Practical Coding-Theoretic Tools for Machine Learning Systems and by Machine Learning Systems

Jack Kosaian

CMU-CS-23-110

April 2023

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by: the National Science Foundation (NSF) under award numbers CCF1943409, CNS1956271, CNS1901410, CCF1910813, and CNS1850483; the NSF Graduate Research Fellowships Program (GRFP) award sponsored by the National Science Foundation; the TCS Presidential Fellowship at Carnegie Mellon University; and the Portuguese Science and Technology Foundation, FCT under award number 1031322. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: fault tolerance, machine learning, coding theory, machine learning systems

Abstract

Machine learning (ML) is now used in many domains, such as web services and safety-critical systems. This has led to the development of ML systems for deploying and training ML models. Beyond achieving high accuracy, ML systems must also use computing infrastructure efficiently and tolerate unreliable infrastructure.

Coding-theoretic tools enable many systems to operate reliably without the significant resource overhead that accompanies replication-based approaches. These tools are used in production storage and communication systems, and there is growing interest in their use for distributed computing.

This thesis explores the interplay between ML systems and practical applications of coding theory. Specifically, we show how ML systems can be made more reliable and efficient via novel uses of coding-theoretic tools, and how coding-theoretic tools can be expanded in reach and be made more efficient through techniques from ML and ML systems. We illustrate this interaction via multiple thrusts:

(1) We show how properties unique to ML systems can be exploited to efficiently integrate coding-theoretic fault tolerance techniques into ML systems. First, we reduce the execution-time overhead of fault-tolerant inference on GPUs by up to 5.3$\times$ by exploiting trends in neural network design and GPU hardware. Second, we show how coding-theoretic tools can be coupled with the unique properties of recommendation models to enable low-overhead fault tolerance in training.

(2) We demonstrate that co-designing coding-theoretic tools with ML systems offers new opportunities to extend these tools beyond prior limitations. Specifically, we enable resource-efficient fault tolerance in distributed prediction serving systems by using ML to overcome a key barrier in prior coding-theoretic tools.

(3) We identify opportunities for ideas inspired by coding theory to be used to improve the performance of ML systems, even when reliability is not a concern. We show that the throughput and GPU utilization of specialized convolutional neural network inference can be improved by up to 2.5$\times$ by combining images in a coding-theory-inspired manner and making small modifications to the model architecture.

(4) Finally, we show that the encoding and decoding functions of one popular class of coding-theoretic tools, linear codes, can operate at higher throughput and with little developer effort via advancements in ML systems. We show how similarities between operations in linear codes and those in ML libraries enable linear codes to be represented via ML libraries, and thus allow libraries for computing linear codes to adopt the many optimizations that have gone into ML libraries. This approach outperforms custom libraries for computing linear codes by up to 2.2$\times$.

Through these thrusts, this thesis demonstrates the promise of using coding-theoretic in ML systems and ideas from ML systems in coding-theoretic tools to bring about the next generation of efficient and reliable systems.

Acknowledgments

Glory be to the Father, and to the Son, and to the Holy Spirit.
As it was in the beginning, is now, and ever shall be, world without end.
Amen.

I have been blessed to have the support of many people throughout graduate school.

I would first like to express my gratitude to my advisor, Rashmi Vinayak. Rashmi has been a supportive advisor throughout our time working together and has given me much flexibility in selecting research projects that interest me. Early on during graduate school, Rashmi encouraged me to explore research that cuts across disciplines outside of computer systems. Though I was reluctant at first, this advice has proven fruitful—the work in this dissertation would not have been possible without Rashmi's fearlessness in diving into the intersection of computer systems, machine learning, and information theory. I am also grateful for Rashmi's patience as my circumstances and preferences both within and outside of work have changed.

I would like to thank the other members of my thesis committee: Phil Gibbons, Zico Kolter, Ion Stoica, and Pramod Viswanath. Their feedback has strengthened this dissertation.

I am grateful to Amar Phanishayee for his support during graduate school. Amar has been a mentor and research collaborator since my second year of graduate school, especially during my internships at Microsoft Research. Specifically relating to this dissertation, I am grateful to Amar for believing in my work when my own belief was wavering; without Amar's encouragement, I would not have persevered through the FoldedCNNs project.

I would like to thank the individuals who contributed to work within this thesis: Kaige Liu, Tianyu Zhang, and Juncheng Yang contributed to Chapter 5; Shivaram Venkataraman contributed to Chapters 6–8; Amar Phanishayee, Matthai Philipose, and Debadeepta Dey contributed to Chapter 9; and Jiyu Hu contributed to Chapter 10. I would also like to thank Lini Mestar and Hamsa Venkataram for our collaboration, which is not featured in this thesis.

I am grateful to the members of the TheSys Lab for their support and feedback: Sanjith Athlur, Saurabh Kadekodi, Francisco Maturana, Michael Rudow, and Juncheng Yang. I would also like to thank the members of the Parallel Data Lab (PDL) for facilitating a fun atmosphere for engaging in systems research and for feedback which has improved my work and presentations.

I am grateful to Amar Phanishayee and Matthai Philipose for their support during my internships at Microsoft Research, and to Haicheng Wu, Aniket Shivam, and Alan Kaatz for their support during my internship at NVIDIA. I am also grateful to my colleagues at NVIDIA for their support as I completed this dissertation.

The many logistics of graduate school have been made significantly simpler thanks to many people. I am particularly grateful for Deb Cavlovich's patience in answering my questions throughout graduate school and especially for helping me navigate administrative aspects unique to my final year. I am grateful to Jenn Landefeld and Tracy Farbacher for their patience in figuring out the many details related to my in-absentia defense. I am also grateful to Karen Lindenfelser, Joan Digney, and many others in PDL for making PDL events so enjoyable.

I would also like to thank those who helped begin my experience in research as an undergraduate at the University of Michigan: Albert Shih and Roland Chen; Vineet Kamat and Suyang

Dong; Mosharaf Chowdhury and Rashmi Vinayak (while she was at Berkeley). I would not have pursued graduate studies without the positive experiences these individuals facilitated.

Many friends have made my time in graduate school far more enjoyable. Matt Butrovich, Graham Gobieski, Michael Rudow, Han Zhang, and Giulio Zhou have been supportive and fun friends from early on in our time in graduate school. Anuj Kalia was a superb officemate and friend during the three years we overlapped. His sense of humor and skills in table tennis made my early years of graduate school much more fun.

The Catholic community in Pittsburgh has been a great blessing throughout graduate school. I am especially grateful to the priests of the Pittsburgh Oratory for their dedication to providing frequent opportunities to celebrate the Mass and to receive the Sacraments.

I have been blessed to have a wonderful family. My sisters, Alli and Ana, and their husbands have shown great love and support through the highs and lows that accompanied my time in graduate school. I am also grateful to my parents in law, William and Rosemary, as well as my brothers and sisters in law for their love and kindness.

I cannot adequately express the gratitude owed to my parents. From my childhood, my parents have been supportive of my many interests. This extended especially during the Ph.D. My parents quickly took it upon themselves to better understand what the Ph.D. entailed and patiently supported me as I navigated it. Thank you, Mom and Dad, for always making clear your unconditional love.

Finally, I am blessed to be married to my wonderful wife, Katherine. Walking this journey alongside Katherine has made both the highs and lows of graduate school much sweeter, and has provided a constant reminder for what truly matters in life. Katherine, thank you for your love, patience, faith, and friendship.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Machine learning (ML) models have become dominant workhorses in a variety of domains, including production and user-facing services [3, 15, 21, 78], scientific applications [8], and safety-critical systems [33]. This has led to the development of many *ML systems* that deploy and train ML models. ML systems range in settings from distributed training and serving systems in datacenters [109, 143, 244] to accelerators performing inference on edge devices [93, 115].

While the primary goal of many ML systems is to deliver high predictive performance in terms of accuracy, many systems must meet additional application-level objectives. For example, user-facing services must often meet strict latency objectives. Additionally, many autonomous systems that interact with the physical world must meet safety standards regarding the frequency of erroneous actions they take [22]. ML systems must therefore be designed both to maintain high accuracy from while also adhering to application-level constraints.

Meeting the application-level objectives of ML systems is made challenging by additional considerations related to the infrastructure on top of which ML systems run. This thesis focuses on two such considerations within the context of ML systems: (1) computing infrastructure is made of unreliable components, and (2) computing infrastructure must be used efficiently. We next describe each of these in greater detail.

1.1 Operating reliably atop unreliable infrastructure

Computer systems are built out of unreliable components: power sources can fail, nodes/jobs can be suddenly preempted, networks can drop packets, bits can be silently flipped, and processors can experience transient latency spikes. Designing computer systems that can tolerate unreliability has thus become a critical aspect of building dependable services. This has become even more important as computer systems have grown in scale [113]; as the number of components in a system increases, the probability that a single component in the system fails also increases.

Meeting the application-level objectives of ML systems frequently necessitates alleviating the effects of hardware unreliability. For example, prediction serving systems used for performing inference often must return predictions with low and predictable latency so as to meet the service-level objectives of user-facing applications. However, a transient slowdown occurring due to multitenancy in public cloud environments or a fail-stop failure due to a preemption or power outage may lead to an inflation in tail latency, and thus a violation of a latency objective. Such a system must, therefore, employ some means of alleviating the effects of slowdowns and failures.

Similarly, safety-critical systems deploying ML models, such as autonomous vehicles, must protect against bit flips in processing logic, which have been shown to cause neural networks to mispredict at rates that violate automotive safety standards [199]. Adhering to the safety standards required of these settings necessitates the use of approaches to tolerating bit flips.

Finally, even if a particular ML system itself is not affected by a particular type of failure, adding failure-detection capabilities to ML systems can assist in quickly identifying faulty hardware. For example, multiple datacenter-scale companies have reported an uptick in manufacturing errors that cause processors to persistently and silently produce incorrect results [121, 155]. Discovering these malfunctioning units is currently a tedious process that can take months, and the effects of these faults have caused errors in important services, such as file decompression [121]. If ML systems were equipped with fault-detection capabilities, they could be used to quickly flag faulty hardware to operators.

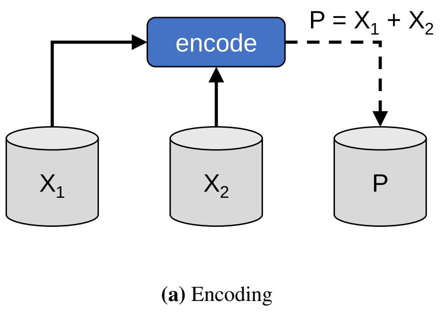
1.2 Using infrastructure efficiently

A second requirement of ML systems is to use computing infrastructure efficiently. Efficient use of infrastructure mandates allocating as few resources as needed to meet a target performance objective, while ensuring that those resources that have been allocated are well utilized. For example, a service should allocate as few servers as it needs to ensure that its tail latency is lower than its service-level objective, but should also strive to minimize the time any server is idle.

Efficiently utilizing computing infrastructure is particularly important for ML systems, as these systems are often run within large-scale computing systems on many servers and make use of specialized accelerators, which are often expensive and power hungry. Overallocating such resources for a particular ML system precludes others from using them in the shared computing infrastructures often employed by large organizations. Furthermore, poorly utilizing allocated infrastructure leads to a poor return on investment for purchasing and deploying accelerators. In a similar sense, ML systems operating on edge devices that poorly utilize hardware can lead to wasted power draw, which is often a scarce resource in these environments [115]. Finally, inefficiently utilizing infrastructure leads to wasted carbon emissions from the manufacturing of hardware, which has been shown to be a significant fraction of the overall carbon footprint of datacenter hardware [320].

1.3 Reliability and efficiency: a losing battle?

The presence of unreliable infrastructure and the need to use infrastructure efficiently each, in isolation, make it challenging to meet the application-level objectives of learning systems. However, the combination of these considerations further exacerbates this challenge: techniques used to increase reliability frequently oppose those used to increase efficiency (and vice versa). For example, ensuring reliable operation when running atop unreliable infrastructure necessitates the use of redundant resources to safeguard against failures (e.g., via replicating computation). How-



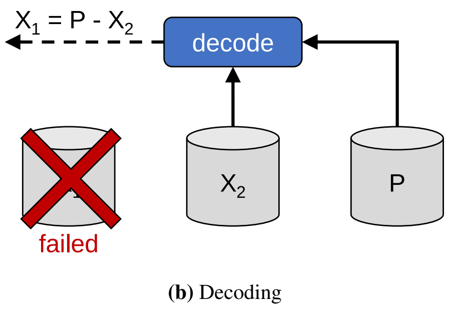


Figure 1.1: Example of (a) encoding and (b) decoding in an erasure-coded storage system with $k = 2$ data units X_1 and X_2 and $r = 1$ parity P.

ever, during normal operation (i.e., in the absence of failures), redundant resources are wasted, resulting in inefficient use of infrastructure. On the other hand, operating with the minimal resources needed to meet an application-level objective leaves a system unable to tolerate failures.

1.3.1 Coding-theoretic tools: hope for brokering peace

To straddle the conflict between reliability and resource efficiency, computer systems have long made use of ideas from coding theory. *Erasure codes* are key coding-theoretic tools used toward this goal [220]. Erasure codes enable a system to tolerate failures/unavailability while using significantly less redundant resources than replication-based approaches. These properties have made erasure codes an attractive alternative to replication-based approaches to reliability in storage, communication, and high-performance-computing systems (e.g., [160, 252, 263, 271, 313]). An erasure code encodes k data units to generate r redundant "parity units" such that any k out of the total $(k + r)$ data and parity units are sufficient for a decoder to recover the original k data units.[1] Therefore, erasure codes operate with a resource overhead of $\frac{k+r}{k}$, which is less than that of replication by setting $r < k$. Figure 1.1 shows an example of using an erasure code with $k = 2$ and $r = 1$ to protect against a single disk failure in a distributed storage system.

Leveraging coding-theoretic tools within ML systems has the potential to similarly enable reliable and resource-efficient operation.

1.4 Overview of thesis

The widespread use of coding-theoretic tools in storage, communication, and computing systems indicates promising potential for the use of such tools to enable resource-efficient reliability for ML systems. The goal of this thesis is to explore the use of coding-theory-inspired approaches for reliability and resource efficiency (both together and separately) across a range of ML systems. In addition, we also investigate how techniques from ML and ML systems can be used to improve coding-theoretic tools.

[1]These properties hold for a class of codes known as "maximum distance separable" (MDS codes). We focus on MDS codes because of their optimality and because many popular erasure codes in computer systems are MDS codes (e.g., Reed-Solomon codes).

Thesis statement: ML systems can be made more reliable and resource efficient through ideas inspired by coding theory, and coding-theoretic tools can be improved through the use of ideas from ML and ML systems.

Specifically, this thesis takes steps toward these goals through the following thrusts:

1. Investigating the interplay between traditional coding-theoretic tools and ML systems to build reliable ML systems
2. Extending the reach of coding-theoretic tools via co-design with ML systems
3. Identifying and exploiting opportunities to use coding-inspired ideas beyond reliability
4. Leveraging advancements in ML systems to accelerate erasure-coded systems

1.4.1 Investigating the interplay between traditional coding-theoretic tools and ML systems to build reliable ML systems (§4, §5)

We begin by illustrating how traditional coding-theoretic tools can be combined with the unique properties of ML systems to enable efficient reliability in these systems. We illustrate this through two ML systems:

Safety-critical neural network inference (§4). We first investigate the efficient use of a traditional coding-theoretic tool, coded computation [160, 195], to detect faults causing erroneous execution on GPUs. We focus on detecting faults caused by soft errors [134], transient errors in combinatorial logic that can result in silently computing erroneously (e.g., $2 + 2 = 3$). Soft errors have been shown to potentially cause neural networks to violate strict safety requirements [22, 199]. Fault tolerance against soft errors is, thus, necessary for systems deploying neural networks in safety-critical environments, such as autonomous vehicles [33], as well as in environments more prone to soft errors, such as in space [92, 115]. Furthermore, Facebook and Google have recently reported an uptick in similar events taking place in their datacenters due to faulty hardware, with Facebook reporting that these events have caused corruption in decompression workloads [120, 121, 155]. Thus, even in cases where a neural network can tolerate soft-error-induced faults, having the ability to efficiently detect faults when training and deploying neural networks can help operators quickly identify persistently faulty hardware that can cause other critical applications to fail.

Prior work has motivated the potential of "coded computation" to enable resource-efficient detection and correction of silent faults in certain classes of computations. Coded computation is a coding-theoretic approach that adds redundant computations employing carefully-designed mathematical structures, and exploits the invariants so introduced to detect faults with significantly lower execution-time overhead than replication-based approaches. The use of coded computation has been explored for fault detection in many applications, such as matrix multiplication (e.g., [86, 87, 160, 336]), LU decomposition [321], and sorting [202].

We show that naively applying coded computation as it has been traditionally used to the new domain of neural network inference leaves significant room for reduction in execution-time overhead. In particular, we show that the unique interplay between the arithmetic intensities of layers of neural networks and the compute and memory bandwidth of inference-optimized GPUs open new opportunities for efficiently performing coded computation, but that traditional approaches to coded computation do not exploit. Based on these insights, we develop *intensity-guided coded computation*, a new approach to leveraging coded computation for fault detection in neural network inference on GPUs that reduces execution-time overhead by 1.09×–5.3× compared to traditional approaches to coded computation across a variety of neural networks.

Fault-tolerant recommendation model training (§5). We next study the potential use of erasure codes in deep-learning-based recommendation model (DLRM) training systems to reduce the training-time overhead of fault tolerance. DLRMs are key tools in serving production, user-facing applications, and are frequently retrained to reflect the freshest data [20]. This makes the timeliness of DLRM training key to serving high-quality predictions to downstream applications. DLRMs are also typically trained in a large-scale, distributed fashion due to the large sizes of embedding tables in DLRMs [60, 169]. Given that DLRM training is resource and time intensive and that failures are common in large-scale settings, it is imperative for DLRM training to be fault tolerant [129, 221].

Checkpointing is the main approach used for fault tolerance in DLRM training [129, 221]. This involves periodically pausing training and writing the current state of training (e.g., model parameters) to stable storage, such as a distributed file system. If a failure occurs, the entire system resets to the most recent checkpoint and restarts training from that point. While simple, checkpointing frequently pauses training to save DLRM state and has to redo work after failure. Thus, checkpointing has been shown to add significant overhead to training production DLRMs; Facebook reports that checkpointing-related overheads add an average of 12% to DLRM training time [221]. Moreover, these overheads are expected to grow with DLRM size, posing a challenge for the training of future DLRMs.

An alternative to checkpointing that does not require pauses is to perform in-memory replication, in which two copies of DLRM parameters are kept up-to-date throughout training. However, given the large size of embedding tables in DLRMs (e.g., hundreds of gigabytes to terabytes), replicating DLRMs in this manner is impractical.

We show that erasure codes offer promise for achieving pause-free fault tolerance for DLRM training (unlike checkpointing) but with only a fraction of the memory overhead of replication. We make the case that the unique characteristics of DLRM training systems call for careful consideration of the use of erasure codes within these systems. For example, we show that an asymmetry between the memory capacity and network bandwidth consumed by certain DLRM parameters leads to it being preferable to take a hybrid approach to redundancy in which some parameters are erasure coded and others replicated. The result of our work is ECRM: a fault-tolerant DLRM training system that leverages erasure codes and replication in new ways to avoid pauses, and to continue training when recovering from failure, while gracefully scaling to larger DLRM sizes. For large DLRMs, ECRM reduces training-time overhead compared to checkpointing by up to 64%.

1.4.2 Extending the reach of coding-theoretic tools via co-design with ML systems (§6, §7, §8)

Neural networks employed in user-facing applications, such as web search and translation, must meet strict latency objectives to maintain satisfactory quality of experience. To support production load demands, these applications are deployed in datacenter settings via *prediction serving systems*. A prediction serving system contains multiple replicas of the same neural network on separate backend servers, and a frontend that distributes incoming queries to backends in a load-balanced fashion. However, within this highly-distributed, multi-tenant setting, a number of events frequently occur that can jeopardize system reliability, such as failures and transient server slowdowns. Left uncorrected, these events inflate a prediction serving system's tail latency and cause violations of latency objectives.

The second part of this thesis investigates the use of coded computation to protect against slowdowns and failures in prediction serving systems in order to maintain low tail latency.

Using coded computation within prediction serving systems raises challenges. Coded computation can be applied efficiently to the many linear operations within neural networks (e.g., matrix multiplication, convolution), but prior approaches are inapplicable for non-linear operations (e.g., activation functions, max pooling). This renders such approaches incapable of detecting/correcting faults across the execution of a neural network as a whole, which would be necessary when using coded computation in a prediction serving system.

This portion of the thesis proposes to overcome the aforementioned barrier of non-linear operations in coded computation through a *learning-based* approach to coded computation. Taking prediction serving systems as a driving application, we show that machine learning can be leveraged within the coded-computation framework to perform coded computation across neural networks as a whole. We propose two ways in which learning can be used to co-design coded-computation schemes with prediction serving systems to enable accurate and resource-efficient recovery of slow or failed predictions resulting from neural networks. Beyond the application of prediction serving, learning-based coded computation offers promising potential for the application of coded computation to more-general non-linear functions.

1.4.3 Identifying and exploiting opportunities to use coding-inspired ideas beyond reliability (§9)

The first two parts of this thesis focus on leveraging coding-theoretic tools to ensure the reliability of ML systems, while using resources efficiently. The third part of this thesis leverages observations made in the first two parts to show that coding-theory-inspired ideas can be used to boost the throughput and accelerator utilization of neural network inference *during normal operation,* even without any reliability concerns.

Specifically, we focus on the use of specialized convolutional neural networks (CNNs) for high-throughput inference in datacenter vision systems [156, 178, 179]. Specialized CNNs are small CNNs designed for highly-specific tasks (e.g., detecting whether a red truck is in a video frame). Through a study of specialized CNNs used at Microsoft, we find that specialized CNNs significantly underutilize accelerators even when using large batch sizes, resulting in suboptimal throughput and a poor return on accelerator investment. Related to insights identified in the first part of this thesis, we find that this low utilization is due to the low arithmetic intensity of specialized CNNs relative to the compute-to-memory-bandwidth ratios of accelerators.

To increase the accelerator utilization and throughput of specialized CNN inference, we design FoldedCNNs. FoldedCNNs take a new approach to CNN design that involves performing inference over multiple input images combined together in a coding-inspired fashion (and motivated, in part, by observations from the second part of this thesis), and widening layers of a CNN. This new design enables FoldedCNNs to increase GPU utilization by up to 2.8$\times$ and throughput by up to 2.5$\times$ for specialized CNN inference workloads.

1.4.4 Leveraging advancements in ML systems to accelerate erasure-coded systems (§10)

The final part of this thesis investigates how ML systems can accelerate erasure-coded systems.

Due to the widespread use of erasure codes in many production storage systems today (e.g., Ceph [9], HDFS [16], Azure Storage [159]), there is significant demand for optimized erasure-coding libraries: software platforms for performing encoding and decoding operations in an erasure code. However, developing high-performance erasure-coding libraries is a challenging process because it requires unique expertise in both the mathematical underpinnings of erasure codes and in techniques for achieving high performance on a given architecture. While this leaves the development of current erasure-coding libraries to a select few experts, we anticipate this challenge being exacerbated in the future, given the growing trend of hardware heterogeneity.

We make the case that the rise of fast ML libraries may serve as a lifeboat for easing the development of current and future optimized erasure-coding libraries: fast erasure-coding libraries across various hardware platforms can be easily implemented by using existing optimized ML libraries. We show that the computation structure of many erasure codes mirrors that common to matrix multiplication, which is heavily optimized in ML libraries. We show one can implement erasure codes using ML libraries in few lines of code and with little knowledge of erasure codes, while immediately adopting the many optimizations within these libraries, without requiring intimate knowledge of high-performance programming. We develop prototypes of our proposed approach using two different ML libraries targeting CPUs and GPUs. Our prototypes achieve up to 2.2$\times$ higher encoding throughput than state-of-the-art erasure-coding libraries.

1.4.5 Summary of contributions

Through these thrusts, this thesis demonstrates how coding-theoretic tools can be efficiently applied to ML systems, how coding-theoretic tools can be advanced by co-designing with ML systems, how coding-theory-inspired ideas can benefit ML systems beyond reliability purposes, and how ML systems can help accelerate traditional applications of coding-theoretic tools.

The findings of this thesis engage experts in various domains:

- **To the designer of ML systems:** this thesis illustrates the benefit of leveraging coding-theoretic tools to make ML systems reliable without significant resource overhead.
- **To the coding theorist:** this thesis provides evidence of the potential for leveraging machine learning to extend the reach of coded computation.

- **To the machine learning researcher:** this thesis motivates the potential to leverage mixtures of multiple inputs at inference time to improve the throughput of inference.
- **To the designer of erasure-coded systems:** this thesis motivates the exploration of how advances in ML systems can accelerate and ease the development of key routines in storage systems.

1.5 Organization

The remainder of this thesis is organized as follows:

Chapter 2 motivates the need for ML systems to operate reliably and efficiently. Various reliability concerns addressed in this thesis are introduced, along with their effects on ML systems. This chapter concludes by describing challenges and opportunities provided by jointly promoting reliability and resource efficiency in ML systems.

Chapter 3 provides background on coding-theoretic tools, coded computation, and the challenges of using coded computation in ML systems. Literature related to each of the contributions of this thesis is relegated to each of the chapters describing each contribution.

Chapter 4 presents our work on efficiently detecting silent data corruptions in neural network inference on GPUs by adapting coded computation to trends in neural network workloads and GPU architectures. This work was presented at the 2021 International Conference on High Performance Computing, Networking, Storage and Analysis (SC) [187]. The research artifact associated with this work is located at
`https://github.com/Thesys-lab/arithmetic-intensity-guided-abft`.

Chapter 5 presents our work on efficient fault tolerance for recommendation model training by exploiting the interplay between erasure codes and the unique performance characteristics of recommendation model training systems. This work is under submission [211].

Chapters 6, 7, and 8 present learning-based coded computation, in which machine learning is used to overcome the challenges of performing coded computation over nonlinear functions, and which enables efficient tolerance of slowdowns and fail-stop failures in prediction serving systems. This work was presented in part at the 2019 ACM Symposium on Operating Systems Principles (SOSP) [188], and in part in the IEEE Journal on Selected Areas of Information Theory [189]. The research artifact associated with this work is located at `https://github.com/Thesys-lab/parity-models`.

Chapter 9 presents FoldedCNNs, which leverage ideas inspired by coding-theory to increase the GPU utilization and throughput and specialized CNN inference. This work was presented at the 2021 International Conference on Machine Learning (ICML) [186]. The research artifact associated with this work is located at `https://github.com/Thesys-lab/folded-cnns`.

Chapter 10 presents our case for using ML libraries to ease the development of optimized erasure-coding libraries, along with supporting evidence through prototypes that outperform custom erasure-coding libraries. This work is under submission.

Chapter 11 provides concluding remarks and outlines future directions.

Chapter 2

Motivation: need for resource-efficient and reliable ML systems

In this chapter, we motivate the need for ML systems to be resource efficient and reliable.

We begin by describing the need for ML systems to be resource efficient, both in terms of minimizing resource overhead as well as in maximizing utilization. ML systems operate at large scale and with expensive, power-hungry infrastructure. Using more resources than necessary and poorly utilizing those resources that are provisioned starves other applications from resources, leads to wasted investments, and hinders efforts toward sustainable infrastructure.

We next describe the need for ML systems to be reliable. We describe three different unreliability events that affect ML systems: transient slowdowns, fail-stop failures, and silent data corruptions. The growing scale at which ML systems operate as well as the trend toward leveraging preemptible infrastructure increases the likelihood that ML systems will experience transient slowdowns and fail-stop failures. We describe how existing techniques for handling these forms of unreliability in ML systems are inefficient for both current and future ML systems. We then detail the causes of silent data corruptions and their effects on ML systems. Using both existing literature and our own experimentation, we highlight the critical effects that silent data corruptions can have on neural network inference applications. We additionally describe opportunities that could be enabled by being able to efficiently detect silent data corruptions in ML systems—even for ML systems that may themselves be unaffected by silent data corruptions.

We conclude this chapter by discussing the interplay between the goals of resource efficiency and reliability in ML systems. We note how these goals often act in opposition with one another, but that carefully investigating each goal within the context of ML systems can enable new opportunities to promote the other goal. These challenges and opportunities motivate multiple portions in the remainder of this thesis.

2.1 Need for resource-efficient systems

We first motivate the need for ML systems to be resource efficient. We begin by describing resource efficiency for general computer systems and then focus specifically on ML systems.

Resources and resource efficiency Computer systems make use of many resources, such as CPUs, memory, storage, networks, and accelerators (e.g., GPUs). Manufacturing, purchasing, and installing these resources incurs one-time, up-front costs (i.e., capital expenses), while powering and maintaining these resources incurs recurring costs (i.e., operational expenses).

Efficiently using resources requires that a computer system strive to (1) use as few resources as possible to meet a desired application-level objective, while (2) ensuring that those resources that are used are well utilized. For example, if a web service's traffic demands can be met by using four servers, it would be less efficient to use five servers. At the same time, even if only four servers are used, these servers will be inefficiently used if the service's software achieves significantly lower computational throughput (e.g., FLOPs/sec) than possible on the servers.

Throughout this thesis, we will refer to inefficiency related to item (1) above as *resource overhead*. A system has resource overhead when it has been allocated more resources than it needs to meet its application-level objective. We refer inefficiency related to item (2) above *underutilization*. A system underutilizes resources if it has been allocated some set of resources, but poorly utilizes the capabilities of these resources.[1]

Resource overhead has many negative effects. In a shared computing system, operating with resource overhead results in unnecessarily occupying resources that could be used by another service/job. For example, in a shared job scheduling system, jobs can experience significant queueing delay if they require four servers, but only three are available [166]. Resource overhead can also lead to wasteful operational expenses, as certain resources that are unallocated to a job (e.g., CPU cores) can often be placed either in a low-power mode or completely shut off. Finally, habitual resource overhead can lead to increases future capital expenses when determining the capacity requirements of future hardware purchases and installments.

Underutilization also has many negative effects. First, underutilization leads wasted capital expenses, as many resources are priced based on the theoretical peak performance they provide (e.g., FLOPs/sec of a GPU). Underutilization also leads to poor return on operational expenses for powering inefficiently utilized resources. While there are techniques to reduce energy consumption during underutilization (e.g., power gating), it is often challenging to reap benefits from these techniques when underutilization manifests in a fine-grained manner (e.g., when waiting on data from a slower level of the memory hierarchy) [70, 165]. While context switching between tasks can improve utilization, doing so at fine-grained time scales is also challenging [75].

In order to operate in a resource-efficient manner, computer systems must strive to minimize resource overhead and maximize utilization.

Effects of resource inefficiency on ML systems While every computer system should ideally use resources efficiently, the increasing scale at which ML systems operate makes resource efficiency particularly important for ML systems. For example, current approaches to training

[1]The reader may note that there is a relationship between resource overhead and underutilization. For example, one could say that a system underutilizes servers when it has been allocated more servers than needed to achieve its application-level objective (i.e., when it has resource overhead). We choose to differentiate between resource overhead and underutilization in this thesis to highlight a difference between resource allocation and usage of allocated resources. We use resource overhead when referring to a system making allocation requests: "I need five servers for this job." We use underutilization when referring to the usage of allocated resources: "I have been allocated a processor that can achieve 10 TFLOPs/sec., but I only maintain an average of 5 TFLOPs/sec."

large language models require up to three thousand NVIDIA A100 GPUs [236]. Inference workloads similarly have a growing resource footprint—the compute capacity used for inference has grown at Facebook by 2.5$\times$ in the past 1.5 years [320]. Similarly, Google has noted that inference workloads accounted for three fifths of Google's energy usage on machine learning from 2017–2021 [251].

Operating at this large scale requires a significant investment in infrastructure. ML systems often use custom accelerators [172] and server setups [237]. These specialized resources are expensive to develop and to manufacture, increasing the cost of platforms for large-scale ML systems. Thus, operating this infrastructure inefficiently leads to a poor return on investment.

In addition to the monetary costs of deploying ML systems, ML systems also have high environmental footprints. Of particular interest from the perspective of resource efficiency is the environmental impact of manufacturing hardware used for ML systems, which Facebook estimates to occupy around 50% of total carbon emissions in the ML lifecycle, and which is expected to exceed emissions from operating resources in the future [320]. Thus, underutilizing hardware results in a poor return on the one-time, up front manufacturing energy cost. Google has also specifically highlighted the need to increase utilization to improve energy efficiency [251]. Alongside underutilization, resource overhead also leads to wasted manufacturing, as it requires procuring more resources to support an organization's overall infrastructure needs.

Beyond large-scale datacenter ML systems, resource efficiency is also important for ML systems that operate at the edge. Many edge systems have limited power sources and strive to be inexpensive to develop and deploy [115, 139]. Operating these edge systems inefficiently can require the procurement of additional hardware resources to meet application-level demands, which increases the price and power draw of these devices.

Therefore, to reduce both the monetary and environmental costs of deploying ML systems, ML systems must efficiently use resources.

2.2 Need for reliable ML systems

We now provide background on unreliability in ML systems. We describe the types of unreliability events that will be addressed in this thesis and motivate the need to address them.

We use the term "unreliability event" to characterize the event in which a computer system deviates from its expected operation. This deviation can manifest in multiple ways:

1. *Fail-stop failure*: a computer system stops operating at all. For example, a fail-stop failure occurs when the power source to a processor is cut off, causing the processor to shut down.
2. *Slowdown*: a system continues operating, but operates slower than expected. For example, a processor may transiently take 30 milliseconds instead of the expected 10 milliseconds to perform a function due to the arrival of a new tenant in a multi-tenant system.
3. *Silent data corruption*: a system continues operating at its expected rate, but produces incorrect results (e.g., computing $2 + 2 = 3$). These errors are referred to as "silent" because they are produced without any notification to the overarching application.

We first motivate and provide background on fail-stop failures and slowdowns in §2.2.1 and then discuss silent data corruption in §2.2.2.

2.2.1 Slowdowns and fail-stop failures

We begin by describing slowdowns and fail-stop failures. We group these two unreliability events together because, as will be shown in §7 and §8, one can often leverage the same techniques to remedy both slowdowns and fail-stop failures.

Transient slowdowns. Many services are built atop the expectation that a computer system will reliably return the results of its computation within a certain time. For example, a web service may expect a database query to complete within 10 milliseconds. However, there are many events that can cause a computer system to transiently return results slower than expected. Examples of these events include resource contention due to multitenancy in cloud settings [151, 161, 325] as well as complex interactions between the application and runtime/operating system (e.g., garbage collection [246, 328]). Within the context of ML systems, transient slowdowns have also been shown to be caused by routines such as loading a new model in a serving system [244].

Transient slowdowns have long been a concern in large-scale services [113]. For example, Google found that increases in search latency of only 100–400 milliseconds reduced the number of daily searches per user by 0.2% to 0.6% [89]. Akamai has similarly reported that a 100 millisecond delay in page load times reduces website conversions by 7% [53].

Within ML systems, transient slowdowns are primarily of concern for user-facing and/or real-time inference systems. A primary example of a platform within this domain are *prediction serving systems*. Prediction serving systems host models for inference and deliver predictions for input queries [7, 14, 23, 26, 109, 244]. We describe prediction serving systems in greater detail in §6; what is relevant for the present discussion is that prediction serving systems operate in a large-scale, distributed fashion in cloud/datacenter settings in order to keep up with the high load of production web services [109].

Similar to other latency-sensitive services, prediction serving systems must adhere to strict service-level objectives (SLOs) (e.g., return predictions within tens of milliseconds [109]). Queries that are not completed by their SLO are often useless to applications [61]. In order to reduce SLO violations, prediction serving systems must minimize tail latency. However, meeting SLOs is made challenging by transient slowdowns, which can inflate tail latency in prediction serving systems. For example, we illustrate in §8 that various forms of resource contention in a prediction serving system can lead to tail latencies that are up to three times slower than median latency. The presence of such transient slowdowns in prediction serving systems makes it challenging to maintain predictable latency for user-facing applications.

Fail-stop failures. There are a number of events that cause fail-stop failures in computer systems. For example, a component of a computer system reaching the end of its useful lifetime and failing to operate properly can cause a fail-stop failure [245]. Within the context of distributed systems, a broken network connection between one node in the system and the remaining nodes also often manifests as a fail-stop failure. Other events, such as detectable, but uncorrectable errors in a device's memory can trigger a computer system to restart, which manifests as a fail-stop failure for many applications [123]. Outside of hardware unreliability, bugs in software can also cause fail-stop failures [117].

Furthermore, the use of preemptible resources also leads to fail-stop failures. Many major cloud-service providers offer preemptible compute instances that can often be obtained for a lower price than a dedicated instance, but that can be reclaimed by the cloud-service provider at any time (e.g., "spot instances" in Amazon Web Services [5]). This reclamation terminates all application logic running atop such a preemptible instance, resulting in a fail-stop failure. Similarly, many organizations leverage preemptible scheduling when time-sharing resources [166, 324]. Preemption within such time-sharing systems results in a fail-stop failure of either part or all of the resources used by an application.

The probability that a system will experience a fail-stop failure increases with the number of computer nodes the system uses. This poses a challenge for the growing trend of distributing the training of large-scale machine learning models. For example, efficient systems for training certain large-scale language models can use over three hundred compute nodes and a total of over three thousand GPUs [236]. Given the large number of components in such systems, fail-stop failures due to hardware unreliability are to be expected. Furthermore, the growing trend of training machine learning models atop preemptable infrastructure (either via "spot instances" or within internal time-shared clusters at an organization) heightens the need for mechanisms to tolerate fail-stop failures in ML systems.

Given the possibility of fail-stop failures and the long duration of many training jobs, distributed training jobs must implement some means of tolerating fail-stop failures. This is particularly critical for models that are frequently retrained to reflect the freshest data in a production deployment, such as recommendation models [20].

Growing need for efficient slowdown and fail-stop-failure tolerance. While there is a pressing need for techniques to mitigate transient slowdowns and fail-stop failures in ML systems, current approaches are costly both in terms of resource usage and execution-time overhead.

Within the context of distributed prediction serving systems, the defacto approach to protecting against transient slowdowns is to issue redundant requests to multiple replicas of a given neural network. However, as we will discuss in greater detail in §8, such approaches require at least $2\times$ resource overhead or incur high latency in recovering from slowdowns.

Within distributed neural network training systems, the most popular means of safeguarding against fail-stop failures is checkpointing, in which training is periodically paused so as to save the current model parameters to a stable storage system, such as a distributed file system [221]. In the event of a fail-stop failure, the training system reloads the most recent checkpoint from stable storage and continues training from the point of that checkpoint. However, checkpointing can incur significant training-time overheads incurred due to such periodic pausing during normal training and rolling back and redoing training iterations on recovery. For example, Facebook has reported that overheads from checkpointing account for, on average, 12% of recommendation model training time, and that these overheads add up to over 1000 machine-years of computation [221]. Facebook has also recently reported that checkpointing can add significant strain to shared network and storage resources [129]. Given that the overhead of checkpointing typically grows with model size, and that popular language and recommendation models have continued to grow in size, the overheads of checkpointing are slated to grow in the future: for example, current large-scale language models require checkpoints of size up to 13.8 TB [236]. Beyond this, Google recently reported that checkpointing overheads amount to the equivalent of wasting 45 software engineer per year [46].

Takeaway. The increasing likelihood of transient slowdowns and fail-stop failures due to the increasing scale of ML systems and trends toward leveraging preemptible resources, coupled with the inefficiency of current approaches to mitigating these events calls for the investigation of new, efficient approaches to tolerating transient slowdowns and fail-stop failures.

2.2.2 Silent data corruption

We next provide background on silent data corruptions. As previously described, a silent data corruption is an event in which a particular computation is corrupted such that it returns an incorrect result without notifying the overarching application that corruption took place.

In this section, we provide general background on silent data corruptions and their causes. We describe how silent data corruptions have become a growing concern for large-scale datacenter operators, such as Google and Facebook [120, 121, 155]. Through both prior work and our own experiments, we illustrate the negative effects that silent data corruptions have on safety-critical ML systems: prior work has shown that silent data corruptions cause many neural networks to produce mispredictions at a rate that violates strict automotive safety requirements [199]; we perform a fault-injection study on a set of neural networks used for air-collision control in autonomous airplanes and find that, on average, a single silent data corruption reduces the accuracy of the network from 96.9% to 93.5%, and certain bit flips reduce accuracy by as much as 20%. We finally illustrate the many opportunities that could be opened up by providing efficient techniques for tolerating silent data corruptions in ML systems: efficiently triaging faulty hardware, safety-critical ML systems, and the use of low-cost components in space systems.

We begin by describing the causes of silent data corruptions. When describing these causes, it is important to distinguish between their immediate effects and their long-term effects. As will be shown below, events that can potentially trigger a silent data corruption begin by triggering a more immediate change in hardware, such as a bit flipping in the output register of a logic unit. These immediate effects are typically referred to as “single-event upsets” (SEUs). The overall effect of an SEU on an application depends on application-level logic; an SEU could cause the application to crash or experience divergent control flow, it could be masked and have no effect on the application, or it could cause silent data corruption.

With the distinction between SEUs and silent data corruptions in mind, we will next highlight a subset of events that cause SEUs:

1. Manufacturing defects
2. Cosmic radiation
3. Voltage and frequency scaling

Cause: manufacturing defects. Defects in the manufacturing of devices can lead to SEUs. For example, both Google and Facebook have recently reported an uptick in processors in their datacenters that consistently produce an incorrect result when performing a given computation [121, 155]. In some cases, these effects have been shown to corrupt file decompression [121]. The silent data corruptions considered by these organizations differ from the sources of silent data corruptions that will be discussed below: these manufacturing defects result in repeatable silent data corruptions, whereas other SEUs typically result in transient silent data corruptions that are unlikely to reoccur if a computation is repeated.

Cause: cosmic radiation. Cosmic radiation is one of the most well-known causes of SEUs. In particular, the collision of high-energy neutrons from Earth's atmosphere with a transistor can generate sufficient electrical charge to cause the transistor to flip [106]. These events can affect both memory (e.g., SRAM, DRAM) and processing logic (e.g., adders, AND gates) [134] of a broad spectrum of devices, such as CPUs [122, 243], GPUs [122, 264, 291], FPGAs [62, 122, 208], and custom ASICs (e.g., Edge TPUs [265]).

The environment in which a device operates plays a significant role in determining the frequency of cosmic-radiation-induced SEUs. Of particular importance is the altitude at which a device operates. At higher altitude, Earth's atmosphere shields less cosmic radiation, causing an increase in neutron flux, and thus potential cosmic-radiation-induced SEUs. For example, devices operating at the altitude commonly flown by commercial airlines experience a rate of SEUs 300 times higher than at sea level [106]. Certain military-grade aircraft flying at 60,000 ft. above the North Pole experience a rate of SEUs 2,000 times higher than that at sea level [106]. Even on ground, altitude has a noticeable effect on the rate of SEUs; the first recorded SEUs on ground occurred at Los Alamos National Laboratory, which sits 7,200 ft. above sea level [240].

Cosmic-radiation-induced SEUs are even more prevalent within outer space. For example, for spacecraft operating in low Earth orbit, which is the orbit of the International Space Station as well as that commonly used by nanosatellites [115], reports indicate that over one hundred SEUs occur in a single device per day [81]. The position of a device within orbit also affects the radiation experienced by a device. For example, devices are subject to a significantly higher rate of SEUs when passing through Earth's South Atlantic Anomaly [286], a region above part of South America in which a portion of Earth's radiation belt comes closest to Earth's surface, and which significantly increases particle flux. Radiation effects outside of Earth's orbit can be even more severe. For example, NASA's upcoming mission to Europa, a moon of Jupiter, must cope with radiation far more intense than that of spacecraft within low Earth orbit [13]. Finally, events such as solar flares can lead to bursts of SEUs, with some observations showing an increase in the rate of SEUs by as much as eight times during a solar flare [81].

Cause: voltage and frequency scaling. SEUs can additionally be caused by techniques aimed at increasing the energy efficiency of a device. One such technique is *undervolting*. The basic idea of undervolting is as follows: For a given clock frequency, processors require a specific minimum voltage to ensure that circuits propagate signals within a clock cycle. If supplied a voltage below this minimum, signals may not propogate within a clock cycle, which can lead to incorrect values being saved in flip-flops, and thus causing SEUs. To ensure correct operation in the face of transient voltage swings that occur based on compute load, the default supply voltage of a device is typically set well above this minimal voltage. Undervolting is the practice of lowering the voltage supplied to a device such that it is closer to the true minimum required supply voltage of the device [198].

By minimizing the excess voltage used in guarding against voltage fluctuations, undervolting can enable one to operate at the same frequency, but with lower energy consumption, leading to increased energy efficiency [198]. However, reducing this voltage guardband also leaves one more vulnerable to SEUs due to signal-propagation-related timing violations [94, 198, 294, 336, 340]. Undervolting has also recently been exploited as a potential attack vector to inject SEUs into specific layers of a neural network during inference [292].

We now describe the effects of SEUs on ML systems. Recall that an SEU can either:

1. Cause a program to crash or to experience divergent control flow
2. Be masked by application logic
3. Cause silent data corruption.

We next highlight each of these effects, paying closest attention to the effects of silent data corruptions on ML systems, which will be the focus of the portions of this thesis devoted to tolerating SEUs in ML systems.

Effect: program crashes and control flow divergence. Certain SEUs cause a program to crash, such as one that changes the value of a pointer, resulting in a segmentation fault. Additionally, systems using single-error-correction, double-error detection (SECDED) error-correcting codes in memory often force an application to stop when a double-bit error is detected [123, 243].

An SEU can also cause divergent control flow in an application. For example, an SEU in the device's program counter or occurring during a jump instruction could cause the device to begin executing instructions in an order that disobeys the correct control flow of the program.

A common approach to tolerating crashes and control-flow divergence is by leveraging watchdog processes that detect or restart an application upon a crash, or detect when a program violates prespecified control-flow invariants (e.g., which basic blocks of a program can be entered after a given basic block) [215, 242]. When a control-flow invariant is violated, an error is raised, which may trigger a program crash. Thus, when detected, such errors can be handled by techniques that tolerate fail-stop failures. Checking control-flow invariants is a particularly good fit for ML systems, as neural networks typically exhibit highly-predictable control flow (e.g., always execute layer 2 after layer 1). Given the availability of these techniques, we do not focus on program crashes and divergent control flow resulting from SEUs in this thesis.

Effect: masked errors. Other SEUs are masked by application-level logic. An SEU will only manifest as silent data corruption if it propagates to a result of the computation being performed by an application. There are many reasons why SEUs might be masked, and thus not result in silent data corruption. For example, an SEU causing a bit flip in a register would be masked if the register is next written before the value is read. Within the context of ML systems, many layers of neural networks mask certain SEUs. For example, an SEU that causes one output element of a convolution to change from -1 to -2048 will be masked if the next layer of the neural network is a ReLU (since the result of ReLU for any negative input is 0) [199].

Effect: silent data corruption. SEUs that do not result in crashes or control-flow divergence and that are not masked by application-level logic result in silent data corruption in the output of the application. However, the net effect of silent data corruption on an application depends on the error tolerance of the application. For example, silent data corruptions in critical functions such as encryption, compression, and erasure coding must be avoided, as silent data corruptions in these applications may result in storing corrupted data. Similarly, silent data corruptions in the linear algebra subroutines of many scientific simulations can have critical effects on the net result of the simulation [88].

The effect of silent data corruptions on ML systems depends on the type of ML system and its application. For example, neural network training systems are unlikely to be effected by silent data corruptions, as the iterative optimization procedures used to train neural networks can likely correct small perturbations caused by silent data corruptions. Thus, silent data corruptions are more of a concern for neural network inference systems. It is within this context that we focus the remainder of our discussion on silent data corruptions.

