeng, J. Zhang, S. Wu, and M. Li, *Attention calibration for transformer in neural machine translation*, in *ACL/IJCNLP (1)*, pp. 1288–1298, Association for Computational Linguistics, 2021.

[293] N. Akoury, K. Krishna, and M. Iyyer, *Syntactically supervised transformers for faster neural machine translation*, in *ACL (1)*, pp. 1269–1281, Association for Computational Linguistics, 2019.

[294] L. Qian, H. Zhou, Y. Bao, M. Wang, L. Qiu, W. Zhang, Y. Yu, and L. Li, *Glancing transformer for non-autoregressive neural machine translation*, in *ACL/IJCNLP (1)*, pp. 1993–2003, Association for Computational Linguistics, 2021.

[295] H. Tang, D. Ji, C. Li, and Q. Zhou, *Dependency graph enhanced dual-transformer structure for aspect-based sentiment classification*, in *ACL*, pp. 6578–6588, Association for Computational Linguistics, 2020.

[296] D. Yin, T. Meng, and K. Chang, *Sentibert: A transferable transformer-based architecture for compositional sentiment semantics*, in *ACL*, pp. 3695–3706, Association for Computational Linguistics, 2020.

[297] J. Cheng, I. Fostiropoulos, B. W. Boehm, and M. Soleymani, *Multimodal phased transformer for sentiment analysis*, in *EMNLP (1)*, pp. 2447–2458, Association for Computational Linguistics, 2021.

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

[299] B. Kim, J. Lee, J. Kang, E. Kim, and H. J. Kim, *HOTR: end-to-end human-object interaction detection with transformers*, in *CVPR*, pp. 74–83, Computer Vision Foundation / IEEE, 2021.

[300] X. Zhu, W. Su, L. Lu, B. Li, X. Wang, and J. Dai, *Deformable DETR: deformable transformers for end-to-end object detection*, in *ICLR*, OpenReview.net, 2021.

[301] L. Ye, M. Rochan, Z. Liu, and Y. Wang, *Cross-modal self-attention network for referring image segmentation*, in *CVPR*, pp. 10502–10511, 2019.

[302] F. Yang, H. Yang, J. Fu, H. Lu, and B. Guo, *Learning texture transformer network for image super-resolution*, in *CVPR*, pp. 5790–5799, Computer Vision Foundation / IEEE, 2020.

[303] M. Alzantot, Y. Sharma, A. Elgohary, B. Ho, M. B. Srivastava, and K. Chang, *Generating natural language adversarial examples*, in *EMNLP*, pp. 2890–2896, Association for Computational Linguistics, 2018.

[304] M. Behjati, S. Moosavi-Dezfooli, M. S. Baghshah, and P. Frossard, *Universal adversarial attacks on text classifiers*, in *ICASSP*, pp. 7345–7349, IEEE, 2019.

[305] Y. Hsieh, M. Cheng, D. Juan, W. Wei, W. Hsu, and C. Hsieh, *On the robustness of self-attentive models*, in *ACL (1)*, pp. 1520–1529, Association for Computational Linguistics, 2019.

[306] D. Jin, Z. Jin, J. T. Zhou, and P. Szolovits, *Is BERT really robust? A strong baseline for natural language attack on text classification and entailment*, in *AAAI*, pp. 8018–8025, AAAI Press, 2020.

[307] J. Li, S. Ji, T. Du, B. Li, and T. Wang, *Textbugger: Generating adversarial text against real-world applications*, in *NDSS*, The Internet Society, 2019.

[308] H. Zhang, T. Weng, P. Chen, C. Hsieh, and L. Daniel, *Efficient neural network robustness certification with general activation functions*, in *NeurIPS*, pp. 4944–4953, 2018.

[309] I. J. Goodfellow, J. Shlens, and C. Szegedy, *Explaining and harnessing adversarial examples*, in *ICLR (Poster)*, 2015.

[310] B. Feng, Y. Wang, G. Chen, W. Zhang, Y. Xie, and Y. Ding, *EGEMM-TC: accelerating scientific computing on tensor cores with extended precision*, in *PPoPP*, pp. 278–291, ACM, 2021.

[311] D. Yan, W. Wang, and X. Chu, *Optimizing batched winograd convolution on gpus*, in *PPoPP*, pp. 32–44, ACM, 2020.

[312] J. Lai and A. Seznec, *Performance upper bound analysis and optimization of SGEMM on fermi and kepler gpus*, in *CGO*, pp. 4:1–4:10, IEEE Computer Society, 2013.