Even for inference systems, not all silent data corruptions are critical. Consider classification tasks as an example. The net classification for an input is typically given as the class with maximal value in the output vector of a neural network. For such tasks, a silent data corruption only changes the prediction made by the neural network if it causes the class with the maximal value to change. For applications that involve predicting in a more-continuous space, such as predicting a bounding box in object detection, the impact of an silent data corruption must be evaluated based on a metric of distance between the corrupted output and the fault-free output [124].

Numerous studies have analyzed the impact of silent data corruption on neural network inference by using controlled fault injection experiments as well as experiments in which a device is placed under a neutron beam to increase the number of cosmic-radiation-induced SEUs experienced in a short experimental window (e.g., [124, 199, 214, 222, 292, 306]). Other studies have shown that silent data corruptions can be induced via targeted voltage-scaling attacks, which can reduce the accuracy of ResNet-18 on CIFAR-10 from around 97% to as low as 36% [292].

A popular application for studying the effects of silent data corruptions on ML systems is safety-critical systems. These applications typically must adhere to strict functional safety standards that dictate their acceptable failure tolerances. For example, the ISO 26262 standard dictates that a fully autonomous vehicle must experience fewer than 10 failures in time (FITs; failures per billion hours) in hardware components [22]. Prior work has shown that the normal rate of silent data corruptions induced by cosmic radiation at sea level causes some popular neural networks to mispredict at a rate that exceeds the FIT rate required by ISO 26262 [124, 199]. It is, thus, necessary for these systems to employ some means of tolerating silent data corruptions.

Case study on autonomous aircraft systems. We study the impact of silent data corruptions on a set of neural networks proposed for the Airborne Collision Avoidance System X for unmanned aircraft (ACAS Xu) [173]. Specifically, we consider a recently-proposed version of the ACAS Xu neural networks [167]. This neural network consists of five fully-connected hidden layers with

50 neurons each, with ReLUs after each fully-connected layer, and a final fully-connected layer projecting onto five output classes. We exhaustively inject a single bit flip into the activations of the neural network that follow a fully-connected layer and precede a ReLU. We use 32-bit IEEE 754 floating point values for all activations and weights. We record whether the bit flip results in a predicted class that differs from the correctly predicted class. We consider only those samples which the untainted neural network correctly classifies. This experiment results in a fault injection run for each bit in each post-fully-connected activation in each correctly-classified sample.

On average, a single bit flip in this network reduces accuracy from 96.9% to 93.5%. As expected, bit flips in more significant bits are more likely to cause a misprediction (e.g., bit flips in the exponent bits in a floating point); for certain bits, the misprediction rate resulting from a bit flip exceeds 20%. While the overall drop in accuracy is significant in its own right, it is compounded by the scale of aircraft systems today. Reliability studies indicate that, within a fleet of 200 aircrafts leveraging common FPGA systems, an SEU will occur every three hours [52]. Scaling this up to the 5400 aircrafts managed by the United States Federal Aviation Administration at peak hours [1] indicates a significantly higher rate of SEUs, and thus mispredictions. The effect of these mispredictions would be even worse when considering so-called "stuck-at faults" in which a bit remains permanently set at either 0 or 1, thus affecting all inferences [306].

Coupling our study on the impact of silent data corruptions on autonomous aircraft with the significant existing literature on the safety-standards-violating effects of silent data corruptions on autonomous vehicles indicates that silent data corruptions can have significant negative effects on critical ML systems.

As described above, the criticality of SEUs depends on the usecase of a ML system. We next describe opportunities that could be enabled by detecting/correcting SEUs in ML systems:

1. Efficiently discovering and diagnosing faulty hardware
2. Safety-critical ML systems
3. Deployment of commercial off-the-shelf (COTS) components in space

Opportunity: discovering and diagnosing faulty hardware. As described above, manufacturing defects can cause devices to produce silent data corruptions. This has recently been noted by both Google and Facebook as affecting their datacenter hardware, in which a particular device consistently returns an incorrect value for a particular computation (e.g., executing $2 + 7$ always produces 10 on a specific core of a specific processor) [121, 155]. These errors have have lead to real application-level bugs, such as the inability to decompress files [121]. Similar to compression, these silent data corruptions could lead to data corruption if they occurred in the cryptography and erasure-coding routines of a storage system. These manufacturing defects are difficult to discover and triage, as evidenced from the painstaking process performed by Facebook engineers to find the device and settings resulting in corrupted compression described above [121]. This has led to a growing desire to have "always-on" fault detection for production applications [120]

Adding redundancy checks to production applications—even those that are not particularly concerned about silent data corruptions—could enable the timely discovery and isolation of faulty hardware by reporting more instances of silent data corruptions. However, traditional approaches to detecting silent data corruptions, such as replication, require significant redundant computation, and thus significant additional infrastructure. Thus, adding replication to all production applications is impractical.

Efficient alternatives to replication that target broad classes of computations leveraged in datacenter applications could allow one to detect hardware errors more quickly, without significant infrastructure overhead. Developing such alternatives for ML systems would be particularly useful given the large footprint of ML systems in datacenters today. Thus, even applications of ML systems that are not particularly susceptible to silent data corruptions could take part in discovering faulty hardware that would cause serious corruption for another application.

Opportunity: safety-critical ML systems. As described above, whether a misprediction resulting from silent data corruption truly matters for a particular ML system depends on the criticality of the application; a misprediction in an advertisement recommendation system is likely less critical than one in an autonomous vehicle. However, there is an increasing push to leverage ML systems in autonomous safety-critical systems, such as autonomous vehicles and aircraft [173, 214]. As we described previously, these applications must adhere to strict functional safety standards (e.g., ISO 26262 [22]). We have previously shown through our own study, as well as through prior work, that silent data corruptions can have significant negative effects on ML systems employed in safety-critical domains. Thus, techniques will be needed to tolerate silent data corruptions to enable future safety-critical ML systems.

Opportunity: deployment of commercial off-the-shelf (COTS) components in space. As described previously, the effects of cosmic radiation are more pronounced within outer space [81]. Future space missions are expected to experience an even higher rate of SEUs than experienced in low-Earth orbit, such as NASA's upcoming mission to Europa [13].

At the same time, there is a growing desire to deploy more computationally-expensive applications atop spacecraft. Within the realm of ML systems, recent proposals aim to use neural networks to perform imaging on satellites [115] and for scientific simulations on the International Space Station to reduce the amount of data that needs to be transmitted to Earth [57, 314].

While spacecraft often leverage radiation-hardened hardware for performing control operations, such devices are not currently practical for supporting such computationally-expensive tasks for multiple reasons: First, the rate of progress of radiation-hardened hardware significantly lags behind the current state-of-the-art hardware. This means that space systems cannot benefit from new edge hardware accelerators such as the Edge TPU [12] or those within the Qualcomm Snapdragon [44]. Second, radiation-hardened hardware is typically more expensive and bulkier than commercial hardware. This poses a challenge to the increasing trend of deploying small, inexpensive nanosatellites [115].

Thus, recent efforts have explored the use of commercial off-the-shelf (COTS) hardware on spacecraft [57, 191, 314]. Deploying COTS components requires employing some means of fault tolerance to ensure reliable operation. The current approach used for fault tolerance in COTS components is triple-modular redundancy (TMR), in which three copies of a computation are run on separate devices, and voting is performed to detect/correct errors. However, TMR comes with significant up-front cost in hardware purchase and results in high energy consumption, which limits the durations of space missions. Thus, alternative techniques are needed for efficiently tolerating silent data corruptions in COTS components on spacecraft.

2.3 Interplay between reliability and resource efficiency

Within this chapter, we have established that ML systems must *both* operate in a resource efficient manner and operate reliably. The interplay between these two requirements gives rise to both a challenge and an opportunity.

Challenge: balancing the need for redundancy with the desire for low resource overhead. As was briefly described in §2.2, a common technique used to make computer systems more reliable is to leverage redundant resources. We will describe this in greater detail in §3. However, leveraging redundant resources leads to high resource overhead; by definition, "redundant" resources are not strictly necessary. Beyond resource overhead, redundancy can also negatively affect application-level objectives, such as latency requirements. For example, a neural network inference system on a satellite that performs inference twice serially for each incoming image would have $2\times$ the latency. Thus, ML systems that leverages redundancy to improve reliability must attempt to minimize the number of redundant resources used as well as the impact on application-level objectives.

Opportunity: leveraging underutilization for more efficient redundancy. While the need for redundancy and the desire for low resource overhead are often in conflict with one another, underutilization can give rise to opportunities to employ redundancy with lower resource overhead and with less effect on application-level objectives. As will be described in §4 and §9, it is challenging to highly utilize many of the resources used in ML systems, even when multiplexing these resources among multiple tenants. This opens up an opportunity for redundant operation for reliability purposes to use the resources that a non-redundant ML system would otherwise waste. In some cases, such resources can essentially be used "for free" and with little impact on application-level objectives.

This chapter has motivated the need for both reliability and resource efficiency in ML systems, as well as the challenges and opportunities in achieving these requirements. These challenges and opportunities motivate some of the contributions of this thesis: we will describe novel techniques to enable ML systems to exploit the aforementioned opportunity caused by underutilization to improve reliability (§4, §5); we will propose approaches to navigate the aforementioned tension between redundancy and resource overhead (§7, §8); and, we will combine insights from the first two parts of this thesis to improve the resource efficiency of ML systems even absent reliability concerns (§9). To begin, we will provide background on the types of redundancy-based approaches we will consider in §3.

Chapter 3

Background: coding theoretic tools and coded computation

Having motivated the need for both resource efficiency and reliability in ML systems, we now provide background on the class of techniques that we will leverage and build upon in this thesis to achieve this goal: coding-theoretic tools. We provide a high level overview of coding-theoretic tools and their benefits, describe coded computation, one of the coding-theoretic tools we focus on in this thesis, and finally conclude by describing the challenges in applying coded computation to ML systems.

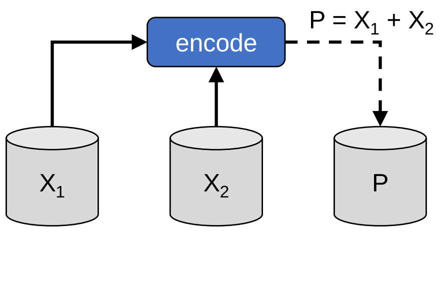
3.1 Leveraging redundancy for fault tolerance

As was briefly described in §2.2, a common approach to building fault-tolerant computer systems is to leverage redundant resources. The simplest form of redundancy is replication, in which a given routine is performed multiple times to detect or correct faults. For example, to protect against a fail-stop failure or transient slowdown of a server in a distributed prediction serving system, inference requests could be replicated to two identical copies of the same neural network on separate servers. To detect a silent data corruption, a system may execute a neural network twice and compare the results from each execution.

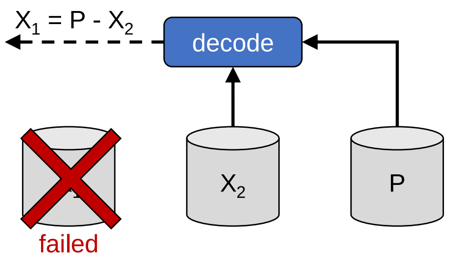
While simple, replication can incur significant overheads. In terms of resource overhead, a replicated system leveraging redundant hardware requires the allocation of at least two times as many resources as a non-replicated system. In cases where replicated execution is performed on the same hardware as the original computation, replication can add significant execution-time overhead. These properties make replication unattractive for building reliable ML systems both due to the significant cost and demand for hardware in large-scale ML systems as well as due to the tight latency requirements of many ML systems.

3.2 Coding-theoretic tools

Many computer systems have turned to the use of coding-theoretic tools for resource-efficient fault tolerance. In this section, we provide a high-level overview of the use of coding-theoretic



(a) Encoding



(b) Decoding

Figure 3.1: Copy of Figure1.1, placed here for ease of reference. Example of (a) encoding and (b) decoding in an erasure-coded storage system with $k = 2$ data units X_1 and X_2 and $r = 1$ parity P.

tools in computer systems and their benefits. We specifically focus on erasure codes and error-correcting codes, referring to these as "codes," collectively. As this thesis deals only with applications of codes within computer systems, we avoid using formal and mathematical notation in descriptions here, favoring examples.

Figure 3.1 shows an example of using an erasure code, a popular coding-theoretic tool, in a distributed storage system in which one would like to reliably store two data units X_1 and X_2. In this example, the erasure-coded storage system provisions an extra disk, on which it stores a "parity unit" that is constructed as $P = X_1 + X_2$ through the erasure code's "encoding function." Suppose that a single fail-stop failure of the disk storing X_1 occurs. The erasure-coded storage system could reconstruct X_1 as $X_1 = P - X_2$, using the code's "decoding function." It is easy to see that the system can recover from the fail-stop failure of any one of the three disks in the system. In a similar way, the decoding function could be used to detect if silent data corruption has occurred in any one of the three disks.

More generally, the codes and we will consider in this thesis operate over k data units and construct r parity units, and will be able to recover from any r fail-stop failures or will be able to detect any r silent data corruptions.

Codes are often preferable to replication-based systems because they can enable the same level of fault tolerance, with lower resource overhead. For example, the erasure-coded storage system in Figure 3.1 can tolerate the same number of fail-stop failures as a replication-based system (one), but does so with only 50% additional servers. Due to this lower resource overhead, codes have been widely deployed in many systems (e.g., [160, 252, 261, 263, 271, 313]).

3.3 Using codes for fault-tolerant computation

Leveraging coding-theoretic tools for fault-tolerant computation systems raises additional challenges compared to the use of codes for storage and communication systems. As many of the contributions of this thesis involve the use of coding-theoretic ideas for computation systems, we now provide additional background on this setting.

The technique of using coding-theoretic tools for fault-tolerant computation systems goes by multiple names, typically being referred to as "coded computation" in coding theory literature and as "algorithm-based fault tolerance" (ABFT) in high-performance-computing literature. For uniformity, we refer to these techniques as "coded computation" throughout this thesis. Fig-

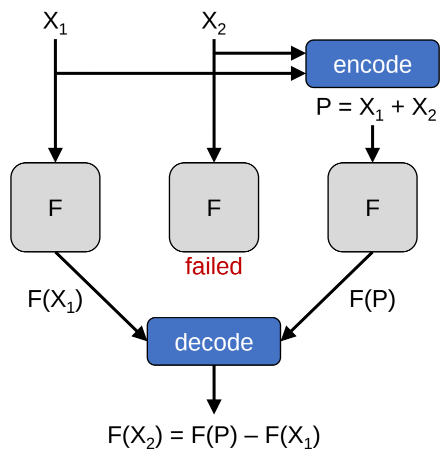


Figure 3.2: Example of coded computation with $k = 2$ and $r = 1$.

Table 3.1: Toy example with parity $P = X_1 + X_2$ showing the challenges of coded computation on non-linear functions.

$\mathcal{F}(\mathbf{X})$	$\mathcal{F}(\mathbf{P})$	**Desired** $\mathcal{F}(\mathbf{P})$
$2X$	$2X_1 + 2X_2$	$2X_1 + 2X_2$
X^2	$X_1^2 + 2X_1X_2 + X_2^2$	$X_1^2 + X_2^2$

ure 3.2 shows an example of coded computation with $k = 2$ and $r = 1$. Under coded computation, rather than the goal being to store/retrieve data units, as in storage systems, or to transmit data units, as in communication systems, the goal is to *compute over data units*. As shown in Figure 3.2, data units X_1 and X_2 are each to be operated on by computation $\mathcal{F}$ in order to produce $\mathcal{F}(X_1)$ and $\mathcal{F}(X_2)$.

The goal of coded computation is to make this system fault tolerant. To do so, coded computation constructs a parity unit P via an encoding function, and operates over P with $\mathcal{F}$ to produce $\mathcal{F}(P)$. In the example in Figure 3.2, encoding is performed as $P = X_1 + X_2$. Then, supposing a fail-stop failure of the computation being performed over X_2 occurs, the *output of computation over* X_2 is reconstructed using the code's decoding function. In the example in Figure 3.2, decoding is performed as $\mathcal{F}(X_2) = \mathcal{F}(P) - \mathcal{F}(X_1)$.

Challenges in coded computation. Coded computation is straightforward when the underlying computation $\mathcal{F}$ is a *linear* function. A function $\mathcal{F}$ is linear if, for any inputs X_1 and X_2, and any constant a: (1) $\mathcal{F}(X_1 + X_2) = \mathcal{F}(X_1) + \mathcal{F}(X_2)$ and (2) $\mathcal{F}(aX_1) = a\mathcal{F}(X_1)$. Many of the codes used in traditional applications, such as Reed-Solomon codes, can recover from unavailability of any linear function [195]. For example, consider having $k = 2$ and $r = 1$. Suppose $\mathcal{F}$ is a linear function as in the first row of Table 3.1. Here, even a simple parity $P = X_1 + X_2$ suffices since $\mathcal{F}(P) = \mathcal{F}(X_1) + \mathcal{F}(X_2)$, and the decoder can subtract the available output from the parity output to recover the unavailable output. The same argument holds for any linear $\mathcal{F}$. However, a non-linear $\mathcal{F}$ significantly complicates the scenario. For example, consider $\mathcal{F}$ to be the simple non-linear function in the second row of Table 3.1. As shown in the table, $\mathcal{F}(P) \neq \mathcal{F}(X_1) + \mathcal{F}(X_2)$, and, even for this simple function, $\mathcal{F}(P)$ involves complex non-linear interactions of the inputs that make decoding difficult.

Due in part to the seamless fit of existing codes to linear computations, there has been a large body of work developing coded-computation techniques for linear computations, such as matrix multiplication (e.g., [86, 87, 160, 195, 227, 336]), convolution [128], and other iterative methods [99, 321]. In contrast, fewer codes have been developed that support non-linear computations. Even those that do are limited to supporting only polynomial functions and require resource overhead that is prohibitive for many practical applications [272, 334]. Of particular note for this thesis, current approaches to coded computation are unable to support cases in which $\mathcal{F}$ is a *neural network*. While neural networks do contain many linear operations (e.g., matrix multiplications, convolutions), they also contain many non-linear operations (e.g., activation functions, max pooling) that render the entire function computed by a neural network non-linear. Thus, the existing approach used for employing coded computation to neural networks is to perform coded computation over the linear layers of a neural network and to replicate the non-linear layers. While such layer-wise decomposition is applicable to detecting/correcting faults on a single device (e.g., detecting silent data corruptions on a single GPU), it is inefficient for settings such as prediction serving systems, in which it would require per-layer inter-server communication that would slow down operation.

The next four chapters of this thesis focus on the use of coding-theoretic tools in ML systems. Chapter 4 focuses on detecting silent data corruptions in neural network inference on GPUs via novel techniques to efficiently perform coded computation for neural network inference when using the technique described above of splitting the neural network into linear and non-linear components. Chapter 5 then enables efficient fault tolerance for recommendation model training by pairing coding-theoretic ideas to the unique characteristics of recommendation model training. Chapters 6, 7, and 8 then focus on tolerating fail-stop failures in prediction serving systems by proposing novel approaches that enable coded computation to be performed over an entire neural network as a whole, including the non-linear layers, without layer-wise decomposition.

Chapter 4

Efficient coded computation for neural network inference on GPUs

This chapter provides an example of the reduction in overhead made possible by taking advantage of specific properties of ML systems when employing coding-theoretic tools for reliability purposes. We illustrate this through the example of applying coded computation to detect silent data corruption in neural network inference on GPUs. Specifically, we focus on the setting described in §3.3 in which coded computation is performed over linear layers of a neural network and replication is performed for non-linear layers. We show that existing approaches to coded computation for linear layers in neural networks fail to account for the diversity of arithmetic intensity and the high compute-to-memory-bandwidth ratios (CMR) of inference-optimized GPUs. We then develop a new approach to coded computation for linear layers that adapts the coded-computation scheme used depending on the arithmetic intensity of the layer and the CMR of the GPU. Compared to traditional approaches to coded computation, the proposed "intensity-guided coded computation" reduces execution-time overhead by 1.09–5.3$\times$.

4.1 Introduction

Chapter 2 described the many causes of soft errors in computer systems, as well as their potentially adverse effects on neural networks. This description motivated the need to protect against soft errors for safety-critical ML systems, as well as the potential benefit of being able to detect soft errors within ML systems so as to quickly discover and triage faulty hardware.

In each of these scenarios, neural networks must be equipped with some means of detecting faults. However, tolerating faults caused by soft errors requires performing redundant execution (e.g., replication and comparison). For fault tolerance to be practical, it is critical that redundant execution operate with low overhead in terms of execution time and cost.

In this work, we focus on software-based approaches for detecting faults that occur in processing logic during neural network inference on GPUs. We focus on detection, rather than correction, as detecting a catastrophic event is often more important to an application than quickly proceeding after such an event [149]. We focus on GPUs because they are commonly used for neural network inference in both cluster and edge settings, including in emerging space applica-

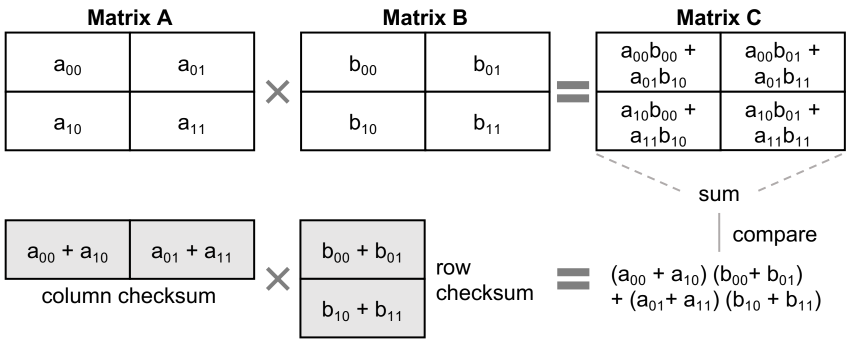


Figure 4.1: Toy example of coded computation for matrix multiplication with $M = N = K = 2$.

tions [115, 314]. We focus on faults that occur in processing logic, rather than in the memory hierarchy, as many modern systems contain ECC-protected memory hierarchies [54]. In contrast, processing logic is not as amenable to lightweight hardware fault tolerance [77].

We specifically focus on the use of coded computation to detect faults (see §3 for background on coded computation). Figure 4.1 shows a toy example of coded-computation-protected matrix multiplication between matrices A and B of size 2×2 to produce output matrix C. Coded computation constructs a *column checksum vector* by summing each column of matrix A and a *row checksum vector* by summing each row of matrix B. It is straightforward to see that the result of taking the dot product of these checksum vectors should, in the absence of a fault, equal the summation of all entries of output matrix C, which we refer to as the output summation. Correspondingly, comparison between the checksum dot-product result and the output summation can detect a single fault in C.

Multiple recent works have explored leveraging coded computation to make neural networks fault tolerant [149, 201, 247, 342]. Since existing coded-computation techniques support only linear computations, these approaches use coded computation for the linear operations of neural networks (e.g., fully-connected and convolutional layers, which are often executed as matrix multiplications), and replicate nonlinear operations (e.g., activation functions). We similarly focus on using coded computation for linear layers implemented as matrix multiplications in this work, and use the terminology “linear layer” to refer to fully-connected and convolutional layers.

Key to efficient operation in any approach to redundant execution is identifying and exploiting underutilized resources. If the computation-to-be-protected underutilizes certain compute units, redundant execution can potentially be performed on those units without adding much execution-time overhead. However, existing approaches to coded computation typically only assume that computations being protected are compute bound, and thus aim to minimize the amount of redundant computation they perform.

In this work, we first present a case for challenging this assumption based on trends in GPU hardware and neural network design: The introduction of processing units optimized for neural networks (e.g., Tensor Cores [38]) has led to an unprecedented increase in FLOPs/sec in inference-optimized GPUs. However, such GPUs have had a far less profound growth in memory bandwidth. This results in inference-optimized GPUs having high compute-to-memory-bandwidth ratios (CMRs). High CMRs require kernels to have high arithmetic intensity to keep computational units highly utilized. However, many convolutional and fully-connected layers in neural networks have *low arithmetic intensity*. Furthermore, many efforts toward reducing neural network latency, such as efficient neural network design [295], model specialization [156, 178, 179, 232, 283], and pruning [83], further reduce arithmetic intensity.

These trends result in many linear layers in neural networks that have arithmetic intensity far lower than the CMR on GPUs, rendering such layers memory-bandwidth[1] bound, rather than compute bound. Such layers are unable to keep computational units highly utilized, opening opportunities for redundant execution to be performed for free. However, current approaches to coded computation for neural network inference, which are well-suited for compute-bound linear layers, cannot exploit this opportunity to squeeze in redundant execution alongside the computation being protected.

To better exploit this nascent opportunity, we (1) investigate coded-computation schemes that exploit the unused computation cycles of linear layers on inference-optimized GPUs, which we refer to as thread-level CC, and (2) propose a new, adaptive approach to coded computation, called intensity-guided CC, that selects among thread-level CC and traditional approaches to coded computation on a per-layer basis, using the layer's arithmetic intensity as a guide.

To design an approach to coded computation that exploits the unused computation cycles of linear layers on inference-optimized GPUs, the key approach we leverage is to perform coded computation at the smallest unit of the parallel subproblem performed by the matrix multiplication for a layer. As illustrated in Figure 4.2, high-performance matrix multiplication on GPUs involves decomposing the overall matrix multiplication into a hierarchy of subproblems across threadblocks, warps, and, at the smallest level, threads. Existing approaches to coded computation for neural network inference on GPUs, which we term "global CC," generate checksums over the full input matrices to minimize the amount of redundant computation performed in checksum dot products. In contrast, we leverage a coded-computation scheme in which each thread performs coded computation over the small matrix multiplication subproblem it is responsible for. We refer to this approach as *thread-level CC*. Under thread-level CC, each thread computes coded-computation checksums and dot products on the fly in tandem with its computation of the original matrix multiplication, and performs its own thread-local checksum equality check.

The approach taken in thread-level CC may at first appear counterintuitive, as it performs more redundant computation than global CC: thread-level CC performs coded computation over many small, thread-local matrix multiplications, whereas global CC performs coded computation over one large matrix multiplication. In fact, thread-level CC results in multiple threads each computing identical checksums (e.g., in Figure 4.1, identical column checksums for threads that compute elements in the same rows in C). However, we show that, through careful design decisions, this approach is effective in exploiting the gaps in compute utilization of bandwidth-bound linear layers. This approach also eliminates any additional loads/stores, which would compete with the matrix multiplication itself for memory bandwidth, which is the bottleneck resource. The net result is low execution-time overhead for bandwidth-bound linear layers.

As described above, thread-level CC primarily benefits linear layers that are bandwidth bound. In contrast, it is not well-suited for compute-bound linear layers, for which global CC suffices. As we show in §4.3, neural networks contain *both bandwidth- and compute-bound linear layers*, making one-size-fits-all approaches inefficient.

Therefore, we propose *intensity-guided CC*, an adaptive coded-computation approach that

[1]We refer to memory-bandwidth-bound layers as "bandwidth-bound" for short.

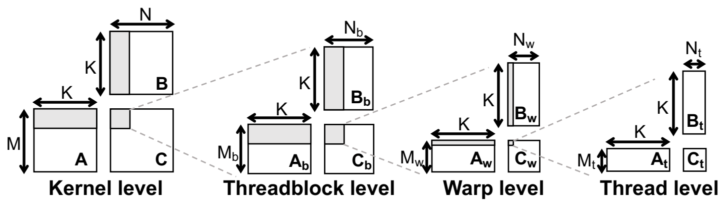


Figure 4.2: Hierarchical matrix multiplication. Shaded regions show inputs/outputs used in the next level.

selects among global CC and thread-level CC for each linear layer of a neural network depending on which approach offers the lowest execution-time overhead, letting the arithmetic intensity of the layer and CMR of the device guide such selection.

We implement and evaluate intensity-guided CC atop CUTLASS [31], a high-performance linear algebra and machine learning library for GPUs from NVIDIA. We evaluate execution-time overhead on the inference-optimized NVIDIA T4 GPU when using Tensor Cores. We consider eight popular convolutional neural networks (CNNs), two neural networks used within recommendation models (DLRM) [237], and four CNNs developed through model specialization and used for video analytics [179]. Compared to an optimized approach to global CC [149], intensity-guided CC reduces execution-time overhead by up to 2.75× for popular CNNs, up to 4.55× for DLRMs, and up to 5.3× for specialized CNNs. These results show the promise of taking an arithmetic-intensity-guided approach to coded computation to reduce the overhead of fault tolerance in neural network inference.

The code used in this chapter is available at:
`https://github.com/Thesys-lab/arithmetic-intensity-guided-abft`.

4.2 Background

In this section, we provide background on matrix multiplication on GPUs and how coded computation is performed and optimized for neural network inference. We refer to §2 for background on the need for and benefits of fault detection in neural network inference.

4.2.1 Efficient matrix multiplication on GPUs

As described in §4.1, our focus is on redundant execution for the convolutional and fully-connected layers of neural networks, which we refer to as "linear layers." For the remainder of this chapter, we describe these operations as matrix multiplications, as high-performance implementations of these layers are often achieved through matrix multiplications [31]. However, the approaches we propose can apply to other implementations as well.

Within this setting, we denote a linear layer as the multiplication of matrix A of size $M \times K$ by matrix B of size $K \times N$ to produce an output matrix C of size $M \times N$. Matrix A contains the inputs to the layer (e.g., activations from the previous layer). Matrix B contains the learned weights of this layer. Weights (matrix B) are known a priori, while activations (matrix A) are known only during computation. Output C contains the output of the layer, which will be fed to the next layer, typically after being operated on by an activation function (e.g., ReLU).

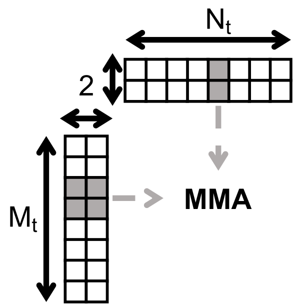


Figure 4.3: One step in the K dimension for a thread using m16n8k8 Tensor Core operations (MMAs). A total of $\frac{M_t N_t}{2}$ MMAs are performed per step, one for each pair of two consecutive rows of A_t and one column of B_t.

GPU terminology. We use NVIDIA's terminology [11] in describing the architectural components and programming abstractions of GPUs. A GPU consists of a number of streaming multiprocessors (SMs), each of which has many cores on which computation is performed along with a register file and shared memory region. Computation is executed on GPUs in kernels consisting of many threads. Threads are grouped into threadblocks, with all threads in a threadblock executing on the same SM and able to communicate with one another via shared memory. Groups of 32 threads within a threadblock execute in lockstep as a so-called warp. Each thread executes on an individual core, except when using Tensor Cores, new processing units that enable warp-wide collaborative execution of matrix multiplications [38].

Hierarchical matrix multiplication. High-performance implementations of matrix multiplication on GPUs decompose the problem solved by the kernel into a number of sub-matrix multiplications solved by threadblocks, warps, and threads. Figure 4.2 shows an example of this decomposition: each threadblock is responsible for computing a subset of C, which it decomposes into subsets to be computed by warps of the threadblock, each of which in turn decomposes the problem into subsets to be computed by individual threads. We denote the portions of A and B used by a thread as A_t and B_t, respectively. A_t is of size $M_t \times K$ and B_t is of size $K \times N_t$.

As our focus is on neural network inference, we focus on low-precision (e.g., FP16) matrix multiplications on Tensor Cores, which are heavily used for accelerating inference. Our description follows the use of such operations in CUTLASS. We focus in particular on the FP16 m16n8k8 Tensor Core operation, though our discussion and proposed solutions apply to other Tensor Core operations as well.

Each m16n8k8 Tensor Core operation is a warp-wide operation that multiplies a 16×8 matrix A_{tc} by an 8×8 matrix B_{tc} and accumulates results into a 16×8 output matrix C_{tc} (we use subscript "tc" to denote Tensor Core operands/outputs) [37]. Each thread in the warp provides four elements of A_{tc} and two elements of B_{tc} to the operation, and obtains four output elements of C_{tc} from the operation. We refer to one such m16n8k8 matrix-multiply-accumulation operation as an "MMA," following NVIDIA's terminology [37].

CUTLASS leverages MMAs within the hierarchical matrix multiplication framework described above. Each thread walks down the K dimension of the problem and loads an $M_t \times 2$ chunk of A_t and a $2 \times N_t$ chunk of B_t. These loaded chunks are then used in $\frac{M_t N_t}{2}$ MMAs, each of which uses two rows of the loaded chunk of A_t and one column from the loaded chunk of B_t from each thread, as shown in Figure 4.3. The results of these operations are accumulated into the thread's $M_t N_t$ registers that store the partial accumulation of the thread's matrix multiplication output. CUTLASS uses standard optimizations to overlap loading the next chunks of A_t and B_t while the current MMAs are performed (e.g., double buffering).

4.2.2 Fault model

We next shift our focus to fault detection for matrix multiplication. To set the stage, we first describe the fault model we consider.

We focus on detecting a single faulty output value in matrix C. We focus on detection, rather than correction, as being able to detect a catastrophic event is often more important than being able to quickly continue after such an event [149]. Following prior work [100, 247, 342], we focus on detecting a single fault because the execution of one layer in a neural network is short enough that the likelihood of more than one soft error occurring during execution is low.

We focus on faults occurring due to soft errors in the processing logic of a GPU. We do not focus on faults in the memory hierarchy, such as in global memory, caches, shared memory, register files, and busses, as these components are more easily protected by ECC [54]. In contrast hardware fault tolerance for processing units is more expensive, typically requiring dual-modular-redundant circuits [77]. As described in §2.2.2, we also assume that control logic on the GPU is protected. This fault model is in line with prior work [100, 200].

4.2.3 Coded computation for matrix multiplication

Coded computation for matrix multiplication typically operates by (1) generating a $1 \times K$ column checksum vector of matrix A and a $K \times 1$ row checksum vector of matrix B, (2) performing the dot product between the column checksum vector and row checksum vector, (3) summing all entries of the output matrix C, and (4) comparing the values generated in (2) and (3) above. Approaches to coded computation typically generate a single column checksum for the entire input matrix A (and similarly for B) [149, 342]. We thus term such approaches "global CC." Global CC results in the minimum additional dot-product computations required for fault detection in matrix multiplication, making it well-suited for compute-bound matrix multiplications.

While we focus on detecting a single fault, coded computation also supports detecting multiple faults. To do so, coded computation generates multiple checksum columns and rows based on independent linear combinations of columns/rows. In this scenario, multiple output checksums are also generated based on these linear combinations and compared to checksum dot products. The approaches to coded computation that we propose in this work can also handle higher fault rates in this way.

4.2.4 Optimizing global CC for neural network inference

Recent works leverage global CC to protect the linear layers of neural networks, and add multiple neural network-specific optimizations [149, 201, 342], which we describe next. Recall that, for neural network inference, matrix A contains input activations and B contains layer weights. We therefore refer to the column checksum of matrix A as the "activation checksum" and the row checksum of matrix B as the "weight checksum."

Offline construction of weight checksum. Since operand B contains the layer's weights, which remain the same for every inference request, the weight checksum of each linear layer in a neural network can be constructed once offline and reused for every inference request [149, 201, 342]. The same does not hold for the activation checksum of operand A, because its contents change for each inference request.

Checksum fusion. Recent work [149] fuses the generation of the output summation used in the coded-computation check to the end of the matrix multiplication kernel. Kernel fusion reduces the amount of data that must be read from memory to form the output summation, which speeds up checksum generation. As the next layer's input A is generated by the current layer, the current layer can also fuse the generation of the next layer's activation checksum to the end of its matrix multiplication kernel (after the activation function is applied) [149].

Flow of coded computation in neural network inference. With these optimizations, the workflow of a coded-computation-protected linear layer is as follows: (1) perform matrix multiplication to generate output C, (2) perform fused output summation generation, (3) apply the layer's activation function to C, (4) perform fused next-layer activation checksum generation, (5) launch a kernel that performs the coded-computation dot product for the current layer and compares the results to the output checksum generated in Step 3. Steps 1–4 must take place sequentially, while Step 5 can take place in parallel with the next layer of the neural network. Step 5 occurs in a separate kernel because it involves a global reduction over the partial checksums generated by threadblocks.

By minimizing redundant computation, global CC offers low execution-time overhead for compute-bound linear layers. However, we next identify trends in GPU hardware and neural networks that lead to many linear layers being memory-bandwidth-bound. This opens new opportunities for efficient redundant execution that current approaches to coded computation for neural network inference are unable to exploit.

4.3 New opportunities for efficient redundant execution

Critical to reducing execution-time overhead for any approach to redundant execution is discovering opportunities to exploit unused resources. In this section, we identify trends in GPU hardware and neural network design that create new, currently unexploited opportunities for efficient redundant execution in neural network inference.

4.3.1 Resource bottlenecks for GPU kernels

GPU kernels are typically either bound by computational throughput or by memory bandwidth. A popular model to determine whether a kernel is compute or memory-bandwidth bound is comparing the the *arithmetic intensity* of the kernel to the *compute-to-memory-bandwidth ratio* (CMR) of the device [32, 317]. Under this model, a kernel is compute bound if the theoretical amount of time it spends performing computation is greater than the theoretical amount of time it spends loading/storing data from/to memory:

$$\frac{\text{FLOPs}}{\text{Compute Bandwidth}} > \frac{\text{Bytes}}{\text{Memory Bandwidth}}$$

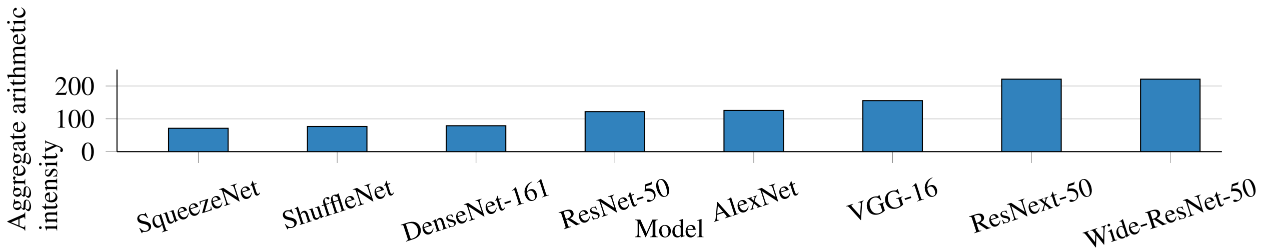


Figure 4.4: FP16 aggregate arithmetic intensity of CNNs on images of size 1080×1920 at batch size of one.

Here, "FLOPs" is the number of arithmetic operations performed by the kernel, "Bytes" is the amount of data it transfers to/from memory, "Compute Bandwidth" is the GPU's peak FLOPs/sec, and "Memory Bandwidth" is the GPU's memory bandwidth (bytes/sec). Rearranging this inequality to pair properties of the kernel on the left-hand side and properties of the GPU on the right-hand gives:

$$\frac{\text{FLOPs}}{\text{Bytes}} > \frac{\text{Compute Bandwidth}}{\text{Memory Bandwidth}} \tag{4.1}$$

The left-hand ratio of Equation 4.1 is the kernel's arithmetic intensity: the ratio between the FLOPs the kernel performs and the bytes it transfers to/from memory. The right-hand ratio is the GPU's CMR.

Takeaway. From the lens of this performance model, it is clear that the arithmetic intensity of a given kernel and CMR of a given GPU play key roles in determining opportunities for redundant execution to leverage unused resources. For example, a kernel with low arithmetic intensity running on a GPU with a high CMR will likely be bandwidth bound and underutilize compute units. This leaves opportunities for redundant execution to leverage such units without hampering the performance of the kernel itself.

We next examine trends in GPU hardware and neural network design to identify opportunities for such efficient redundant execution.

4.3.2 Wide range of arithmetic intensities among neural network layers

We first examine the arithmetic intensities of current neural networks and their individual linear layers under various operational settings. In this analysis, we consider only "linear layers", such as convolutional and fully-connected layers, which are often implemented as matrix multiplications. Other operations, such as activation functions, are typically fused to these linear layers and contribute far less to overall arithmetic intensity and execution time.

The "aggregate arithmetic intensity" of a neural network as a whole is computed by summing the FLOPs performed across all linear layers, summing the bytes read/written across all linear layers, and dividing these quantities. This metric provides an estimate of whether the neural network as a whole is more compute or memory-bandwidth bound.

Figure 4.4 shows the FP16 aggregate arithmetic intensities of eight widely-used CNNs from the popular PyTorch Torchvision library [48].[2] The figure shows a wide range of aggregate

[2]We replace the group convolutions in ShuffleNet and ResNext-50 with non-grouped convolutions to ease their conversion to matrix multiplications. The reported aggregate arithmetic intensities of these neural networks are, thus, higher than they would be with grouped convolutions, which typically decrease arithmetic intensity.

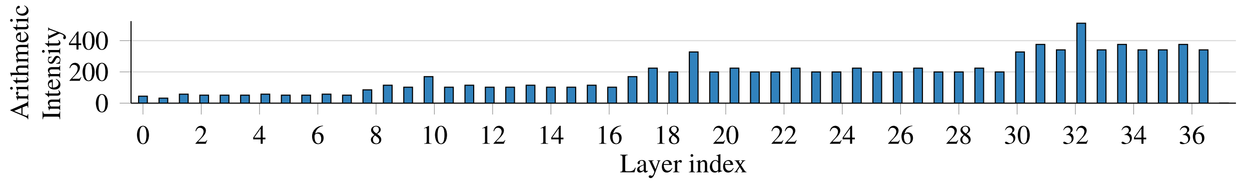


Figure 4.5: FP16 arithmetic intensity of convolutional and fully-connected layers of ResNet-50 on HD images (resolution 1080×1920) with batch size of one.

arithmetic intensities among such CNNs (from 71 to 220).

Furthermore, many domains leverage neural networks that are significantly smaller than those described above, and thus have even lower aggregate arithmetic intensities. For example, neural networks used for recommendation serving, such as Facebook's popular DLRM [237], leverage small neural networks consisting of a few fully-connected layers. Consequently, these neural networks have low aggregate arithmetic intensities (e.g., 7 in FP16).

Figure 4.5 shows the arithmetic intensities of individual convolutional and fully-connected layers of ResNet-50. As illustrated, there is a wide range of arithmetic intensities (1–511) among even various linear layers of the same neural network (other neural networks are similar).

Finally, arithmetic intensity also varies with settings of the applications in which neural networks operate, such as the size of inputs to the neural network. For example, increasing the batch size used in inference typically increases arithmetic intensity by amortizing the overhead of loading neural network weights from memory. Thus, the many applications that use small batch sizes for low-latency inference are likely to have low arithmetic intensity [105, 341], while those that can aggressively batch inputs may have higher arithmetic intensity. For example, the FP16 aggregate arithmetic intensities of the neural networks used in DLRM increase from 7 at batch size of 1 to 70–109 at batch size 256. For CNNs, the resolution of input images also affects arithmetic intensity for similar reasons, as operating over large images amortizes the cost of loading convolutional filters from memory. For example, the FP16 aggregate arithmetic intensity of ResNet-50 is 72 when operating over images of resolution 224×224 (the resolution typically used for ImageNet [273]), but increases to 122 when operating over images of resolution 1080×1920 (typically considered HD).

Takeaway. Neural networks exhibit wide variance in arithmetic intensity across neural networks, across individual linear layers within a neural network, and across application settings. This renders some neural networks, some layers, and some application settings likely to underutilize computational resources.

4.3.3 Inference-optimized GPUs have high CMR

We now discuss trends in CMR, the right-hand ratio in Equation 4.1.

GPUs have been a workhorse for neural networks since the early 2010s [190]. Recent GPUs have further bolstered neural network acceleration by adding hardware units specifically designed for the matrix multiplications found in neural networks, such as NVIDIA's Tensor Cores [38]. These hardware units offer unprecedented performance in terms of FLOPs/sec, particularly when using low-precision arithmetic (e.g., FP16), as is common in neural network inference.

For example, the inference-optimized T4 GPU offers 65 FP16 TFLOPs/sec [55], a marked increase from the 11 FP16 TFLOPs/sec offered by its predecessor, the P4 [40], which did not contain Tensor Cores. Such high performance is also offered in other server-grade GPUs, such as the V100 and A100 GPUs, which offer 125 and 312 FP16 TFLOPs/sec, respectively [27, 54]. This trend has also made its way to edge devices, as GPUs in the NVIDIA Jetson family now offer up to 32 INT8 TOPs/sec via Tensor Cores, whereas predecessors were bound to single-digit INT8 TOPs/sec [35].

While the FLOPs/sec offered by inference-optimized GPUs has drastically increased, memory bandwidth has not increased at the same rate. For example, while the T4 GPU increases FP16 FLOPs/sec by 5.9$\times$ compared to the P4 GPU, it offers only a 1.7$\times$ increase in memory bandwidth. Similar trends hold for other GPUs.

The net result of these trends in compute and memory bandwidth is a *significant increase in the CMR of GPUs*. For example, the FP16 CMR of the T4 GPU is 203, while that of the P4 was 58. Even GPUs with high-bandwidth memory (e.g., HBM2) have high CMRs (139 and 201 in FP16 for V100 and A100, respectively [27, 54]), as do edge GPUs (235 in INT8 for Jetson AGX Xavier [35]).

Takeaway. The introduction of specialized hardware units for matrix multiplications that drastically increase computational throughput, paired with a significantly slower increase in memory bandwidth, results in inference-optimized GPUs with high CMRs. This trend "raises the bar" for GPU kernels, *making them more likely to be memory-bandwidth bound and underutilize GPU compute units.*

4.3.4 Many neural network optimizations reduce arithmetic intensity

A secondary trend further exacerbates the growing bandwidth-bound nature of many neural networks: designing small neural networks to perform tasks with high throughput or low latency. The neural networks shown in Figure 4.4 are large, general-purpose models designed to classify a wide variety of objects (e.g., from ImageNet [273]). There is a growing body of work on designing more efficient neural network architectures that can accomplish the same task as such general-purpose neural networks, but with a significantly smaller model. There are many techniques along these lines, including efficient neural architecture search [295, 348], pruning [83], and model specialization [156, 178, 179, 232, 283]. These techniques often result in deploying neural networks with lower aggregate arithmetic intensity than the general-purpose neural networks shown in Figure 4.4.

For example, in model specialization for offline video analytics, a small, specialized CNN is designed to answers specific queries (e.g., find red trucks), and which consults a larger, general-purpose CNN only when unsure [156, 179, 283]. By targeting a focused query, specialized CNNs can typically be made smaller and faster than general-purpose neural networks, but have lower aggregate arithmetic intensity: the specialized CNNs from the NoScope video analytics system [179] have FP16 aggregate arithmetic intensities of 15–53, even with large batch size.

Takeaway. Current trends in efficient neural network design result in neural networks that have lower arithmetic intensity, making current and future workloads likely to underutilize GPU compute units.

4.3.5 Takeaways and new opportunities

The previous sections have identified trends that lead to the conclusion that *the current and future landscape of neural network inference will contain a significant number of memory-bandwidth bound linear layers*: §4.3.2 illustrated that current neural networks, the linear layers within them, and their application settings exhibit a wide range of arithmetic intensities (including many with low arithmetic intensity), and §4.3.4 described trends in neural network design that often reduce arithmetic intensity. Coupling this with the dramatic growth in CMR for GPUs described in §4.3.3 drives home the conclusion that current and future neural networks will contain bandwidth-bound linear layers that underutilize the computational capabilities of GPUs.

Such bandwidth-bound linear layers leave room open for redundant execution to fill gaps in compute utilization during matrix multiplication. However, current approaches to coded computation for neural network inference are unable to exploit these fine-grained opportunities for efficient redundant execution. As described in §4.2.4, global CC operates at a much higher level (specifically, kernel level), and hence is unable to exploit compute underutilization that occurs at *finer granularity within the matrix multiplication operation.*

This calls for investigating approaches to redundant execution that can exploit the fine-grained compute underutilization exhibited by current and future matrix multiplication kernels in neural network inference. Such an approach would complement global CC, which is well-suited for the compute-bound linear layers in neural networks.

We next turn our focus toward investigating such an approach.

Key design principle. Driven by the opportunities outlined above, we use the following principle when considering approaches to redundant execution for memory-bandwidth-bound matrix multiplications: *avoid performing additional memory accesses whenever possible even if doing so comes at the expense of additional computation*. Adhering to this principle avoids competing with the matrix multiplication for its bottleneck resource, memory bandwidth.

4.4 Thread-level replication?

A natural question that arises when considering options for redundant execution for bandwidth-bound linear layers is whether it is beneficial to use *thread-level replication*, rather than coded computation. After all, coded computation is primarily designed to reduce the number of redundant operations performed compared to replication, while spare compute cycles are plentiful in bandwidth-bound linear layers. Furthermore, thread-level replication easily satisfies the design principle stated in §4.3.5, by sharing loads with the original matrix multiplication.

We began our exploration of redundant execution for bandwidth-bound linear layers with replication for these very reasons, but ultimately found it to have higher execution-time overhead than coded computation, as we next describe. We focus on matrix multiplications using m16n8k8 FP16 Tensor Core operations (MMAs) (described in §4.2.1). Recall that, in matrix multiplication using this operation, each thread participates in $\frac{M_t N_t}{2}$ MMAs on each iteration along the K dimension. For each MMA, a thread provides four elements from A_t, two elements from B_t, and receives four output elements.

We have considered two approaches to thread-level replication:

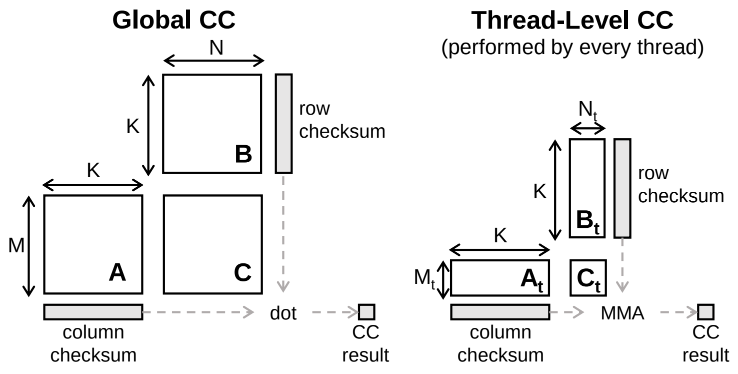


Figure 4.6: Global and two-sided thread-level coded computation.

Traditional replication. The traditional approach to performing thread-level replication is to perform $\frac{M_t N_t}{2}$ additional MMAs per step down the K dimension, accumulate the results in a separate set of $M_t N_t$ registers, and compare these registers to the original $M_t N_t$ matrix multiplication output registers. However, we found that the $2\times$ increase in output register usage per thread in this approach limits the number of threadblocks that can be co-scheduled on a single SM (so-called "occupancy" [51]), and leads to significant slowdowns compared to the original matrix multiplication kernel.

Replicated MMA, single accumulation. Based on this limitation, we next explored replicating MMAs, but accumulating results to a single set of four output registers. Under this approach, one still performs $\frac{M_t N_t}{2}$ additional MMAs per step along the K dimension, but each redundant MMA accumulates results to a single set of four registers. By the end of the thread-level matrix multiplication, in the absence of a fault, the summation of these four registers equals the summation of the thread's "original" $M_t N_t$ output registers.

We find that the limited additional register usage of this approach alleviates the occupancy-related slowdowns described above, and thus significantly reduces execution-time overhead compared to the traditional form of replication. However, as we will show in §4.6.5, doubling the number of MMAs performed results in higher execution-time overhead than coded computation.

We thus turn our focus to investigating coded-computation schemes that can exploit the compute underutilization identified in §4.3.

4.5 Arithmetic-intensity-guided coded computation

In this section, we first investigate approaches to coded computation that can exploit the fine-grained compute underutilization of bandwidth-bound linear layers identified in §4.3. We then describe the design of an adaptive approach to coded computation that selects a coded-computation scheme for each linear layer guided by the layer's arithmetic intensity.

4.5.1 At which level should coded computation be performed?

The hierarchical decomposition of matrix multiplications described in §4.2.1 offers multiple levels at which coded computation can be performed: the kernel level (as in global CC), threadblock level, warp level, or thread level. However, performing coded computation at any level other than the thread level requires performing additional loads/stores to generate checksums. For example, performing coded computation at the level of a threadblock requires individual threads to cooperate to generate threadblock-wide checksums, which requires storing and loading thread-local partial checksums. These additional loads and stores violate the design principle described in §4.3.5 and compete for bandwidth with the matrix multiplication itself.

In contrast, performing coded computation at the level of individual threads avoids additional loads/stores. Figure 4.6 compares one approach to thread-level CC with global CC at a high level. Concretely, thread-level CC involves threads in the matrix multiplication kernel performing their own, local coded computation calculations across their own, local sub-matrix multiplications. Thread-level CC eliminates additional loads/stores by (1) sharing the loads of operands that will be used for checksum generation with those that were already performed for thread-level matrix multiplication in a step along the K dimension, and (2) eliminating stores of partial checksums for use in threadblock- or warp-wide checksum generation.

Thus, we conclude that performing coded computation at the thread level is the appropriate fit for coded computation optimized for bandwidth-bound linear layers. This conclusion is heavily driven by the design principle established in §4.3.5 of avoiding additional loads/stores. In cases where this principle can be relaxed, performing coded computation at other levels may be appropriate. Even with this somewhat-extreme stance taken, we will show in §4.6 that thread-level CC significantly reduces execution-time overhead for bandwidth-bound linear layers.

4.5.2 Design decisions for thread-level CC

Even having narrowed our focus to performing coded computation at thread level for bandwidth-bound linear layers, there remain multiple design decisions that affect performance, which we discuss next. Similar to §4.4, we focus on m16n8k8 Tensor Core operations (MMAs), which are described in detail in §4.2.1.

Online computation of weight checksums. Recall from §4.2.4 that optimized approaches to global CC for neural networks typically compute the weight checksum of B once offline and load it upon every inference request. We do not employ this technique for thread-level CC, as doing so would require threads to load weight checksums from memory, violating the design principle described in §4.3.5. Thus, thread-level CC recomputes thread-local weight checksums alongside the thread-level matrix multiplication.

Balancing checksum generation and redundant MMAs. Adopting the coded-computation approach described in §4.2.3 at thread level would involve performing the following for each step the thread takes along the K dimension: (1) computing a thread-level activation checksum from A_t, (2) computing a thread-level weight checksum from B_t, and (3) performing a single MMA over these checksums to generate coded-computation output values. These steps are illustrated in the left-hand side of Figure 4.7, and are repeated for each iteration along the K dimension, accumulating into the same coded-computation output registers. Once the thread has completed all iterations along the K dimension, it generates a thread-local output summation

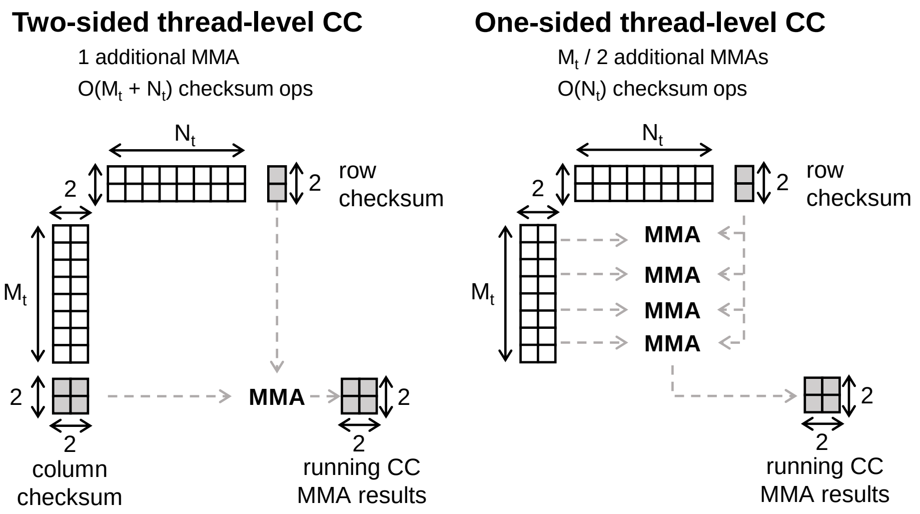


Figure 4.7: Comparison of two-sided and one-sided thread-level CC for a single step in the K dimension.

Table 4.1: Additional Tensor Core MMAs and checksum operations done by thread-level replication, two-sided coded computation, and one-sided coded computation per step in the K dimension.

	Replication	**Two-sided**	**One-sided**
Tensor Core MMAs	$M_tN_t/2$	1	$M_t/2$
Checksum operations	0	$\mathcal{O}(M_t + N_t)$	$\mathcal{O}(N_t)$

and compares it to the final coded-computation output registers. We call this approach *two-sided thread-level CC*, as it generates checksums for both A_t and B_t.

Two-sided thread-level CC minimizes the number of redundant MMAs performed by thread-level CC, as it performs only one extra MMA for every step along the K dimension. However, it maximizes the amount of computation performed in generating thread-local checksums.

It is important to note that checksum generation involves summations that will execute on traditional arithmetic units on the GPU (e.g., using `HADD2` instructions), rather than on Tensor Cores. In contrast, redundant MMA operations will execute on Tensor Cores. Thus two-sided thread-level CC will more significantly utilize traditional arithmetic units than Tensor Cores because it performs $\mathcal{O}(M_t + N_t)$ additional checksum generation operations but only one additional MMA per step along the K dimension.

Given that Tensor Cores are the drivers behind the math performed in matrix multiplication, it is Tensor Cores that are heavily underutilized by bandwidth-bound linear layers, rather than traditional arithmetic units. Traditional arithmetic units are likely not as underutilized in bandwidth-bound linear layers, as they are also used by threads to carry out general control flow (e.g., updating loop counters) and to assist in loading/storing data (e.g., computing addresses). Thus, minimizing the number of additional MMAs performed in two-sided thread-level CC may insufficiently exploit underutilized Tensor Cores. At the same time, our experience with replication in §4.4 indicates that adding too many additional MMAs can also lead to high overhead.

To straddle this tradeoff between added operations to Tensor Cores and added operations to traditional arithmetic units, we leverage a *one-sided thread-level CC scheme*. Rather than computing checksums for both A_t and B_t and performing a single MMA across these checksums, one-sided thread-level CC instead generates a checksum only for B_t and multiplies the entirety of A_t with this checksum.[3] As illustrated in the right-hand side of Figure 4.7 this results in performing $\frac{M_t}{2}$ additional MMAs, and $\mathcal{O}(N_t)$ checksum generation operations for each step along the K dimension.

As shown in Table 4.1, one-sided thread-level CC sits between thread-level replication and two-sided thread-level CC in terms of additional MMAs and checksum operations performed. We illustrate in §4.6.5 that this enables one-sided thread-level CC to provide the lowest execution-time overhead among these approaches to thread-level redundant execution.

4.5.3 Per-layer, intensity-guided adaptation

As shown in §4.3, neural networks have a mix of compute- and bandwidth-bound linear layers. The coded-computation scheme with the lowest execution-time overhead for a given layer depends on the bottleneck of the layer, with global CC preferable for compute-bound layers and thread-level CC preferable for bandwidth-bound layers.

Rather than selecting one coded-computation scheme to be applied to all linear layers of a neural network, we propose *intensity-guided CC*, which selects among global CC and thread-level CC for each individual linear layer. Prior to deploying a neural network, intensity-guided CC measures the execution-time overhead of each linear layer under global CC and thread-level CC, and chooses the scheme with the lowest overhead for that layer. As we show in §4.6, in conforming to the ideas presented in this chapter, linear layers with higher arithmetic intensity typically benefit from global CC, while those with lower arithmetic intensity typically benefit from thread-level CC. Thus, intensity-guided CC uses arithmetic intensity as a guide in selecting the best coded-computation scheme for each layer. Our evaluation in §4.6 shows that intensity-guided CC significantly reduces overhead compared to either global or thread-level CC alone.

Integration with pre-deployment optimizers. Intensity-guided CC fits alongside the popular approach of pre-deployment optimization in neural network inference, as performed by frameworks like TensorRT [39], TVM [97], cuDNN [29], and CUTLASS. This process takes in a neural network and an input size (e.g., image resolution, batch size) that will be used during inference and enumerates and executes many configurations of each layer in the neural network (e.g., tile sizes, matrix layouts). The configuration with the lowest execution time for a layer is chosen for that layer and used for all inference requests during deployment. A pre-deployment optimizer using intensity-guided CC will include global CC and thread-level CC in its enumeration of configurations of a matrix multiplication. Intensity-guided CC chooses the fastest among these, which typically aligns with the arithmetic intensity of the layer, as we show in §4.6.

4.6 Implementation and evaluation

We now evaluate the execution-time overhead of intensity-guided CC. The highlights of the evaluation are as follows:

[3]One can alternatively multiply a checksum of A_t with B_t.

- Across eight popular CNNs, two neural networks used in DLRMs, and four specialized CNNs, intensity-guided CC reduces execution-time overhead compared to global CC by 1.09–5.3$\times$.
- Intensity-guided CC provides the largest reductions in execution-time overhead for neural networks that have many linear layers with low arithmetic intensity, such as DLRMs (up to 4.9$\times$ reduction) and specialized CNNs (up to 5.3$\times$ reduction).
- Even for neural networks that have many linear layers with high arithmetic intensity, intensity-guided CC still significantly reduces execution-time overhead (e.g., 1.5$\times$ for Wide-ResNet-50). This shows the benefit of intensity-guided CC's adaptive approach to coded computation, as even neural networks that are primarily compute bound often have some linear layers with low arithmetic intensity.
- Intensity-guided CC provides similar benefits across various input resolutions (§4.6.4) and batch sizes (§4.6.4).
- The one-sided thread-level CC approach motivated in §4.5.2 significantly reduces execution-time overhead compared to two-sided thread-level CC and thread-level replication (§4.6.5).

4.6.1 Implementation

Recall that intensity-guided CC adapts to each linear layer in a neural network by choosing between thread-level CC and global CC. We implement thread-level CC and global CC in CUDA/C++ atop CUTLASS [31], a high-performance, open-source linear algebra and machine learning library developed by NVIDIA. For thread-level CC, we modify existing thread-level inner loops in CUTLASS to perform checksum generation, redundant MMAs, and final checksum comparison. We implement the global CC scheme based on the state-of-the-art approach from Hari et al. [149] (discussed in §4.2.4), using NVIDIA's CUB library [28] when possible.

Recall from §4.5.3 that intensity-guided CC fits alongside common pre-deployment neural network optimizers. We integrate intensity-guided CC into the pre-deployment workflow of the CUTLASS profiler, which selects the fastest matrix multiplication kernel and configuration (e.g., tile size, layout) for a given matrix multiplication size.

4.6.2 Evaluation setup

Baselines. Our main comparison is between intensity-guided CC and the state-of-the-art approach to global CC for neural network inference on GPUs described in §4.2.4. We also evaluate one-sided thread-level CC alone (referred to as "thread-level CC"), and in §4.6.5 compare to two-sided thread-level CC and thread-level replication.

Metrics. Execution-time overhead is one of the primary metrics of interest for redundant execution. For each linear layer of a neural network, we obtain the execution time of the original matrix multiplication without redundancy (T_o), as well as that of the redundant version (T_r) and report the percentage increase in execution time ($\frac{T_r - T_o}{T_o} * 100$). We report execution-time overhead for an entire neural network by summing the per-layer execution times and using these in the equation above. We include only linear layers, as these layers typically dominate the end-to-end execution time of a neural network. Moreover, aggregating the execution times of each linear layer in this fashion is representative of overall execution-time overhead for the neural network as a whole, because, for all of the neural networks we consider, the subsequent layer of the neural network cannot begin executing until the current layer has completed execution. We report the mean of 10 trials of 1000 runs after 100 warmup runs. Error bars show the maximum and minimum time overheads across trials. In many cases, error bars are imperceptible due to their tightness. We do not plot error bars in Figure 4.8 to avoid clutter and because all error bars for neural networks in Figure 4.8 are plotted in Figures 4.9, 4.10, and 4.11. Note that while in some cases error bars may give the incorrect impression that intensity-guided CC performs worse than global CC, intensity-guided CC, by design, always performs at least as well as global CC.

We also report "aggregate arithmetic intensity" (defined in §4.3.2). In each figure, the FP16 aggregate arithmetic intensity is listed in parentheses below each model.

Evaluation setting. We evaluate on an NVIDIA T4 GPU [55], which is an inference-optimized GPU, on an AWS g4dn.xlarge instance. The T4 offers 65 FP16 TFLOPs/sec and 320 GB/sec of memory bandwidth, giving it an FP16 CMR of 203. We use CUDA 11.0 and configure the clock rate of the GPU according to that used in CUTLASS [30]. We perform all experiments using FP16 datatypes and we use the m16n8k8 matrix multiplications targeting Tensor Cores described in §4.2.1. Note that it is standard to perform neural network inference in low precision, such as FP16. We pad matrix dimensions M, N, and K to be multiples of eight when needed to operate with the m16n8k8 operation. We find CUTLASS's m16n8k8 matrix multiplication with $M = N = K = 2048$ to achieve similar TFLOPs/sec to the highest reported on the T4 GPU [167].

Workloads. We consider workloads from multiple domains:

General-purpose CNNs. We consider eight widely-used CNNs from the popular PyTorch Torchvision library [48]: ResNet-50, VGG-16, AlexNet, SqueezeNet, ShuffleNet, DenseNet-161, ResNext-50, and Wide-ResNet-50. Each of these CNNs has 1000 output classes, as is standard for ImageNet. We primarily report performance when operating over HD images of size 1080×1920 with batch size of one, though we consider other image resolutions in §4.6.4.

Recommendation models. We consider Facebook's DLRM [237], which has two neural networks consisting of fully-connected layers (also called multilayer perceptrons, or MLPs): MLP-Bottom, which has three hidden layers with 512, 256, and 64 nodes each, and MLP-Top which has two hidden layers with 512 and 256 nodes, and produces one output value. We primarily consider DLRMs with batch size of one as this is the common case for low-latency, user-facing inference [105, 341]. For completeness, we also consider large batch size in §4.6.4.

Specialized CNNs. We also evaluate on neural networks representative of ongoing efforts to deploy small neural networks (described in §4.3.4). We consider four specialized CNNs used within the NoScope system [179]: Coral, Roundabout, Taipei, Amsterdam. These CNNs act as lightweight filters performing binary classification in front of large, general-purpose CNNs for high-throughput offline video analytics in cluster settings. These CNNs have 2–4 convolutional layers, each with 16–64 channels, at most two fully-connected layers, and operate over regions of video frames of size 50×50 pixels. As these CNNs are used for offline analytics, we use a large batch size of 64 for experiments.

Square matrix multiplications. We finally perform a more detailed comparison of one-sided thread-level CC and global CC, along with two-sided thread-level CC and thread-level replication on matrix multiplications with $M = N = K$ of various sizes (§4.6.5).

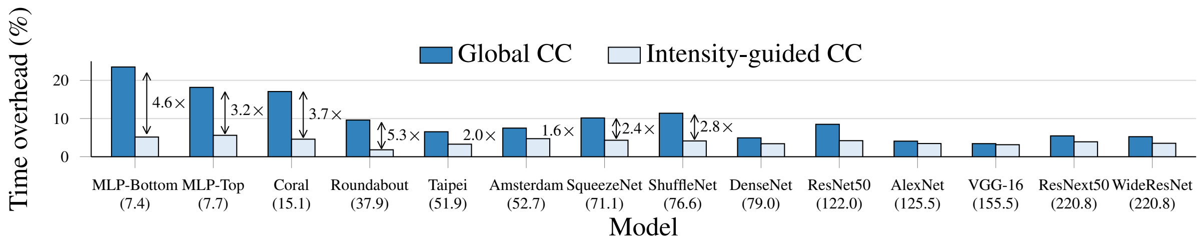


Figure 4.8: Execution-time overhead on all neural networks considered. To avoid clutter, error bars are not plotted in this figure, but are plotted in Figures 4.9, 4.10, and 4.11 for each neural network.

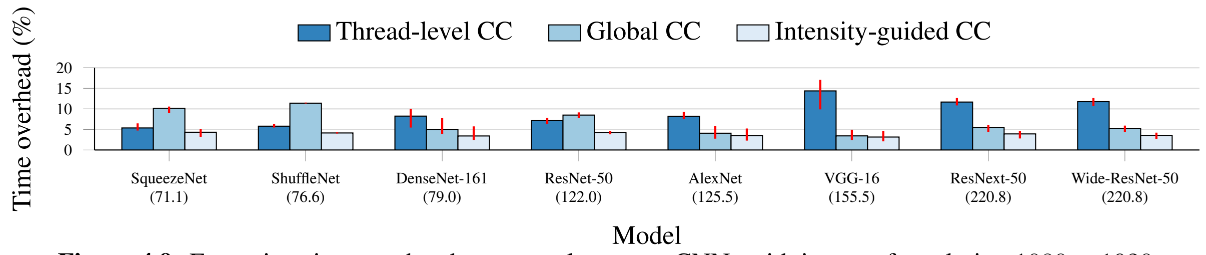


Figure 4.9: Execution-time overhead on general-purpose CNNs with inputs of resolution 1080×1920.

4.6.3 Summary of results

Figure 4.8 compares the execution-time overhead of global CC to that of intensity-guided CC on all neural networks we consider (listed in order of increasing aggregate arithmetic intensity).[4] Compared to global CC, intensity-guided CC reduces execution-time overhead by up to 5.3×. For example, for the Coral specialized CNN, intensity-guided CC reduces execution-time overhead from 17% to 4.6%. As expected, intensity-guided CC achieves the largest reduction in execution-time overhead for neural networks with low aggregate arithmetic intensity, as these neural networks contain more bandwidth-bound linear layers that benefit from thread-level CC. That said, intensity-guided CC reduces execution-time overhead even for neural networks with high aggregate arithmetic intensity. For example, intensity-guided CC reduces the execution-time overhead on Wide-ResNet-50 by 1.5× compared to global CC (from 5.3% to 3.5%). Even though such neural networks have high aggregate arithmetic intensity, they still contain bandwidth-bound linear layers, for which using thread-level CC over global CC reduces overhead.

4.6.4 Evaluation across various neural network domains

General-purpose CNNs. Figure 4.9 shows the execution-time overhead for thread-level CC, global CC, and intensity-guided CC on eight popular general-purpose CNNs operating over HD images of size 1080×1920 at batch size one. Compared to global CC, intensity-guided CC reduces execution-time overhead by 1.09–2.75×. As expected, thread-level CC achieves lower execution-time overhead than global CC for CNNs with low aggregate arithmetic intensity, while global CC has lower overhead for CNNs with higher aggregate arithmetic intensity. Intensity-guided CC achieves the lowest execution-time overhead across all the CNNs, motivating its per-layer, arithmetic intensity-guided approach.

[4]Note that the execution-time overheads do not monotonically decrease with increasing aggregate arithmetic intensity since execution is performed layer-wise whereas aggregate arithmetic intensity is not layer-wise.

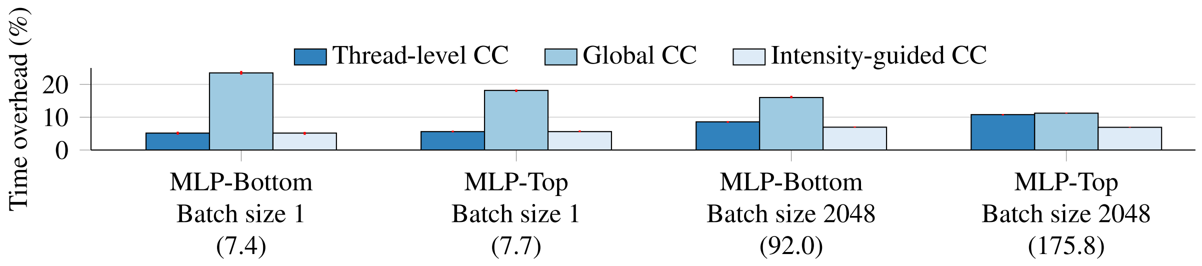


Figure 4.10: Execution-time overheads on neural networks from DLRM. Error bars are tight to the point of being imperceptible.

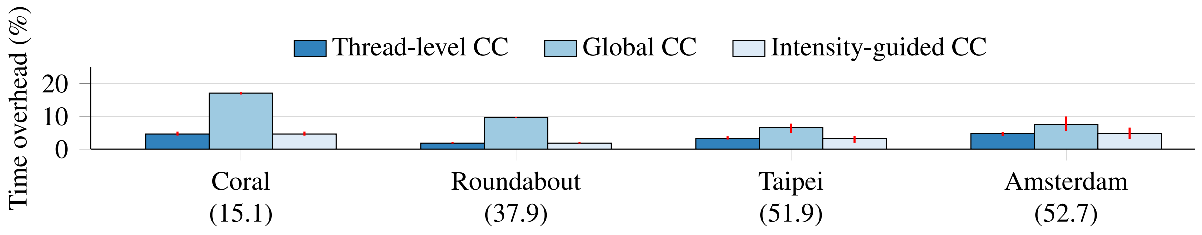


Figure 4.11: Execution-time overheads on specialized CNNs.

Effect of image resolution. When operating on images of size 224×224 (the standard resolution in ImageNet), intensity-guided CC reduces execution-time overhead by 1.3–3.3$\times$ compared to global CC. This larger reduction compared to operating on HD images stems from the lower aggregate arithmetic intensity of CNNs when operating on images with smaller resolution (described in §4.3.2). This leads to more linear layers being bandwidth-bound and benefiting from thread-level CC in intensity-guided CC.

Recommendation models (DLRM). We next consider the neural networks used in Facebook's DLRM. Figure 4.10 plots execution-time overheads on MLP-Bottom and MLP-Top. At batch size of one, which corresponds to low-latency deployments of DLRMs for user-facing services, both MLP-Bottom and MLP-Top have low aggregate arithmetic intensity. This results in intensity-guided CC reducing execution-time overhead compared to global CC by 4.55$\times$ and 3.24$\times$ for MLP-Bottom and MLP-Top, respectively. At a very large batch size of 2048, the aggregate arithmetic intensity of both MLP-Bottom and MLP-Top increase, but at different rates. The aggregate arithmetic intensity of MLP-Top increases from 7.7 to 175.8, resulting in the difference between global and thread-level coded computation decreasing. In contrast, the aggregate arithmetic intensity of MLP-Bottom grows only from 7.4 to 92, resulting in thread-level CC continuing to have lower overhead. In both cases, intensity-guided CC achieves the lowest overhead, illustrating the need for coded computation to consider the resource bottlenecks of each linear layer of a neural network.

Specialized CNNs. Figure 4.11 shows the execution-time overheads on specialized CNNs from NoScope [179] at batch size 64. For these primarily bandwidth-bound CNNs with low aggregate arithmetic intensity, intensity-guided CC reduces execution-time overhead by 1.6–5.3$\times$. These results are particularly promising when considering the growing trends described in §4.3 of designing lightweight neural networks, coupled with the increasing CMR of GPUs, which will

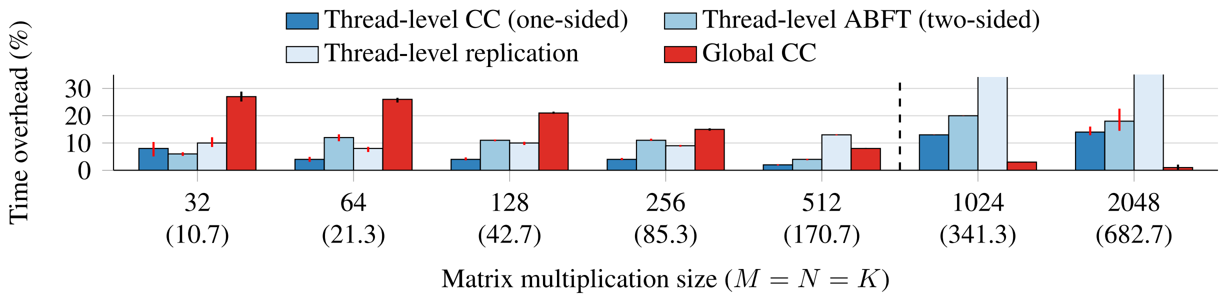


Figure 4.12: Execution-time overhead on square matrix multiplications. Sizes left of the dashed line have arithmetic intensity below the T4's FP16 CMR. The overhead for replication is above 70% for the final two sizes, and thus is cut off.

likely result in more neural networks being bandwidth-bound.

4.6.5 Evaluation of thread-level design decisions

To evaluate the design decisions made in leveraging thread-level CC for neural network inference on GPUs, we now evaluate global CC and the various approaches to thread-level redundant execution described in §4.4 and §4.5. We perform such evaluation on square matrix multiplications (i.e., $M = N = K$) of varying size, allowing us to control arithmetic intensity and best illustrate the tradeoffs.

Figure 4.12 shows the execution-time overhead of each approach with $M = N = K$ ranging from 32 to 2048, corresponding to FP16 arithmetic intensities of 10 to 683. We first compare only the final version of thread-level CC we leverage (one-sided) to global CC. As expected, for matrix sizes with arithmetic intensity less than the FP16 CMR of the T4 (203), thread-level CC achieves an execution-time overhead up to 6.5$\times$ lower than that of global CC, while for matrix sizes with higher arithmetic intensity, global CC achieves overheads up to 14$\times$ lower than thread-level CC. It is clear that taking a one-size-fits-all approach to coded computation will lead to suboptimal performance on certain matrix sizes, motivating our adaptive approach in intensity-guided CC.

Figure 4.12 also shows that one-sided thread-level CC almost always exhibits lower execution-time overhead than two-sided thread-level CC and thread-level replication. This reinforces our decision to use one-sided coded computation for thread-level CC. The differences between replication and coded computation are particularly stark for larger sizes (512 and beyond), where the overhead of replication sharply spikes due to increasing competition for Tensor Cores.

4.7 Related work

Fault tolerance in general programs. There are various techniques for tolerating soft-error-induced faults in general programs:

One approach is hardware-based fault tolerance, such as through using ECC in memory, and radiation-hardened processing units [77, 114, 290]. While certain approaches to hardware-based fault tolerance are widely used, such as ECC-protected memory subsystems, hardware protection for processing units is less widely used due to its high overhead. Furthermore, hardware-based fault tolerance is inflexible to changes in the required fault tolerance of applications or the fault rate of operating environments. Thus, we focus on software-based fault tolerance.

Software-based fault tolerance for general programs is typically achieved through techniques like instruction duplication [225, 268], replication of threads/warps [119, 165, 305, 330], and compiler-driven re-execution [183, 212]. In contrast, we focus on using application-level features of neural network inference to reduce the overhead of fault detection.

Fault tolerance in neural networks. Recent works have illustrated the potentially-catastrophic effects of soft errors on neural networks through fault injection tools [101, 102, 199, 222] and neutron beam experiments [122]. This has spurred many approaches for fault tolerance in neural networks, such as leveraging the "inherent robustness" of neural networks [248, 300, 340], training neural networks to tolerate faults [185], anomalous activation suppression [100, 249], selective hardening [74, 205, 223, 224], and learning to detect faults [204, 277, 278]. Our focus in this work is on leveraging coded computation to detect errors in neural network inference. Compared to the approaches listed above, coded computation provides clearer fault-tolerance guarantees and does not require retraining a neural network or understanding its behavior.

Coded computation for neural networks. Due to the heavy use of linear algebra in neural networks, coded computation is a natural fit for fault tolerance in neural networks, and a number of recent works have explored using coded computation for neural networks [126, 149, 201, 247, 342]. Ozen et al. [247] leverage coded computation to protect convolutional and fully-connected layers and propose integration of coded computation into a systolic array architecture. Zhao et al. [342] propose a systematic workflow of coded-computation checks for CNNs to provide a high degree of protection against faults with low execution-time overhead on CPUs. Li et al. [201] propose optimizations for coded computation in low-bitwidth DLRM inference on CPUs. Most closely related to our work is the work of Hari et al. [149], which proposes the optimized global CC scheme for GPUs (described in §4.2.4), and which forms a component of our proposed intensity-guided CC. Intensity-guided CC complements the work of Hari et al. [149] with coded-computation schemes well-suited for bandwidth-bound linear layers, and by adaptively selecting between the two, using arithmetic intensity as a guide.

Compared to these works, the present work is unique in multiple aspects. First, the works listed above all focus on employing a single coded-computation scheme across all linear layers of a neural network. In contrast, we illustrate that different linear layers within a neural network have varying resource bottlenecks that benefit from per-layer-optimization in the proposed intensity-guided CC. Second, to the best of our knowledge, our work is the first to analyze the bottleneck of memory bandwidth for neural network inference in the context of exploiting it for efficient redundant execution. Careful analysis of this trend lends itself to developing optimizations that have been overlooked by prior works. Finally, to the best of our knowledge, this work presents the first thread-level approach to coded computation for neural networks on GPUs.

Coded computation in other domains. Coded computation has been widely studied for making linear algebra routines fault tolerant [86, 87, 160, 321, 336], iterative methods [96, 99], and other applications [202]. Our work differs from these works in its focus on the specific characteristics of neural networks and GPUs, and its adaptive, intensity-guided approach of selecting coded-computation schemes based on the resource bottlenecks of the problem.

Smith et al. [289] investigated fusing coded-computation operations alongside matrix multiplications on CPUs. While similar to the approach to thread-level CC that we consider as part of intensity-guided CC, the techniques employed by Smith et al. [289] differ in that they do not perform coded computation at the level of the smallest unit of the parallel subproblem in the matrix multiplication. Thus, this approach generates checksums collaboratively across CPU threads (although not globally), which requires additional loads and stores, albeit, at higher levels of the memory hierarchy. In contrast, we leverage thread-level CC specifically for bandwidth-bound linear layers in neural networks on GPUs, and thus avoid performing any additional loads and stores (which would compete for the layer's bottleneck resource). This results in thread-level CC performing coded computation at the smallest parallel sub-matrix multiplication solved (GPU thread level), requiring no coordination between threads. Furthermore, intensity-guided CC takes an adaptive approach to coded computation based on the resource bottleneck of a given matrix multiplication, whereas Smith et al. [289] use a one-size-fits-all approach.

Concurrent work with ours, FT-BLAS [337], proposes to choose between replication and coded computation in BLAS routines on CPUs depending on the BLAS level of an operation. Specifically, FT-BLAS [337] uses replication for operations in Levels 1 and 2 (vector-vector and matrix-vector operations), and coded computation for those in Level 3 (matrix-matrix operations). However, for a given operation in a BLAS level (e.g., for all matrix-matrix multiplications), FT-BLAS [337] takes a one-size-fits-all approach. In contrast, we show that the unique characteristics of neural network inference on GPUs lead to *neural networks containing a mix of compute- and bandwidth-bound matrix-matrix multiplications*, rendering one-size-fits-all approaches inefficient. Moreover, as shown in §4.6.5, leveraging replication even for bandwidth-bound matrix multiplications used in neural networks on GPUs can lead to significant overhead, motivating intensity-guided CC's approach of selecting between various coded computation schemes for each matrix multiplication.

4.8 Conclusion

Intensity-guided CC illustrates the reduction in resource overhead possible by considering specific properties of ML systems and the infrastructure on which they run when leveraging existing coding-theoretic tools. Specifically, we showed that the most efficient implementation of coded computation for linear layers of neural networks depends on the arithmetic intensity of the layer and the CMR of the GPU on which it is run. Leveraging this insight and the diversity of arithmetic intensities of linear layers of neural networks, intensity-guided CC adapts the approach to coded computation it uses on a per-layer basis to significantly reduce execution-time overheads compared to a one-size-fits-all approach.

Chapter 5

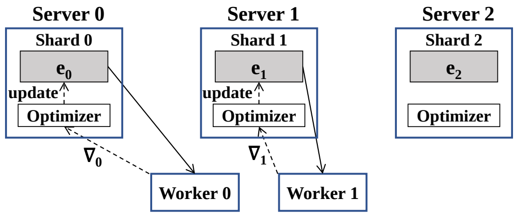
Efficiently using erasure codes in recommendation model training

This chapter provides a second example making fault tolerance for ML systems more efficient through properties specific to ML systems. We focus on the problem of fault-tolerant training of distributed deep-learning based recommendation models (DLRMs). Prior work from Facebook has shown that checkpointing, the current approach to fault tolerance in DLRM training, adds an average of 12% to the total training time of production DLRMs. We investigate the potential of using erasure codes to overcome the downsides of checkpointing. Our study reveals multiple ways in which the unique characteristics of DLRM training call for nuance in leveraging erasure codes in these systems. The result of our work is ECRM, a fault-tolerant DLRM training system that leverages a hybrid redundancy scheme by selecting between erasure codes and replication for different DLRM parameters, enables training to continue when recovering from a failure, and correctly and maintains the consistency of recovered parameters. Compared to checkpointing, ECRM reduces training-time overhead on large DLRMs by up to 66%, recovers from failure up to $9.8\times$ faster, and continues training during recovery with only a 7–13% drop in throughput (whereas checkpointing must pause).

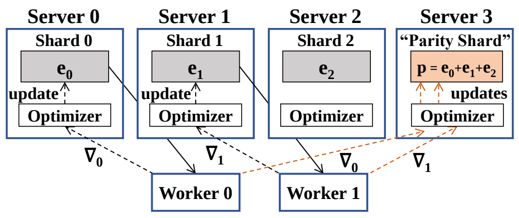
5.1 Introduction

Deep-learning-based recommendation models (DLRMs) are key tools in serving personalized content to users at Internet scale [107, 237]. As the value generated by DLRMs often relies on the ability to reflect recent data, production DLRMs are frequently retrained [20]. Reducing DLRM training time is thus critical to maintaining an accurate and up-to-date model.

DLRMs consist of embedding tables and neural networks (NNs). Embedding tables map sparse categorical features (e.g., location of a client) to a learned dense representation. Embedding tables resemble lookup tables in which millions or billions [130, 169] of sparse features each map to a small dense vector representation of tens/hundreds of floating-point values. We refer to a single dense embedding vector as an "embedding table entry," or "entry" for short. A small NN processes dense vectors resulting from embedding table "lookups" to produce a prediction (e.g., whether a client likes a video).



(a) Example of the distributed DLRM training.



(b) Naive erasure-coded DLRM with $k = 3$ and $r = 1$.

Figure 5.1: Example of (a) normal and (b) (naive) erasure-coded training with parameters e_0, e_1, and e_2, and gradients ∇_0 and ∇_1. For simplicity, only accesses and updates for embedding table entries are shown.

Embedding tables are typically large, ranging from hundreds of gigabytes to terabytes in size [169]. Such large models are trained in a distributed fashion across tens/hundreds of nodes [60, 169], as depicted (at a small scale) in Figure 5.1a. Embedding tables and NN parameters are sharded across *servers* and kept in memory for fast access. *Workers* operate in a data-parallel fashion to perform NN training by accessing model parameters from servers and sending gradients to servers to update parameters via an optimizer (e.g., Adam [118]).

Since model parameters are stored in memory, any server failure requires training to restart from scratch. Given that DLRM training is resource and time intensive and that failures are common in large-scale settings, it is imperative for DLRM training to be fault tolerant [129, 221]. In this work, we focus on tolerating server failures. Server failures are critical to handle because they result in the loss of a fraction of the DLRM parameters. In contrast, worker failures are less critical because workers do not contain training state.

Checkpointing is the main approach used for fault tolerance in DLRM training [129, 221]. This involves periodically pausing training and writing the current parameters and optimizer state to stable storage, such as a distributed file system. If a failure occurs, the entire system resets to the most recent checkpoint and restarts training from that point. While simple, checkpointing frequently pauses training to save DLRM state and has to redo work after failure. Thus, checkpointing has been shown to add significant overhead to training production DLRMs, such as at Facebook [221]. Checkpointing also consumes significant network and storage bandwidth in datacenters and results in a large storage footprint [129]. This adds up to additional cost in training DLRMs: Google recently reported that checkpointing-related overheads amount in an additional cost per year of roughly 45 sofware engineers [46]. Even more concerning, time and resource overheads increase with DLRM size. Given the trend of increasing model size [280, 329] (which we describe in greater detail in §5.2.2), *checkpointing is slated to incur even larger overhead for training future DLRMs.*

An alternative to checkpointing that does not require stalls, a lengthy recovery process, or significant storage bandwidth and capacity is to replicate DLRM parameters on separate servers. However, replication requires at least twice as much memory as a checkpointing-based system, which is impractical given the large sizes of DLRMs. Another alternative is to reduce the overhead of checkpointing by taking approximate checkpoints [98, 129, 221, 258]. However, this can result in accuracy loss in training, which makes debugging production systems harder due to uncertainty in the accuracy of the model recovered from the checkpoint. Furthermore, even small drops in accuracy have been noted to result in a significant reduction in the business value generated by DLRMs [343], making potential accuracy loss induced by an approach to fault tolerance undesirable.

An ideal approach to fault-tolerant DLRM training would (1) operate with low training-time and memory overhead, and (2) recover quickly from failures, while (3) not introducing potential accuracy loss (and the associated uncertainty). Finally, such a solution should scale well with increases in DLRM size so as to support emerging DLRMs. Designing such an approach is the goal of this chapter.

We question the potential of using erasure codes as an efficient alternative means of fault tolerance for DLRM training. Like replication and traditional checkpointing, erasure coding would not alter the accuracy of training. Due to their low memory overhead, erasure codes offer potential for efficient fault tolerance in DLRM training. As shown in Figure 5.1b, a DLRM training system could potentially construct "parity parameters" by encoding k parameters from separate servers. In this example, a parity p is formed from parameters e_0, e_1, and e_2 via the encoding function $p = e_0 + e_1 + e_2$, and placed on a separate server. If a server fails, lost parameters can be recovered by reading the k available parameters and performing the erasure code's decoding process (e.g., $e_1 = p - e_0 - e_2$).

While erasure codes appear promising for fault-tolerant DLRM training, applying them to this setting comes with challenges due to the interaction between erasure codes and the unique characteristics of DLRM training. In this work, we thoroughly investigate the use of erasure codes in DLRM training systems, uncover these challenges, and propose solutions to overcome them. The result of our work is ECRM,[1] a DLRM training system that achieves efficient fault tolerance by leveraging insights into the unique characteristics of DLRM training along with careful system design. We describe the challenges in this process and how ECRM overcomes them below.

Hybrid redundancy. We show in §5.3.2 that correctly using erasure codes in DLRM training necessitates more communication overhead than replication. Thus, ECRM must carefully determine which parameters should be erasure coded so as to straddle a tradeoff between memory and network overhead. ECRM approaches this decision based on unique characteristics of many DLRMs: while embedding tables account for the vast majority of the memory use of DLRMs, gradients for NN parameters dominate the network bandwidth used in updating parameters. For example, for the DLRM trained on the Criteo dataset [10] in MLPerf [228], embedding tables account for 99% of the DLRM parameters, but only 35% of the network bandwidth for gradients, while NN parameters account for 1% of the DLRM parameters, but 65% of the network bandwidth.

Based on this asymmetry of resource consumption, ECRM takes a hybrid approach to redundancy by *erasure coding embedding tables and replicating NN parameters.* Keeping embedding tables erasure coded ensures that ECRM has small memory overhead, while replicating NN parameters significantly reduces the network bandwidth consumed by ECRM, without adding much memory overhead.

Maintaining correctness and consistency. Redundant parameters in ECRM must be kept up-to-date with DLRM parameters to ensure correct recovery. As will be shown in §5.3.2 and §5.3.5, correctly and consistently updating parities when using optimizers that store internal state (e.g., Adagrad, Adam) is challenging without incurring large memory overhead. ECRM circumvents these challenges by delegating the responsibility for updating parities to servers,

[1]ECRM: Erasure-Coded Recommendation Model

rather than workers, via an approach we call "difference propagation," and by leveraging two-phase commit to update parameters.

Pause-free recovery. An erasure code's recovery process can be resource intensive because it involves reading all available data to a single server and performing decoding [263, 276]. This can lead to long recovery times during which training is stalled. ECRM recovers quickly from failure by enabling training to continue during recovery. ECRM leverages on-demand reconstruction of failed DLRM parameters to service new training iterations while full recovery proceeds in the background. ECRM carefully ensures that new training updates do not conflict with the background recovery process.

We implement ECRM atop XDL [169], an open-source, industrial-scale DLRM training system, and evaluate using variants of the Criteo DLRM in MLPerf [228]. ECRM recovers from failures faster than checkpointing and with lower training-time overhead for large DLRMs. For example, ECRM recovers from failure up to 9.8$\times$ faster than the average case for checkpointing, and enables training to continue during recovery with only a 7–13% drop in throughput, while checkpointing pauses training during recovery. This improved recovery makes it easier for DLRM training to meet tight deadlines for deploying a new model even when failures occur. Furthermore, ECRM reduces training-time overhead for a large DLRM by up to 66% compared to checkpointing. ECRM's benefits increase with DLRM size, showing promise to bring efficient fault tolerance to current and future DLRMs.

5.2 Challenges in fault-tolerant DLRM training

We first describe DLRM training, the insufficiency of current approaches to fault-tolerant DLRM training, and opportunities for more-efficient fault tolerance in DLRM training.

5.2.1 DLRM training systems

As described in §5.1, DLRMs are large in size due to their use of embedding tables that span hundreds of gigabytes to terabytes in size, and DLRM training is typically distributed across a set of servers and workers (see Figure 5.1a). Model parameters (both for embedding tables and for NNs) are sharded across server memory. In a given training iteration over a batch of data, a worker reads embedding table entries relevant to that batch and all NN parameters, performs a forward and backward pass to generate gradients (for both NN parameters and embedding table entries), and sends gradients back to the servers hosting the parameters that were read. An optimizer (e.g., Adam) on each server uses gradients received from workers to update parameters. Finally, many systems use asynchronous training when training DLRMs (e.g., Facebook [60] and Alibaba [169]). We thus focus on asynchronous training in this work, but the techniques we propose could extend to synchronous training.

Stateful optimizers. Many popular optimizers use per-parameter state in updating parameters (e.g., Adam [118], Adagrad [125], momentum SGD). We refer to such optimizers as "stateful optimizers." For example, Adagrad tracks the sum of squared gradients for each parameter over time and uses this when updating the parameter. Per-parameter optimizer state is kept in memory on servers and is updated when the corresponding parameter is updated. As per-parameter state grows with DLRM size, optimizer state for embedding tables can consume significant memory.

5.2.2 Unique characteristics of DLRMs

DLRMs contain unique characteristics compared to other deep networks. First, while many deep networks today leverage large NNs, DLRMs typically leverage small NNs but large embedding tables [145]. For example, a DLRM commonly used to train on the Criteo dataset contains over 100 GB of embedding tables, but less than 1 GB of NN parameters. Second, components of DLRMs have diverse access patterns. Each training sample typically accesses only a few embedding table entries, but all NN parameters. For example, the average number of embedding table entries accessed by a batch of 2048 training samples on the Criteo dataset is only 8900. This is a small fraction of the roughly 200 million entries in the DLRM. Thus, embedding table entries are updated sparsely, while all NN parameters are updated on every training batch.

Scaling trends. Similar to other deep models, increasing parameter count in DLRMs has led to increased accuracy. Thus, DLRMs have drastically increased in size over the years: whereas in 2020, Facebook used DLRMs with 100s of billions of parameters, today's DLRMs now use well over one trillion parameters [231]. This scaling is heavily driven by the increased number of embedding table entries in DLRMs: Facebook reports that the number of embedding table entries in DLRMs increased by 17.5$\times$ from 2017–2021 [280]. Moreover, this scaling trend is expected to increase in the future [329].

5.2.3 Checkpointing and its downsides

Given the large number of nodes on which DLRMs are trained, failures are common [129, 221]. Due to the time it takes to train such models, it is critical that DLRM training be fault tolerant.

We focus on tolerating server failures. Handling server failures is critical, as failure of a single server results in loss of fraction of embedding table entries and NN parameters, as well as any optimizer state. In contrast, fault tolerance is not as critical for workers, as workers do not hold DLRM parameters. Losing a worker may reduce training throughput, but will not result in loss of training state. In the event of a worker failure, a replacement worker can be allocated while the system continues training.