[313] X. Li, Y. Liang, S. Yan, L. Jia, and Y. Li, *A coordinated tiling and batching framework for efficient GEMM on gpus*, in *PPoPP*, pp. 229–241, ACM, 2019.

[314] P. Micikevicius, "Local memory and register spilling." `https://developer.download.nvidia.com/CUDA/training/register_spilling.pdf`.

[315] J. Roesch, S. Lyubomirsky, L. Weber, J. Pollock, M. Kirisame, T. Chen, and Z. Tatlock, *Relay: a new IR for machine learning frameworks*, in *MAPL@PLDI*, pp. 58–68, ACM, 2018.

[316] X. Zhang, J. J. Zhao, and Y. LeCun, *Character-level convolutional networks for text classification*, in *NIPS*, pp. 649–657, 2015.

[317] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Y. Ng, and C. Potts, *Recursive deep models for semantic compositionality over a sentiment treebank*, in *EMNLP*, pp. 1631–1642, ACL, 2013.

[318] Y. Wang, B. Feng, and Y. Ding, *QGTC: accelerating quantized GNN via GPU tensor core*, in *PPoPP*, ACM, 2022.

[319] Y. Wang, B. Feng, G. Li, L. Deng, Y. Xie, and Y. Ding, *Stpacc: Structural ti-based pruning for accelerating distance-related algorithms on cpu-fpga platforms*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021) 1–1.

[320] Y. Wang, B. Feng, G. Li, G. Tzimpragos, L. Deng, Y. Xie, and Y. Ding, *Tiacc: Triangle-inequality based hardware accelerator for k-means on fpgas*, in *CCGRID*, pp. 133–142, IEEE, 2021.

[321] Y. Huang, Y. Zhang, B. Feng, X. Guo, Y. Zhang, and Y. Ding, *A close look at multi-tenant parallel CNN inference for autonomous driving*, in *NPC*, vol. 12639 of *Lecture Notes in Computer Science*, pp. 92–104, Springer, 2020.

[322] Y. Wang, Z. Zeng, B. Feng, L. Deng, and Y. Ding, *Kpynq: A work-efficient triangle-inequality based k-means on FPGA*, in *FCCM*, p. 320, IEEE, 2019.

[323] Y. Wang, B. Feng, G. Li, S. Li, L. Deng, Y. Xie, and Y. Ding, *Gnnadvisor: An efficient runtime system for gnn acceleration on gpus*, .

[324] B. Feng, Y. Wang, and Y. Ding, *UAG: uncertainty-aware attention graph neural network for defending adversarial attacks*, in *AAAI*, pp. 7404–7412, AAAI Press, 2021.

[325] B. Feng, Y. Wang, and Y. Ding, *Saga: Sparse adversarial attack on eeg-based brain computer interface*, in *ICASSP*, pp. 975–979, IEEE, 2021.

[326] B. Feng, Y. Wang, X. Li, and Y. Ding, *Scalable adversarial attack on graph neural networks with alternating direction method of multipliers*, *CoRR* **abs/2009.10233** (2020).

[327] B. Feng, Y. Wang, X. Li, S. Yang, X. Peng, and Y. Ding, *Sgquant: Squeezing the last bit on graph neural networks with specialized quantization*, in *ICTAI*, pp. 1044–1052, IEEE, 2020.

[328] Y. Wang, B. Feng, X. Peng, and Y. Ding, *An efficient quantitative approach for optimizing convolutional neural networks*, in *CIKM*, pp. 2050–2059, ACM, 2021.

[329] K. Wan, S. Yang, B. Feng, Y. Ding, and L. Xie, *Reconciling feature-reuse and overfitting in densenet with specialized dropout*, in *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 760–767, 2019.

[330] F. Shi, H. Cao, Y. Wang, B. Feng, and Y. Ding, *Chatter detection in high-speed milling processes based on on-lstm and pbt*, in *The International Journal of Advanced Manufacturing Technology*, pp. 3361–3378, Springer Nature, 2020.

[331] L. Xie, S. He, Z. Zhang, K. Lin, X. Bo, S. Yang, B. Feng, K. Wan, K. Yang, J. Yang, and Y. Ding, *Domain-adversarial multi-task framework for novel therapeutic property prediction of compounds*, *Bioinformatics* **36** (01, 2020) 2848–2855, [https://academic.oup.com/bioinformatics/article-pdf/36/9/2848/33180831/btaa063.pdf].