Checkpointing is the primary approach used for fault tolerance in DLRM training [129, 221]. Under checkpointing, training is periodically paused and DLRM parameters and optimizer state are writen to stable storage (e.g., a distributed file system). Upon failure, the most recent checkpoint is read from stable storage, and the entire system restarts training from this checkpoint, redoing any training iterations that occurred between the most recent checkpoint and the failure.

Recently, *Facebook reported that overheads from checkpointing account for, on average, 12% of DLRM training time*, and that these overheads add up to *over 1000 machine-years of computation* [221]. We next describe two primary time penalties that make up the overhead of checkpointing on training time, which we empirically evaluate in §5.4.

1. Time penalty during normal operation. Writing checkpoints to stable storage is a slow process given the large sizes of DLRMs, and training is paused during this time so that the saved model is consistent. Intuitively, the overhead of checkpointing on normal operation increases the more frequently checkpoints are taken and the longer it takes to write a checkpoint (and thus the larger the DLRM). This is illustrated empirically in §5.4.2.

2. Time penalty during recovery. Upon failure, a system using checkpointing must roll back the DLRM to the state of the most recent checkpoint by reading it from stable storage, and redo all training iterations that occurred between this checkpoint and the failure. New training iterations are paused during this time. The time needed to read checkpoints from storage can be significant [221] and grows with DLRM size. The expected time to redo iterations grows with the time between checkpoints: if checkpoints are written every T time units, this time will be 0 at best (failing just after writing a checkpoint), T at worst (failing just before writing a checkpoint), and $\frac{T}{2}$ on average. We demonstrate the recovery performance of checkpointing in §5.4.3.

Takeaway. Checkpointing suffers a fundamental tradeoff between training-time overhead in normal operation and that when recovering from failure. Increasing the time between checkpoints reduces the fraction of time paused when saving checkpoints, but increases the expected work to be redone in recovery. Facebook has also recently noted that reducing the storage and network bandwidth consumed by checkpointing DLRMs is critical for reducing load on these shared resources [129]. These overheads in training time and resource consumption increase with model size. Given the trends of increasing model size noted in §5.2.2 *checkpointing is slated to become an even larger overhead in training future DLRMs.*

This calls for alternatives for fault tolerance in DLRM training that scale to large DLRMs without a severe tradeoff between training-time overhead and recovery performance.

5.2.4 Reducing overhead of checkpointing

Approximation? Several recent approaches aim to reduce the overheads of checkpointing by taking approximate checkpoints or via approximate recovery [98, 129, 221, 258]. However, in the event of a failure, such techniques roll back an approximation of the true DLRM, which can potentially alter convergence and final accuracy. Given the significant business value generated by DLRMs, prior work has noted that even small drops in DLRM accuracy must be avoided [343]. Furthermore, our personal conversations with multiple practitioners working on large-scale DLRM training indicate that this potential accuracy drop introduces a source of uncertainty that makes debugging production DLRMs difficult, and thus is less desirable. Hence, ideally, one would reduce the overhead of checkpointing without compromising accuracy.

Asynchronous checkpointing? Another way to reduce the overhead of checkpointing is to asynchronously write checkpoints while training progresses by writing updates to stable storage as they are generated. This is feasible only if writing to stable storage can keep pace with the rate at which gradients are generated. As DLRM training systems have many workers operating asynchronously, gradients are generated at a high rate that stable storage cannot keep pace with. In fact, if storage could keep pace, then DLRM parameters could be kept in stable storage, rather than in memory. Thus, asynchronous checkpointing is not viable for DLRM training.

5.2.5 In-memory redundancy?

An alternative to checkpointing is to provision extra memory in the system to *redundantly* store DLRM parameters and optimizer state in memory in a fault-tolerant manner.

Replication. The simplest approach to redundancy is replication, in which a system proactively stores copies of the DLRM parameters in memory on separate servers that are kept up-to-date throughout training. Replicated DLRM training would use twice as much memory to store copies of each parameter on two servers. Gradients for a given parameter are sent to and applied on both copies. The system seamlessly continues training if a single server fails by accessing the replica, avoiding the need to redo work after a failure that checkpointing requires. Similar to traditional checkpointing, replicated training would preserve the accuracy guarantees of the underlying training system. However, such a replicated system requires at least twice as much memory as a non-replicated one. Given the large and growing sizes of embedding tables and optimizer state, this memory overhead is impractical.

Erasure codes. As described in §3, erasure codes often enable one to achieve the same level of fault tolerance as a replicated system, but with lower storage/memory overhead. It is, thus, interesting to consider whether erasure codes could be used for DLRM training. For example, consider the naive erasure-coded DLRM training system in Figure 5.1b in which parameters e_0, e_1, and e_2 are stored on three separate servers. Suppose the system must tolerate one server failure. An erasure code with parameters $k = 3$ and $r = 1$ could do so by encoding a parity unit as $p = e_0 + e_1 + e_2$ and storing this parity unit on a fourth server. Suppose the server holding e_1 fails. The system could recover e_1 using the erasure code's subtraction decoder: $e_1 = p - e_0 - e_2$. This setup can recover from any one of the servers failing by using only 33% more memory, while replication would require 100% more memory.

Takeaway. An ideal approach to fault-tolerant DLRM training would (1) avoid pauses during both normal operation and recovery, (2) have low memory overhead, (3) introduce no potential for accuracy loss, and (4) scale to large DLRMs. Erasure codes offer promising potential for achieving these goals. However, there are several challenges in using erasure codes for DLRM training. We describe these in detail and how they can be overcome in the next section.

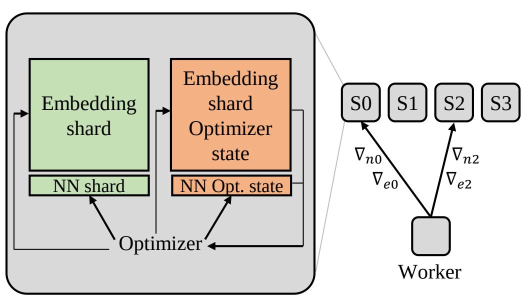
5.3 ECRM: erasure-coded DLRM training

We propose ECRM, a system that provides efficient fault tolerance to DLRM training via a careful design based on investigating the interplay between erasure codes and the unique characteristics of DLRM training systems. ECRM requires no pausing during training nor rolling back during recovery and provides the same accuracy guarantees as the underlying training system.

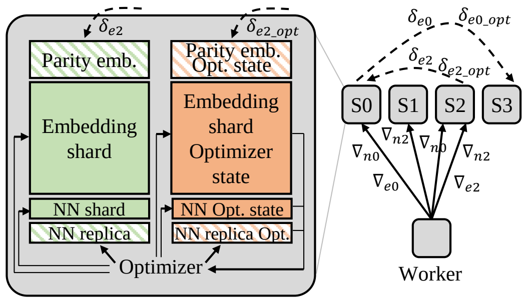
5.3.1 Overview of ECRM

Figure 5.2 shows a toy example comparing the high-level operation of traditional DLRM training systems and that of ECRM, as well as the detailed contents of an individual server in each system. Arrows illustrate the high-level flow of data when performing an update from a single worker.

In the original system (Figure 5.2a), a worker sends gradients for NN parameters (∇_{n0}, ∇_{n2}) and for embedding table entries (∇_{e0}, ∇_{e2}) to the servers hosting these parameters. As shown in the inset, the optimizer on a server reads optimizer state for the corresponding entries and NN parameters and uses this with the received gradients to compute updates, which are applied to relevant NN and embedding table parameters.



(a) Original DLRM training system



(b) ECRM (with $k = 3$ and $r = 1$). Dotted lines indicate values propagated by difference propagation.

Figure 5.2: Example of an update from a single worker with three servers (a) in a traditional DLRM training system and (b) in ECRM with $k = 3$. A detailed view of a single server is shown in the inset to the left.

Hybrid redundancy. As shown in the inset of Figure 5.2b, in addition to original shards of embedding tables and NN parameters (as well as their optimizer state), ECRM also maintains redundant versions of each of these components (hashed boxes). ECRM selects the type of redundancy to use for each type of parameter based on its unique resource footprint. In particular, ECRM considers the size of a given parameter type in memory, as well as the network bandwidth consumed for updating a parameter type.

To minimize memory overhead, all parameters in a DLRM would ideally be erasure coded. However, as we show in §5.3.3, correctly updating parities during DLRM training necessitates more communication than updating a replica. Thus, using erasure coding for parameters that have a high network-bandwidth footprint may lead to considerable overhead.

ECRM efficiently balances this tradeoff based on an observation related to the asymmetry between memory footprint and network bandwidth footprint for of parameters of many DLRMs: embedding tables (and their optimizer state) account for the vast majority of the memory footprint in DLRMs, while NN parameters (and their optimizer state) account for a minor fraction [145]. On the other hand, gradients for NN parameters account for the majority of network traffic during updates, while those for embedding table entries account for a much smaller portion. We discuss this in greater detail in §5.3.3.

Based on this asymmetry and the additional network traffic needed for updating erasure-coded parameters, ECRM *erasure codes embedding tables and their optimizer state, and replicates NN parameter and their optimizer state*. Doing so enables ECRM to operate with low memory overhead, as the vast majority of the DLRM's memory footprint is erasure coded, while reducing network bandwidth overhead. We describe specifically how ECRM overcomes challenges in using erasure codes for embedding tables in §5.3.2, and why ECRM leverages replication for NNs in §5.3.3.

Updating redundant parameters. Figure 5.2b also shows how ECRM keeps redundant parameters up-to-date. Workers send gradients for NN parameters (∇_{n0}, ∇_{n2}) to each server hosting a replica of an NN parameter. However, as will be described in detail in §5.3.2, this same process is insufficient for correctly updating parity embedding table entries and their optimizer state. To overcome this issue, we leverage "difference propagation" in §5.3.2 (denoted with dashed lines in Figure 5.2b), in which a server hosting an embedding table entry forwards the differences resulting from an update for that entry and its optimizer state to the server holding the corresponding parity.

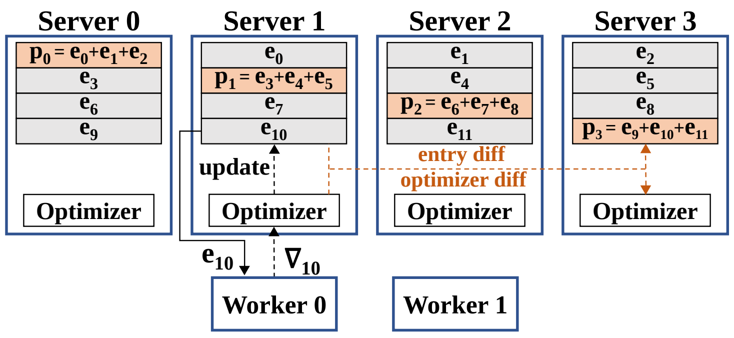


Figure 5.3: Example of rotating parity placement and difference propagation with $k = 3$, $r = 1$.

Recovery and consistency. Finally, ECRM enables training to continue during recovery from a server failure, and maintains the same consistency guarantees as the underlying DLRM training system. We describe these features in §5.3.4 and §5.3.5, respectively.

5.3.2 Erasure-coded embedding table entries

Encoding and placing embedding table entries. As described in §5.2.1, embedding tables and optimizer state are sharded across servers. ECRM encodes k embedding table entries from different shards to produce a "parity entry," and places the parity entry on a separate server. Optimizer state is similarly encoded to form "parity optimizer state," and placed on the same server as the corresponding parity entry.

Parities in ECRM are updated whenever any of the k corresponding embedding table entries are updated. Hence, parities are updated more frequently than the original entries, and must be placed carefully within the cluster so as not to introduce load imbalance among servers. ECRM uses rotating parity placement to distribute parities among servers, resulting in an equal number of parities per server. An example of this is shown in Figure 5.3 with $k = 3$: each server is chosen to host a parity in a rotating fashion, and the entries used to encode the parity are hosted on the three other servers. This approach is inspired by parity placement in RAID-5 systems [252].

Encoder and decoder. We focus on using erasure codes with parameter $r = 1$ (i.e., one parity per k entries, and recovering from a single failure) since it represents the most common failure scenario experienced by a cluster in datacenters [263]. Within the setting of $r = 1$, ECRM uses the simple summation encoder shown in Figure 5.3, and the corresponding subtraction decoder. For example, with $k = 3$, embedding table entries e_0, e_1, and e_2 are encoded to form parity $p = e_0 + e_1 + e_2$. If the server holding e_1 fails, e_1 will be recovered as $e_1 = p - e_0 - e_2$.

Correctly updating parities. We next describe challenges in correctly updating parities, and how ECRM overcomes them.

Challenges in keeping up-to-date parities. The naive approach to erasure-coded DLRM training in Figure 5.1b suffers a fundamental challenge in *correctly* updating parity entries when using a stateful optimizer (e.g., Adam, Adagrad).

Consider the example in Figure 5.1b with the Adagrad [125] optimizer. Let $e_{i,t}$ denote the value of embedding table entry e_i after t updates, and $\nabla_{i,t}$ denote the gradient for $e_{i,t}$. The update performed by Adagrad for $e_{0,t}$ with gradient $\nabla_{0,t}$ is:

$$e_{0,t+1} = e_{0,t} - \frac{\alpha}{\sqrt{G_{0,t} + \epsilon}} \nabla_{0,t} \tag{5.1}$$

where $G_{0,t} = \nabla^2_{0,0} + \nabla^2_{0,1} + \ldots + \nabla^2_{0,t}$, α is a constant learning rate, and ϵ is a small constant. $G_{0,t}$, which we call e_0's "accumulator," is an example of per-parameter optimizer state.

As described in §5.3.1, ECRM maintains one "parity optimizer parameter" for every k original optimizer parameters. In the example above, a "parity accumulator" would be $G_p = G_0 + G_1 + G_2$. This is easily kept up-to-date by adding to G_p the squared gradients for updates to each of the k original entries (e.g., $G_{p,t+1} = G_{p,t} + \nabla^2_{0,t+1}$). However, using this parity accumulator to update the parity entry based on $\nabla_{0,t}$ (i.e., replacing e_0 with e_p and G_0 with G_p in Eqn. 5.1) would result in an incorrect parity entry, since $G_{0,t} \neq G_{p,t}$.

The issue illustrated above arises for any stateful optimizer, such as Adagrad, Adam, and momentum SGD. Given the popularity of such optimizers, ECRM must employ some means of maintaining correct parities when using stateful optimizers. This could be overcome by replicating the k original optimizer parameters on the server hosting the parity. However, optimizer state for embedding tables is large and grows with embedding tables, making replication impractical.

Difference propagation. The challenge described above stems from sending gradients directly to the servers hosting parities: servers holding parity entries receive only the gradient for the original embedding table entry and must correctly update the parity entry and optimizer state. To overcome this challenge, ECRM leverages *difference propagation*. Under difference propagation, workers send gradients only to the servers holding embedding table entries for that gradient. After applying the optimizer to entries and updating optimizer state, the server sends the *differences* in entry and optimizer state to the server holding the corresponding parity entry. The receiving server adds these differences to the parity entry and optimizer state. This is shown in Figure 5.2b with the worker sending gradients for embedding table entries (∇_{e0}, ∇_{e2}) to Servers 0 and 2, which then send differences in entries (δ_{e0}, δ_{e2}) and optimizer state (δ_{e0_opt}, δ_{e2_opt}) to servers hosting the corresponding parity. By sending differences to servers, rather than sending gradients, difference propagation updates parity entries correctly when using stateful optimizers.

5.3.3 Replicated neural network parameters

We next describe how ECRM applies fault tolerance to neural network parameters and their optimizer state.

Is additional fault tolerance needed for NNs? Recall from §5.2.1 that workers in DLRM training pull all NN parameters from servers on each training iteration. Thus, each worker contains an approximate replica of the current NN parameters.[2] This may lead one to question whether ECRM can simply leverage the NN parameters pulled by workers as "natural" replicas of the NN sharded across servers.

While such a strategy may be promising for recovering NN parameters, it does not provide fault tolerance for optimizer state used for NN parameters. Because optimizer state is kept on

[2]This replica is only approximate because, as described in §5.2.1, workers operate asynchronously. Thus, the NN that a worker currently holds may not reflect the latest updates from other workers.

servers and is not read by workers, optimizer state for NNs is lost if a server fails. Thus, an alternative that keeps both NN parameters and their optimizer state redundant is needed.

Erasure coding NN parameters? A natural solution to keeping NN parameters and their optimizer state fault tolerant is to leverage erasure coding in a similar fashion to that described in §5.3.2. After all, these parameters are sharded across servers just like embedding tables, so the same technique could be applied to all DLRM parameters.

However, we find that leveraging erasure coding as described in §5.3.2 for NN parameters leads to significant performance overhead. Recall from §5.2.2 that, while embedding tables are updated sparsely, all NN parameters and their optimizer state are updated on every training iteration. We find that this leads to an imbalance in the amount of network bandwidth consumed for updating embedding table entries and NN parameters, with NN parameters consuming significantly more bandwidth. For example, for the DLRM used for the Criteo dataset [10], we find that the *network traffic incurred in a given training iteration for updating NN parameters is over 1.8$\times$ higher than that for embedding table entries*.

Performing erasure coding with difference propagation as described in §5.3.2 requires 200% network bandwidth overhead for a given update, as differences for both the original parameter and its associated optimizer state (each of which are the same size as the original gradient) must be forwarded to the server hosting the corresponding parity. Given the aforementioned dominance of NN parameters on network bandwidth, performing erasure coding with difference propagation for NNs can add considerable training-time overhead for DLRM training. However, as described in §5.3.2, difference propagation is necessary for correctly keeping parities up-to-date.

Replicating NN parameters. ECRM exploits the asymmetry between the network bandwidth consumed by gradients for NN parameters and the size of NN parameters. While gradients for NNs account for the majority of network bandwidth during updates, NN parameters represent a minor portion of the overall DLRM size. For example, for the DLRM used for Criteo [10], NN parameters and their optimizer state account for less than 1% of the overall DLRM size. Thus, NN parameters and their optimizer state can be replicated without adding significant memory overhead to the DLRM training system.

Replicating NN parameters and their optimizer state allows ECRM to avoid performing difference propagation for updating NNs (and its associated network bandwidth overhead). In ECRM, gradients for a given NN parameter are sent from workers to both servers hosting replicas of the given parameter. Each server containing a replica locally updates the parameter and its replica of the optimizer state. In this way, ECRM incurs half of the network bandwidth overhead for NN parameters as that incurred by erasure coding NN parameters with difference propagation: whereas difference propagation additionally sends both the difference for the NN parameter and the difference for its optimizer state, replication sends *only* the gradient for the NN parameters an additional time.

5.3.4 Pause-free recovery from failure

We next describe how ECRM recovers from failure without requiring training to pause.

Due to the property of erasure codes that any k out of the $(k + 1)$ original and parity units suffice to recover the original k units, ECRM can continue training even if a single server fails.

For example, a worker in ECRM could access the embedding table entry e_1 in Figure 5.3 even if Server 2 fails by reading e_0, e_2, and p, and decoding $e_1 = p - e_0 - e_2$. Such read operations that require decoding are referred to as "degraded reads" in erasure-coded systems. Similarly, NN parameters can be accessed via replicas on another server.

Challenges in erasure-coded recovery. Despite the ability to perform degraded reads, ECRM must still fully recover failed servers to remain tolerant of future failures. This is straightforward and efficient for NN parameters, as they are simply copied from a replica server and are small in size. However, prior work on erasure-coded storage has shown that full recovery can be time-intensive [263, 276]. Full recovery in ECRM requires decoding all embedding table entries and optimizer state held by the failed server. This consumes significant network bandwidth in transferring entries for decoding, and server CPU in performing decoding. Given the large sizes of embedding tables and their optimizer state, fully recovering before resuming training can significantly pause training.

Training during recovery via granular locking. Rather than solely performing degraded reads after a failure or pausing until full recovery is complete, ECRM *enables training to continue while full recovery takes place.* Upon failure, ECRM begins full recovery of lost embedding table entries and optimizer state. In the meantime, the system continues to perform new training iterations, with workers performing degraded reads to access entries from the failed server.

ECRM must avoid updating an entry in parallel with its use for recovery. If the recovery process reads the new value of the entry, but the old value of the parity entry (e.g., because the update has not yet reached the parity), then the recovered entry will be incorrect. ECRM uses granular locking to avoid this. The recovery process locks a fraction of the lost entries that it will decode. While this lock is held, updates to entries that will be used in recovery for the locked entries are buffered in memory on servers. Workers reading an updated, but locked entry do so by reading from the buffer. When a lock is released, buffered updates are applied to the embedding tables, and the next set of entries is locked. The number of entries covered by each lock introduces a tradeoff between time overhead in switching locks and server memory overhead for buffering, which can be navigated based on the requirements of a deployment.

5.3.5 Recovering a consistent DLRM

As discussed in §5.2.4, one of the goals of ECRM is to avoid introducing additional sources of accuracy loss beyond the original DLRM training system. We next describe how ECRM maintains the same consistency guarantees as the general asynchronous DLRM training system it builds upon.

Under general asynchronous training, concurrent updates to parameters from different workers can occur in an arbitrary order and can potentially overwrite one another. The same can occur with the redundant parameters employed by ECRM, matching the consistency guarantees of the original system.

However, one case requires care: server failure while an update is in flight. To better illustrate this, first consider that each training iteration updates parameters hosted on different servers. Suppose one server that holds the original copy of an embedding table entry used in the current iteration fails during the update step. The erasure code's decoding function would correctly recover if the corresponding parity entry was updated through difference propagation before the failure occurred. However, if the server failed before propagating its difference, the recovered DLRM would be inconsistent: other parameters involved in this iteration would be recovered to the state including this iteration's update, while the recovered version of the entry in question would not reflect this update.

The issue underlying this example is a lack of knowledge of whether a parity entry has been updated through difference propagation before a failure occurs.

Two-phase commit in ECRM. To avoid this potential inconsistency, ECRM employs two-phase commit (2PC) when updating parameters. The 2PC protocol is coordinated by individual workers and occurs for each training iteration a worker performs. 2PC in ECRM splits the process of updating a set of parameters on separate servers into two phases: In the first phase, updates for parameters (original and redundant) are computed and staged. In the second phase, staged updates are applied to parameters (original and redundant). Operating in this manner ensures that DLRM parameters are in a consistent state before recovery begins.

Leveraging 2PC in ECRM adds an extra round of communication for updating parameters. As will be shown in §5.4, this adds training-time overhead. Given that ECRM uses 2PC solely to protect against losing portions of an update when a failure occurs, a reader may question how important it is to preserve these full updates. After all, the training system on top of which ECRM is built is asynchronous, which means that concurrent updates from workers can potentially overwrite one another. Nevertheless, we have chosen to implement 2PC in ECRM so as to adhere to the design goal in §5.2.4 of introducing no additional sources of inaccuracy in training. While potentially a heavy-handed solution, 2PC in ECRM ensures that the fault tolerance technique used in training does not open additional sources of inconsistency (and the related uncertainty when debugging model accuracy). We show in §5.4.2 that the overhead of ECRM can be reduced if one is willing to forgo these guarantees by turning off 2PC.

5.3.6 Tradeoffs in ECRM

ECRM encodes k embedding table entries into a single parity entry (for $r = 1$) (similarly for optimizer state). Parameter k results in the following tradeoffs in ECRM:

Increasing k decreases memory overhead and fault tolerance. As ECRM encodes one parity entry for every k embedding table entries (and similarly for optimizer state), less memory is required for storing parities with increased k. However, since the erasure codes employed by ECRM can recover from any one out of $(k + 1)$ failures, increasing k decreases the fraction of failed servers ECRM can tolerate.

Increasing k *does not* change load during normal operation. As each embedding table entry in ECRM is encoded to produce a single parity entry, each update applied to an entry is also be applied to one parity. Thus, the total increase in load in terms of the number of updates performed due to ECRM is $2\times$, regardless of the value of k. In addition to this constant load increase, we show in §5.4.2 that ECRM balances the overall load for updates across servers.

Increasing k increases the time to fully recover. Recovering embedding tables in ECRM involves reading k entries from separate servers and decoding. Thus, the network traffic and computation used during recovery increases with k, which increases the time to fully recover. However, as described in §5.3.4, ECRM allows training to continue during this time.

5.4 Evaluation

We next evaluate ECRM. The highlights are as follows:

- ECRM recovers from failure up to $9.8\times$ faster than the average recovery time for checkpointing. Faster recovery enables DLRM training to meet the tight deadlines for deploying new versions of a model even when failures occur.
- ECRM enables training to proceed during recovery with only a 7%–13% drop in throughput, whereas checkpointing requires training to completely pause.
- ECRM reduces training-time overhead on large DLRMs by up to 66% compared to checkpointing. ECRM scales gracefully with DLRM size, showing promise for training both current and future DLRMs.
- While ECRM introduces additional load for updating parities, the impact of this increased load on training throughput is alleviated by improved cluster load balance.

5.4.1 Evaluation setup

We implement ECRM in C++ atop XDL, an open-source DLRM training system from Alibaba [169].

Dataset. We evaluate with the Criteo Terabyte dataset [10], which is commonly used for evaluating DLRM training systems. We randomly draw one day of samples from the dataset by picking each sample with probability $\frac{1}{24}$ in one pass through the dataset, and use this subset in evaluation to reduce storage requirements. This random sampling results in a sampled dataset that mimics the full dataset.

Models. We evaluate on various DLRMs based on the DLRM for the Criteo dataset from MLPerf [237], which has 13 embedding tables, for a total of around 200M entries each with 128 dense features. We use SGD with momentum as the optimizer, which adds one floating point value of optimizer state per parameter. Any other optimizer can also be used. The total size of the embedding tables and optimizer state is 220 GB. The DLRM uses a seven-layer multilayer perceptron with 128–1024 features per layer as a NN [25].

We evaluate on DLRMs of different size. First, we increase the number of embedding table entries in the DLRM (i.e., sparse dimension). This increases the memory required per server and the amount of data that must be checkpointed/kept redundant and recovered. We consider four variants of the original Criteo DLRM described above, with one-, two-, four-, and eight-times increase in the number of embedding table entries. We refer to each of these as Criteo-Original, Criteo-2S, Criteo-4S, and Criteo-8S, which have size 220, 440, 880, and 1760 GB, respectively. *We primarily focus on scaling the sparse dimension of embedding tables, as this reflects a prominent scaling trend observed today:* Facebook reports that from 2017 to 2021, the number of embedding table entries in DLRMs has increased by $17.5\times$ [280].

For completeness, we also evaluate on a DLRM in which we increase the size of embedding table *entries* (i.e., dense dimension). This increases the memory required per server, the amount of data that must be checkpointed/kept redundant, the network bandwidth in transferring entries/gradients, and the work done by workers and servers. Thus, this form of scaling complements model scaling in the sparse dimension. We consider a variant of the original Criteo-2S

DLRM described above but in which each embedding table entry is twice as large. The width of the input layer of the NN is also increased to accommodate the larger entry size. We refer to this DLRM as Criteo-2S-2D, and it has a total size of 880 GB.

Coding parameters. We evaluate ECRM with k of 2 and 4, which have 50%, 25% memory overhead, respectively. We use one lock during recovery by default (see §5.3.4), but also evaluate finer locking granularity.

Baselines. We compare ECRM to taking checkpoints to HDFS (1) every 30 minutes (Ckpt-30) and (2) every 60 minutes (Ckpt-60). We also compare ECRM to running (3) without any checkpointing or fault tolerance at all (No FT). As recovery time for checkpointing depends on when failure occurs (see §5.2.3), we additionally compare against the best-, average-, and worst-case scenarios for checkpointing when evaluating recovery from failure. Checkpointing to HDFS is representative of production DLRM training environments, which leverage HDFS-like distributed file systems [60, 129]. Furthermore, the checkpointing baselines we use have competitive performance: we find that checkpointing via HDFS is only 7%–27% slower than a (purposely unrealistic) baseline of writing directly to a local SSD. In addition, for the Criteo-Original DLRM, which is representative of current DLRMs, the checkpoint-writing overhead we report is similar to that reported in production training jobs by Facebook [221].

Cluster setup. We evaluate on AWS with 5 servers of type r5n.8xlarge, each with 32 vCPUs, 256 GB of memory, and 25 Gbps network bandwidth (due to memory requirements, r5n.12xlarge and r5n.24xlarge are used for DLRMs larger than 440 GB and 880 GB, respectively). We use 15 workers of type p3.2xlarge, each with a V100 GPU, 8 vCPUs, and 10 Gbps of network bandwidth. This ratio of worker to server nodes is inspired by XDL [169]. Workers use batch size of 2048. For checkpointing, we use 15 additional nodes of type i3en.xlarge as HDFS nodes, each equipped with NVMe SSDs and 25 Gbps of network bandwidth. All instances use AWS ENA networking.

Metrics. For performance during normal operation, we measure training throughput and training-time overhead, which is the percent increase in the time to train a certain number of samples. For performance during recovery, we measure the time to fully recover a failed server and training throughput during recovery (samples/second).

5.4.2 Performance during normal operation

We first compare the performance of ECRM and checkpointing during normal operation.

Figure 5.4 shows the training-time overhead of ECRM and checkpointing as compared to a system with no fault tolerance (and thus no overhead) in a two-hour training run. As DLRM size increases, ECRM's training-time overhead grows only slightly, while that of checkpointing increases significantly. For example, going from Criteo-Original to Criteo-8S, ECRM's training-time overhead with $k = 4$ increases by only 1.2×, while those of Ckpt-30 and Ckpt-60 increase by 7.2× and 7×, respectively. This leads to ECRM significantly reducing training-time overhead for large DLRMs: for Criteo-8S, ECRM has training-time overhead with $k = 4$ of 22%, while Ckpt-30 and Ckpt-60 have overheads of 65% and 31%, respectively. While one could checkpoint less frequently for such large DLRMs, doing so comes with the adverse effect of prolonged recovery times (as will be shown in §5.4.3).

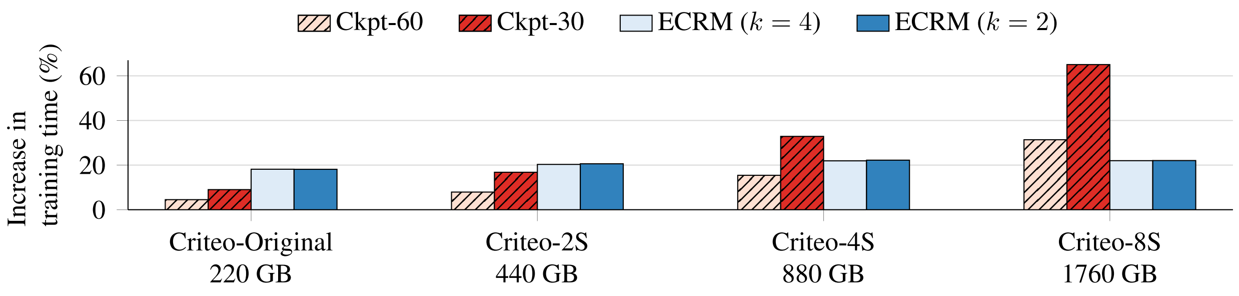


Figure 5.4: Training-time overhead.

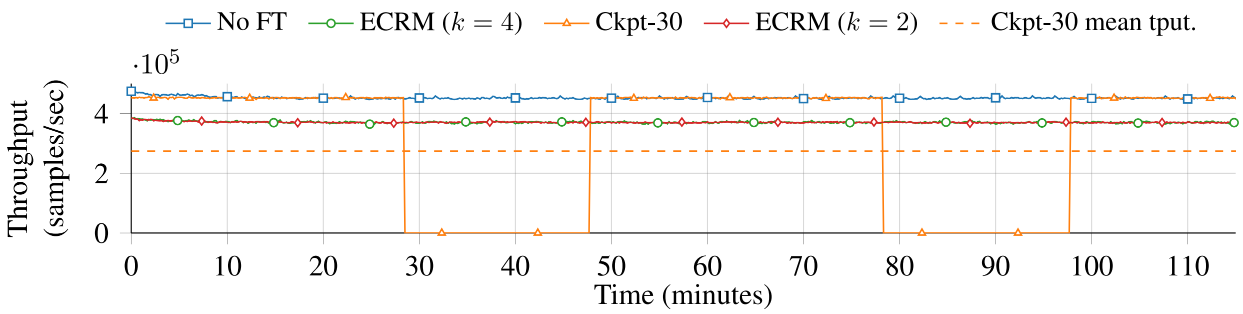


Figure 5.5: Throughput of training Criteo-8S.

As shown in Figure 5.4, ECRM does have higher training-time overhead than checkpointing for smaller DLRMs. However, it is important to note that DLRMs are expected to increase in size [280] (see §5.2.2). Furthermore, as will be shown in §5.4.3, ECRM significantly improves performance during recovery compared to checkpointing. Thus, ECRM is poised to remain a scalable solution for future DLRMs, without requiring one to severely trade normal-mode and recovery performance.

Figure 5.5 shows training throughput on Criteo-8S. ECRM has slightly lower throughput compared to No FT, while Ckpt-30 causes throughput to fluctuate from that of No FT, to zero when writing a checkpoint. This makes the average training throughput of Ckpt-30 (dashed line) lower than that of ECRM. The effects of this fluctuation are further shown in Figure 5.6: Ckpt-30 progresses slower than ECRM.

Effect of parameter k. As described in §5.3.6, ECRM has constant network bandwidth and CPU overhead during normal operation regardless of the value of parameter k. This is illustrated in Figures 5.4, 5.5, and 5.6, in which ECRM has nearly equal performance with $k = 2$ and $k = 4$.

Effect of ECRM on load imbalance. We next evaluate the effect of parity placement in ECRM on cluster load imbalance. We measure load by counting the number of updates that occur on each server when training Criteo-Original.

Without erasure coding, the most-heavily loaded server performs $2.28\times$ more updates than the least-heavily loaded server. In contrast, in ECRM with $k = 2$ and $k = 4$, this difference in load is $1.64\times$ and $1.58\times$, respectively. This indicates that the increased load introduced by ECRM is alleviated by *improved load balance.* Under ECRM, parities corresponding to the embedding table entries of a given server are distributed among all other servers. Thus, the same amount of load that an individual server experiences for non-parity updates will also be

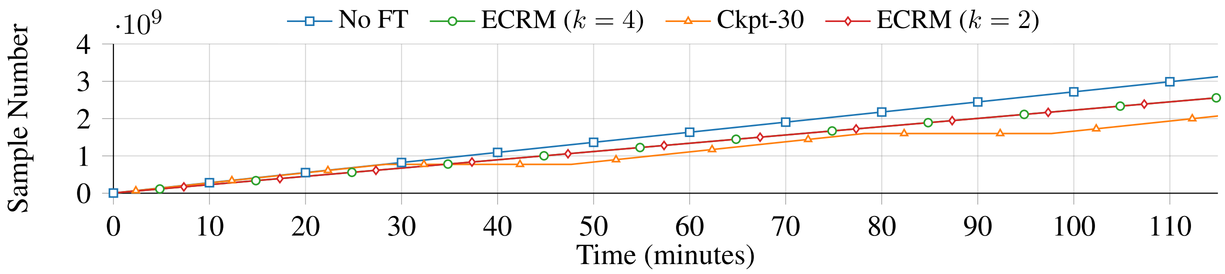


Figure 5.6: Progress of training Criteo-8S.

distributed among the other servers to update parities. While all servers experience increased load, the most-loaded (least-loaded) server in the absence of erasure coding is likely to experience the smallest (largest) increase in load due to the addition of erasure coding because all other servers for whom it hosts parities have lower (higher) load. Hence, the expected difference in load between the most- and least-loaded servers decreases. Thus, while ECRM doubles the total number of updates on the servers, its impact is alleviated by improved load balancing provided by its approach to parity placement.

Effect of embedding table entry width. We now evaluate ECRM with an increase in the size of each embedding table entry (i.e., the dense dimension). We compare the training-time overhead of ECRM with $k = 4$ on Criteo-4S and that on Criteo-2S-2D. These DLRMs have the same total size, but with Criteo-2S-2D having half of the embedding table entries as Criteo-4S, and with each entry being twice as large.

While ECRM's training-time overhead with $k = 4$ on Criteo-4S is 22%, that on Criteo-2S-2D is 27%. The higher training-time overhead on Criteo-2S-2D can be explained by the increased network traffic when training Criteo-2S-2D: because each entry in Criteo-2S-2D is twice as large as each in Criteo-4S, transmitting embedding table entries (and their gradients) with difference propagation consumes twice as much network bandwidth in Criteo-2S-2D as in Criteo-4S.

Ablation study. We next investigate the contributions to training-time overhead of ECRM's components.

We first consider the training-time overhead incurred for replicating NN parameters in ECRM. We compare ECRM with $k = 4$ to ECRM-NoRep, a version of ECRM that does not replicate NNs. On Criteo-Original, ECRM has training-time overhead of 18.2%, while that of ECRM-NoRep is only 5.2%. This indicates that the extra network traffic of keeping NN replicas up-to-date in ECRM adds considerable training-time overhead. Recall from §5.3.3, that replication of NN parameters could be avoided if one uses an optimizer that does not have optimizer state (e.g., SGD). Users leveraging such optimizers can potentially achieve significantly reduced overhead with ECRM by turning off NN replication.

We next consider the training-time overhead incurred by using two-phase commit (2PC) in ECRM (see §5.3.5). On Criteo-Original, the training-time overhead of ECRM-NoRep in the absence of 2PC further reduces from 5.2% down to only 2.6%. As described in §5.3.5, 2PC is used in ECRM to avoid losing updates that were in-flight when a failure occurrs. Applications that are willing to forgo this (likely small) potential hit in accuracy can turn off 2PC.

5.4.3 Performance during recovery

We next evaluate ECRM and checkpointing in recovering from failure. Recovery performance is best compared in Figure 5.8, which plots the throughput and training progress of ECRM and Ckpt-30 on Criteo-4S after a single server failure at time 15 minutes. ECRM fully recovers faster than the average case for Ckpt-30, and, critically, maintains throughput within 7%–13% of that during normal operation during recovery. In contrast, Ckpt-30 cannot perform new iterations during recovery. As shown in the bottom plot of Figure 5.8, ECRM's high throughput during recovery enables it to progress faster than the average case for Ckpt-30.

Figure 5.7 shows the time it takes for ECRM, Ckpt-30, and Ckpt-60 to recover a failed server. ECRM with $k = 4$ recovers 1.2–6.7× and 0.8–3.5× faster than the average case for Ckpt-60 and Ckpt-30, respectively (and up to 9.8× faster with $k = 2$). More importantly, unlike checkpointing, *ECRM enables training to continue with high throughput during recovery within half a minute of the failure occurring.*

Effect of parameter k and DLRM size. Figure 5.7 illustrates the discussion from §5.3.6 that it takes longer for ECRM to fully recover with higher value of parameter k. However, ECRM maintains high throughput during recovery for each value of k: after a failure occurs, ECRM resumes training with high throughput within 30 seconds (also see Figure 5.8).

Figure 5.7 also shows that the time to fully recover increases with DLRM size for both ECRM and checkpointing, as expected (see §5.2.3 and §5.3.6). ECRM's recovery time increases more quickly with DLRM size than checkpointing due to the k-fold increase in data read and compute performed by a single server in ECRM when decoding. However, this does not significantly affect training in ECRM because ECRM can continue training during recovery with high throughput.

Effect of lock granularity. We next evaluate the recovery performance of ECRM with varying lock granularity (see §5.3.4). We compare the full recovery time of ECRM with $k = 4$ when using one and ten locks. Using ten locks increases recovery time by 6.5% for Criteo-8S and 24.8% for Criteo-Original. Even when employing locks with finer granularity, and thus having longer recovery time, *ECRM continues to provide high training throughput during recovery*, unlike checkpointing. Switching locks involves (1) momentarily synchronizing workers and servers, and (2) copying updated embedding table entries from buffers to the original entries. Synchronization time is constant regardless of DLRM size, whereas the time to copy buffers grows with DLRM size. Thus, synchronization time is better amortized on larger DLRMs, reducing the overhead of lock switching for larger DLRMs.

5.5 Related Work

DLRM systems. System support for DLRM training and inference has recently received significant attention. Solutions tailored toward improving DLRM inference range from workload/system analysis [145, 216], model-system codesign [130, 144], and specialized hardware support [170, 316]. More recently, work has emerged for improving the performance of training DLRMs. For example, recent works from organizations that train large-scale DLRMs have described systems designed for training DLRMs [60, 169, 177, 231, 237, 280, 343]. Other works

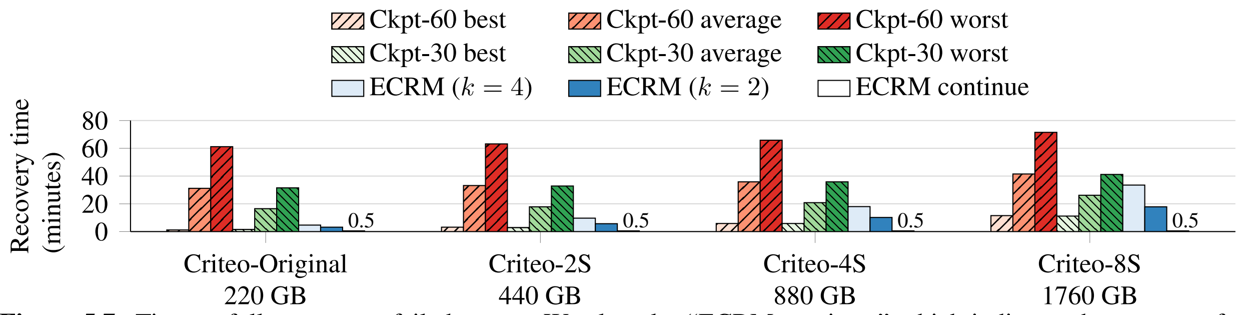


Figure 5.7: Time to fully recover a failed server. We also plot "ECRM continue," which indicates the amount of time between when a failure occurs and when ECRM begins continuing training during recovery. This value is small enough to be imperceptible, so we also indicate it in text.

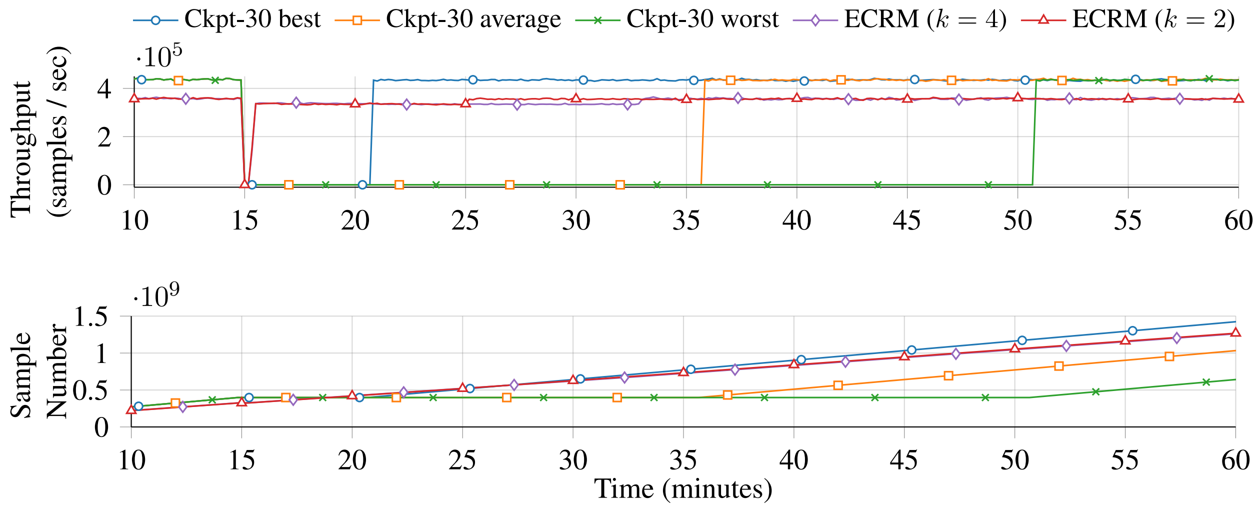


Figure 5.8: Training throughput (top) and progress (bottom) when recovering from failure at 15 minutes on Criteo-4S.

have focused on model-system codesign, such as reducing the sizes of embedding tables through compression and precision-reduction techniques [137, 329, 331].

ECRM differs from these works by its focus on fault tolerance for DLRM training and its use of erasure codes therein. ECRM could operate atop many of these works.

Checkpointing. Computer systems have long used checkpointing for fault tolerance (e.g., [112, 184, 230]). Some recent works optimize checkpointing in NN training [229, 238], but do not focus on DLRM training. In contrast, ECRM leverages the unique characteristics of DLRM training to use erasure codes for efficient fault tolerance. Other works have developed approximation-based checkpointing techniques to reduce the overhead of checkpointing in machine learning training [98, 258]. Unlike these approaches, ECRM does not change the accuracy guarantees of the underlying training system.

Most closely related to ECRM are two works that focus on reducing the overhead of checkpointing in DLRM training. Maeng et al. [221] use partial recovery to reduce the overhead of rolling back after failure: when a failure occurs, only the failed node rolls back to its most recent checkpoint. Eisenman et al. [129] use a combination of incremental checkpointing and reducing the numerical precision of checkpointed parameters. Both of these works potentially reduce the accuracy of DLRM training upon recovering from failure. While both works empirically demonstrate only small accuracy drops, they cannot provide the same accuracy guarantees as the underlying training system. ECRM differs from these works in two regards: (1) ECRM maintains the same accuracy guarantees as the underlying training system. This avoids uncertainty about whether the fault tolerance approach will deliver a model with the accuracy needed for deployment, and reduces effort in debugging model accuracy when there are multiple sources of inaccuracy present. (2) ECRM leverages in-memory redundancy to reduce the overhead of fault tolerance.

5.6 Conclusion

ECRM is a new approach to fault-tolerant DLRM training that employs erasure coding to overcome the downsides of checkpointing. ECRM exploits the unique characteristics of DLRM training to take a hybrid approach to in-memory redundancy by erasure coding the large embedding tables of DLRMs, while replicating the NN parameters. ECRM maintains up-to-date redundant parameters with low overhead, and enables training to continue during recovery, while maintaining the same accuracy guarantees as the underlying training system. Compared to checkpointing, ECRM reduces training-time overhead by up to 66%, recovers from failures up to 9.8$\times$ faster, and allows training to proceed without pauses both during normal operation and recovery. ECRM scales gracefully with increased DLRM size without enforcing a severe tradeoff between training-time overhead and recovery performance. While ECRM's benefits come with additional memory requirements and load on servers, the impact of these is alleviated by the fact that memory overhead is only fractional and that load gets evenly distributed. ECRM shows the potential of erasure coding as a superior alternative to checkpointing for fault tolerance in training current and future DLRMs.

Chapter 6

Overcoming the nonlinearity challenge via learning-based coded computation

Chapter 4 showed opportunities to improve the efficiency of existing coded-computation techniques applied to ML systems by taking advantage of opportunities unique to the ML system at hand. However, this approach required one to split a neural network into linear and non-linear layers. As we will describe below, this is not possible or efficient in all ML systems. For such ML systems, one would ideally perform coded computation over the entire neural network as a whole. However, as described in §3.3, existing approaches to coded computation cannot support non-linear functions, such as neural networks, without high resource overhead.

In the next three chapters of this thesis, we describe our work on expanding the reach of coded-computation approaches to be able to operate over neural networks as a whole by co-designing coded computation to the ML system at hand. We investigate the problem of protecting against slowdowns and fail-stop failures in distributed prediction serving systems used for low-latency inference. We leverage the property of many of these systems that returning an approximate prediction is better than returning no prediction at all to develop *learning-based approaches to coded computation.* We propose two different ways that learning can be used in the coded-computation framework: (1) learning encoders and decoders (§7); and (2) using simple encoders and decoders but learning a new "parity model" that operates over encoded data (§8). Learning-based coded computation enables accurate reconstruction of unavailable predictions resulting from neural network inference, and, when integrated into an open-source prediction serving system, enables significant reduction in tail latency in the presence of resource contention.

The rest of this section motivates learning-based coded computation at a high level.

6.1 Prediction serving systems

Machine learning is widely deployed in production services and user-facing applications [3, 15, 21, 61, 78]. This has increased the importance of inference, the process of returning a prediction from a trained machine learning model. *Prediction serving systems* are platforms operating in datacenter/cluster settings that host models for inference and deliver predictions for input queries.

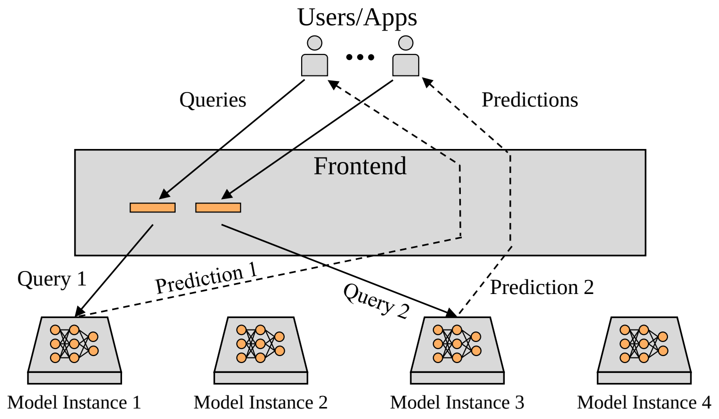


Figure 6.1: Prediction serving system

We refer to a model hosted for inference as a "deployed model." Numerous prediction serving systems are being developed by service providers [7, 14, 23] and open-source communities [26, 109, 244].

As depicted in Figure 6.1, prediction serving systems have two types of components: a frontend and model instances. The frontend receives queries and dispatches them to model instances for inference. Model instances are containers or processes that contain a copy of the deployed model and return predictions by performing inference on the deployed model.

Prediction serving systems employ scale-out architectures to serve predictions with low latency and high throughput and to overcome the memory and processing limitations of a single server [197]. In such a setup, multiple model instances are deployed on separate servers, each containing a copy of the same deployed model [109]. The frontend distributes queries to model instances according to a load-balancing strategy (e.g., single queue, round robin).

To meet the demands of user-facing production services, prediction serving systems must deliver predictions with low latency (e.g., within tens of milliseconds [109]). Similar to other latency-sensitive services, prediction services must adhere to strict service-level objectives (SLOs). Queries that are not completed by their SLO are often useless to applications [61]. In order to reduce SLO violations, prediction serving systems must minimize tail latency.

6.2 Need for slowdown and failure tolerance in prediction serving systems

As described above, prediction serving systems are often run in a distributed fashion and make use of many cluster resources (e.g., compute, network). These systems are thus prone to the slowdowns and failures common to cloud and cluster settings that cause inflated tail latency, such as those described in §2. There are numerous causes of inflated tail latency in these settings, such as multi-tenancy and resource contention [151, 161, 325], hardware unreliability and failures [66], and other complex runtime interactions [65]. Left unmitigated, these slowdowns inflate tail latency and cause violations of latency targets. Prediction serving systems must therefore employ some means to mitigate the effects of slowdowns and fail-stop failures.

A popular approach to tolerating slowdowns and failures in serving systems (e.g., web services) while remaining agnostic to the cause of slowdown is to issue redundant requests to multiple servers [113]. Previous work has theoretically motivated the potential for redundancy-based techniques to reduce latency [133, 171, 207, 281]. Redundancy-based approaches commonly navigate a tradeoff space between resource-overhead and latency. We next characterize existing redundancy-based techniques and where they operate within this tradeoff space.

Proactive approaches. A common technique used to achieve the lowest latency in recovering from slowdowns and failures is to proactively issue redundant requests to multiple servers and to wait only for the first replica to respond. Under such techniques, a system that replicates each query to d servers can tolerate $(d-1)$ slow/failed servers. By issuing redundant queries as soon as a query arrives in the system, proactive approaches mitigate slowdowns and failures with low latency. However, current proactive approaches result in high resource-overhead, as replicating queries to d servers requires d-times as many resources to handle increased load.

Reactive approaches. Reactive approaches operate with lower resource-overhead than replication-based approaches by issuing redundant queries only when confident that a slowdown or failure has occurred [45, 113, 335]. Under such approaches, each query is dispatched to a single server, and redundant requests are only issued if a certain amount of time has elapsed without receiving the result from the server. Such waiting time is commonly chosen to be long enough such that one is confident a server is running slowly if a result has not been returned by this time [113]. By issuing redundant queries only when confident that a slowdown or failure has occurred, reactive approaches reduce the amount of additional load introduced to a system, and thus the amount of resource-overhead necessary. However, by waiting for a significant amount of time before issuing redundant requests, reactive approaches are unable to mitigate slowdowns and failures with latency as low as proactive approaches.

An ideal approach to slowdown and failure tolerance for prediction serving systems would, thus, have resource efficiency similar to that of reactive approaches with recovery latency similar to proactive approaches.

6.3 Coded computation for prediction serving systems: opportunities and challenges

We now describe how coded computation could ideally be performed in a prediction serving system and challenges therein.

Recall the general coded computation setup described in §3.3 with $k = 2$ and $r = 1$. Given inputs X_1 and X_2, the goal is to return $\mathcal{F}(X_1)$ and $\mathcal{F}(X_2)$ for a given function $\mathcal{F}$, and for the setup to be able to tolerate one of the two computations failing. Coded computation achieves this by constructing a third "parity" unit P by encoding X_1 and X_2, operating over this parity using a third copy of function $\mathcal{F}$, and decoding by using any two of the three function outputs among $\mathcal{F}(X_1)$, $\mathcal{F}(X_2)$, and $\mathcal{F}(P)$.

Mapping this notation to the setting of prediction serving systems, function $\mathcal{F}$ would be the copies of the deployed model placed on each model instance, X_1 and X_2 would be input queries, and $\mathcal{F}(X_1)$ and $\mathcal{F}(X_2)$ would be predictions resulting from inference over the deployed model. Within this setup, the logical place to perform the encoding and decoding operations in coded computation is on the prediction serving system's frontend, as this is the entity in the system through which all queries and predictions flow.

Leveraging coded computation in prediction serving systems has the potential to enable a new position in the tradeoff between proactive and reactive approaches to handling slowdowns and failures: coded computation could bring resource efficiency similar to that of reactive approaches, but with the a recovery latency similar to that of proactive approaches.

However, applying coded computation to this setting, once again, raises the challenge described in §3.3 of handling non-linear operations within $\mathcal{F}$. The setting in §4 was able to handle this challenge by splitting a neural network $\mathcal{F}$ into linear and non-linear layers, and performing coded computation over the linear layers and replication over the non-linear layers. However, this same approach is not practical when applying coded computation to a prediction serving system. In a prediction serving system, each copy of $\mathcal{F}$ executes on a separate model instance. Thus, encoding and decoding operations in this setting require transmitting data from each model instance to the frontend. Performing this expensive network communication following each layer of a neural network would significantly slow down inference.

Thus, we desire a coded computation scheme that is capable of operating over a neural network $\mathcal{F}$ *as a whole.* Such an approach would encode only the inputs to the neural networks and decode using only the predictions resulting from neural networks.

6.4 Leveraging learning for coded computation

Machine learning has recently led to significant advances in complex tasks, such as image classification, natural language processing, and even in designing codes for communication [69, 182, 233]. This leads one to question: *can machine learning similarly help overcome the challenges of coded computation for non-linear functions?*

We answer this question in the affirmative by proposing and evaluating a *learning-based* coded-computation framework. We describe two distinct paradigms for leveraging machine learning for coded computation: (1) *Learning a code:* using neural networks as encoders and decoders to learn a code that enables coded computation over non-linear functions. (2) *Learning a parity computation:* using simple encoders and decoders (e.g., addition/subtraction), and instead learning a new computation over parities that enables reconstruction of unavailable outputs. These two methods are fundamentally new approaches to coded computation.

Each of the approaches to learning-based coded computation that we propose are capable of returning only *approximations* of unavailable function outputs. This is in contrast to traditional approaches to coded computation, which aim to reconstruct the exact unavailable function output. However, reconstructing approximations of unavailable predictions is often appropriate for the specific setting of coded computation within prediction serving systems for the following reasons: (1) Predictions resulting from inference are already approximations themselves, and (2) Approximate reconstructions resulting from coded computation in this setting are returned only when a prediction from the deployed model would otherwise be slow or failed. In this scenario, it is often preferable to return an approximate prediction rather than a late one or no prediction at all [61]. These reasons, in part, motivated our approach of using machine learning for coded computation in prediction serving systems.

The next two chapters describe these two approaches to learning-based coded computation.

Notational recap. Following §3.3, we consider a setting in which k copies of a neural network $\mathcal{F}$ are performed on separate servers. Each input X_i, is sent to one of the copies of $\mathcal{F}$ to compute and return $\mathcal{F}(X_i)$. Thus, given k inputs $X_1, X_2, \ldots, X_k$, the goal is to compute $\mathcal{F}(X_1), \mathcal{F}(X_2), \ldots, \mathcal{F}(X_k)$. The coded-computation schemes we consider will use an encoder $\mathcal{E}$ to generate r parities $P_1, P_2, \ldots, P_r$ which are each computed on by some function. A decoder $\mathcal{D}$ then takes in any k out of the total $(k+r)$ outputs from original and parity units to reconstruct any r unavailable outputs. Within the context of a prediction serving system, each input is a query and each output is a prediction. We focus on the scenario in which $r = 1$. This represents the typical unavailability faced by groups of $(k + r)$ servers ("stripes") for typical values of k and r in datacenters [263].

Chapter 7

Learning encoders and decoders for coded computation

This chapter describes the first of the two approaches to learning-based coded computation that we propose: learning encoders and decoders that enable approximate coded computation over neural networks. We thus term this "learning an erasure code."

Learning an erasure code involves learning the encoding and the decoding functions ($\mathcal{E}$ and $\mathcal{D}$). Unlike the traditional approach in erasure coding, we allow the outputs of the decoding function to be an *approximation* of the unavailable outputs. Approximate outputs are sufficient for many applications, such as machine learning algorithms since many of these algorithms themselves are approximate. For any input X and a given function $\mathcal{F}$, we denote the (approximate) reconstruction of $\mathcal{F}(X)$ as $\widehat{\mathcal{F}(X)}$. We express encoding and the decoding functions as neural networks and learn to approximate reconstructions. We train the neural networks for the encoding and the decoding functions in tandem via backpropagation through the given function $\mathcal{F}$.

Our approach is applicable for designing codes for coded computation over any differentiable non-linear function $\mathcal{F}$.[1] As described in §6, however, we focus our attention on learning codes for neural networks in prediction serving systems. We use the term "base model" to refer to $\mathcal{F}$. However, we emphasize that our solution extends to any differentiable function, making it applicable to a large class of tasks in machine learning and beyond.

We evaluate our framework using two neural-network based image classifiers as base models (a multi-layer perceptron (MLP) and ResNet-18) using MNIST [193], Fashion-MNIST [323], and CIFAR-10 [63] datasets. Our experimental results show that the proposed approach can *accurately reconstruct a significant fraction of the unavailable outputs*: for example, 98.87%, 92.06%, and 80.84% of ResNet-18 classifier outputs are accurately reconstructed on MNIST, Fashion-MNIST, and CIFAR-10 datasets respectively.

[1]Although our approach is applicable for linear functions as well, we focus primarily on non-linear functions. There are several existing works (e.g., [127, 128, 195, 203, 227, 269, 309, 333]) that address only linear functions. These approaches may be more suitable for linear functions as they guarantee exact reconstruction of unavailable outputs.

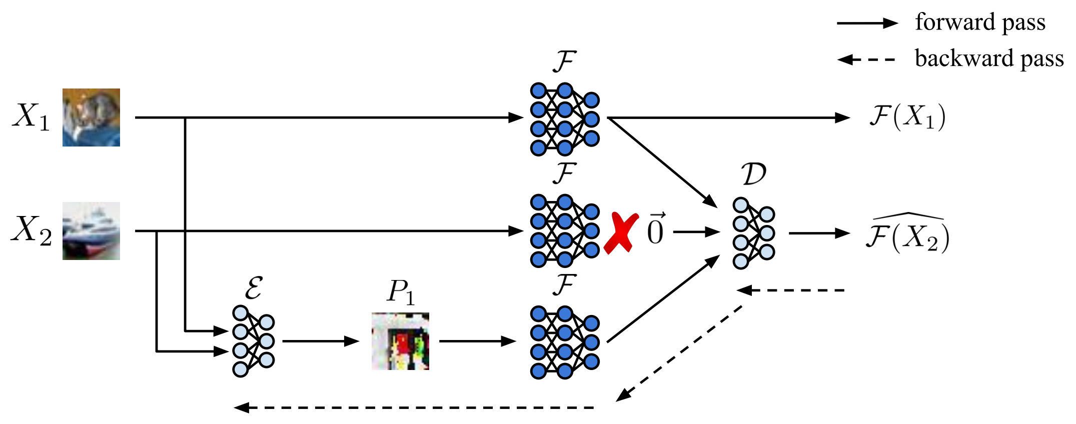


Figure 7.1: A forward and a backward pass in training the encoding and decoding functions for ($k = 2$, $r = 1$).

7.1 Learning a Code

In this section, we describe our proposed approach for learning erasure codes. We first present our training methodology, and subsequently describe the neural network architectures for learning the encoding and decoding functions.

7.1.1 Training methodology

Recall that our overall architecture has three functions: the given function $\mathcal{F}$ whose execution is to be protected using the learned codes, the encoding function $\mathcal{E}$, and the decoding function $\mathcal{D}$. During training the goal is to train the parameters of the neural networks for the encoding and the decoding functions. Note that the given function $\mathcal{F}$ is not modified during this training.

When the given function $\mathcal{F}$ is a machine learning algorithm, we train the encoding and the decoding functions using the same training dataset (whenever available) that was used to train $\mathcal{F}$. When such a training dataset is not available, which will be the case for generic functions $\mathcal{F}$ outside the realm of machine learning, one can instead generate a training dataset comprising pairs $(X, \mathcal{F}(X))$ for various values of X in the domain of $\mathcal{F}$. Each sample for training the encoding and decoding functions uses a set of k (randomly chosen) inputs from the training dataset. For any sample, we perform a forward and a backward pass for each of $\binom{k+r}{r}$ possible unavailability scenarios, except for the case where all unavailable outputs correspond to parity inputs (since the only role of parities is to aid in the reconstruction of unavailable outputs corresponding to the data inputs). Any iterative optimization algorithm, such as gradient descent and its variants, may be used for training.

A forward and a backward pass under our training method is illustrated in Figure 7.1. A forward pass involves the following steps. The k data inputs $X_1, X_2, \ldots, X_k$ are fed through the encoding function to generate r parity inputs $P_1, P_2, \ldots, P_r$. Each of the $(k + r)$ inputs (data and parity) are then fed through the given function $\mathcal{F}$. The resulting $(k + r)$ outputs $\mathcal{F}(X_1), \ldots, \mathcal{F}(X_k), \mathcal{F}(P_1), \ldots, \mathcal{F}(P_r)$ are fed through the decoding function $\mathcal{D}$, out of which no more than r are made unavailable (discussed in detail in §7.1.3). The decoding function outputs an (approximate) reconstruction for the unavailable function outputs among $\mathcal{F}(X_1), \ldots, \mathcal{F}(X_k)$.

Table 7.1: Neural network architectures for encoding functions employing fully-connected (FC) and convolutional (Conv) layers. All convolutional layers have stride of 1. In each network, ReLU activation functions are used after all but the final layer. The activation functions are omitted above for brevity.

(a) MLPEncoder

Layer	Layer Type
1	FC: $kn^2 \times kn^2$
2	FC: $kn^2 \times rn^2$

(b) ConvEncoder

Layer	Layer Type
1	Conv: 3×3, dilation: 1
2	Conv: 3×3, dilation: 1
3	Conv: 3×3, dilation: 2
4	Conv: 3×3, dilation: 4
5	Conv: 3×3, dilation: 8
6	Conv: 3×3, dilation: 1
7	Conv: 1×1, dilation: 1

The corresponding backward pass involves using any chosen loss function (discussed in detail below) for backpropogation through $\mathcal{D}$, $\mathcal{F}$, and $\mathcal{E}$. We train the encoding and decoding functions in tandem via backpropagation of losses directly through $\mathcal{F}$. In other words, the parameters of the encoding and the decoding functions are updated by backpropagating through $\mathcal{F}$. Since training backpropagates directly through $\mathcal{F}$, this approach is applicable to any given differentiable function $\mathcal{F}$.

We consider two types of losses when training the encoding and the decoding functions:

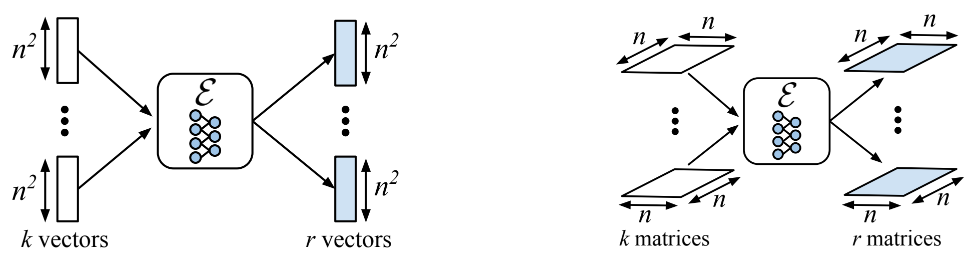
1. *Loss with respect to function outputs:* Loss is computed between the function output $\mathcal{F}(X)$ and its approximate reconstruction $\widehat{\mathcal{F}(X)}$ produced by the decoding function. This approach can be employed for any given function $\mathcal{F}$.
2. *Loss with respect to true labels:* When $\mathcal{F}$ is a machine learning algorithm, there is an additional option of calculating the loss using the true labels (when available in the training dataset). For example, consider $\mathcal{F}$ to be a neural network for image classification, and let Y represent the true label for an input image X. Under this approach, the loss is computed between the true label Y and the label predicted using $\widehat{\mathcal{F}(X)}$.

The specific loss functions employed in our evaluation under both of the above approaches are discussed in §7.2.

7.1.2 Encoding function architectures

We consider two neural network architectures for learning the encoding function. For concreteness, we describe the proposed architectures below by setting the given function $\mathcal{F}$ as a neural network for image classification over m classes, and use the term "base model" to refer to $\mathcal{F}$. For such an $\mathcal{F}$, each data input X is an $n \times n$ pixel image. Each function output $\mathcal{F}(X)$ is an m-length vector representing output from the last layer of the neural network classifier.

We now describe two neural network architectures for learning the encoding function. Recall that the encoding function acts on k data inputs to create r parity inputs. We first describe the



(a) MLPEncoder

(b) ConvEncoder

Figure 7.2: Encoding function architecture: (a) MLPEncoder flattens inputs into n^2-length vectors and produces n^2-length parity vectors. (b) ConvEncoder encodes over inputs in their original dimension, as $n \times n$ matrices. The k input matrices are treated as k input channels.

architecture considering single-channel images as inputs, and consider multi-channel images in §7.1.2.

MLPEncoder We first consider a simple 2-layer multilayer-perceptron (MLP) encoding function architecture, which we call *MLPEncoder*. Under this architecture, the $n \times n$ data inputs are flattened into n^2-length vectors, as illustrated in Figure 7.2a. The k flattened vectors from inputs $X_1, X_2, \ldots, X_k$, are concatenated to form a single kn^2-length input vector to the MLP. The first fully-connected layer of the MLP produces a kn^2-length hidden vector. The second fully-connected layer produces an rn^2-length output vector, which represents the r parity inputs. Each layer used in MLPEncoder is outlined in Table 7.1a.

The fully-connected nature of the MLP allows for computation of arbitrary combinations from the kn^2 total inputs with a small number of layers. While simple in design and effective for many scenarios (as will be shown in §7.2.2), the high parameter count of the fully-connected layers can lead to overfitting. We next describe an alternate encoding function architecture that avoids overfitting, which we call *ConvEncoder*.

ConvEncoder The ConvEncoder architecture makes use of multiple convolutional layers as detailed in Table 7.1b. Unlike MLPEncoder, ConvEncoder computes over data inputs in their original $n \times n$ representation. As depicted in Figure 7.2b, the k inputs to the encoding function are treated as k *input channels* to the first convolution layer. This is similar to feeding the RGB representations of an image to a convolutional neural network for image classification. We explain how the encoder handles multi-channel inputs in §7.1.2.

The traditional use of convolutional layers for image classification involves repeated downsampling of an input image to gradually expand the receptive field of convolutional filters. This approach works well when the output dimension of the network is significantly smaller than the input dimension, which is often the case for image classification. However, the encoding function of a code produces outputs that have the same dimension as the inputs (see Figure 7.2b). Hence, using convolutional layers with downsampling would necessitate subsequent upsampling to bring the outputs back to the input dimension. This has been shown to be inefficient in the context of image segmentation [132]. To overcome this issue, we employ *dilated convolutions* [132]. As shown in Figure 7.3, this approach increases the receptive field of a convolutional filter

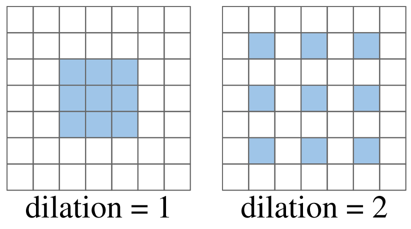


Figure 7.3: (3×3) Dilated Convolution. Traditional convolution (left) operates on adjacent cells. Dilated convolution (right) considers cells spread apart from one another.

exponentially with linear increase in the number of layers.

Table 7.1b shows each layer of ConvEncoder. The first layer has k input channels and the final layer has r output channels, one for each parity to be produced. Each of the intermediate layers has $20k$ input channels and $20k$ output channels. We increase the receptive field of convolutions by increasing the dilation factor, borrowing this architecture from Yu et al. [132], where it was used for image segmentation.

ConvEncoder uses less parameters than MLPEncoder but requires more layers to enable combinations of all input pixels. The lower parameter count compared to MLPEncoder helps avoid overfitting, as will be shown in §7.2.2.

Multi-channel input It is common to represent color images as having multiple channels. For example, a 32×32 RGB image would consist of 3 channels, each 32×32 in size, representing the pixel values of each of the red, green, and blue components. Our encoding function architectures handle multi-channel inputs by encoding across each channel independently. For example, an encoding function with k RGB images as inputs would encode across the k red channels to produce r "red" parity channels, and similarly for green and blue channels. The r "red", "green", and "blue" parity channels are combined together to create r parity "RGB" images.

7.1.3 Decoding function architecture

As in §7.1.2, for concreteness, we describe our decoding function architecture by setting the given function $\mathcal{F}$ as a neural network for image classification over m classes. Recall that we refer to $\mathcal{F}$ as the base model. The base model output $\mathcal{F}(X)$ for any input X is an m-length vector representing output from the last layer of the neural-network classifier. Further recall that the base model $\mathcal{F}$ is applied on the $(k + r)$ inputs $X_1, X_2, \ldots, X_k, P_1, \ldots, P_r$ on separate, unreliable compute nodes that can fail or straggle arbitrarily. The decoding function $\mathcal{D}$ operates on all the available base model outputs and reconstructs approximations of up to r unavailable base model outputs among $\mathcal{F}(X_1), \mathcal{F}(X_2), \ldots, \mathcal{F}(X_k)$.

Figure 7.4 presents the overall architecture of our decoding process. The two key design choices for the decoding function architecture are: (a) representation of the unavailable base model outputs at the input layer of the neural network for the decoding function, and (b) the neural network architecture used for learning the decoding function.

Representing unavailability A key design consideration for the decoding function is in the representation of the unavailable base model outputs at its input layer. We design the decod-

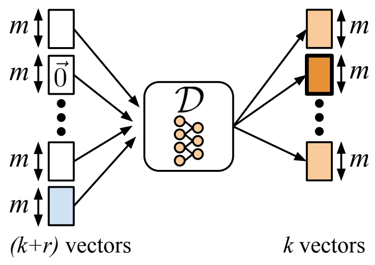


Figure 7.4: Decoding function architecture: The second input is unavailable and is set to a vector of zeros and the second (bolded) output represents its reconstruction. Parity inputs are shaded.

Table 7.2: Neural network architecture for decoding function employing fully-connected (FC) layers. ReLU activation functions are used after all but the final layer. The activation functions are omitted above for brevity.

Layer	Layer Type
1	FC: $(k + r)m \times km$
2	FC: $km \times km$
3	FC: $km \times km$

ing function to take the $(k + r)$ vectors of length m, $\mathcal{F}(X_1), \dots, \mathcal{F}(X_k), \mathcal{F}(P_1), \dots, \mathcal{F}(P_r)$, as inputs. Some of these inputs to the decoding function will be unavailable. In place of any unavailable input, we insert a vector of all zeros. Note that an alternative approach is to provide the decoding function with only the (concatenated) available inputs. We chose the former as it allows us to learn a decoding function that depends on the relative position of the unavailable inputs; providing only the available inputs would hide this information. This approach is inspired by traditional (hand-crafted) erasure codes whose decoding functions leverage positional information. Correspondingly, the output of the decoding function maintains positional information and consists of k vectors $\widehat{\mathcal{F}(X_1)}, \dots, \widehat{\mathcal{F}(X_k)}$, each representing an approximate reconstruction of corresponding potentially unavailable function output. Only those approximate reconstructions corresponding to unavailable function outputs will be used.

Decoding function architecture We design the neural network for learning the decoding function as a 3-layer MLP as described in Table 7.2. We use the raw outputs of the base model $\mathcal{F}$ as input to the decoding function. Note that we do not convert such outputs to a probability distribution (via a softmax operation) as is typically done during training of classifiers.

7.2 Evaluation

As discussed in §7.1, we evaluate our approach of learning codes for coded computation over non-linear computations by setting the base model $\mathcal{F}$ as inference on neural-network based image classifiers. For any input X, $\mathcal{F}(X)$ represents the output from the last layer of the neural network used as the base model. We start by describing our experimental setup and then present results using two neural-network based image classifiers as base models (a multi-layer perceptron (MLP) and ResNet-18) on the MNIST [193], Fashion-MNIST [323], and CIFAR-10 [63]

Table 7.3: Test accuracies for the base models used in our experiments on the MNIST, Fashion-MNIST, and CIFAR-10 datasets.

Base Model	MNIST	Fashion-MNIST	CIFAR-10
ResNet-18	0.9920	0.9285	0.9347
Base-MLP	0.9793	0.8947	-

datasets. Finally, we present a more detailed analysis of the accuracy attained by the learned codes and the quality of the predictions obtained from the reconstructed outputs.

7.2.1 Experimental setup

We implement all the encoding and decoding function architectures as well as the training methodology using PyTorch [43].

Loss functions used in training. As discussed in §7.1.1, we use two approaches for calculating the loss when training the neural networks for the encoding and the decoding functions: (a) calculating the loss with respect to the base model output and (b) calculating the loss with respect to the true label (when available in the training dataset). When calculating the loss with respect to the base model output, we experiment with two different loss functions: (1) mean-squared error (denoted by *MSE-Base*) and (2) KL-divergence (denoted by *KL-Base*) between $\widehat{\mathcal{F}(X)}$ and $\mathcal{F}(X)$. When calculating loss with respect to the true labels of the underlying task, we use the cross-entropy between $\widehat{\mathcal{F}(X)}$ and the true label of X (denoted by *XENT-Label*).

Overall, we find only marginal differences in the accuracies attained with these different loss functions.

Base models. We experiment with two neural network architectures as base models: Base-MLP and ResNet-18. Base-MLP is a 3-layer multilayer-perceptron used for the MNIST and Fashion-MNIST datasets containing three fully-connected layers with dimensions 784×200, 200×100, and 100×10 with ReLU activation functions following all but the final layer. We choose an MLP model due to its simplicity and its reported success on MNIST [193]. ResNet-18 [153] is an 18-layer state-of-the-art neural network for image classification consisting of convolutional, pooling, and fully-connected layers.[2]. We choose to use ResNet-18 for two reasons: (a) it has been shown to provide high classification accuracy on both CIFAR-10 and Fashion-MNIST, and (b) it is a significantly more complex model than Base-MLP and thus provides a good alternative evaluation point for our proposed approach. Table 7.3 shows the classification accuracies of the base models. We do not use Base-MLP as a base model for CIFAR-10 because it achieves low accuracy.

Encoding and decoding function architectures. Recall that, in §7.1.2 we presented two architectures for learning the encoding function, MLPEncoder and ConvEncoder, and an MLP-

[2]We use the ResNet-18 model from `https://github.com/zalandoresearch/fashion-mnist`

based architecture for learning the decoding function. We present experimental results for all the proposed architectures.

Parameters and training details. We perform experiments for all combinations of the configuration settings discussed above for $k = 2$ and $k = 5$ with $r = 1$. We focus on $r = 1$ because this corresponds to the case of typical unavailability faced in today's data centers as shown from measurements on Facebook's data analytics cluster [262, 263]. With $r = 1$, the parameter settings with $k = 2$ and $k = 5$ correspond to 50% and 20% redundant computation, respectively.

Training uses minibatches of 64 samples for $k = 2$ and 32 samples for $k = 5$. Each sample in the minibatch consists of k images from the dataset drawn randomly without replacement (i.e., no image is used more than once per epoch). Thus each minibatch for $k = 2$ consists of 128 images and for $k = 5$ consists of 160 images from the dataset. The encoding and decoding functions are trained in tandem using the Adam optimizer [118] with learning rate of 0.001 and L2-regularization of 1×10^{-5}. The weights for the convolutional layers are initialized via uniform Xavier initialization [138] and weights for the fully-connected layer are initialized according to $\mathcal{N}(0, 0.01)$. All bias values are initialized to zero.

Accuracy metrics. We measure the accuracy of the reconstructed output with respect to the machine learning task at hand using the following two metrics:

1. *Prediction accuracy*: This metric measures the accuracy of the reconstructed output based on its ability to recover the label predicted by the base model output. For example, when $\mathcal{F}$ is a classifier, for any input X, a reconstructed output $\widehat{\mathcal{F}(X)}$ is considered accurate if the classes predicted using $\widehat{\mathcal{F}(X)}$ and $\mathcal{F}(X)$ are identical. More formally, let $\mathcal{C}(.)$ denote the argmax operator (which is typically used to predict the class label from the output layer of a neural network classifier). For an input X, a reconstructed output $\widehat{\mathcal{F}(X)}$ is considered accurate if $\mathcal{C}(\widehat{\mathcal{F}(X)}) = \mathcal{C}(\mathcal{F}(X))$. This metric helps decouple the accuracy of the learned code in its ability to reconstruct unavailable base model outputs and the classification accuracy of the base model itself.
2. *Label accuracy*: This metric measures the accuracy of the reconstructed output based on the true label. For example, when $\mathcal{F}$ is a classifier, for any input X with true label Y, a reconstructed output $\widehat{\mathcal{F}(X)}$ is considered accurate if the class predicted using $\widehat{\mathcal{F}(X)}$ and Y are identical. More formally, using the terminology defined above, a reconstructed output $\widehat{\mathcal{F}(X)}$ is considered accurate if $\mathcal{C}(\widehat{\mathcal{F}(X)}) = Y$.

In the results presented, for both the metrics, we calculate the aggregate accuracy by averaging the accuracy over all unavailability scenarios. If unavailability statistics are known, one can instead weigh different unavailability scenarios based on the statistics.

7.2.2 Experimental results

Main results. Table 7.4 presents evaluation results on test datasets for all combinations of datasets and deployed models with values of k of 2 and 5, and $r = 1$. We consider a simple MLP architecture, Base-MLP, as well as ResNet-18 on the MNIST, Fashion-MNIST, and CIFAR-10

Table 7.4: Prediction accuracy and label accuracy for $r = 1$ for all configuration settings.

				MLPEncoder		ConvEncoder	
Dataset	Base Model	k	Training Loss Function	Prediction Accuracy	Label Accuracy	Prediction Accuracy	Label Accuracy
MNIST	Base-MLP	2	KL-Base	0.9769	0.9758	0.9831	0.9854
			MSE-Base	0.9885	0.9776	0.9767	0.9770
			XENT-Label	0.9737	0.9768	0.9769	0.9893
		5	KL-Base	0.9371	0.9340	0.9485	0.9518
			MSE-Base	0.9480	0.9424	0.9339	0.9357
			XENT-Label	0.9251	0.9232	0.9474	0.9533
	ResNet-18	2	KL-Base	0.9742	0.9760	0.9836	0.9854
			MSE-Base	0.9788	0.9806	0.9887	0.9888
			XENT-Label	0.9774	0.9796	0.9904	0.9925
		5	KL-Base	0.9460	0.9466	0.9571	0.9585
			MSE-Base	0.9349	0.9359	0.9415	0.9433
			XENT-Label	0.9401	0.9407	0.9171	0.9178
Fashion-MNIST	Base-MLP	2	KL-Base	0.9215	0.8800	0.9128	0.9080
			MSE-Base	0.8484	0.8196	0.8471	0.8253
			XENT-Label	0.9107	0.8808	0.9036	0.9185
		5	KL-Base	0.8275	0.7997	0.8300	0.8153
			MSE-Base	0.7133	0.6987	0.7302	0.7193
			XENT-Label	0.8259	0.8037	0.8364	0.8282
	ResNet-18	2	KL-Base	0.9002	0.8845	0.9206	0.9031
			MSE-Base	0.8960	0.8815	0.8982	0.8892
			XENT-Label	0.8947	0.8880	0.9242	0.9164
		5	KL-Base	0.8219	0.8133	0.8033	0.7960
			MSE-Base	0.7726	0.7672	0.7939	0.7885
			XENT-Label	0.8277	0.8203	0.8303	0.8248
CIFAR-10	ResNet-18	2	KL-Base	0.4293	0.4283	0.7889	0.8002
			MSE-Base	0.4107	0.4116	0.8074	0.8204
			XENT-Label	0.4284	0.4238	0.7980	0.8106
		5	KL-Base	0.1889	0.1895	0.5368	0.5382
			MSE-Base	0.1913	0.1936	0.6431	0.6466
			XENT-Label	0.1874	0.1890	0.5224	0.5287

datasets. With $k = 2$, across all datasets and configurations, the best of the proposed learned encoder and decoder pair achieve a label accuracy of no more than 11.5% lower than available accuracy. This is a small drop in accuracy compared to the available-mode accuracies when considering that reconstructions only come into play when a function output would otherwise be slow or failed. This represents a promising step forward in coded computation for neural networks, as existing coded-computation approaches are inapplicable even for simple neural networks.

Value of parameter k. Across all datasets, base models, and encoding function architectures, we find that test accuracy is significantly higher when $k = 2$ than when $k = 5$. We believe this is because for $k = 5$, a single parity needs to pack information about 5 input images, whereas for $k = 2$, a single parity contains information about only 2 inputs images. Note that with r fixed, the value of k controls the amount of redundant base model computation. For $r = 1$, having $k = 2$ corresponds to 50% redundant base model computation and having $k = 5$ corresponds to 20% redundant base model computation. The above observation hints towards a potential fundamental tradeoff between prediction accuracy and the amount of redundant computation. The difference between $k = 2$ and $k = 5$ is more pronounced for the Fashion-MNIST and CIFAR-10 datasets, which we attribute to the increased complexity of the dataset.

Effect of base model complexity. We find that the complexity of the base model does not have an adverse effect on the accuracy of the learned code. As discussed in §7.2.1, ResNet-18 is a significantly more complex model than Base-MLP, including many more layers of non-linearities. Despite this higher complexity, we see that the learned codes achieve similar accuracies for both Base-MLP and ResNet-18. This is promising, since it suggests that the proposed approach is effective even for complex base models.

Encoding function architectures. For the MNIST and Fashion-MNIST datasets, there is little difference in the accuracies attained by the two proposed neural network encoding function architectures, MLPEncoder and ConvEncoder. The difference between the two architectures comes to fore in the more complex dataset CIFAR-10, where ConvEncoder greatly outperforms MLPEncoder. MLPEncoder's high parameter count causes it to overfit and plateau at low accuracy on CIFAR-10, while ConvEncoder is able to reach significantly higher accuracy.

We next take a deeper look at the prediction accuracy attained on the configurations discussed above and analyze cases where the predicted class from reconstructed outputs does not match that from the base model outputs.

Prediction accuracy stratified based on accuracy of the base model. We next take a deeper look at the prediction accuracy attained on the configurations discussed above and analyze cases where the predicted class from reconstructed outputs does not match that from the base model outputs.

We find that the learned codes achieve a significantly higher prediction accuracy on the set of samples that the base model classifies correctly as compared to the set of samples that the base model classifies incorrectly. Figure 7.5 shows the prediction accuracy on these two sets of samples in the highest-performing configurations of experiments listed in Table 7.4 (which are highlighted in Table 7.5 for clarity). We find that the learned codes achieve, on average, 2.19

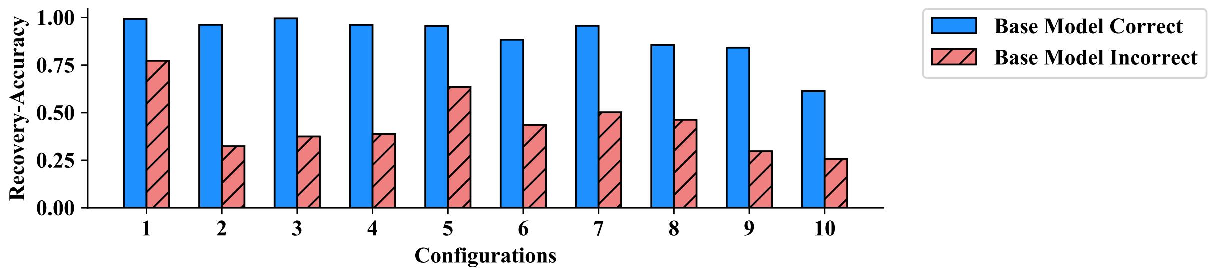


Figure 7.5: Prediction accuracy attained on samples which the base model correctly classifies ("Base Model Correct") and those which the base model incorrectly classifies ("Base Model Incorrect"). The configuration for each bar is listed in Table 7.5.

Table 7.5: Highest performing configurations in terms of prediction accuracy from Table 7.4.

Cfg.	Dataset	Base Model	k	Prediction accuracy	Encoder	Loss function
1	MNIST	Base-MLP	2	0.9885	MLPEncoder	KL-Base
2	MNIST	Base-MLP	5	0.9485	ConvEncoder	KL-Base
3	MNIST	ResNet-18	2	0.9904	ConvEncoder	XENT-Label
4	MNIST	ResNet-18	5	0.9571	ConvEncoder	KL-Base
5	Fashion-MNIST	Base-MLP	2	0.9215	MLPEncoder	KL-Base
6	Fashion-MNIST	Base-MLP	5	0.9364	ConvEncoder	XENT-Label
7	Fashion-MNIST	ResNet-18	2	0.9242	ConvEncoder	XENT-Label
8	Fashion-MNIST	ResNet-18	5	0.8277	MLPEncoder	XENT-Label
9	CIFAR-10	ResNet-18	2	0.8074	ConvEncoder	MSE-Base
10	CIFAR-10	ResNet-18	5	0.6431	ConvEncoder	MSE-Base

times higher prediction accuracy on the set of samples that the base model classifies correctly ("Base Model Correct" in Figure 7.5) as compared to the set of samples that the base model classifies incorrectly ("Base Model Incorrect" in Figure 7.5). Thus, the prediction accuracy of the learned codes is higher on samples where it indeed matters more to reconstruct accurately.

Analysis of errors in the learned code. Here we analyze how poor is the class predicted from inaccurate reconstructions. Specifically, we look at the samples for which the reconstructed output is inaccurate with respect to the base model output , and analyze how far the resulting predicted class label is from the label predicted by the base model output. We quantify the quality of the label predicted from an inaccurate reconstruction by its rank in the base model output. A rank of 2 means that the class predicted using the reconstruction was ranked second in the base model output.[3] Figure 7.6 shows the fraction of inaccurate reconstructions which lead to predicted labels that have rank 2 and rank 3 in the base model output for the configurations considered in Table 7.5. We see that, on average, 61.29% of the inaccurate reconstructions result

[3]Note that rank 1 is unattainable since we are analyzing only those instances for which the predicted class from the reconstruction does not match that of the base model output.

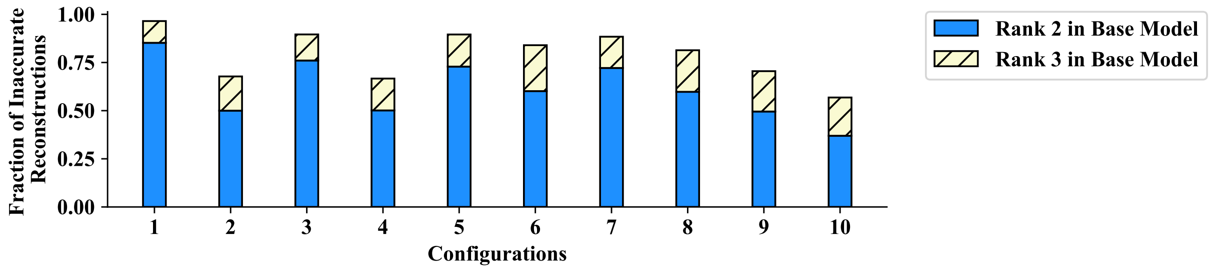


Figure 7.6: Fraction of inaccurate reconstructions that are rank 2 and 3 in the base model output. The configuration for each bar is listed in Table 7.5.

in a class prediction that is the second best in the base model output. Furthermore, on average, 79.12% of the inaccurate reconstructions result in a class prediction among the top 3 predictions of the base model output. Thus, even when the class prediction resulting from a reconstructed output does not match that of the base model output, the predicted class is not far off.

7.3 Related Work

A host of recent works have explored using coding-theoretic approaches to fault-tolerant execution of distributed linear computations such as matrix multiplication. Lee et al. [195] use a family of codes called "maximum-distance-separable" (MDS) codes to mitigate stragglers in distributed matrix-vector multiplication. Dutta et al. [127] propose Short-Dot codes to decompose long dot products that arise in certain matrix-vector multiplications into smaller products which facilitates parallel computation of such products. Li et al. [203] present a framework for navigating the tradeoff between computation time and communication time in coded computation schemes for matrix multiplication. Yu et al. [333] propose Polynomial Codes for distributed matrix multiplication, which reconstruct the full matrix multiplication result using the minimal number of results from workers. Sparse Codes are introduced by Wang et al. [309] to exploit the sparsity of matrix operands in order to reduce decoding complexity in coded matrix-matrix multiplication. Dutta et al. [128] employ linear codes for fault-tolerant distributed convolution between two vectors. Reisizadeh et al. [269] propose a scheme to balance the load across compute nodes for coded, distributed matrix-multiplication by taking into account heterogeneity of compute resources. Mallick et al. [227] propose using rateless codes for distributed matrix-vector multiplication in order to make use of partial work completed by straggling nodes. In comparison to the above works which are applicable to *only linear* computations, we present a learning-based approach that learns codes that can handle any differentiable *non-linear* computation.

In another direction in coded computation, several recent works present approaches to using codes to bring fault tolerance to specific iterative optimization algorithms that are employed during training of machine learning algorithms. Tandon et al. [296] propose a straggler mitigation scheme for data-parallel gradient descent which involves having multiple copies of the data across the worker nodes. Under this scheme, each worker node sends a carefully constructed linear combination of its computed gradients to a master node such that the master node can complete a gradient descent iteration without having to wait for results from all the worker nodes. Karakus et al. [180, 181] propose a coded-computation approach wherein both the data and labels of a training set are encoded, and the original optimization algorithm is directly run on the encoded training dataset. For specific optimization algorithms (e.g., gradient descent and L-BFGS) and machine learning tasks (e.g., ridge regression, matrix factorization, and logistic regression), the authors present code constructions that achieve stable convergence and reduced runtime as compared to replication-based approaches. Maity et al. [226] encode the second moment of the data matrix using LDPC codes in order to mitigate the effect of stragglers on gradient descent. The authors show that encoding the second moment reduces the number of aggregation steps necessary per training iteration compared to directly encoding the data matrix. In contrast to these lines of work that focus on specific iterative optimization algorithms that arise during the training phase of machine learning, the focus of our work is to add fault tolerance through redundant computation to any differentiable non-linear computation that arise during the *inference* phase of machine learning.

Multiple recent works have explored taking a learning approach to designing decoding algorithms for *existing* error-correcting-codes employed in the domain of communication. For example, Nachmani et al. [233] propose using feed-forward and recurrent neural networks for decoding a family of codes called "block codes". Kim et al. [182] show that recurrent neural networks can learn close-to-optimal decoding algorithms for several classes of well known codes employed in the domain of communication. In comparison with these works, we propose and establish the feasibility of taking a learning-based approach to coded computation, rather than for channel communication.

Another related line of work is on using neural networks for image compression and cryptography [59, 297, 299]. While these lines of work are similar in spirit to learning an erasure code (transforming input data into alternate representation for later reconstruction), the overall goal, and thus the structure of the architecture and the training methodology differ significantly.

7.4 Conclusion

Coded computation is an emerging technique which makes use of coding-theoretic tools to protect against failures and stragglers in distributed computation. However, the applicability of current techniques to general computation, including machine learning algorithms, is limited due to the lack of codes that can handle non-linear functions. We propose a novel learning-based approach for designing erasure codes that approximate unavailable outputs for any differentiable non-linear function. We present carefully-designed neural-network architectures and a training methodology for learning the encoding and decoding functions. We show that our learned codes can accurately reconstruct many of the unavailable class predictions from image classifiers for MNIST, Fashion-MNIST, and CIFAR-10 datasets, respectively. These results are highly promising as they show the potential of using learning-based approaches for designing erasure codes and herald a new direction for coded computation by handling general non-linear computations.

Practical considerations. While the results presented above show the promise of learning encoders and decoders for coded computation, there are several practical challenges with deploying this approach in prediction serving systems. As described previously, within the setting of prediction serving systems, the frontend is the logical place to perform encoding and decoding operations. As neural networks are often computationally expensive, using neural network encoders and decoders could increase the latency of recovering from a slowdown or failure. To be feasible, this approach necessitates using hardware acceleration for encoders and decoders, which requires using a more expensive frontend node.

We next describe a second approach to learning-based coded computation that is designed to overcome these practical concerns.

Chapter 8

Practical learning-based coded computation via parity models

The previous chapter illustrated the promise of using learning to overcome the challenge of non-linearity in coded computation, but raised practical challenges for deployment within prediction serving systems: learned encoding and decoding functions may be too expensive to deploy on a prediction serving system's frontend without hardware acceleration.

In this chapter, we propose a fundamentally new approach to coded computation that overcomes these practical challenges. Rather than designing new encoders and decoders, as done by prior approaches to coded computation, we propose to use simple, fast encoders and decoders (such as addition and subtraction) and instead design a *new computation over parities*. Within the context of prediction serving systems, this new computation is a separate model, which we call a "parity model." As depicted in Figure 8.1, instead of the extra copy of the deployed neural network used by current coded-computation approaches ($\mathcal{F}$, in formal notation), we introduce a parity model (which we denote as $\mathcal{F}_P$). The challenge of this approach is to design a parity model that enables accurate reconstruction of unavailable function outputs. We address this by designing parity models as neural networks, and learning a parity model that enables simple encoders and decoders to reconstruct slow or failed function outputs.

By learning a parity model and using simple, fast encoders and decoders, this approach enables coded computation over non-linear computations, like neural networks, while operating with low latency without requiring hardware acceleration for encoding and decoding. We illustrate these practical benefits by implementing parity models in ParM ($\overline{\text{par}}$ity $\underline{\text{m}}$odels), a prediction serving system designed to make use of erasure codes to protect against slowdowns and failures. As depicted in Figure 8.1, ParM encodes multiple queries together into a parity query. A parity model transforms the parity query such that its output enables a decoder to reconstruct slow or failed predictions.

The predictions returned by ParM are the same as any prediction serving system in the absence of slowdowns and failures. When slowdowns and failures do occur, the output of ParM's decoder is an approximate reconstruction of a slow or failed predictions. Reconstructing approximations of unavailable predictions is appropriate for inference, as predictions are already approximations and because ParM's reconstructions are returned only when a prediction from the deployed model would otherwise be slow or failed. In this scenario, it is preferable to return

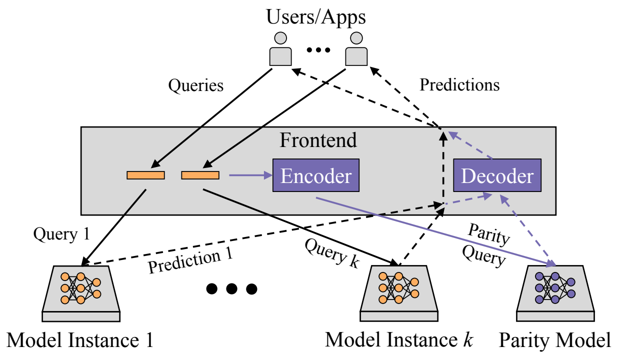


Figure 8.1: Architecture of a prediction serving system and components introduced by ParM (shown in purple).

an approximate prediction rather than a late one or no prediction at all [61].

We have built ParM atop Clipper [109], an open-source prediction serving system. We evaluate the accuracy of ParM's reconstructions on a variety of neural networks and inference tasks such as image classification, speech recognition, and object localization. We also evaluate ParM's ability to reduce tail latency across varying query rates, levels of background load, and amounts of redundancy. ParM reconstructs unavailable predictions with high accuracy and reduces tail latency while using 2-4$\times$ less additional resources than replication-based approaches. For example, using only half of the additional resources as replication, ParM's reconstructions from ResNet-18 models on various tasks are within a 6.5% difference in accuracy compared to if the original predictions were not slow or failed. This enables ParM to reign in tail latency: ParM reduces 99.9th percentile latency in the presence of load imbalance for a ResNet-18 model by up to 48% compared to a baseline that uses the same amount of resources as ParM, while maintaining the same median. This brings tail latency up to $3.5\times$ closer to median latency, enabling ParM to maintain predictable latencies in the face of slowdowns and failures. These results show the promise of parity models to open new doors for imparting efficient slowdown and failure tolerance to prediction serving systems.

8.1 Design of ParM

We first describe ParM in detail for protecting against any one out of k predictions experiencing slowdown or failure (i.e., $r = 1$). This setting is motivated by measurements of production clusters [261, 263]. Section 8.4 describes how the proposed approach can tolerate multiple unavailabilities (i.e., $r > 1$) as well. We will continue to use the notation of $\mathcal{F}$ to represent the deployed model, X_i to represent a query, $\mathcal{F}(X_i)$ to represent a prediction resulting from inference on $\mathcal{F}$ with X_i, and $\widehat{\mathcal{F}(X_i)}$ to represent a reconstruction of $\mathcal{F}(X_i)$ when $\mathcal{F}(X_i)$ is unavailable.

8.1.1 System architecture

The architecture of ParM is shown in Figure 8.2. ParM builds atop a typical prediction serving system architecture that has m instances of a deployed model. Queries sent to the frontend are batched (according to a batching policy) and dispatched to a model instance for inference on

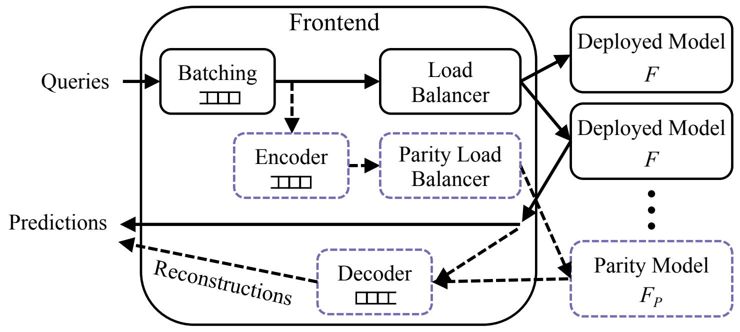


Figure 8.2: Components of a prediction serving system and those added by ParM (dotted). Queues indicate components which may group queries/predictions (e.g., coding group).

the deployed model. Query batches[1] are dispatched to model instances according to a provided load-balancing strategy.

ParM adds an encoder and a decoder on the frontend along with $\frac{m}{k}$ instances of a parity model. Each parity model uses the same amount of resources (e.g., compute, network) as a deployed model. ParM thus adds $\frac{1}{k}$ resource-overhead.

As query batches are dispatched, they are placed in a *coding group* consisting of k batches that have been consecutively dispatched. A coding group acts similarly to a "stripe" in erasure-coded storage systems: the query batches of a coding group are encoded to create a single "parity batch." ParM treats queries as though they are independent from one another. In particular, the queries that form a coding group need not have been sent by the same application or user. Encoding takes place across individual queries of a coding group: the ith queries of each of the k query batches in a coding group are encoded to produce the ith query of the parity batch. Encoding *does not* delay query dispatching as query batches are immediately handled by the load balancer when they are formed, and placed in a coding group for later encoding. The parity batch is dispatched to a parity model and the output resulting from inference over the parity model is returned to the frontend. Encoding is performed on the frontend rather than on a parity model so as to incur only $\frac{1}{k}$ network bandwidth overhead. Otherwise, all queries would need to be replicated to a parity model prior to encoding, which would incur $2\times$ bandwidth overhead.

Predictions that are returned to the frontend are immediately returned to clients. ParM's decoder is only used when any one of the k prediction batches from a coding group is unavailable. ParM enables flexibility in determining when a prediction is considered unavailable, and thus when decoding is necessary. A prediction could be considered unavailable if the $(k-1)$ other predictions from its coding group and the output of the parity model have been returned before the prediction in question. Alternatively, a system could set a timeout after which a prediction is considered unavailable if the conditions above are still not met.

When a prediction is considered unavailable, the decoder uses the outputs of the parity model and the $(k-1)$ available model instances to reconstruct an approximation of the unavailable prediction batch. Approximate predictions are returned only when predictions from the deployed model are unavailable. ParM thus reduces tail latency when an unavailable prediction is reconstructed before the actual prediction for the query returns (e.g., from a slow model instance).

[1]We use the terms "batch" and "query batch" to refer to one or more queries dispatched to a model instance at a single point in time.

8.1.2 Encoder and decoder

Introducing and learning parity models enables ParM to use simple, fast erasure codes to reconstruct unavailable predictions. ParM can support many different encoder and decoder designs, opening up a rich design space. We will illustrate the power of parity models by using a simple addition/subtraction erasure code, and showing that even with this simple encoder and decoder, ParM significantly reduces tail latency and accurately reconstructs unavailable predictions. This simple encoder and decoder is applicable to a wide range of inference tasks, including image classification, speech recognition, and object localization, allowing us to showcase ParM's applicability to a variety of inference tasks. A prediction serving system that is specialized to a specific inference task could potentially benefit from designing task-specific encoders and decoders for use in ParM, such as an encoder that downsamples and concatenates image queries for image classification. We evaluate an example of such a task-specific code in §8.2.2.

Under the simple addition/subtraction encoder and decoder showcased in this chapter, the encoder produces a parity as the summation of queries in an coding group, i.e., $P = \sum_{i=1}^{k} X_i$. Queries are normalized to a common size prior to encoding, and summation is performed across corresponding features of each query (e.g., top-right pixel of each image query). The decoder subtracts $(k-1)$ available predictions from the output of the parity model $\mathcal{F}_P(P)$ to reconstruct an unavailable prediction. Thus, an unavailable prediction $\mathcal{F}(X_j)$ is reconstructed as $\widehat{\mathcal{F}(X_j)} = \mathcal{F}_P(P) - \sum_{i \neq j}^{k} \mathcal{F}(X_i)$.

8.1.3 Parity model design

ParM uses neural networks for parity models to learn a model that transforms parities into a form that enables decoding. In order for a parity model to help in mitigating slowdowns, the average runtime of a parity model should be similar to that of the deployed model. One simple way of enforcing this is by using the same neural network architecture for the parity model as is used for the deployed model (i.e., same number and size of layers). Thus, if the deployed model is a ResNet-18 architecture, the parity model also uses ResNet-18, but with parameters trained using the procedure that will be described in §8.1.4. As a neural network's architecture determines its runtime, this approach ensures that the parity model has the same average runtime as the deployed model. We use this approach in our evaluations.

In general, a parity model is not required to use the same architecture as the deployed model. In cases where it is necessary or preferable to use a different architecture for a parity model, such as when the deployed model is not a neural network, a parity model could be designed via architecture search [348]. However, we do not focus on this scenario.

It is common in classification tasks to use a softmax function to convert the output of a neural network into a probability distribution. We do not apply a softmax function to the output of a parity model, as the desired output of a parity model is not necessarily constrained to be a probability distribution. As we will describe in §8.1.4, we employ a loss function in training that does not require the output of parity model to be a probability distribution.

8.1.4 Training a parity model

A parity model is trained prior to being deployed.

Training data. The training data are the parities generated by the encoder, and training labels are the transformations expected by the decoder. For the simple encoder and decoder described in §8.1.2, with $k = 2$, training data from queries X_1 and X_2 are $(X_1 + X_2)$ and labels are $(\mathcal{F}(X_1) + \mathcal{F}(X_2))$.

Training data is generated using queries that are representative of those issued to the deployed model for inference. A parity model is trained using the same dataset used for training the deployed model, whenever available. Thus, if the deployed model was trained using the CIFAR-10 [63] dataset, samples from CIFAR-10 are used as queries $X_1, \ldots, X_k$ that are encoded together to generate training samples for the parity model. Labels are generated by performing inference with the deployed model to obtain $\mathcal{F}(X_1), \ldots, \mathcal{F}(X_k)$ and summing these predictions to form the desired parity model output. For example, if the outputs of a deployed model are vectors of n floating points, as is the case in a classification task with n classes, a label would be generated as the element-wise summation of these vectors. ParM can also use as labels the summation of the true labels for queries.

If the dataset used for training the deployed model is not available, a parity model can be trained using queries that have been issued to ParM for inference on the deployed model. The predictions that result from inference on the deployed model are used to form labels for the parity model. In this case, as expected, ParM can deliver benefits only after the parity model has been trained to a sufficient degree.

Loss function. While there are many loss functions that could be used in training a parity model, we use the mean-squared-error (MSE) between the output of the parity model and the desired output as the loss function. We choose MSE rather than a task-specific loss function (e.g., cross entropy) to make ParM applicable to many inference tasks.

Training procedure. Training a parity model involves the same optimization procedure commonly used for training neural networks. In each iteration, k samples are drawn at random from the deployed model's training dataset and encoded to form a parity sample. The parity model performs inference over this parity query (forward pass) to generate $\mathcal{F}_P(P)$. A loss value for this parity query is calculated between $\mathcal{F}_P(P)$ and the desired parity model output (e.g., $\mathcal{F}(X_1) + \mathcal{F}(X_2)$ for the addition/subtraction code with $k = 2$). Parity model parameters are updated based on this loss value using the standard backpropogation algorithm (backward pass). This iterative process continues until a parity model reaches sufficient accuracy on a validation dataset.

Reducing the time to train a parity model was not a goal of this work; we describe inefficiencies of the procedure above and their potential solutions in §8.4.

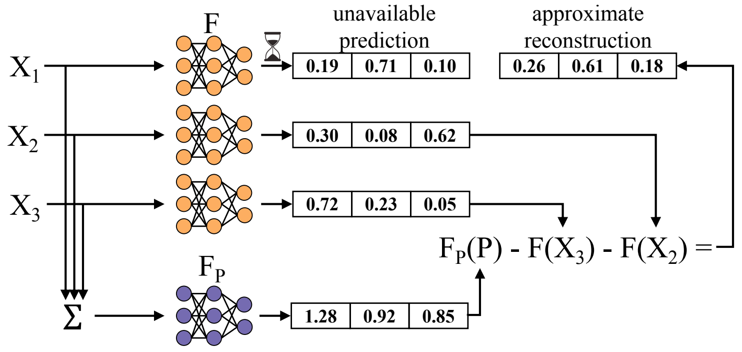


Figure 8.3: Example of ParM ($k = 3$) mitigating a slowdown.

8.1.5 Example

Figure 8.3 shows an example of how ParM mitigates unavailability of any one of three model instances (i.e., $k = 3$). Queries X_1, X_2, X_3 are dispatched to three separate model instances for inference on deployed model $\mathcal{F}$ to return predictions $\mathcal{F}(X_1), \mathcal{F}(X_2), \mathcal{F}(X_3)$. The learning task here is classification across n classes. Each $\mathcal{F}(X_i)$ is thus a vector of n floating-points ($n = 3$ in Figure 8.3). As queries are dispatched to model instances, they are encoded (Σ) to generate a parity $P = (X_1 + X_2 + X_3)$. The parity is dispatched to a parity model $\mathcal{F}_P$ to produce $\mathcal{F}_P(P)$. In this example, the model instance processing X_1 is slow. The decoder reconstructs this unavailable prediction as $(\mathcal{F}_P(P) - \mathcal{F}(X_3) - \mathcal{F}(X_2))$. The reconstruction provides a reasonable approximation of the true prediction that would have been returned had the model instance not been slow (labeled as “unavailable prediction”).

In this example, ParM protects against any one out of three model instances being unavailable by using one extra instance to serve a parity model. To achieve the same level of failure tolerance, a replication-based approach requires three extra model instances. Thus, in this example, ParM operates with $3\times$ less additional resources than replication-based approaches. More generally, ParM operates with k-times less additional resources of replication-based approaches.

8.2 Evaluation of Accuracy

In this section, we evaluate ParM’s ability to accurately reconstruct unavailable predictions.

8.2.1 Experimental setup

We use PyTorch [43] to train separate parity models for each parameter k, dataset, and deployed model.

Inference tasks and models. Learning a parity model to enable reconstruction of predictions represents a fundamentally new learning task. Therefore, we evaluate ParM on popular inference tasks and datasets to establish the potential of using parity models. Specifically, we use popular image classification (CIFAR-10 and 100 [63], Cat v. Dog [50], Fashion-MNIST [323], and MNIST [193]), speech recognition (Google Commands [312]), and object localization (CUB-200 [315]) tasks. We evaluate ParM on the ImageNet dataset [273] in §8.2.2.

As described in §8.1.3, a parity model uses the same neural network architecture as the deployed model. We consider five different architectures: a multi-layer perception (MLP),[2] LeNet-5 [194], VGG-11 [287], ResNet-18, and ResNet-152 [153]. The former two are simpler neural networks while the others are variants of state-of-the-art neural networks.

Hyperparameters. We consider values for parameter k of 2, 3, and 4, corresponding to 33%, 25%, and 20% redundancy. We use Adam [118], learning rate of 0.001, L2-regularization of 10^{-5}, and batch sizes between 32 and 64. Convolutional layers use Xavier initialization [138], biases are initialized to zero, and other weights are initialized from $\mathcal{N}(0, 0.01)$.

Encoder and decoder. Unless otherwise specified, we use the generic addition encoder and subtraction decoder described in §8.1.2. We showcase the benefit of employing task-specific encoders and decoders within ParM in §8.2.2.

Metrics. Analyzing erasure codes for storage and communication involves reasoning about performance under normal operation (when unavailability does not occur) and performance in "degraded mode" (when unavailability occurs and reconstruction is required). These different modes of operation are similarly present for inference. The overall accuracy of any approach is calculated based on its accuracy when predictions from the deployed model are available (A_a) and its accuracy when these predictions are unavailable (A_d, "degraded mode"). If f fraction of deployed model predictions are unavailable, the overall accuracy (A_o) is:

$$A_o = (1 - f)A_a + fA_d \tag{8.1}$$

ParM aims to achieve high A_d; it does not change the accuracy when predictions from the deployed model are available (A_a). We report both A_o and A_d.

We report accuracies on test datasets, which are not used in training. Test samples are randomly placed into groups of k and encoded to produce a parity. For each parity P, we compute the output of inference on the parity model as $\mathcal{F}_P(P)$. During decoding, we use $\mathcal{F}_P(P)$ to reconstruct $\widehat{\mathcal{F}(X_i)}$ for each X_i that was used in encoding P, simulating every scenario of one prediction being unavailable. Each $\widehat{\mathcal{F}(X_i)}$ is compared to the true label for X_i. For CIFAR-100, we report top-5 accuracy, as is common (i.e., the fraction for which the true class of X_i is in the top 5 of $\widehat{\mathcal{F}(X_i)}$).

8.2.2 Results

Figure 8.4 shows the accuracy of the deployed model (A_a) along with the degraded mode accuracy (A_d) of ParM with $k = 2$ for image classification and speech recognition tasks using ResNet-18. VGG-11 is used for the speech dataset and ResNet-152 for CIFAR-100. ParM's degraded mode accuracy is no more than 6.5% lower than that when predictions from the deployed model are available. As Figure 8.5 illustrates, this enables ParM to maintain high overall

[2]The MLP has two hidden layers with 200 and 100 units and ReLUs.

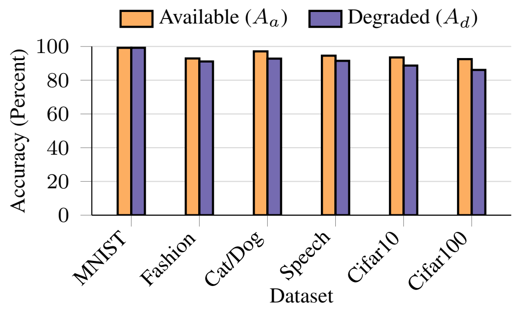


Figure 8.4: Comparison of ParM's accuracy when predictions from the deployed model are available (A_a, "Available") and when ParM's reconstructions are necessary (A_d, "Degraded"). ParM uses $k = 2$.

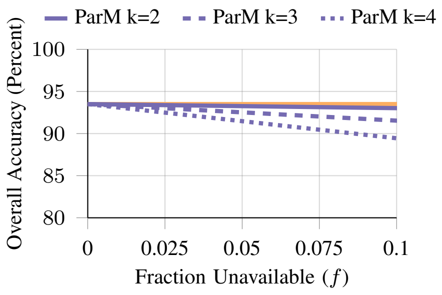


Figure 8.5: Overall accuracy (A_o) of predictions on CIFAR-10 as the fraction of predictions that are unavailable (f) increases. The horizontal orange line shows the accuracy of the ResNet-18 deployed model (A_a).

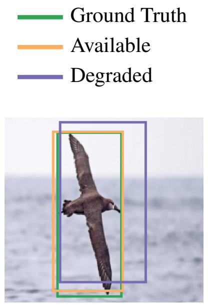


Figure 8.6: Example of ParM's reconstruction for object localization.

accuracy (A_o) in the face of unavailability. For this example, at expected levels of unavailability (i.e., f less than 10%), ParM's overall accuracy is at most 0.4%, 1.9%, and 4.1% lower than when all predictions are available at k values of 2, 3, and 4, respectively. This indicates a tradeoff between ParM's parameter k, which controls resource-efficiency and resilience, and the accuracy of reconstructed predictions, which we discuss in §8.2.2. Finally, ParM's framework enables encoders and decoders to be specialized to the inference task at hand, allowing for further increase in degraded mode accuracy. We showcase task-specific specialization with an image-classification-specific encoder on CIFAR and ImageNet datasets (§8.2.2).

Inference tasks. ParM achieves high degraded mode accuracy with $k = 2$ for the image classification and speech recognition datasets in Figure 8.4. For these tasks, degraded mode accuracy is at most 6.5% lower than when predictions are not slow or failed. We observe similar results for a variety of neural network architectures. For example, on the Fashion-MNIST dataset, ParM's degraded mode accuracy for the MLP, LeNet-5, and ResNet-18 models are only 1.7-9.8% lower than the accuracy when predictions from the deployed model are available.

Object localization task. We next evaluate ParM on object localization, which is a regression task. The goal in this task is to predict the coordinates of a bounding box surrounding an object of interest in an image. As a proof of concept of ParM's applicability for this task, we evaluate ParM on the Caltech-UCSD Birds dataset [315] using ResNet-18. The performance metric for localization tasks is the intersection over union (IoU): the IoU between two bounding boxes is computed as the area of their intersection divided by the area of their union. IoU values fall between 0 and 1, with an IoU of 1 corresponding to identical boxes, and an IoU of 0 corresponding to boxes with no overlap.

Figure 8.6 shows an example of the bounding boxes returned by the deployed model and ParM's reconstruction. For this example, the deployed model has an IoU of 0.88 and ParM's reconstruction has an IoU of 0.61. ParM's reconstruction captures the gist of the localization and would serve as a reasonable approximation in the face of unavailability. On the entire dataset, the deployed model achieves an average IoU of 0.95 with ground-truth bounding boxes. In degraded mode, ParM with $k = 2$ achieves an average IoU of 0.67.

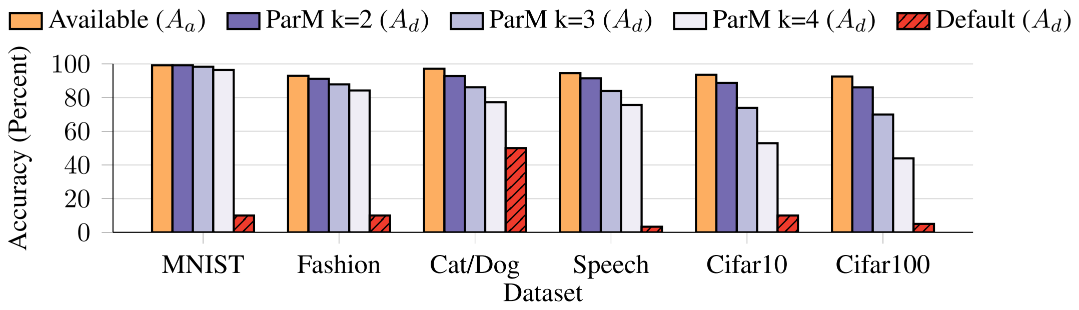


Figure 8.7: Accuracies of predictions reconstructed by ParM with $k = 2, 3, 4$ compared to returning a default response when deployed model predictions are unavailable (A_d).

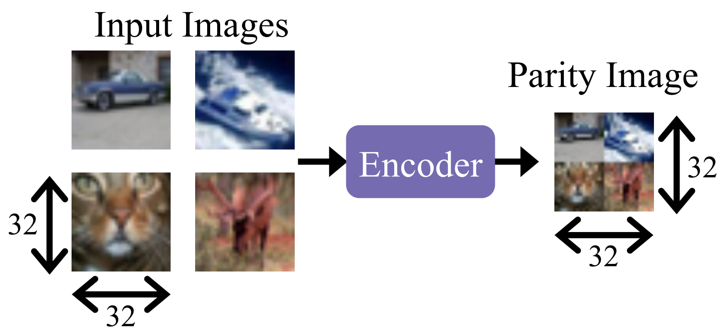


Figure 8.8: Example of an image-classification-specific encoder with $k = 4$ on the CIFAR-10 dataset.

Varying redundancy via parameter k. Figure 8.7 shows ParM's degraded mode accuracy with $k = 2, 3, 4$. ParM's degraded mode accuracy decreases with increasing parameter k. As parameter k increases, features from more queries are packed into a single parity query, making the parity query noisier and making it challenging to learn a parity model. This indicates a tradeoff between the value of parameter k and degraded mode accuracy.

To put these accuracies in perspective, consider a scenario in which no redundant computation is used to mitigate unavailability. In this case, when the output from any deployed model is unavailable, the best one can do is to return a random output as a "default" prediction. The option to provide such a default prediction is available in Clipper. The degraded mode accuracy when returning default predictions depends on the number of possible outputs of an inference task (e.g., the number of classes). For example, on a classification task with ten classes, the expected degraded mode accuracy of this technique would be 10%. The degraded mode accuracy of default predictions provides a lower bound on degraded mode accuracy and an indicator of the difficulty of a particular inference task. For all datasets in Figure 8.7, ParM's degraded mode accuracy is significantly above this lower bound, indicating that ParM makes significant progress in the task of reconstructing predictions.

Inference task-specific encoders and decoders. As described in §8.1.2, ParM's framework enables a large design space for possible encoders and decoders. So far, all evaluation results have used the simple, general addition encoder and subtraction decoder, which is applicable to many inference tasks. We now showcase the breadth of ParM's framework by evaluating ParM's accuracy with alternate encoders and decoders that are inference-task specific.

We design an encoder specialized for image classification which takes in k image queries, and downsizes and concatenates them into a parity query. For example, as shown in Figure 8.8, given $k = 4$ images from the CIFAR-10 dataset (each with $32 \times 32 \times 3$ features), each image is resized to have $16 \times 16 \times 3$ features and concatenated together. The resulting parity query is a 2×2 grid of these resized images, and thus has a total of $32 \times 32 \times 3$ features, the same amount as a single image query. We use the subtraction decoder alongside this encoder. In training parity models with this encoder, we use a loss function specialized to classification: the cross entropy between the output of the parity model and the summation of the one-hot-encoded labels for concatenated images (after normalizing this summation to be a probability distribution).

By specializing to the task at hand, this encoder improves degraded mode accuracy compared to the general addition encoder. For example, at k values of 2 and 4 on CIFAR-10, the task-specific encoder achieves a degraded mode accuracy of 89% and 74%, respectively. This represents a 2% and 22% improvement compared to the general encoder, respectively.

On the 1000-class ImageNet dataset (ILSVRC 2012 [273]) with $k = 2$ and using ResNet-50 models, this approach achieves a 61% top-5 degraded mode accuracy. To put these results in perspective, we note that the first use of neural networks for the ImageNet classification task resulted in a top-5 accuracy of 84.7% [190]. Our results similarly represent the first use of learning and neural networks for coded computation. As the task of a parity model is considerably more difficult than that of image classification, these results show the promise of using parity models for coded computation, even on massive datasets. We expect that improvement in degraded mode accuracy may be achieved by further exploring the design space for encoders, decoders, and the model architecture used for parity models.

8.3 Evaluation of Tail Latency Reduction

We next evaluate ParM's ability to reduce tail latency. The highlights of the evaluation results are as follows:

- ParM significantly reduces tail latency: in the presence of load imbalance, ParM reduces 99.9th percentile latency by up to 48%, bringing tail latency up to $3.5\times$ closer to median latency, while maintaining the same median (§8.3.2). Even with little load imbalance, ParM reduces the gap between tail and median latency by up to $2.3\times$ (§8.3.2). These benefits hold for a variety of inference hardware, query rates, and batch sizes.
- ParM's approach of introducing and learning parity models enables using encoders and decoders with low latencies (less than $200\,\mu$s and $20\,\mu$s, respectively) (§8.3.2).
- ParM reduces tail latency while maintaining simpler development and deployment than other hand-crafted approaches, such as deploying approximate models (§8.3.2).

8.3.1 Implementation and Evaluation Setup

We have built ParM atop Clipper [109], an open-source prediction serving system. We implement ParM's encoder and decoder on the Clipper frontend in C++. Inference runs in Docker containers on model instances, as is standard in Clipper, and we use PyTorch [43] to implement models. We disable the prediction caching feature in Clipper to evaluate end-to-end latency, though ParM does not preclude the use of prediction caching. We use OpenCV [41] for pixel-level encoder operations. We use the addition encoder and subtraction decoder described in §8.2.2 in all latency evaluations.

Baselines. We consider as a baseline a prediction serving system with the same number of instances as ParM but using all additional instances for deploying extra copies of the deployed model. We call this baseline "Equal-Resources." For a setting of parameter k on a cluster with m model instances for deployed models, both ParM and Equal-Resources use $\frac{m}{k}$ additional model instances. ParM uses these extra instances to deploy parity models, whereas this baseline hosts extra deployed models on these instances. These extra instances enable the baseline to reduce system load, which reduces tail latency and provides a fair comparison. We compare ParM to another baseline, deploying approximate models, in §8.3.2.

Cluster setup. All experiments are run on Amazon EC2. We evaluate ParM on two different cluster setups to mimic various production prediction-serving settings.

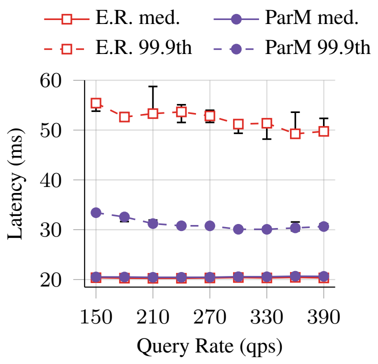
- **GPU cluster.** Each model instance is a `p2.xlarge` instance with one NVIDIA K80 GPU. We use 12 instances for deployed models and $\frac{12}{k}$ additional instances for redundancy. With $k = 2$ there are thus 18 instances.
- **CPU cluster.** Each model instance is a `c5.xlarge` instance, which AWS recommends for inference [4]. We use 24 instances for deployed models and $\frac{24}{k}$ additional instances for redundancy. This emulates production settings that use CPUs for inference [152, 250, 341]. This cluster is larger than the GPU cluster since the CPU instances are less expensive than GPU instances.

We use a single frontend of type `c5.9xlarge`. We use this larger instance for the frontend to sustain high aggregate network bandwidth to model instances (10 Gbps). Each instance uses AWS ENA networking. We observe bandwidth of 1-2 Gbps between each GPU instance and the frontend and of 4-5 Gbps between each CPU instance and the frontend.

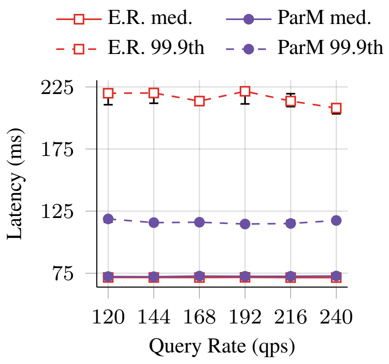
Queries and deployed models. Recall that accuracy results were presented for various tasks and deployed models in §8.2. For latency evaluations we choose one of these models and tasks, ResNet-18 [153] for image classification. We use ResNet-18 rather than a larger model like ResNet-152, which would have a longer runtime, to provide a more challenging scenario in which ParM must reconstruct predictions with low latency. Queries are drawn from the Cat v. Dog [50] dataset. These higher-resolution images test the ability of ParM's encoder to operate with low latency.[3] We modify deployed models and parity models to return 1000 values as predictions to create a more computationally challenging decoding scenario in which there are 1000 classes in each prediction, rather than the usual 2 classes for this task.

Load balancing. Both ParM and the baseline use a single-queue load balancing strategy for dispatching queries to model instances. This strategy is used in Clipper and is optimal in reducing average response time [148]. The frontend maintains a single queue to which all queries are

[3]While CIFAR-10/100 are more difficult tasks for training a model than Cat v. Dog, their low resolution makes them computationally inexpensive. This makes Cat v. Dog a more realistic workload for evaluating latency.



(a) GPU cluster



(b) CPU cluster

Figure 8.9: Latencies of ParM and Equal-Resources (E.R.). The CPU cluster has twice as many instances as the GPU cluster and thus sustains comparable load.

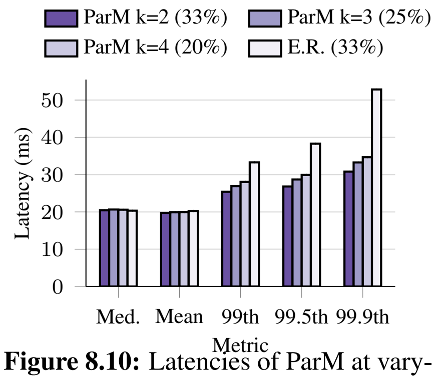


Figure 8.10: Latencies of ParM at varying values of k compared to the strongest baseline. The amount of redundancy used in each configuration is listed in parentheses.

added. Model instances pull queries from this queue when they are available. Similarly, ParM adds parity queries to a single queue which parity models pull from. Evaluation on other, sub-optimal, load balancing strategies (e.g., round robin) revealed results that are even more favorable for ParM than those showcased below.

Background traffic. Like any redundancy-based technique, ParM is most beneficial in operating scenarios prone to unpredictable latency spikes and failures; if unavailability is absent or entirely predictable, redundancy-based approaches are not necessary. Therefore, we focus the evaluation of ParM on these scenarios by inducing background load on the clusters running ParM. The main form of background load we use emulates network traffic typical of data analytics workloads. Specifically, two model instances are chosen randomly to transfer data to one another of size randomly drawn between 128-256 MB. Unless otherwise mentioned, four shuffles take place concurrently. In this setting *only* the cluster network is imbalanced; we do not introduce computational multitenancy. We experiment with light multitenant computation and varying the number of shuffles in §8.3.2.

Latency metric. Clients send 100,000 queries to the frontend using a variety of Poisson arrival rates. All latencies measure the time between when the frontend receives a query and when the corresponding prediction is returned to the frontend (from a deployed model or reconstructed). We report the median of three runs of each configuration, with error bars showing the minimum and maximum. Unless otherwise noted, all experiments are run with batch size of one, as this is the preferred batch size for low latency [105, 341]. We evaluate ParM with larger batch sizes in §8.3.2.

8.3.2 Results

We now report ParM's reduction of tail latency.

Varying query rate. Figure 8.9 shows median and 99.9th percentile latencies with $k = 2$ (i.e., both ParM and Equal-Resources have 33% redundancy) on the GPU and CPU clusters. We consider query rates up until a point in which a prediction serving system with no redundancy (i.e., with m instances) experiences tail latency dominated by queueing. Beyond this point, ParM could be used alongside techniques that reduce queueing [108].

ParM reduces the gap between 99.9th percentile and median latency by 2.6-3.2× compared to Equal-Resources on the GPU cluster, and by 3-3.5× on the CPU cluster. ParM's 99.9th percentile latencies are thus 38-43% lower on the GPU cluster and 44-48% lower on the CPU cluster. This enables ParM to operate with more predictable latency. As expected from any redundancy-based approach, ParM adds additional system load by issuing redundant queries, leading to a slight increase in median latency (less than $0.5\,\mathrm{ms}$).

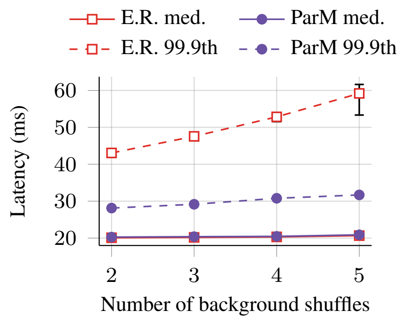
Varying redundancy via parameter k. Figure 8.10 shows the latencies achieved by ParM with k being 2, 3, and 4, when operating at 270 qps on the GPU cluster. As k increases, ParM's tail latency also increases. This is due to two factors. First, at higher values of k, ParM is more vulnerable to multiple predictions in a coding group being unavailable, as the decoder requires $k - 1$ predictions from the deployed model to be available (in addition to the output of the parity model). Second, increasing k increases the amount of time ParM needs to wait for k queries to arrive before encoding into a parity query. This increases the latency of the end-to-end path of reconstructing an unavailable prediction.

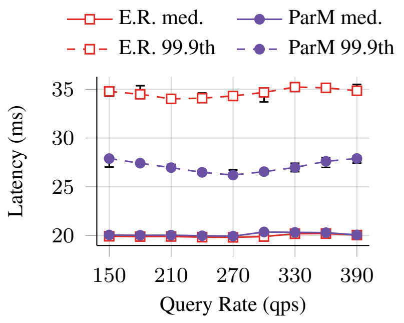
Despite these factors, ParM still reduces tail latency, even when using less resources than the baseline. At k values of 3 and 4, which have 25% and 20% redundancy, ParM reduces the gap between tail and median latency by up to 2.5× compared to when Equal-Resources has 33% redundancy.

Varying batch size. Due to the low latencies required by user-facing applications, many prediction serving systems perform no or minimal query batching [105, 152, 341]. For completeness, we evaluate ParM when queries are batched on the GPU cluster. ParM uses $k = 2$ in these experiments and query rate is set to 460 qps and 584 qps for batch sizes of 2 and 4, respectively. These query rates are obtained by scaling from 300 qps used at batch size 1 based on the throughput improvement observed with increasing batch sizes. ParM outperforms Equal-Resources at all batch sizes: at batch sizes of 2 and 4, ParM reduces 99.9th percentile latency by 43% and 47%, respectively.

Varying degrees and types of load imbalance. All experiments so far were run with background network imbalance, as described in §8.3.1. ParM reduces tail latency even with lighter background network load: Figure 8.11 shows that when 2 and 3 concurrent background shuffles take place (as opposed to the 4 used for most experiments), ParM reduces 99.9th percentile latency over Equal-Resources by 35% and 39%, respectively on the GPU cluster with query rate of 270 qps. ParM's benefits increase with higher load imbalance, as ParM reduces the gap between 99.9th and median latency by $3.5\times$ over Equal-Resources with 5 background shuffles.

To evaluate ParM's resilience to a different, lighter form of load imbalance, we run light background inference tasks on model instances. Specifically, we deploy ResNet-18 models on





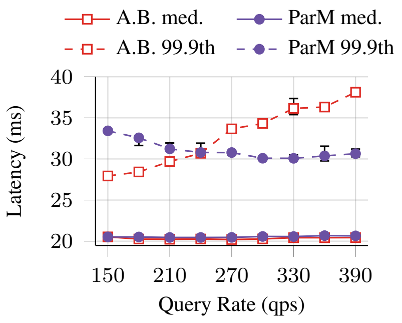


Figure 8.11: ParM and Equal-Resources (E.R.) with varying network imbalance.

Figure 8.12: ParM and Equal-Resources (E.R.) with light inference multitenancy.

Figure 8.13: Latencies of ParM and using approximate backup models (A.B.).

one ninth of instances using a separate copy of Clipper, and send an average query rate of less than 5% of what the cluster can maintain. We do not add network imbalance in this setting. Figure 8.12 shows latencies at $k = 2$ on the GPU cluster with varying query rate. Even with this light form of imbalance, ParM reduces the gap between 99.9th percentile and median latency by up to 2.3× over Equal-Resources.

Latency of ParM's components. ParM's latency of reconstructing unavailable predictions consists of encoding, parity model inference, and decoding. ParM has median encoding latencies of 93 µs, 153 µs, and 193 µs, and median decoding latencies of 8 µs, 14 µs, and 19 µs for k values of 2, 3, and 4, respectively. As the latency of parity model inference is tens of milliseconds, ParM's encoding and decoding make up a very small fraction of end-to-end reconstruction latency. These fast encoders and decoders are enabled by introducing and learning parity models.

Comparison to approximate backup models. An alternative to ParM is to replace parity models with less-computationally-expensive models that approximate the predictions of the deployed model, and to replicate queries to these approximate models. This approach has a number of drawbacks: (1) it is unstable at expected query rates, (2) it is inflexible to changes in hardware, limiting deployment flexibility, and (3) it requires 2× network bandwidth. To showcase these drawbacks of the alternative approach, we compare ParM (with $k = 2$) to the aforementioned alternative using $\frac{m}{k}$ extra model instances for approximate models. We use MobileNet-V2 [274] (with a width factor of 0.25) as the approximate models because this model has similar accuracy (87.6%) as ParM's reconstructions (87.4%) for CIFAR-10.

Figure 8.13 shows the latencies of these approaches on the GPU cluster with varying query rate. While ParM's 99.9th percentile latency varies only modestly, using approximate models results in tail latency variations of over 36%. This variance occurs because all queries are replicated to approximate models even though there are only $\frac{1}{k}$ as many approximate models as there are deployed models. Thus, approximate models must be k-times faster than the deployed model for this system to be stable. The approximate model in this case is not k-times faster than the deployed model, leading to inflated tail latency due to queueing as query rate increases.

Even if one crafted an approximate model satisfying the runtime requirement described above, the model may not be appropriate for different hardware. We find that the speedup achieved by the approximate model over the deployed model varies substantially across different inference hardware. For example, the MobileNet-V2 approximate model is $1.4\times$ faster than the ResNet-18 deployed model on the CPU cluster, but only $1.15\times$ faster on the GPU cluster. Thus, an approximate model designed for one hardware setup may not provide benefits on other hardware, limiting deployment flexibility.

Finally, this approach uses $2\times$ network bandwidth by replicating queries. This can be problematic, as limited bandwidth has been shown to hinder prediction-serving [109, 151].

Some of the limitations of this approach may be mitigated by reactively issuing redundant queries to approximate backup models after a timeout. However, like other reactive approaches, doing so reduces the potential for such an approach to reduce tail latency.

ParM does not have the drawbacks described above. As described in §8.1, ParM's parity models have the same average runtime as deployed models, and ParM encodes k queries into one parity query prior to dispatching to a parity model. The $\frac{m}{k}$ parity models therefore receive $\frac{1}{k}$ the query rate of the m deployed models, and thus naturally keep pace. This reduced query rate also means that ParM adds only minor network bandwidth overhead. Further, by using the same architecture for parity models as is used for deployed models, ParM does not face hardware-related deployment issues.

8.4 Discussion

Training time. We find that the time to train a parity model can be 3-$12\times$ longer than that of a deployed model. Reducing training time was not a goal of this work. We next describe ways in which training time may potentially be reduced.

As described in §8.1.4, the training procedure for a parity model draws k samples from the deployed model's training dataset to form a single training sample for the parity model. Thus, the number of possible combinations of k samples that could be used grows exponentially with k. For example, training a parity model using a deployed model dataset with 1000 samples would lead to an effective dataset size of 1000^k. For large deployed model datasets, the effective parity model dataset is too large to train on every possible combination. We currently randomly sample combinations of k queries from the deployed model dataset without keeping track of which combinations have been used. As was shown in §8.2, even this simple approach enables accurate training of parity models. A more principled approach to sampling from the deployed model's dataset may help reduce training time and improve accuracy, especially for massive datasets and those with imbalance in the number of samples of a particular type (e.g., fewer examples of fish than dogs).

Throughput. Like any redundancy-based approach, the achievable throughput when using coded computation is lower than it could be if one used all resources for serving copies of a deployed model. Specifically, as ParM uses $\frac{1}{k}$ of all model instances for parity models, ParM's maximum achievable throughput is $(1 - \frac{1}{k})$-times that of an approach which uses no redundancy. However, as shown in §8.3, reserving some resources to be used for redundancy aids in reducing tail latency. ParM enables one to span this tradeoff between tail latency and throughput by changing parameter k.

Concurrent unavailabilities. ParM can accommodate concurrent unavailabilities by using decoders parameterized with $r > 1$. In this case, r separate parity models can be trained to produce different transformations of a parity query. For example, consider having $k = 2$, $r = 2$, queries X_1 and X_2, and parity $P = (X_1 + X_2)$. One parity model can be trained to transform P into $\mathcal{F}(X_1) + \mathcal{F}(X_2)$, while the second can be trained to transform P into $\mathcal{F}(X_1) + 2\mathcal{F}(X_2)$. The decoder reconstructs the initial k predictions using any k out of the $(k + r)$ predictions from deployed models and parity models.

Parity model design space. In this work, we have chosen to keep encoders and decoders simple and to specialize parity models so as to reduce the additional computation introduced on prediction serving system frontends. However, as described in §8.1 and shown empirically in §8.2.2 using an image-classification-task specific encoder, ParM's framework opens a rich design space for encoders, decoders, and parity models. Navigating this design space by co-designing these components may yield interesting opportunities for improving the accuracy of reconstructions.

Applicability to other workloads. We have showcased the use of parity models on image classification, speech recognition, and object localization tasks. However, the core idea of learning-based coded computation has the potential to be applied more broadly. For example, parity models may potentially be applicable to sequence-to-sequence models, such as those for translation, though further research is necessary to accommodate the use of parity models to these tasks.

8.5 Related Work

Mitigating slowdowns. Many approaches target specific causes of slowdown. Examples of such techniques include configuration selection [64, 206, 303, 327], isolation [141, 161, 219, 325], replica selection [147, 293], predicting slowdowns [326], and autoscaling [108, 142]. As these techniques apply only to specific slowdowns, they are unable to mitigate all slowdowns. In contrast, ParM is agnostic to the cause of slowdown.

Many techniques mitigate slowdowns in *training* [150, 154, 266]. These techniques exploit iterative computations specific to training and are thus inapplicable to inference.

Accuracy-latency tradeoff. A number of systems trade accuracy for lower latency [310, 338]. This enables handling query rate variation efficiently, but may degrade accuracy. In contrast, ParM does not proactively degrade accuracy; any inaccuracy due to ParM is incurred only when a prediction experiences slowdown or failure.

Algorithmic techniques like cascades [304, 311] and anytime neural networks [157] enable "early exit" during inference for queries that are easier to complete or those taking longer than a predefined deadline. These techniques are only applicable to reducing the latency of inference, and thus will not help mitigate tail latency induced by other sources, like network congestion or failure. In contrast, ParM alleviates slowdowns and failures regardless of their cause.

High performance inference. Many techniques improve the average latency and throughput of inference [39, 97, 163, 168, 196, 213]. These techniques are complementary to ParM, which is designed for mitigating slowdowns and failures.

8.6 Conclusion

We present a fundamentally new, learning-based approach for enabling the use of ideas from erasure coding to impart low-latency, resource-efficient resilience to slowdowns and failures in prediction serving systems. Through judicious use of learning, parity models overcome the limitations of existing coded-computation approaches and enable the use of simple, fast encoders and decoders to reconstruct unavailable predictions for a variety of neural network inference tasks. We have built ParM, a prediction serving system that makes use of parity models, and shown the ability of ParM to reduce tail latency while maintaining high overall prediction accuracy in the face of load imbalance. These results suggest that our approach may open new doors for enabling the use of erasure-coded slowdown and failure tolerance for a broader class of workloads.

Chapter 9

Leveraging ideas inspired by coding theory beyond reliability purposes

Thus far, this thesis has focused on the use and enhancement of coding-theoretic tools to build resource-efficient, reliable ML systems. This part of the thesis focuses on the use of coding-theory-inspired ideas for a different purpose: to improve the performance and resource utilization of ML systems, even when reliability is not a concern.

In particular, this chapter focuses on improving the throughput and resource utilization of high-throughput specialized CNN inference on accelerators. We provide more background on this setting in §9.2. We identify that specialized CNNs significantly underutilize server-grade hardware accelerators due to having low arithmetic intensity, an occurrence observed in §4 of this thesis. To improve the accelerator utilization of specialized CNN inference, we propose transformations to specialized CNNs that involve widening individual layers of the CNN and operating over "stacks" of input images in a manner similar to the learned encoders described in §7 of this thesis. A toy example of this is shown in Figure 9.2. The resultant FoldedCNNs significantly increase the arithmetic intensity, accelerator utilization, and throughput of specialized CNN inference. This shows the intriguing promise of leveraging coding-theory-inspired ideas to improve ML systems beyond reliability purposes.

9.1 Introduction

Convolutional neural networks (CNNs) are widely deployed for high-throughput vision tasks. Many such tasks target highly specific events for which general-purpose CNNs trained on diverse data (e.g., ResNet-50 on ImageNet) are overkill; an application detecting red trucks does not need a CNN capable of classifying animals. It has thus become popular to employ small *specialized CNNs* trained only for such focused tasks [156, 178, 179, 283]. In being trained for highly specific tasks, specialized CNNs can typically be much smaller than general-purpose CNNs, and thus operate at higher application-level throughput (e.g., images/sec).

Specialized CNNs are heavily used for inference in both datacenters and edge clusters [82, 156, 179, 232], and occasionally on constrained devices (e.g., cameras) [93]. *We focus on specialized CNNs used for high-throughput vision in datacenters/clusters.* A popular usecase in

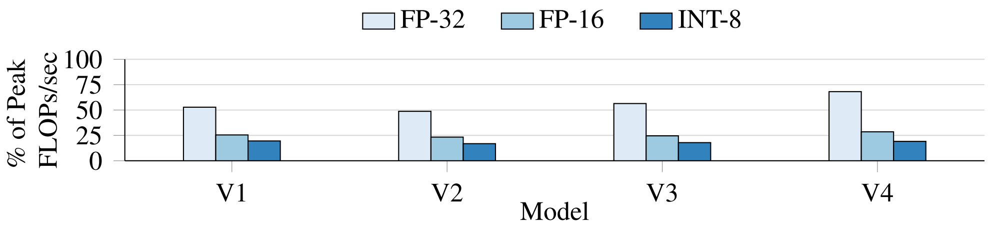


Figure 9.1: Utilization of production specialized CNNs (see §9.2.1) at various precisions and maximum batch size on a T4 GPU. Each bar is relative to the peak FLOPs/sec of the T4 in that precision.

this setting is offline video analytics, in which all video frames are processed by a specialized CNN, and only frames for which the specialized CNN is uncertain are processed by a slower, general-purpose CNN [179]. The throughput of the specialized CNN is critical to that of the overall system, as all frames are processed by the specialized CNN and only a small fraction by the general-purpose CNN.

Aiding the case for high-throughput CNNs, server-grade deep learning hardware accelerators offer unprecedented performance in FLOPs/sec, and thus are used for inference in datacenters (e.g., V100 and T4 GPUs, TPUs) and edge clusters (e.g., AWS Outposts [71] and Azure Stack Edge [72] with T4 GPUs). It is critical that these accelerators be highly utilized, with software running on an accelerator ideally achieving FLOPs/sec near the accelerator's theoretical peak FLOPs/sec. Given the high cost of accelerators and the operational costs incurred in deploying them (e.g., power) [76], poorly utilizing an accelerator leads to a poor return on investment. From a sustainability perspective, Google has noted that machine learning inference makes up over half of the energy used for machine learning, and that higher utilization enables more efficient use of infrastructure [251]. Furthermore, underutilization results in suboptimal application-level throughput.

However, current specialized CNNs significantly underutilize server-grade accelerators: we find that specialized CNNs used in production at Microsoft achieve less than 20% of the peak FLOPs/sec of GPUs employed in datacenters, even with large batch sizes (which are common for high-throughput inference), and when using techniques that improve throughput, such as reduced precision (see Figure 9.1). While specialized CNNs might better utilize weaker devices, we find that server-grade GPUs, such as V100 and T4, offer the highest cost-normalized throughput for the CNNs described above, motivating their deployment in production.

The main cause for the poor accelerator utilization of specialized CNNs is low *arithmetic intensity*: the ratio between the number of arithmetic operations performed by a computational kernel (i.e., FLOPs) and the number of bytes read from or written to memory by the kernel [317]. As the bandwidth of performing arithmetic on accelerators is far higher than memory bandwidth (e.g., over 200$\times$ on T4 [55]), a CNN with low arithmetic intensity incurs frequent memory stalls, leaving arithmetic units idle and underutilized. High arithmetic intensity is, thus, a prerequisite to high utilization. However, we will show in §9.2 that specialized CNNs have arithmetic intensities far lower than needed for peak utilization on accelerators.

The arithmetic intensities of specialized CNNs must be increased to improve utilization of server-grade accelerators, but achieving this requires care: we show that common techniques that increase application-level throughput can reduce arithmetic intensity, while naive approaches to increasing arithmetic intensity reduce application-level throughput.

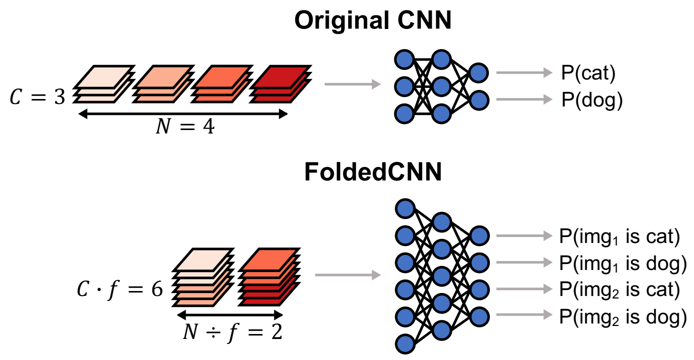


Figure 9.2: Abstract illustration of a FoldedCNN with $f = 2$.

Increasing the batch size over which inference is performed *can* increase arithmetic intensity, utilization, and application-level throughput by amortizing the cost of loading a CNN's weights from memory. However, doing so leads to diminishing returns in these quantities: for example, we show in §9.2 that specialized CNNs achieve at most 17% of the peak FLOPs/sec of a V100 at large batch sizes. *An alternative is needed to further improve the utilization and throughput of specialized CNNs beyond the limits of increasing batch size.*

We propose *FoldedCNNs*, a new approach to the design of specialized CNNs that boosts inference utilization and throughput beyond increasing batch size. We show that convolutional and fully-connected layers in specialized CNNs at large batch size can be transformed to perform an equal number of FLOPs, but with higher arithmetic intensity. Our key insight is that, once arithmetic intensity has plateaued due to increased batch size, reading/writing activations accounts for most of the memory traffic in specialized CNNs. We show that this memory traffic can be significantly reduced, while performing the same number of FLOPs, by jointly decreasing the size of the batch of input/output activations for a layer and increasing the layer's width. By decreasing memory traffic while performing the same number of FLOPs, this transformation increases arithmetic intensity.

FoldedCNNs take a new approach to structuring the inputs of a CNN to apply this transformation, inspired in part from the interplay of machine learning and coding theory in the learned encoders described in §7. As shown in Figure 9.2, rather than operating over a batch of N images each with C channels, a FoldedCNN instead operates over a batch of $\frac{N}{f}$ "folded" inputs each with fC channels formed by concatenating f images along the channels dimension. These f images are jointly classified: if the original CNN had C_L output classes, the FoldedCNN now has fC_L output classes. FoldedCNNs increase the number of channels for all middle layers by $\sqrt{f}\times$, while maintaining an $f\times$ reduction in batch size. This reduces memory traffic over N images by $\sqrt{f}\times$ while performing a similar number of FLOPs, thus increasing arithmetic intensity, utilization, and throughput.

We evaluate FoldedCNNs on four specialized CNNs used at Microsoft and four from the NoScope video-processing system [179]. FoldedCNNs improve the GPU utilization of specialized CNNs by up to 2.8× and throughput by up to 2.5×, while maintaining accuracy close to the original CNN in most cases. Compared to the compound scaling used in EfficientNets [295], FoldedCNNs achieve higher accuracy, throughput, and utilization for specialized CNNs. These results show the promise of FoldedCNNs in increasing the utilization and throughput of specialized CNNs beyond increased batch size, and open doors for future high-performance specialized CNNs. The code used in this chapter is available at `https://github.com/msr-fiddle/folded-cnns`.

9.2 Challenges in Achieving High Utilization

We now describe challenges in achieving high accelerator utilization in specialized CNN inference.

9.2.1 Specialized CNNs

As described in §9.1, specialized CNNs are small CNNs designed to target highly specific visual tasks and to achieve higher throughput than large, general-purpose CNNs. We focus on two motivating usecases of specialized CNNs:

Usecase 1: filters. A popular use of specialized CNNs is as lightweight filters in front of slower, general-purpose CNNs. In such systems, all video frames/images pass through a specialized CNN, and are processed by the general-purpose CNN only if the specialized CNN is uncertain [179]. In other cases, the specialized CNN builds an approximate index to accelerate later ad-hoc queries by a general CNN [156]. These applications desire high throughput, so batching is heavily exploited. We use specialized CNNs from the NoScope video-processing system [179] as examples of this usecase.

Usecase 2: game scraping. We also consider specialized CNNs used in production at Microsoft to classify events in video game streams by scraping in-game text appearing in frames (e.g., score). Separate CNNs are specialized for each game and event type. The service handles thousands of streams at once, and thus heavily batches images.

Comparison of general and specialized CNNs. General-purpose CNNs, such as those used for ImageNet, have many convolutional layers, each with many channels. For example, ResNet-50 has 49 convolutional layers, each with 64–2048 channels. In contrast, specialized CNNs have far fewer layers and channels: the specialized CNNs used in NoScope (Usecase 1) have 2–4 convolutional layers, each with 16–64 channels; those used at Microsoft (Usecase 2) have 5–8 convolutional layers with at most 32 channels. Further details on these CNNs are given in Table 9.1.

9.2.2 High utilization requires high arithmetic intensity

We now provide a brief recap on arithmetic intensity and the need for high arithmetic intensity to achieve high utilization. Further background on this topic was previously provided in §4.

As described in §9.1, achieving high utilization of accelerators is critical for operational efficiency. Ideally, a CNN would operate near the peak FLOPs/sec offered by an accelerator. However, achieving this is confounded by the need to transfer data to/from memory, as cycles stalled on memory are wasted if they cannot be masked by computation.

A computational kernel must be compute bound to achieve peak FLOPs/sec: a compute-bound kernel uses all arithmetic units on an accelerator at all times. Under the popular Roofline

Table 9.1: Specialized CNNs used in this work.

Group	ID	Name	Resol.	Convs.	Classes	Description
NoScope	N1	coral	(50, 50)	2	2	Classifies whether there is a person in front of an aquarium exhibit.
NoScope	N2	night	(50, 50)	2	2	Classifies whether a car is in the frame.
NoScope	N3	roundabout	(50, 50)	4	2	Classifies whether a car is in the frame.
NoScope	N4	taipei	(50, 50)	2	2	Classifies whether a bus is in the frame.
Microsoft	V1	lol-gold1	(22, 52)	5	11	Classifies fractional value of amount accumulated (e.g., “7” in “14.7k”). Classes are digits 0–9 and “other” indicating blank.
Microsoft	V2	apex-count	(19, 25)	5	22	Classifies number of members of a squad remaining. Classes are integers 0–20 and “other” indicating blank.
Microsoft	V3	sot-coin	(17, 40)	5	15	Classifies the thousands-place of a count on the number of coins a player has (e.g., “10” for “10,438”). Classes are integers 0–14 and “other” indicating blank.
Microsoft	V4	sot-time	(22, 30)	8	27	Classifies time remaining. Classes are integers 0–25 and “other” indicating blank.

performance model [317], a kernel can only be compute bound if it theoretically spends more time computing than it does reading/writing memory:

$$\frac{\text{FLOPs}}{\text{Compute Bandwidth}} > \frac{\text{Bytes}}{\text{Memory Bandwidth}}$$

Here, “FLOPs” is the number of arithmetic operations performed, “Bytes” is the amount of data transferred to/from memory (memory traffic), “Compute Bandwidth” is the accelerator’s peak FLOPs/sec, and “Memory Bandwidth” is the accelerator’s memory bandwidth (bytes/sec). Rearranging this to pair properties of the kernel on the left-hand side and properties of the accelerator on the right-hand gives:

$$\frac{\text{FLOPs}}{\text{Bytes}} > \frac{\text{Compute Bandwidth}}{\text{Memory Bandwidth}} \tag{9.1}$$

The left-hand ratio of Equation 9.1 is termed “arithmetic intensity”: the ratio between the FLOPs performed by the kernel and the bytes it transfers to/from memory. The arithmetic intensity of a given layer in a CNN is (abstractly) written as:

$$\frac{\text{FLOPs}}{\text{Input bytes} + \text{Weight bytes} + \text{Output bytes}} \tag{9.2}$$

where “Input bytes” is the size of the layer’s input activations, “Output bytes” is the size of output activations written by the layer to memory for processing by the next layer, and “Weight bytes” is the size of the layer’s weights. For example, using the terminology in Table 9.2, the arithmetic

Table 9.2: Parameters of a 2D convolution with stride of 1.

Parameter(s)	**Variable(s)**
batch size	N
output height, width	H, W
input, output channels	C_i, C_o
conv. kernel height, width	K_H, K_W

intensity of a 2D convolutional layer with a stride of 1 is:

$$\frac{2NHWC_oC_iK_HK_W}{B(NHWC_i + C_iK_HK_WC_o + NHWC_o)} \tag{9.3}$$

where B is numerical precision in bytes (e.g., 2 for FP-16).[1] The aggregate arithmetic intensity of a CNN as a whole is computed by summing the FLOPs performed by each layer of the CNN, summing the bytes read/written by each layer, and dividing these quantities. This accounts for optimizations like layer fusion that reduce memory traffic.

Equation 9.1 indicates that, for a kernel to achieve the peak FLOPs/sec of an accelerator, the kernel's arithmetic intensity must be higher than the ratio between the accelerator's compute bandwidth and memory bandwidth [317].[2] For example, this ratio is 139 in half precision on a V100 GPU [241], 203 on a T4 GPU [55], and 1350 on TPUv1 [172]. It is often necessary for arithmetic intensity to be far higher than this ratio, as arithmetic intensity calculations typically assume perfect memory reuse, which can be difficult to achieve in practice.

Specialized CNNs have low arithmetic intensity. While high arithmetic intensity is needed for high utilization of accelerators, specialized CNNs have low arithmetic intensity due to their small sizes. For example, the half-precision arithmetic intensities of the CNNs used in the game-scraping tasks are 88–102 at large batch sizes, much lower than the minimum of 139 required for peak utilization of a V100 GPU, which is used for specialized CNN inference in datacenters [232]. Thus, these CNNs achieve at most 17% of the V100's peak FLOPs/sec, even at large batch sizes and when running on the TensorRT inference library that performs optimizations like layer fusion. To improve their utilization of accelerators, specialized CNNs must be modified to increase arithmetic intensity.

As described above, high arithmetic intensity alone is insufficient to achieve high utilization, as implementations must efficiently use accelerator resources (e.g., memory hierarchy). Nevertheless, high arithmetic intensity is a prerequisite for high utilization. For specialized CNNs, increasing arithmetic intensity is thus necessary to increase utilization. We will show that simply increasing arithmetic intensity greatly increases the utilization and throughput of specialized CNN inference atop an optimized inference library.

[1]Here, we show arithmetic intensity for direct- and GEMM-based convolutions, though the arguments we make also apply to other implementations (e.g., Winograd).

[2]This condition is necessary, but not sufficient, as inefficiencies in implementation can limit performance [317].

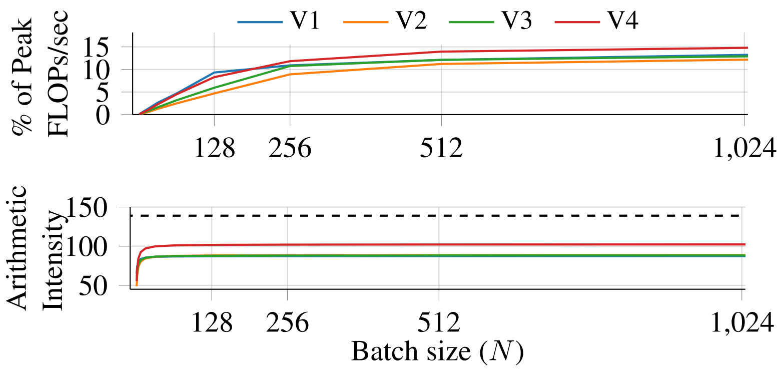


Figure 9.3: FP-16 utilization and arithmetic intensity of game-scraping CNNs on a V100 GPU. The dashed line is the minimum arithmetic intensity needed for peak utilization of a V100 GPU.

9.2.3 Improving arithmetic intensity is non-trivial

To increase the arithmetic intensity of convolutional and fully-connected layers, one must increase the ratio in Equation 9.2. For concreteness, we focus on convolutional layers in this subsection, and thus on increasing Equation 9.3.

Low precision? One way to increase Equation 9.3 is to decrease numerical precision B, which reduces memory traffic by representing operands/outputs using fewer bits. However, modern accelerators have compute units that offer increased FLOPs/sec in low precision (e.g., T4 GPUs). Reducing precision thus increases both the left-hand side of Equation 9.1 (by reducing bytes) and the right-hand side (by increasing compute bandwidth). When these quantities change at equal rates, as is common in accelerators [55], the inequality remains the same: *kernels that did not satisfy this inequality at a high precision will not satisfy it at low precision.* Figure 9.1 illustrates this on a T4 GPU: specialized CNNs have low utilization at both full (FP-32) and low precisions (FP-16, INT-8). Thus, while reducing precision can accelerate inference, it *does not* increase utilization.

Large batch size? Increasing batch size N can increase arithmetic intensity by amortizing the cost of loading layer weights. However, doing so leads to diminishing returns in arithmetic intensity (A), as (ignoring B in Equation 9.3):

$$A = \frac{2NHWC_oC_iK_HK_W}{NHWC_i + C_iK_HK_WC_o + NHWC_o}$$

$$\lim_{N\to\infty} A = \frac{2C_oC_iK_HK_W}{C_i + C_o} \tag{9.4}$$

When batch size is large enough that arithmetic intensity is determined by Equation 9.4, we say that a layer is in the "batch-limited regime." Figure 9.3 shows this on the game-scraping CNNs: arithmetic intensity and utilization plateau with large batch size at 17% of the peak FLOPs/sec of a V100.

To further increase arithmetic intensity beyond the limits of increased batch size, Equation 9.4 indicates that one must increase C_i, C_o, K_H, or K_W. However, doing so increases the number of

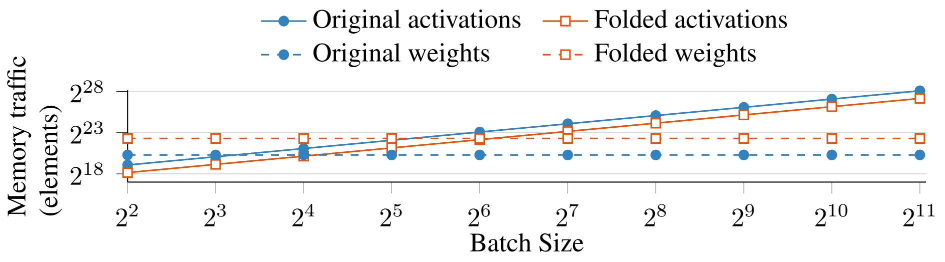


Figure 9.4: Memory traffic of activations and weights of the N1 and folded ($f = 4$) CNN. Axes are in log scale. The y-axis is in elements, rather than bytes, as the trends hold for any bitwidth.

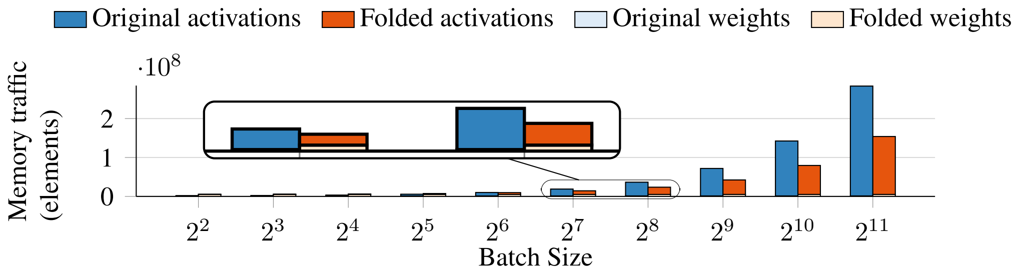


Figure 9.5: Total memory traffic of the N1 and folded ($f = 4$) CNN. As shown in the inset, weights account for a minor fraction of memory traffic with large batch size. The y-axis is in elements, rather than bytes, as the trends hold for any bitwidth used.

FLOPs performed by the layer per image, which typically *decreases application-level throughput*.

Takeaway. To increase utilization beyond increasing batch size, while maintaining high throughput, one must increase arithmetic intensity without greatly increasing FLOP count. We next propose techniques to achieve this goal.

9.3 Boosting Intensity via Folding

We now propose transformations to increase the arithmetic intensity of layers of specialized CNNs operating over large batches without increasing FLOPs. For clarity, we focus on convolutional layers, though the transformations also apply to fully-connected layers (as will be shown in §9.4).

To increase arithmetic intensity while performing the same number of FLOPs, one must decrease memory traffic, the denominator in Equation 9.3. Our key insight is that the total memory traffic of specialized CNNs with large batch size is dominated by reading/writing the input/output activations of convolutional and fully-connected layers ($NHWC_i$ and $NHWC_o$ in the denominator of Equation 9.3),[3] rather than by reading layer weights ($C_iK_HK_WC_o$). Figures 9.4 and 9.5 (focus only on blue parts) depict this for one CNN: with batch size 1024, activations make up over 99% of total memory traffic.

Due to the dominance of input/output activations on a layer's total memory traffic, we note that a joint decrease in NHW and increase in $C_iK_HK_WC_o$ can reduce memory traffic while

[3]The common practice of fusing activation functions to the preceding layer eliminates their contribution to total memory traffic.

Table 9.3: Example of increasing arithmetic intensity by folding a convolutional layer with $f = 4$. The layer has $K_H = K_W = 3$, $H = 11$, $W = 26$, and uses half precision (i.e., $B = 2$).

	Original Equation	**Original Value**	**Folded Equation**	**Folded Value**
Batch size	N	1024	N/f	256
Input, output channels	C_i, C_o	32, 32	$C_i\sqrt{f}$, $C_o\sqrt{f}$	64, 64
Input + output elts. (E_{io})	$NHWC_i + NHWC_o$	18.74M	$\frac{\sqrt{f}}{f}NHWC_i + \frac{\sqrt{f}}{f}NHWC_o$	9.37M
Layer elts. (E_l)	$C_iK_HK_WC_o$	0.01M	$fC_iK_HK_WC_o$	0.04M
Mem. traffic (bytes) (M)	$B(E_{io} + E_l)$	37.51M	$B(E_{io} + E_l)$	18.82M
Operations (O)	$2NHWC_oC_iK_HK_W$	5398.07M	$2NHWC_oC_iK_HK_W$	5398.07M
Arithmetic intensity	O/M	**143.93**	O/M	**286.87**

maintaining the same number of FLOPs. Suppose one decreased NHW by a factor of f (with $f > 1$) and increased C_i and C_o by a factor of $\sqrt{f}$. We call this transformation *folding* and layers transformed by it *folded*. The folded layer has the following properties: (1) It performs the same number of FLOPs: $\frac{NHW}{f}(C_o\sqrt{f})(C_i\sqrt{f})(K_HK_W) = NHWC_oC_iK_HK_W$. (2) It decreases the size of layer inputs/outputs by a factor of $\sqrt{f}$ from $NHWC_i$ to $\frac{\sqrt{f}}{f}NHWC_i$ (similarly for outputs with C_o). (3) It increases the number of layer weights by a factor of f from $C_iK_HK_WC_o$ to $(C_i\sqrt{f})K_HK_W(C_o\sqrt{f})$.

Properties 2 and 3 are shown in Figure 9.4 when folding a representative specialized CNN from the NoScope system with $f = 4$: the folded convolutions have $2\times$ lower memory traffic for activations and $4\times$ higher memory traffic for weights. At large batch sizes, the decrease in memory traffic for activations is larger than the increase for weights. For example, at batch size 1024, memory traffic for activations decreases by 66.7M, while that for weights increases by only 3.9M. The increase in memory traffic from layer weights is dwarfed by the decrease for activations, resulting in a reduction in total memory traffic. Figure 9.5 illustrates this reduction in memory traffic for the same CNN.

As the folded layer performs as many FLOPs as the original layer, but with reduced memory traffic, it has higher arithmetic intensity. If a layer is in the batch-limited regime, in which arithmetic intensity is determined by Equation 9.4, folding increases arithmetic intensity by $\sqrt{f}\times$, as the numerator and denominator in Equation 9.4 increase by $f\times$ and $\sqrt{f}\times$, respectively. An example of this is shown in Table 9.3.

Proof of reduction in memory traffic. We will now prove that the folding transformation described in reduces total memory traffic if:

$$NHW > \frac{(f-1)C_iK_HK_WC_o}{(1-\frac{1}{\sqrt{f}})(C_i+C_o)} \tag{9.5}$$

We will first show that the new layer performs an equal number of operations as the original layer (i.e., the numerator in Equation 9.3 stays the same) and then show that the new layer layer has reduced memory traffic compared to the original layer (i.e., the denominator in Equation 9.3 decreases), provided that the inequality holds. These two changes will result in the new layer having an increased arithmetic intensity.

Equal number of operations. The initial convolutional layer performs $2NHWC_oC_iK_HK_W$ operations. The transformed convolutional layer performs $\frac{2NHW}{f}(\sqrt{f}C_o)(\sqrt{f}C_i)(K_HK_W) = 2NHWC_oC_iK_HK_W$ operations, which is equal to that of the original model.

Reduced memory traffic. We wish to show that the inequality is equivalent to the memory traffic of the transformed layer being lower than that of the original layer.

We first note that, ignoring the bytes per element B, the memory traffic of the original convolutional layer is $NHWC_i + C_iK_HK_WC_o + NHWC_o$, while that of the transformed layer is $\frac{\sqrt{f}}{f}NHWC_i + fC_iK_HK_WC_o + \frac{\sqrt{f}}{f}NHWC_o$.

We wish to show that:

$$NHWC_i + C_iK_HK_WC_o + NHWC_o > \frac{\sqrt{f}}{f}NHWC_i + fC_iK_HK_WC_o + \frac{\sqrt{f}}{f}NHWC_o$$

We first rearrange the righthand side of the inequality as:

$$\frac{\sqrt{f}}{f}NHWC_i + fC_iK_HK_WC_o + \frac{\sqrt{f}}{f}NHWC_o = \frac{NHW}{\sqrt{f}}C_i + fC_iK_HK_WC_o + \frac{NHW}{\sqrt{f}}C_o$$

Grouping by similar terms gives:

$$NHWC_i - \frac{NHW}{\sqrt{f}}C_i + NHWC_o - \frac{NHW}{\sqrt{f}}C_o > fC_iK_HK_WC_o - C_iK_HK_WC_o$$

Which implies:

$$(1 - \frac{1}{\sqrt{f}})NHWC_i + (1 - \frac{1}{\sqrt{f}})NHWC_o > (f-1)C_iK_HK_WC_o$$

Which implies:

$$NHW((1 - \frac{1}{\sqrt{f}})(C_i + C_o)) > (f-1)C_iK_HK_WC_o$$

Which ultimately leads to our desired inequality:

$$NHW > \frac{(f-1)C_iK_HK_WC_o}{(1 - \frac{1}{\sqrt{f}})(C_i + C_o)}$$

When does folding help? As described in §9.2, increasing arithmetic intensity in this manner has the potential to increase both the throughput and accelerator utilization of specialized CNN inference. Additionally, because the folding transformation reduces memory accesses, it has the potential to reduce the energy consumed during inference, which is heavily tied to the number of memory accesses performed [251].

Folding will most heavily increase the utilization and throughput of layers that have arithmetic intensity in the batch-limited regime that is below that needed for peak FLOPs/sec on an accelerator. Specialized CNNs are thus ideal targets for folding, as they have low arithmetic intensity even at large batch size. Meanwhile, large CNNs or those with small batch size are less likely to benefit. Thus, we focus on folding specialized CNNs.

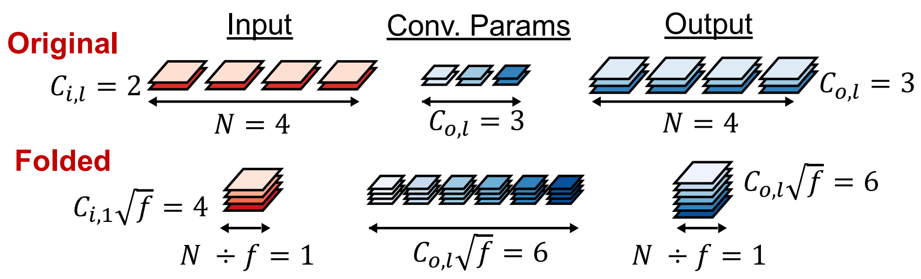


Figure 9.6: Middle layer of a FoldedCNN with $f = 4$. Both the number of input and output channels increase by a factor of $\sqrt{f}$.

9.4 FoldedCNNs

We now propose *FoldedCNNs*, a new approach CNN design based on the folding transformation proposed in §9.3.

Folding involves (1) decreasing NHW by $f\times$ and (2) increasing $C_i K_H K_W C_o$ by $f\times$. There are many ways to achieve these effects. FoldedCNNs achieve them by (1) decreasing batch size N by $f\times$, (2) increasing the number of input and output channels C_i and C_o each by $\sqrt{f}\times$. We do not reduce resolution (H, W) or increase receptive field (K_H, K_W), as specialized CNNs often operate over small images to begin with [179]; we find that such changes can decrease accuracy compared to FoldedCNNs.

9.4.1 Applying folding to a full CNN

We now describe folding for a specialized CNN with L convolutional/fully-connected layers and C_L classes. Let $C_{i,l}$ denote the number of input channels to layer l of the original CNN, and $C'_{i,l}$ that in the FoldedCNN. Similar notation is used for all parameters in Table 9.2. While we focus on plain convolutions in this section, FoldedCNNs also apply to other convolutional variants. We evaluate folding for group convolutions and Winograd convolutions in §9.5.5 and §9.5.6.

We first transform a layer l in the middle of the CNN, as shown in Figure 9.6. As described above, FoldedCNNs decrease batch size: $N' = \frac{N}{f}$ and increase the number of input and output channels: $C'_{i,l} = C_{i,l}\sqrt{f}$ and $C'_{o,l} = C_{o,l}\sqrt{f}$. Folded fully-connected layers in the middle of the CNN also have $\sqrt{f}\times$ the number of input and output features. As folding is applied to all convolutional and fully-connected layers, the increase in output channels in one layer naturally fits the increase in input channels for the next layer.

Folding batches of images. As described in §9.3, each layer in a FoldedCNN performs the same number of FLOPs as the corresponding layer of the original CNN. However, a Folded-CNN performs these FLOPs over $\frac{N}{f}$ images, whereas the original CNN operates over N images. Left uncorrected, FoldedCNNs would thus perform $f\times$ more FLOPs per image, and thus would *reduce* application-level throughput.

To rectify this, FoldedCNNs "fold" a batch of images into "stacks" of images, as shown in Figures 9.2 and 9.7. Suppose the original CNN takes in N images each with $C_{i,1}$ channels (e.g., $C_{i,1} = 3$ for RGB). A FoldedCNN instead takes in $\frac{N}{f}$ inputs each with $C_{i,1}f$ channels, formed by concatenating f images along the channels dimension. Each folded input represents f images, so the number of images in a batch of $\frac{N}{f}$ such inputs is equal to that of the original CNN (N).

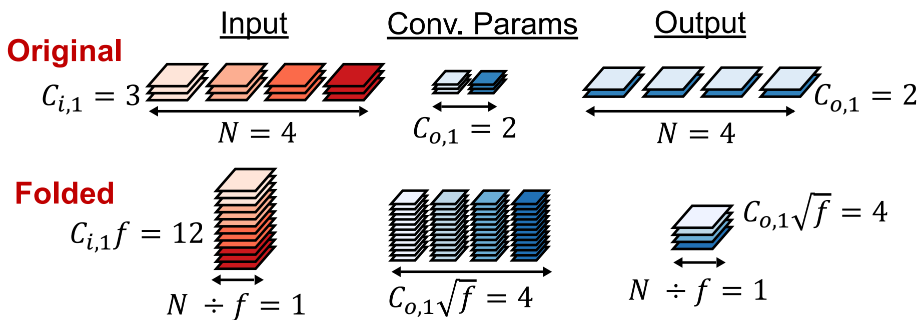


Figure 9.7: First layer of a FoldedCNN with $f = 4$. Unlike other layers, this layer increases input channels by a factor of f.

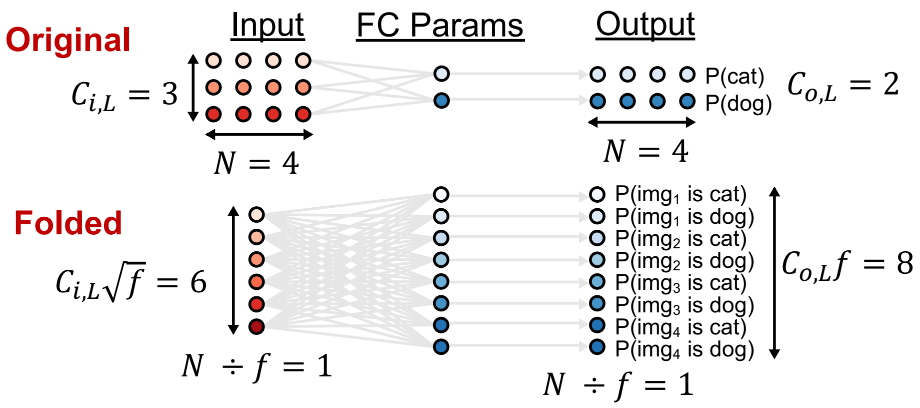


Figure 9.8: Output layer of a FoldedCNN with $f = 4$ and 2 classes. Unlike other layers, this layer has $f\times$ the number of outputs.

As a FoldedCNN performs inference over f images in a single input, it must return classification results for f images. To accommodate this, the output layer of a FoldedCNN produces outputs for fC_L classes, C_L for each of the f images stacked in a single input. This is illustrated in Figure 9.8. Note that this is similar to the manner in which k image inputs were represented to the learned encoders in §7.

These adjustments result in the first and last layers of FoldedCNNs performing slightly more FLOPs than those of the original CNN. The first layer of a FoldedCNN sets $C'_{i,1} = C_{i,1}f$, whereas other layers have $C'_{i,l} = C_{i,l}\sqrt{f}$. As the number of output channels in the first layer is also increased by $\sqrt{f}\times$, the first layer performs $\sqrt{f}\times$ more FLOPs than the original first layer (see Figure 9.7). This is also the case for the last layer of the FoldedCNN due to returning predictions for f images (see Figure 9.8). All other layers in the FoldedCNN perform the same number of FLOPs as those in the original CNN, as described previously. Despite this slight increase in FLOPs, §9.5 will show that FoldedCNNs, in fact, *achieve higher throughput* than the original CNN due to their increased arithmetic intensity.

9.4.2 Training a FoldedCNN

Training a FoldedCNN is similar to training the original CNN. Let N_T denote the training batch size. Each training iteration, N_T images are sampled and transformed into $\frac{N_T}{f}$ folded inputs as described above. A forward pass through the FoldedCNN results in an output of size $\frac{N_T}{f} \times fC_L$, as shown in Figure 9.8. This output is reshaped to be of size $N_T \times C_L$, and loss is computed on each of the N_T rows.

As each folded input consists of f images, and each image belongs to one of C_L classes, the effective number of classes for a FoldedCNN is C_L^f. This large increase in the number of classes can make it difficult to train a FoldedCNN for tasks with many classes to begin with. To combat this issue, we use a form of curriculum learning [80] specialized for FoldedCNNs. Training begins by sampling from only $I < C_L$ classes of the original CNN's dataset, and introducing Δ more classes every E epochs. We hypothesize that starting with a small number of classes I avoids overloading the FoldedCNN with a difficult task early on in training, as $I^f \ll C_L^f$. We find this form of training beneficial when C_L and f are large, and it yielded only marginal improvements in other settings.

9.5 Evaluation

9.5.1 Evaluation setup

We consider CNNs and tasks from the usecases described in §9.2.1: specialized CNNs from NoScope[4] as lightweight filters, and specialized CNNs from Microsoft. Each task and CNN is described in [58]. While the focus of this work is on specialized CNNs, we also evaluate on the more general ResNet-18 on CIFAR-10 and CIFAR-100.

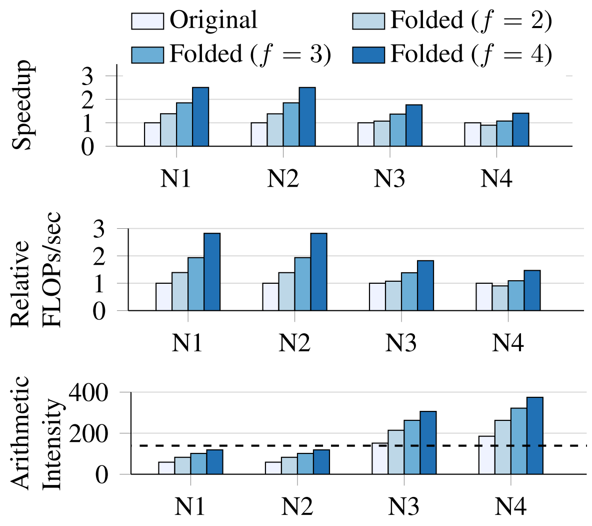
We evaluate FoldedCNNs with f of 2, 3, and 4, which increase the channels per layer by factors of roughly 1.41, 1.73, and 2, respectively ($\sqrt{f}\times$).[5] We compare FoldedCNNs to the compound scaling used in EfficientNets in §9.5.3.

Training setup. When training FoldedCNNs, we randomly assign images from the training set into groups of size f each epoch. Test sets are formed by randomly placing images from the test data into groups of f. Such randomization at test time avoids simpler settings, such folding f sequential frames in a video, thus providing a challenging scenario for FoldedCNNs. We also evaluate the sensitivity of FoldedCNNs to the order in which images are folded in §9.5.3.

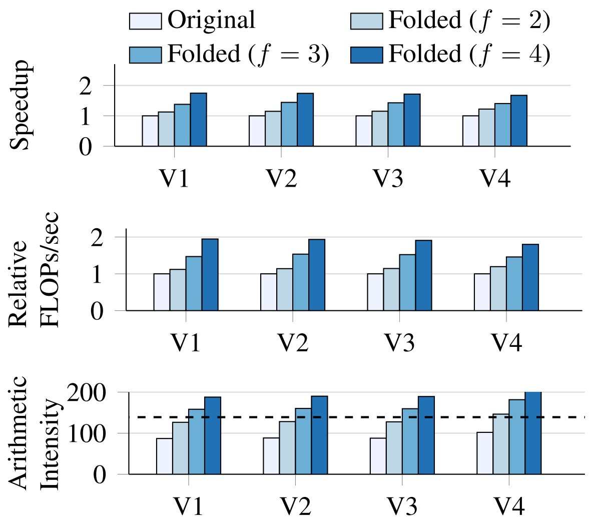
We train all CNNs using cross entropy loss. Training takes place for for 50 epochs with batch size of 128 for the NoScope tasks and for 1500 epochs with batch size of 32 for the game-scraping tasks. We use the curriculum learning in §9.4.2 for FoldedCNNs only on the game-scraping tasks. For these scenarios that use curriculum learning, we use $I = \max(f, \lfloor C_L/10 \rfloor)$, $\Delta = \lfloor C_L/10 \rfloor$, and $E = 60$. Such curriculum learning did not improve the accuracy of the original CNN. We use hyperparameters from NoScope [179] to train NoScope CNNs: RMSprop with learning rate 6.6×10^{-4} and Dropout of 0.25 after the second layer and before the last layer. All other models use Adam with learning rate 10^{-4} and weight decay of 10^{-5}.

[4]Our evaluation focuses only on specialized CNNs, and thus does not reflect the performance of the full NoScope system.

[5]The number of channels resulting from folding are rounded down to avoid a non-integer number of channels (e.g., $\lfloor C_i\sqrt{f} \rfloor$).



(a) Specialized CNNs use in NoScope



(b) Specialized CNNs used at Microsoft

Figure 9.9: Inference performance of FoldedCNNs relative to the original CNN on a V100 GPU. Arithmetic intensity is plotted in absolute numbers, and the dashed line shows the minimum arithmetic intensity required to reach peak FLOPs/sec on a V100 GPU.

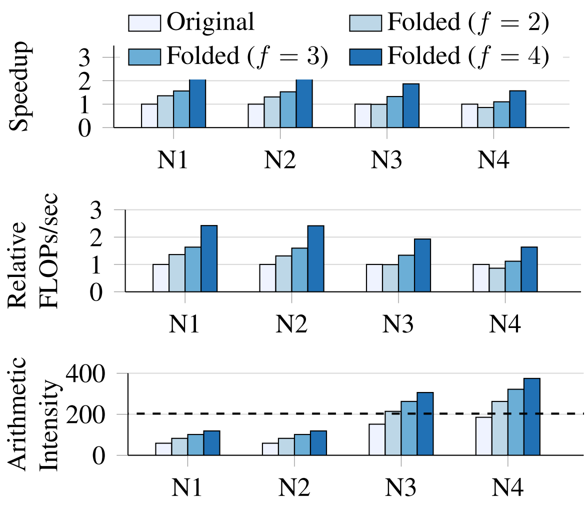
Table 9.4: Accuracy and speedup of FoldedCNNs for NoScope CNNs. Changes in accuracy are shown in parentheses.

	Original	**FoldedCNN (f = 2)**		**FoldedCNN (f = 3)**		**FoldedCNN (f = 4)**	
CNN	**Accuracy**	**Accuracy**	**Speedup**	**Accuracy**	**Speedup**	**Accuracy**	**Speedup**
N1	98.82	98.64 (-0.18)	1.39	98.35 (-0.47)	1.85	97.93 (-0.89)	2.51
N2	96.96	96.99 (0.03)	1.38	96.93 (-0.03)	1.85	96.75 (-0.21)	2.50
N3	94.84	94.95 (0.11)	1.07	94.82 (-0.02)	1.37	94.72 (-0.12)	1.76
N4	91.66	91.91 (0.25)	0.90	91.39 (-0.27)	1.07	91.21 (-0.45)	1.41

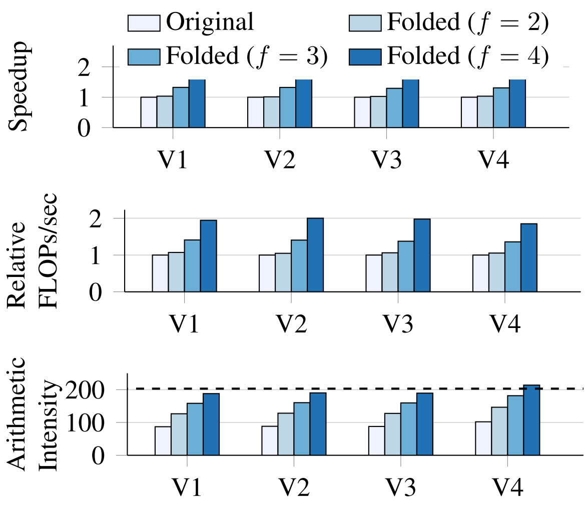
Inference setup. We evaluate inference on a V100 GPU (p3.2xlarge AWS instance), which is typical of hardware used for specialized CNN inference in datacenters [232]. We also evaluate on T4 GPUs (g4dn.xlarge AWS instance), which are common both in datacenters and edge clusters. When not mentioned explicitly, inference performance is evaluated on V100. Inference is performed in PyTorch with TensorRT [39] on CUDA 10.2. While FoldedCNNs can improve utilization for any numerical precision, we use half precision (FP-16) to use Tensor Cores. We report utilization (FLOPs/sec) and application-level throughput (images/sec) relative to the original CNN via the mean of 10 trials of 10000 inferences of batch size 1024. We use other batch sizes in §9.5.4. We call relative throughput "speedup."

9.5.2 Evaluation on specialized CNNs used in NoScope

Utilization and throughput. Figure 9.9a shows the speedup and FLOPs/sec of FoldedCNNs relative to the original CNN and the arithmetic intensity of each CNN on a V100 GPU. FoldedCNNs increase FLOPs/sec by up to 2.8× and throughput by up to 2.5×. Increased throughput



(a) Specialized CNNs used in NoScope



(b) Specialized CNNs used at Microsoft

Figure 9.10: Inference performance of FoldedCNNs relative to the original CNN on a T4 GPU. Arithmetic intensity is plotted in absolute numbers, and the dashed line shows the minimum arithmetic intensity required to reach peak FLOPs/sec on a T4 GPU.

speeds up tasks like offline analytics, while increased utilization enables higher throughput on a single accelerator and a better return on investment for deploying accelerators. FoldedCNNs match the $\sqrt{f}\times$ theoretical increase in arithmetic intensity described in §9.3, thus increasing utilization and throughput with higher f. Figure 9.10a illustrates similar performance on a T4 GPU.

FoldedCNNs result in larger improvements in utilization and throughput for the N1 and N2 CNNs (up to 2.8×) than for the N3 and N4 CNNs (up to 1.76×). This can be explained by arithmetic intensity: the N1 and N2 CNNs originally have very low arithmetic intensity. FoldedCNNs bring this arithmetic intensity much closer to that needed for peak performance on the V100 GPU, resulting in significantly higher utilization and throughput. In contrast, both N3 and N4 already have arithmetic intensity above the minimum needed for peak utilization, leaving less room for improvement. Despite this lower potential, FoldedCNNs still deliver up to 1.76× higher utilization and throughput for these CNNs.

Effect of tile quantization. There is only one case in which FoldedCNNs decrease throughput/utilization (N4, $f = 2$). This is caused by *GPU tile quantization*: when the problem size does not divide evenly into a chosen tile size (i.e., the size of partitions of the overall kernel) [241]. Many deep learning libraries are best optimized for cases in which certain parameters of a convolutional layer, such as input and output channels, are divisible by large powers of two (e.g., divisible by 64 or 128) [241]. Parameters that do not meet this requirement typically use kernels optimized for a larger tile, resulting in wasted work. For more details on the inefficiency resulting from tile quantization, please see NVIDIA's deep learning performance guide [241].

Many CNNs are already designed to have a number of input and output channels that are a power of two. However, FoldedCNN's increase the number of input and output channels by a factor of $\sqrt{f}$. For non-square values of f, such as 2 and 3, applying ParM to such a layer may result in a number of input or output channels that is no longer a power of two or is no longer divisible by a power of two. For example, applying ParM with $f = 2$ to a convolutional layer with 64 input and output channels will result in a convolutional layer with $\lfloor 64\sqrt{2} \rfloor = 90$ channels.

For the values of f considered in this work, we find that tile quantization primarily affects convolutions with a number of input and output channels greater than or equal to 64; we do not observe the negative effects often associated with tile quantization for convolutions with fewer channels, such as 32 or 16.

The N4 CNN contains two convolutions with 64 intermediate channels, followed by a fully-connected layer with 32 output neurons. The FoldedCNNs with f of 2 and 3 will thus lead to the negative effects of tile quantization for the convolutions in this CNN, but not for the fully-connected layer, which will receive the full benefits of folding. With $f = 2$, the benefit from folding does not outweigh the inefficiency due to tile quantization, resulting in a net decrease in utilization and throughput. In contrast, with $f = 3$, the benefits of folding outweigh the cost of tile quantization, resulting in an increase in utilization and throughput, albeit less pronounced than expected for $f = 3$.

It is important to note that this case with decreased utilization and throughput is not due to incorrectness of the transformations performed by FoldedCNNs. FoldedCNNs with f of 2 and 3 for the N4 CNN result in the expected $\sqrt{f}\times$ improvements in arithmetic intensity.

Accuracy. Table 9.4 shows the accuracy of FoldedCNNs on the NoScope tasks. FoldedCNNs maintain high accuracy: the accuracy of FoldedCNNs with $f = 2$ is, in fact, higher than that of the original CNN for three of CNNs, and only 0.18% lower on the fourth. For these cases, FoldedCNNs provide up to a $1.39\times$ speedup with the same accuracy.

As f increases, a FoldedCNN classifies more images per input, making the task of the FoldedCNN more challenging. As shown in Table 9.4 and Figure 9.9a, increasing f reduces accuracy but increases utilization and throughput, introducing a tradeoff that can be spanned based on the requirements of applications. We next analyze an example of this tradeoff.

When reasoning about the potential tradeoff between accuracy and throughput/utilization present with FoldedCNNs, it is important to consider the usecases of specialized CNNs. As described in §9.2.1, it is common to use specialized CNNs as a lightweight filter in front of a large, general-purpose CNN. In such systems, most inputs are processed only by the specialized CNN, rather than by both the specialized CNN and the general-purpose CNN. Thus, the throughput of the specialized CNN typically dominates the total throughput of the system.

Given the heavy use of the specialized CNN in this setup, improving the throughput of the specialized CNN at the expense passing more inputs to the general-purpose CNN may increase system throughput. For example, a FoldedCNN with $f = 4$ speeds up the N2 CNN by $2.50\times$ with a 0.21% drop in accuracy. We show below that this FoldedCNN increases system throughput unless the general-purpose CNN is over $285\times$ slower than the N2 CNN. Thus, the improved utilization and throughput of specialized CNNs made possible by FoldedCNNs can compensate for reduced their accuracy to improve total system throughput.

We now walk through this accuracy-throughput tradeoff via an abstract example. Figure 9.11

shows an abstract example of using a specialized CNN (e.g., those from NoScope) as a lightweight filter in front of a large, general-purpose CNN (e.g., ResNet-50). As shown in the figure, all inputs pass through the specialized CNN, which has a latency of T_s. The specialized CNN is unsure about u fraction of those inputs, and thus forwards these inputs to the general-purpose CNN, which has a latency of T_g. For the remaining $(1 - u)$ fraction of inputs, the specialized CNN is sure of its answer, and returns the prediction directly.

The expected latency for a given input to this system is thus:

$$E[T] = T_s + uT_g$$

Suppose that one replaced the specialized CNN used in such an application with a FoldedCNN that increases throughput by a factor of x, but decreases accuracy by a. Under the reasonable assumption that an increase in throughput leads to a corresponding decrease in latency, the latency of the FoldedCNN can be given as $\frac{T_s}{x}$. Furthermore, under the assumption that all incorrectly classified inputs from the specialized CNN are forwarded to the general-purpose CNN (i.e., u is equivalent to the error of the specialized CNN), then the FoldedCNN lets $u + a$ fraction of frames through to the general-purpose CNN. Thus, the expected latency for a given input to the system with a FoldedCNN is:

$$E[T] = \frac{T_s}{x} + (u + a)T_g$$

Clearly, for high values of x and small values a, the FoldedCNN can result in improved total system throughput (reciprocal of latency). A secondary question of interest is: given specific values of x and a, for what values of T_s and T_g does the FoldedCNN increase overall system throughput?

To answer this question, we focus on the ratio $\frac{T_g}{T_s}$. Intuitively, the higher this ratio, the larger the effect of inaccuracy of the FoldedCNN on overall system throughput. We next calculate the maximum value this ratio can be for a FoldedCNN to improve overall system throughput:

$$\begin{aligned}
T_s + uT_g &> \frac{T_s}{x} + (u + a)T_g \\
T_s - \frac{T_s}{x} &> (u + a)T_g - uT_g \\
T_s(1 - \frac{1}{x}) &> aT_g \\
\frac{1}{a}(1 - \frac{1}{x}) &> \frac{T_g}{T_s}
\end{aligned}$$

Consider the FoldedCNN with $f = 4$ for the N2 dataset. This FoldedCNN results in an increase in throughput of $x = 2.5\times$ and a decrease in accuracy of $a = 0.0021$. Plugging these values into the inequality above shows that this FoldedCNN will result in an overall improvement in system throughput so long as the general-purpose CNN is less than $285\times$ slower than the original specialized CNN. If we consider ResNet-50 as an example of a general-purpose CNN, this is easily satisfied for the N2 CNN: ResNet-50 is $83\times$ slower than the original specialized CNN.

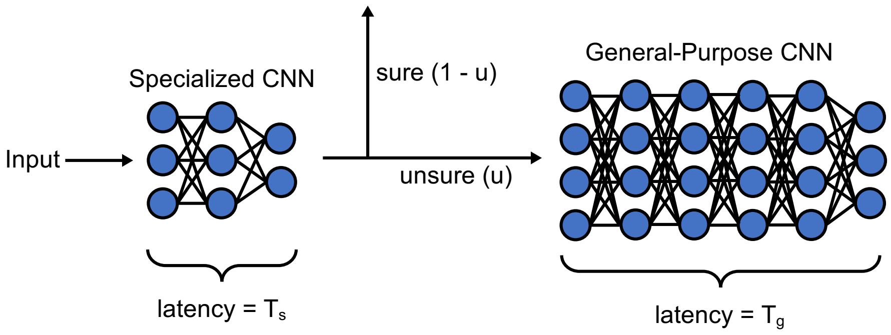


Figure 9.11: Abstract example of the use of a specialized CNN as a lightweight filter in front of a larger, general-purpose CNN.

Table 9.5: Performance of FoldedCNNs on specialized CNNs used at Microsoft. Differences in accuracy are listed in parentheses. Input resolution ("Res.") and the number of classes ("Cls.") for each CNN are also listed.

			Original	FoldedCNN (f = 2)		FoldedCNN (f = 3)		FoldedCNN (f = 4)	
CNN	**Res.**	**Cls.**	**Accuracy**	**Accuracy**	**Speedup**	**Accuracy**	**Speedup**	**Accuracy**	**Speedup**
V1	(22, 52)	11	97.64	97.64 (0.00)	1.13	97.18 (-0.46)	1.38	95.27 (-2.37)	1.75
V2	(19, 25)	22	93.45	92.09 (-1.36)	1.15	90.00 (-3.45)	1.44	89.91 (-3.54)	1.74
V3	(17, 40)	15	98.50	97.43 (-1.07)	1.15	97.20 (-1.30)	1.43	96.87 (-1.63)	1.71
V4	(22, 30)	27	96.52	96.52 (0.00)	1.22	96.00 (-0.52)	1.40	94.41 (-2.11)	1.67

9.5.3 Evaluation on production game-scraping CNNs

Figure 9.9b and Figure 9.10b show the utilization, throughput, and arithmetic intensity of FoldedCNNs on the production game-scraping tasks. FoldedCNNs increase FLOPs/sec by up to 1.95× and throughput by up to 1.75× compared to the original CNN. Table 9.5 shows that FoldedCNNs have accuracy drops of 0–1.36%, 0.46–3.45%, and 1.63–3.54% with f of 2, 3, and 4 on these tasks. These drops are larger than those on the NoScope tasks due to the higher number of classes in the game-scraping tasks. While the NoScope tasks have only two classes, the game-scraping tasks have 11–27 classes. Thus, lower accuracy on the game-scraping tasks is expected from FoldedCNNs. That said, FoldedCNNs still enable large improvements, such as a 1.22× speedup with no accuracy loss for V4 with $f = 2$.

Effect of image order. As FoldedCNNs jointly classify f distinct images concatenated over the channels dimension, a natural question is how sensitive FoldedCNNs are to the order in which images are folded. To investigate this, we measure how often the predictions made by FoldedCNNs for each image match for all $f!$ permutations of f images folded together (e.g., how often do predictions for image X_1 match in folded inputs (X_1, X_2) and (X_2, X_1) for $f = 2$). With f of 2, 3, and 4, the average percentage of matching predictions for all $f!$ permutations on the V1 task is 98.8%, 98.4%, and 98.0%, showing high invariance to image order.

Comparison to EfficientNet scaling. We next compare FoldedCNNs to the techniques used in EfficientNets [295]. EfficientNets trade FLOPs and accuracy by jointly scaling the number of layers, the width, and the input resolution of a CNN. While such scaling can increase through-

Table 9.6: FoldedCNNs and EfficientNet compound scaling on game-scraping tasks. Speedup, utilization, and arithmetic intensity are relative to the original CNN.

		Higher values are better			
CNN	**Mode**	**Accuracy**	**Speedup**	**Utilization**	**Arithmetic intensity**
V1	EfficientNet	93.27%	1.32	0.83	0.91
	Fold (f = 4)	95.27%	1.75	1.95	2.16
V2	EfficientNet	84.91%	1.51	0.80	0.88
	Fold (f = 4)	89.91%	1.74	1.93	2.15
V3	EfficientNet	96.40%	1.46	0.75	0.87
	Fold (f = 4)	96.87%	1.71	1.91	2.16
V4	EfficientNet	95.19%	1.34	0.83	0.91
	Fold (f = 3)	96.00%	1.40	1.46	1.78
	Fold (f = 4)	94.41%	1.67	1.80	2.10

put by reducing FLOP count, reducing FLOP count in this manner can also decrease arithmetic intensity and utilization. To illustrate this, we transform the game-scraping CNNs with EfficientNet compound scaling[6] with the recommended parameters from the EfficientNet paper [295]: using terminology from the paper, $\phi = -1$, $\alpha = 1.1$, $\beta = 1.2$, and $\gamma = 1.15$. This transforms a CNN to perform roughly $2\times$ fewer FLOPs, which increases throughput.

Table 9.6 compares FoldedCNNs and EfficientNets on the game-scraping CNNs. For each task, a FoldedCNN achieves both higher accuracy and throughput than the EfficientNet variant. For example, for V1, a FoldedCNN has 2% higher accuracy and 33% higher throughput than the EfficientNet variant. Furthermore, whereas EfficientNets reduce arithmetic intensity and utilization for all CNNs due to decreased FLOP count, FoldedCNNs uniformly increase arithmetic intensity and utilization. These results show the promise of the new approaches proposed in FoldedCNNs targeted specifically for large-batch, specialized CNN inference.

9.5.4 Varying batch size

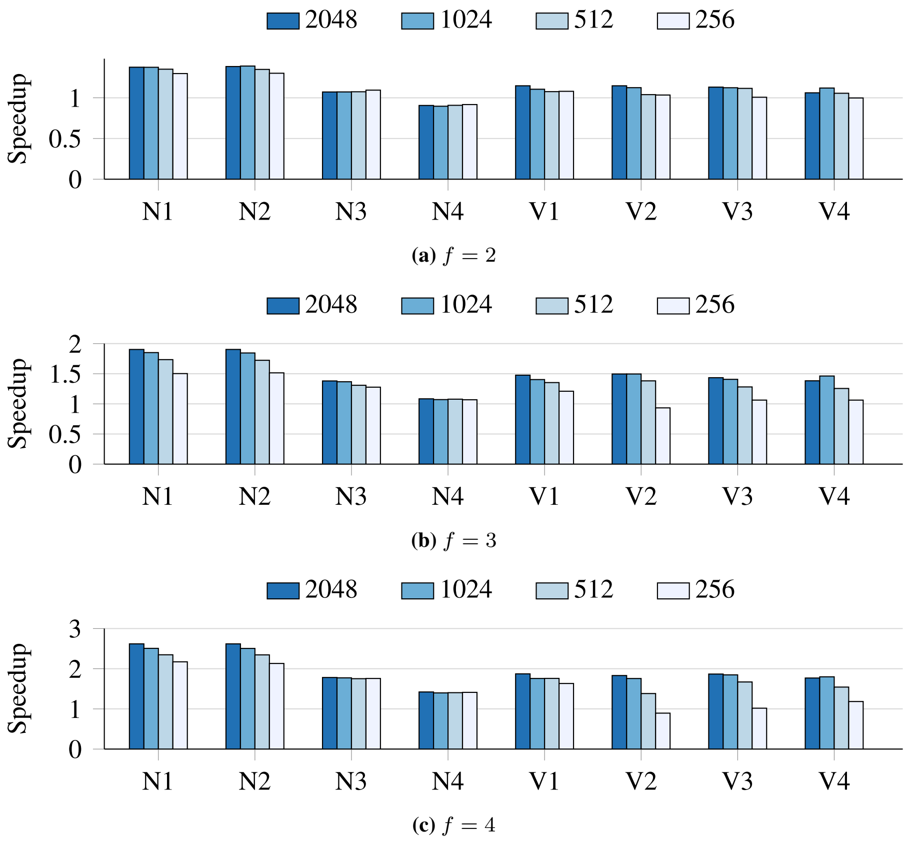
Figure 9.12 shows the throughput improvement when using FoldedCNNs with various values of f relative to the original CNN at varying batch sizes. As shown in the figure, the throughput improvement resulting from folding is largest at a batch size of 2048, and decreases with decreasing batch size. This behavior is expected, as decreasing batch size N decreases the likelihood that the inequality proved in §9.3 will hold, and thus that folding will benefit.

9.5.5 Folding grouped convolutions

In this section, we describe how folding is applied to group convolutions.

Background on group convolutions. In a group convolution, the input and output channels of the convolution are split into G groups. Each output channel in a particular group is computed via

[6]We do not use the EfficientNet-B0 architecture because it is significantly larger than typical specialized CNNs.



(a) $f = 2$

(b) $f = 3$

(c) $f = 4$

Figure 9.12: Speedup of FoldedCNNs with varying f at various batch sizes relative to the throughput of the original CNN at corresponding batch sizes.

convolution over only those input channels in the corresponding group. This results in a G-fold decrease in operations and a G-fold decrease in the number of parameters in the convolutional layer. The resultant arithmetic intensity for a group convolution is thus:

$$\frac{2NHWC_oC_iK_HK_W/G}{B(NHWC_i + \frac{C_iK_HK_WC_o}{G} + NHWC_o)}$$

The arithmetic intensity of a group convolution in the batch-limited regime is determined as follows (recalling from that calculating arithmetic intensity in the batch-limited regime involves removing the variable B):

$$A = \frac{2NHWC_oC_iK_HK_W/G}{NHWC_i + \frac{C_iK_HK_WC_o}{G} + NHWC_o}$$
$$\lim_{N\to\infty} A = \frac{2C_oC_iK_HK_W}{C_i + C_o} * \frac{1}{G}$$

Comparing this arithmetic intensity to that in Equation 9.3, the arithmetic intensity of a group convolution with G groups in the batch-limited regime is $G\times$ lower than a corresponding "vanilla" convolution. This makes group convolutions a promising target for increasing arithmetic intensity via folding.

Table 9.7: Group convolutions evaluated

Name	C_i	C_o	G	K_H	K_W	H	W
G32	32	32	4	3	3	50	50
G64	64	64					

Applying folding to group convolutions. Folding group convolutions is straightforward. Similar to folding "vanilla" convolutions, a FoldedCNN for a group convolution with C_i input channels and C_o output channels reduces batch size by a factor of f and increases C_i and C_o each by a factor of $\sqrt{f}\times$. This results in increasing the number of channels per group in the group convolution by a factor of $\sqrt{f}$, and thus also increases arithmetic intensity in the batch-limited regime by a factor of $\sqrt{f}$.

Inference performance of folded group convolutions. We evaluate the throughput and utilization of folding on two group convolutions shown in Table 9.7. The two group convolutions are identical other than the number of total input and output channels, with G32 having 32 and G64 having 64. Each setting uses 4 groups, leading to 8 and 16 channels per group for G32 and G64, respectively. We compare the throughput and utilization of these convolutions to the corresponding folded version with $f = 4$. Folding results in 64 input and output channels with 16 channels per group for G32, and 128 input and output channels with 32 channels per group for G64.

With batch size of 1024, folding with $f = 4$ increase throughput and utilization of these grouped convolutions by $1.74\times$ for G32 and by $1.59\times$ for G64 on a V100 GPU. Folding increases arithmetic intensity by nearly a factor of two for each convolution. The larger improvement for G32 compared to G64 comes from the lower arithmetic intensity of G32; due to having half the number of input and output channels of G64, G32 has half the arithmetic intensity. Thus, there is more room for improving the utilization of G32 by increasing arithmetic intensity alone via FoldedCNNs. These results show the effectiveness of folding on group convolutions.

9.5.6 Folding for Winograd convolutions

FoldedCNNs can benefit a wide variety of convolutional implementations, such as direct convolutions, matrix-multiplication-based convolutions, and Winograd convolutions. In fact, our prior evaluation results runs atop TensorRT, which selects among convolutional implementations, including Winograd. To more clearly illustrate the performance of FoldedCNNs on Winograd convolutions, we also directly run FoldedCNNs using Winograd convolutions in cuDNN. Here, on the video scraping CNNs, FoldedCNNs with $f = 4$ provided a median speedup of $1.66\times$ over the original CNN, matching the speedups found for other CNNs above.

9.5.7 FoldedCNNs in non-target settings

As described in §9.3, our focus in FoldedCNNs is on small CNNs with low arithmetic intensity even at large batch size, and specialized tasks with few classes. For completeness, we now

Table 9.8: Performance of FoldedCNNs for CNNs with many classes. Differences in accuracy are listed in parentheses. Input resolution ("Res.") and the number of classes ("Cls.") for each CNN are also listed.

			Original	**FoldedCNN (f = 2)**		**FoldedCNN (f = 3)**		**FoldedCNN (f = 4)**	
CNN	**Res.**	**Cls.**	**Accuracy**	**Accuracy**	**Speedup**	**Accuracy**	**Speedup**	**Accuracy**	**Speedup**
V5	(22, 52)	111	93.95	92.50 (-1.45)	1.12	90.78 (-3.17)	1.42	87.51 (-6.44)	1.75
V6	(15, 35)	62	89.71	88.03 (-1.68)	1.08	85.48 (-4.23)	1.42	84.92 (-4.79)	1.71

evaluate FoldedCNNs on general-purpose CNNs and tasks, which are not in this target regime. We also evaluate small CNNs for tasks with many classes.

Accuracy on general tasks. To evaluate the accuracy of FoldedCNNs on general-purpose tasks, we consider ResNet-18 FoldedCNNs on CIFAR-10 and CIFAR-100.

For CIFAR-10, we train a FoldedCNN with $f = 4$ via distillation with the original CNN as the "teacher" [73]. The original ResNet-18 has an accuracy of 92.98%, while the FoldedCNN has an accuracy of 92.10%. This small accuracy drop even with high f shows the potential applicability of FoldedCNNs to general-purpose tasks.

For CIFAR-100, we do not observe benefit from the same distillation used for CIFAR-10. The original ResNet-18 on CIFAR-100 achieves 70.3% accuracy, while FoldedCNNs have accuracies of 68.11% (2.19% drop), 67.44% (2.86% drop), and 65.76% (4.54% drop) with f of 2, 3, and 4. These larger drops compared to CIFAR-10 can be attributed to the higher number of classes in CIFAR-100, which makes the task of a FoldedCNN more challenging (see §9.4.2).

Speedup on general CNNs. We now evaluate the speedup of FoldedCNNs when the original CNN is the general-purpose ResNet-18 operating on CIFAR-10. A FoldedCNN with $f = 4$ in this setup improves throughput by 8.1%. This speedup is smaller than those observed in Figure 9.9 because ResNet-18 has arithmetic intensity of 430, much higher than the minimum needed for peak FLOPs/sec on a V100 (139). This places ResNet-18 outside the target regime of FoldedCNNs. FoldedCNNs still do provide 8.1% speedup, as 24% of the layers in ResNet-18 have low arithmetic intensity.

Accuracy on small CNNs with many classes. We also consider CNNs that have the same size as specialized CNNs, but which operate over many classes. We consider two new game-scraping tasks: a task with 111 classes (V5), and one with 62 classes (V6[7]). We use the same CNN as that used for V1. Table 9.8 shows that FoldedCNNs exhibit larger drops in accuracy on these tasks due to the larger number of classes, but still increase utilization/throughput by up to 1.75×.

Takeaway. Coupling these moderate benefits in non-target settings with large benefits in target settings, FoldedCNNs show promise for increasing the utilization and throughput of specialized CNN inference beyond increased batch size.

[7]For this CNN, we find that the small input resolution and large number of classes requires using more specially-tuned curriculum learning parameters. Specifically, when training a FoldedCNN with $f = 4$ on this dataset, we use $I = 4$, $\Delta = 3$, and $E = 120$, and train the CNN for 3000 epochs.

9.6 Related Work

Efficient neural architectures. There is a large body of work on designing CNNs for efficient inference (e.g., [90, 91, 218, 295, 319, 346]). Many of these works aim to reduce latency, but often do not consider accelerator utilization, which is a primary objective of FoldedCNNs. Some of these approaches, such as EfficientNets [295], reduce the number of FLOPs performed by a CNN to achieve lower latency. However, we show in §9.5 that doing so can, in fact, reduce accelerator utilization. Furthermore, compared to these approaches, FoldedCNNs employ a fundamentally new structure to CNN inputs and classification, which could be integrated into existing architecture search techniques. Finally, FoldedCNNs are designed primarily for large-batch, specialized CNN inference, whereas existing works typically target general-purpose CNNs.

Improving throughput. Many other techniques have been proposed to accelerate inference, but which do not target utilization. Network pruning [83] can improve throughput by reducing the FLOP count of a CNN, but, similar to the approaches described above, can reduce utilization. Reducing the numerical precision used during inference can increase throughput [308], but is insufficient for increasing utilization on modern accelerators (as we show in §9.2.3). Folding can be applied on top of these techniques to further improve the utilization and throughout of specialized CNN inference. In fact, our evaluation in §9.5 applies FoldedCNNs atop low-precision specialized CNNs.

Multitenancy. There is a growing body of work on increasing accelerator utilization by performing inference for multiple models on the same device [116, 164, 235, 282, 332]. These works do not improve the utilization of individual models, which is the goal of FoldedCNNs. Thus, these works are complementary to FoldedCNNs.

9.7 Conclusion

This section of the thesis illustrated that coding-theory-inspired ideas can be used to boost the throughput and accelerator utilization of ML systems—a goal that falls outside the usual target of codes of improving reliability. Specifically, following the insights developed in §4.3 of this thesis, we show that specialized CNNs poorly utilize server-grade accelerators due to their low arithmetic intensity, even at large batch sizes. We then show that leveraging ideas inspired by the coding-based approaches proposed in §7 of this thesis, combined with small modifications to CNN architecture, can boost arithmetic intensity beyond the limits of increased batch size. This allows the resultant FoldedCNNs to significantly increase throughput and accelerator utilization, while maintaining similar accuracy to the original CNN.

Chapter 10

Accelerating erasure codes with little developer expertise via ML libraries

The previous portions of this thesis have focused on ways in which coding-theoretic ideas can benefit ML systems and how machine learning can benefit coding-theoretic tools. This chapter explores a different, but related, interaction between these domains: ways in which advancements in ML *systems* can improve the development of erasure-coded systems.

We focus on the development of high-performance erasure-coding libraries, which are critical given the widespread use of erasure codes in production many storage systems. However developing optimized erasure-coding libraries currently requires a deep understanding of both the mathematical underpinnings of erasure codes and techniques to achieve high performance on a given hardware platform. The need for this unique skillset makes developing optimized erasure-coding libraries challenging enough today, and likely even more challenging in the future given increasing trends in hardware heterogeneity.

We make the case that the growth of fast ML libraries may serve as a lifeboat for easing the development of current and future optimized erasure-coding libraries: fast erasure-coding libraries for various hardware platforms can be easily implemented by using existing optimized ML libraries. We show that the computation structure of many erasure codes mirrors that common to matrix multiplication, which is heavily optimized in ML libraries. Due to this similarity, one can implement erasure codes using ML libraries with few lines of code and with little knowledge of erasure codes, while immediately adopting the many optimizations within these libraries, without requiring intimate knowledge of high-performance programming. We develop prototypes of our proposed approach using two different ML libraries targeting CPUs and GPUs. Our prototypes are up to 2.2$\times$ faster than state-of-the-art erasure-coding libraries.

10.1 Introduction

Recall that erasure codes enable storage systems to reliably store data with significantly less storage overhead than replication [220], and, thus, are widely used in storage systems (e.g., [135, 159, 252, 263, 313]).

Significant work has been devoted to developing high-performance software libraries for erasure coding (e.g., [19, 253, 302, 345]). Many system-level optimizations have been used, such as vectorization and cache optimization [256, 302, 345]. Algorithmic techniques have also been developed, such as exploiting properties of the primitive operations performed in erasure coding (e.g., XORs) to minimize the number of operations performed [257]. Beyond targeting CPUs [19, 253, 302, 345], erasure codes have also been optimized for GPUs [110, 210], FPGAs [95], and network devices [259, 284, 285].

Developing erasure-coding libraries is challenging.

However, developing optimized erasure-coding libraries is currently challenging because it requires expertise in both the mathematical underpinnings of erasure codes and methods to achieve high performance on a given hardware platform. While there are individuals with this skillset, we argue that they are few and far between. This limits the velocity with which these libraries can be developed and leads to software fragility: relying on a select-few experts to maintain a library threatens the library when, for example, such experts retire.

While the current state of developing optimized erasure-coding libraries is far from ideal, we argue that trends in computer hardware will further exacerbate this problem. In particular, as computer systems leverage increasingly heterogeneous hardware, efficient erasure-coding libraries will be required for a variety of hardware platforms. This requirement stems from two observations: (1) With the rise of applications that are primarily run on accelerators, such as deep-learning training, data that needs to be erasure coded will more frequently reside on accelerators, rather than in host memory. (2) Accelerators are being equipped with enhanced I/O paths for accessing storage and the network [260]. For example, NVIDIA's GPUDirect Storage technology [34] enables GPUs to bypass the CPU when accessing storage devices.

The combination of these trends gives rise to an enticing opportunity to perform erasure coding on accelerators, rather than shipping data to CPU to perform erasure coding. However, developing optimized erasure-coding libraries for a variety of hardware platforms only exacerbates the aforementioned challenge: not only do developers need to understand the mathematics of erasure codes, they now must also understand architectural details of *a variety of hardware platforms.*

To ease the development of current erasure-coding libraries and usher in future erasure-coded systems, a new approach to developing optimized erasure-coding libraries is needed.

Toward simpler erasure-coding library development.

In this section of the thesis, we make the case that current trends in optimized *machine learning (ML) libraries* offer a promising lifeboat for the development of current and future erasure-coding libraries. Our case rests on the following observations:

Observation 1: Erasure codes have similar structure to general matrix multiplication (GEMM), which is heavily optimized in ML libraries. However, erasure coding requires the arithmetic to be performed over mathematical structures called finite fields which are not supported by ML libraries. Performing arithmetic over finite fields is generally more time consuming than traditional binary arithmetic. A common optimization used by many high-performance erasure-coding libraries is to avoid finite-field arithmetic via "bitmatrix erasure coding," which requires only the primitive operations of bitwise AND and XOR [85]. Although optimized implementations of bitmatrix erasure coding forgo matrix operations for high performance, one can use this construction to decompose erasure coding into a matrix-matrix multiplication, but with multiplication performed as bitwise AND and addition as bitwise XOR. The net result is a computation structure that matches nearly-identically with that of a GEMM, but with a different primitive operation, as will be shown in §10.4. Thus, *many of the components needed to implement high-performance erasure codes already exist in ML libraries.*

Observation 2: ML libraries are optimized to exploit the latest hardware features. Due to the popularity of machine learning, significant effort has been undertaken to ensure that ML libraries make the best use of the hardware on top of which they run. These libraries often achieve near-peak performance on a given platform when executing GEMMs and are kept up-to-date with the latest hardware advancements [56]. Thus, *ML libraries are expected to continue to achieve high performance as hardware evolves.*

Observation 3: Significant effort has been devoted to making ML libraries portable across hardware platforms. The aforementioned challenge of developing high-performance software for multiple hardware platforms is not unique to erasure coding. In fact, this challenge is being addressed at a rapid pace in the development of ML libraries. Because ML libraries must run on the many ML accelerators under development, significant effort has been devoted to developing ML libraries that achieve high performance on a variety of hardware platforms. For example, so-called "ML compilers," such as Apache TVM [97], aim to generate high-performance implementations of ML operations for various hardware platforms. In many cases, such compilers achieve similar performance as vendor-optimized solutions that target specific hardware platforms. Thus, *ML libraries are likely to be portable across the increasingly-diverse spectrum of hardware platforms.*

Combining these observations, we propose that (1) By representing erasure coding as "matrix multiplication" (but replacing multiplication with AND, and addition with XOR), erasure codes can easily be implemented via ML libraries (and, thus, require little development effort); (2) Erasure codes implemented via ML libraries will immediately adopt the many performance optimizations currently within ML libraries and those that will come as hardware evolves (and, thus, require little optimization and maintenance effort); and (3) Erasure codes implemented via ML libraries are likely to be able to execute on a variety of hardware platforms.

While this proposal appears promising for reducing development and optimization effort, it comes with the downside of forgoing performance optimizations specific to erasure coding, which are not included in ML libraries. However, as we show in §10.4.4, many of the most-critical optimizations performed by erasure-coding libraries are also used in optimizing GEMMs. Thus, erasure codes implemented via ML libraries retain the critical optimizations that would be found in custom erasure-coding libraries. As will be shown in §10.6, even without erasure-coding-specific optimizations, our proposed approach often outperforms custom erasure-coding libraries.

To support these claims, we develop erasure-coding libraries using two different popular ML libraries: Apache TVM [97] and NVIDIA CUTLASS [31]. Atop these libraries, we develop TVM-EC and CUTLASS-EC, which leverage these respective libraries to perform erasure coding. We compare the performance of these prototypes to state-of-the-art erasure-coding libraries targeting CPUs [19, 302] and GPUs [210]. TVM-EC and CUTLASS-EC achieve up to 1.75$\times$ and 2.2$\times$ higher encoding and decoding throughput than the state-of-the-art custom erasure-coding libraries for CPUs and GPUs, respectively. Furthermore, these prototypes required little development effort: (1) each prototype was implemented in only tens of lines of code, and (2) TVM-EC enables one to develop in the user-friendly Python language and export high-performance im-

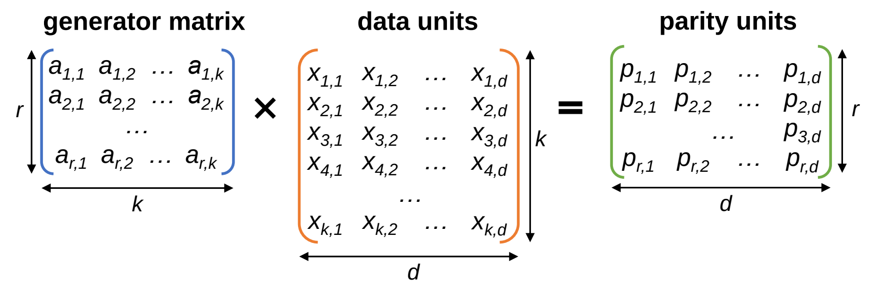


Figure 10.1: High-level depiction of encoding operation

plementations found by TVM to lower-level languages (e.g., C++) for later use. Finally, our prototypes show the benefit of learning-based autotuning in ML libraries to automatically discover complex system-level optimizations that developers would otherwise need to implement by hand.

These results showcase the promise of leveraging ML libraries to easily develop optimized erasure-coding libraries targeting current and future hardware platforms, and, thus, to usher in the next generation of erasure-coded systems.

10.2 Background on erasure coding

An erasure code *encodes* k data units to produce r parity units and stores all $(k + r)$ data units on separate storage devices.[1] Parity units are formed such that reading any k of the total $(k + r)$ data and parity units suffices for a *decoder* to recover the original k data units. Thus, an erasure code withstands up to r lost units with storage overhead of only $\frac{k+r}{k}$. These properties hold for a class of codes known as "maximum distance separable" (MDS) codes. We focus on MDS codes because many popular erasure codes in storage systems are MDS codes (e.g., Reed-Solomon codes). However, as we show in §10.5 and §10.6, our proposal is applicable to the broader class of "linear" erasure codes. To the best of our knowledge, all erasure codes used in storage systems are linear.

We next describe the high-level operation of encoding and decoding in an erasure code. As is common, we use matrix notation to illustrate these procedures. However, as we describe in §10.2, many current high-performance erasure-coding libraries *do not use matrix operations under the hood in order to leverage several erasure-coding-specific optimizations.*

Encoding. The encoding process of an erasure code takes in k data units and produces r parity units, where each unit contains d elements. Each parity unit is formed via a linear combination of the k data units. The encoding of a single parity element can thus be viewed as the dot product between a vector of k coefficients and a vector of k data elements. Expanding this to the encoding of r parity units, encoding can be viewed as a matrix-matrix multiplication between a *generator matrix* E of size $(r \times k)$ and a *data matrix* D of size $(k \times d)$ to produce a matrix P of size $(r \times d)$ containing parity units. This is illustrated in Figure 10.1. Using simple matrix notation, this is represented as $P = ED$. However, as we describe in §10.2, erasure codes require performing different arithmetic than normally performed in matrix multiplication.

[1]It is also common to describe erasure codes via parameters n and k, where n is the total number of data and parity units (i.e., $n = k + r$).

Decoding. Recall that the erasure codes we consider in this section of the thesis have the ability to decode (i.e., reconstruct) the k original data units given any k out of the total $(k + r)$ data and parity units. This decoding process takes place as follows: (1) Form a new matrix C of size $((k + r) \times k)$ by performing a row-wise concatenation between a $k \times k$ identity matrix and the generator matrix E of size $r \times k$. (2) Erase from C the r rows corresponding to the missing units. The resultant $k \times k$ matrix is labelled C_e. (3) Create a $k \times d$ matrix A consisting of the k available data/parity units. The given operands now satisfy the following equation: $A = C_e D$. (4) To find D, the decoding process then computes $D = C_e^{-1} A$. All possible $k \times k$ matrices C_e resulting from erasing r rows of C are guaranteed to be invertable based on the construction of the MDS code.

The steps required to generate the inverted matrix C_e^{-1} are frequently performed offline, and the resultant inverted matrices are stored in a lookup table. This reduces the decoding process to one of gathering available data/parity units, and performing the matrix-matrix product for decoding.

Implementing erasure codes. We now describe how the encoding and decoding operations described above are implemented. Due to the similarity between the two operations, we focus only on encoding here.

The arithmetic operations performed in erasure codes take place over finite fields (Galois Fields). Each element in the generator matrix is from a finite field, and all multiplication and addition operations take place using the finite-field arithmetic. Finite-field arithmetic is more computationally intensive than traditional arithmetic: for the fields typically used in erasure coding, additions take place via XOR, whereas multiplication requires multiplication and modulus over polynomials [255]. As will be described in §10.2.1, high-performance erasure-coding libraries typically use optimizations to avoid performing such computationally-expensive multiplications.

Finally, it is important to note that, while the descriptions of encoding/decoding in an erasure code typically use matrix notation, *many high-performance erasure-coding libraries today do not leverage matrix operations for erasure codes* [253, 302]. Instead, many implementations treat the equations for each individual parity separately. For example, such libraries may represent the encoding of parity p_i from data units $X_1, X_2, \ldots, X_k$ as $p_i = \alpha_{i,1} X_1 + \alpha_{i,2} X_2 + \ldots + \alpha_{i,k} X_k$, where $\alpha_{i,1}, \alpha_{i,2}, \ldots, \alpha_{i,k}$ are the k elements of the ith row of the generator matrix. This enables one to reuse identical partial summations performed in encoding two distinct parities (e.g., reusing $\alpha_{i,1} X_1 + \alpha_{i,2} X_2$ in the encoding of p_j when $\alpha_{i,1} = \alpha_{j,1}$ and $\alpha_{i,2} = \alpha_{j,2}$).

10.2.1 Optimizing erasure code implementations

Erasure codes are heavily used in production storage systems (e.g., Ceph [9], HDFS [16], Azure Storage [159]). Thus, it is critical that erasure-coding libraries operate with high performance. We next describe techniques that have been used for developing high-performance erasure-coding libraries.

Bitmatrix erasure coding. As described above, the arithmetic used in encoding and decoding in erasure codes is done via finite-field arithmetic, which is often far more computationally expensive than traditional arithmetic. To ameliorate this expense, the so-called "bitmatrix" erasure-coding procedure transforms an erasure code that operates over a Galois Field of size 2^w into one operating over a Galois Field of size 2, that is, *binary* [85]. This enables all arithmetic operations to be performed using computationally-inexpensive operations, such as bitwise AND and XOR [85, 253, 345]. To do so, each element in the generator matrix is converted to a $w \times w$ matrix of binary elements, and each entry in the data matrix is converted to a length-w column vector of binary elements. We refer the reader to the works of Bloemer et al. [85], Plank et al. [255], and Zhu et al. [345] for further details. The result of this conversion is an generator matrix of size $rw \times kw$ and a data matrix of size $kw \times d$, each of binary values.

The primary benefit of bitmatrix erasure codes is that all encoding and decoding operations are now performed via the primitive bitwise AND and XOR operations. Recall from §10.2 that the encoding of a single parity unit p_i using a Galois Field of size 2^w (GF(2^w)) can be viewed as the equation:

$$p_i = \alpha_{i,1}X_1 + \alpha_{i,2}X_2 + \ldots + \alpha_{i,k}X_k,$$

where each $\alpha_{i,j}$ belongs to GF(2^w). Transitioning this to the corresponding bitmatrix erasure code, we now have

$$p_i = \beta_{i,1}b_1 + \beta_{i,2}b_2 + \ldots + \beta_{i,kw}b_{kw},$$

where each $\beta_{i,j}$ and b_j belongs to GF(2), that is, they are *binary*. This results in products of the form $\beta_{i,j}b_j$ being performed as the bitwise AND between $\beta_{i,j}$ and b_j, and the summations being performed as bitwise XOR. Note that, due to the bitmatrix transformation, the equation above now has kw data elements and coefficients, rather than the k in the original equation. Similarly, there will be w such equations in the new construction for each parity in the original construction.

This representation enables simplification of the equation used in encoding a parity: since all coefficients $\beta_{i,j}$ are either 0 or 1, the equation for computing a parity p_i can be rewritten as the sum (XOR) of all data units b_j for which $\beta_{i,j}$ is 1. For example, encoding $p_i = \beta_{i,1}b_1 + \beta_{i,2}b_2 + \beta_{i,3}b_3$ with $\beta_{i,1} = 1$, $\beta_{i,2} = 0$, and $\beta_{i,3} = 1$ can be simplified as $p_i = b_1 \oplus b_3$.

Finally, while the equations above perform ANDs and XORs in a bit-by-bit fashion, bitwise AND and XOR operations are typically implemented via the bitwise AND/XOR of two regions of 8–512 contiguous bits for efficiency.

System-level optimizations. Many classic techniques for developing high-performance software on CPUs have similarly been used to accelerate erasure codes on CPUs. Examples of these include vectorization [256, 345] and techniques to make the best use of the memory hierarchy [217, 345]. Erasure codes have also been executed on GPUs [110, 210], FPGAs [95], SmartNICs [284, 285], and programmable switches [259]. These works each consider specific features of the target hardware in implementing erasure codes, and, thus, are typically not portable across different classes of hardware. As an example, recent Intel CPUs are equipped with Galois Field New Instructions (GFNI) [17], which provide hardware-accelerated arithmetic over Galois Fields. However, the same functionality is not available on AMD CPUs, which increases the complexity of developing portable high-performance erasure-coding libraries using this functionality.

Algorithmic optimizations. In addition to system-level optimizations, many optimizations have been developed that exploit properties specific to bitmatrix erasure codes.

One such technique is to search for generator matrices that achieve the desired level of fault tolerance with as few ones in the matrix as possible [84, 254]. As illustrated above, reducing the number of ones in the generator matrix reduces the number of XORs that must be performed in encoding.

Another algorithmic technique used to accelerate bitmatrix erasure codes is to schedule the XORs performed in encoding parities so as to minimize the total number of XORs performed. Multiple works propose to exploit partial work completed in computing one parity to compute other parities. While doing so can minimizes the number of XOR operations performed, this does not always result in optimal performance on modern hardware. Nevertheless, multiple works have shown that this heuristic is effective in accelerating bitmatrix erasure codes [158, 257, 302, 345].

Landscape of optimized erasure coding. Multiple highly-optimized erasure-coding libraries have been developed, such as Intel ISA-L [19], Jerasure [253], and research libraries (e.g., [210, 302, 345]). To exploit the optimizations described above, each of these are *custom* erasure-coding libraries designed only to perform erasure coding.[2] While this enables such libraries to use optimizations specific to erasure coding, we will describe next how such customization hinders the development of erasure-coding libraries.

10.3 Need for rethinking erasure-coding libraries

While erasure codes are critical to many storage systems, we argue that the current approach to developing optimized erasure-coding libraries leaves much to be desired.

As described in §10.1 and §10.2.1, developing optimized erasure-coding libraries is currently challenging because it requires exploiting low-level hardware features *and* knowledge of the mathematical underpinnings of erasure codes. Expertise in these areas typically comes from the disparate domains of computer architecture and information theory, respectively, which makes it difficult to field a team of engineers well-equipped for developing an optimized erasure-coding library.

While the current development process of optimized erasure-coding libraries leaves much to be desired, we argue that it will be even more challenging in the future.

Hardware is becoming increasingly heterogeneous, with GPUs, FPGAs, and ASICs complementing, and at times replacing, CPUs. At the same time, accelerators are gaining better access to storage and network devices [260]. For example, NVIDIA GPUDirect enables GPUs to access storage and network without transferring data to the host CPU [34].

Alongside, and perhaps driving, this increase in accelerators is a growth in "accelerator-native" applications: applications that run primarily on an accelerator, rather than primarily on a host CPU. A popular accelerator-native application is ML training, in which most application logic and state is kept on accelerators, such as GPUs. Similarly, many scientific simulations are increasingly run atop accelerators (e.g., [318]).

The growth of accelerator-native applications also indicates that *much of the data that needs to be erasure coded in future systems will be generated on accelerators*. Consider ML training as an example. Training ML models is a resource- and time-intensive process that often requires hundreds of GPUs [236]. Due to the scale at which training takes place, it is common for nodes to fail [221]. Thus, a common practice to ensure fault tolerance in ML training is to periodically checkpoint the state of training [129, 221]. High-performance checkpointing libraries often leverage in-memory erasure coding across multiple nodes to reduce the time-overhead of writing

[2]While Intel ISA-L accelerates other storage functions, its implementation of erasure coding (which contains ∼33000 lines of code) is largely custom.

checkpoints to stable storage [68, 230, 239]. Additionally, Chapter 5 showed that erasure coding can potentially overcome overheads in checkpointing for ML systems. It would be ideal for such applications to be able to perform erasure coding directly on the accelerator on top of which they run, rather than transferring data to the host CPU for erasure coding.

While the rise of accelerator-native applications and fast accelerator-storage datapaths calls for the ability to perform erasure coding on various hardware platforms, developing optimized erasure-coding libraries for this future exacerbates the challenges described above: a developer is now tasked with understanding the architectures of a variety of accelerators.

To usher in future erasure-coded systems in a developer-friendly way, a new approach to developing optimized erasure-coding libraries is needed.

10.4 Case for erasure coding via ML libraries

We now make the case that developing erasure-coding libraries using machine learning (ML) libraries offers the potential to overcome the challenges in developing optimized erasure-coding libraries. We summarize our case as follows:

1. ML libraries are heavily optimized to support new hardware features, as well as to support efficient execution on a variety of hardware platforms (§10.4.2). This alleviates the burden of understanding hardware details from users of ML libraries, and eases portability across platforms.

2. Erasure codes have a structure closely matching a key operation accelerated by ML libraries (§10.4.3). Thus, one can easily represent erasure codes using ML libraries, and doing so enables one to adopt existing optimizations in ML libraries and their support for hardware heterogeneity.

3. Implementing erasure codes via ML libraries may lose opportunities to apply optimizations specific to erasure codes, but improves performance in optimizations common to both ML libraries and erasure codes. This often results in a net improvement in performance (§10.4.4).

10.4.1 Background on ML libraries

We classify as "ML libraries" software packages that are used for carrying out the computations commonly performed by neural networks. There are many examples of ML libraries that sit at different levels of the software stack:

High-level frameworks, such as PyTorch [43] and TensorFlow [47], define operators needed to implement neural networks (e.g., matrix multiplications, convolutions) as well as routines needed to train neural networks (e.g., automatic differentiation). They typically expose APIs in a high-level language, such as Python, but call into lower-level libraries to achieve high performance among individual operators.

Low-level ML libraries provide high-performance implementations of individual operators (e.g., matrix multiplications) targeting a specific hardware platform. Examples of these include NVIDIA CUTLASS [31] and AMD MIOpen [6]. While low-level libraries are aimed at achieving high-performance, they are typically inconvenient to use on their own for ML training and inference because they lack support for common auxiliary methods needed in these settings, such as differentiation and data loading.

Finally, ML compilers translate individual ML operators, often referred to as "kernels," from high-level frameworks into high-performance implementations on various hardware platforms. ML compilers typically sit between high-level frameworks and low-level libraries, though they sometimes bypass low-level libraries entirely. Examples include Apache TVM [97] and XLA [49]. ML compilers typically translate code written in high-level libraries into an intermediate representation, perform transformations on this intermediate representation, and generate code targeting a hardware platform. It is common for ML compilers to leverage autotuning to search for high-performance implementations of a given kernel [97].

10.4.2 ML libraries satisfy many of the desiderata of erasure-coding libraries

Recall from §10.3 that a desirable property of an erasure-coding library is the ability to achieve high performance on a variety of hardware platforms without requiring expertise in each of these platforms, and to keep pace with hardware as it evolves.

ML libraries are good examples of software platforms structured toward achieving these goals. With the rise of accelerators targeting ML workloads, it has been recognized that it is untenable to develop custom optimizations for each platform. Thus, there has been an increasing focus on developing ML libraries that seamlessly achieve high performance on a variety of hardware platforms. For example, as described previously, ML compilers generate high-performance implementations of key ML operators and tune them for specific hardware backends. This enables one to achieve high performance without having expertise in computer architecture.

Furthermore, ML libraries are frequently updated to best exploit the latest hardware features. For example, CUTLASS is considered a go-to platform for learning how to best use new NVIDIA hardware. Thus, users of ML libraries can expect to continue to achieve high performance on new hardware.

10.4.3 Erasure coding via ML libraries?

The ability of ML libraries to achieve high performance across various hardware platforms and as hardware evolves draws attention to whether similar strategies could be used in the design of erasure-coding libraries. In this section, we move one step further: we question whether erasure codes can be developed directly by ML libraries themselves.

We next make the case for why it is indeed possible and potentially fruitful to forgo custom erasure-coding libraries and instead implement erasure codes via ML libraries.

ML libraries heavily optimize GEMM. General matrix multiplication (GEMM) is a fundamental operator in many ML libraries due to its wide use in neural networks (e.g., for fully-connected layers and some convolutional layers). A GEMM is defined as the multiplication of matrix A of size $M \times K$ by matrix B of size $K \times N$ to produce matrix C of size $M \times N$. Listing 1 shows unoptimized pseudocode for GEMM.

Decades of effort has been devoted to optimizing implementations of GEMMs. Such implementations build atop the naive version in Listing 1 by reordering loops, splitting loops into cache-friendly chunks, performing vectorized operations, and decomposing the problem for various types of parallel execution (e.g., threads on CPUs; threadblocks, warps, and threads on

```
for i in range(M):
    for j in range(N):
        for k in range(K):
            C[i, j] += (A[i, k] * B[k, j])
```

Listing 1: (Unoptimized) GEMM

```
for i in range(ec_r * ec_w):
    for j in range(ec_d):
        for k in range(ec_k * ec_w):
            C[i, j] ^= (A[i, k] & B[k, j])
```

Listing 2: (Unoptimized) bitmatrix erasure code encoding

GPUs). A significant amount of work has gone into making best use of the memory hierarchy (e.g., [31, 140, 288]). These optimizations have led to GEMMs being so well optimized that they are often used as benchmarks for determining the peak achievable performance of a hardware platform [167].

Similarity between erasure codes and GEMMs. Recall from §10.2 that a bitmatrix erasure code is equivalent to a GEMM, except with a bitwise XOR replacing summation and bitwise AND replacing multiplication. This is shown in Listing 2. Comparing Listings 1 and 2 illustrates the similarity between GEMM and bitmatrix erasure coding. The entire nested looping structure of a bitmatrix erasure code matches that of a GEMM. The only difference comes in the innermost operation: whereas a GEMM performs a multiply-accumulate operation, bitmatrix erasure codes perform an AND-XOR.

Due to the similarity between GEMM and bitmatrix erasure coding, a bitmatrix erasure code implemented via an ML library could potentially *automatically exploit the many optimizations performed in these libraries for GEMM*. In particular, we note that most of the optimizations performed in ML libraries for GEMMs target the portions of the code in Listings 1 and 2 that are identical: the nested looping structure. Examples of such optimizations include vectorization, loop reordering, and cache blocking. Thus, *erasure codes implemented via an ML library could immediately adopt many of the optimizations in ML libraries.*

The similarities between Listings 1 and 2 lead to another important conclusion: *bitmatrix erasure codes could be implemented via ML libraries with few additional lines of code*. As described above, all that is needed to support a bitmatrix erasure code in an ML library is changing the primitive operations performed on matrix elements from multiply-accumulate to AND-XOR. By studying ML libraries, we find that the portions of these codebases defining such primitive operations make up a minor fraction of the overall codebase; the vast majority of the code in these libraries for GEMM is devoted to optimizing the nested looping structure. We illustrate this further through our prototype implementations in §10.5, which require few lines of code that do not interfere with the main functionality of the ML library in question.

Why not HPC libraries? It is natural to question why we choose to implement erasure codes via ML libraries, rather than via traditional high-performance computing (HPC) libraries—after all, GEMM has been heavily optimized within HPC libraries for decades [67]. However, ML libraries provide multiple benefits over HPC libraries for this purpose:

First, because ML is driving many advancements in accelerators, ML libraries are better suited for exploiting current and future hardware platforms. In contrast, HPC libraries have often been slower to adopt new ML-centric hardware features: it took multiple years after the release of NVIDIA's Tensor Cores for HPC libraries to begin using them. Implementing erasure codes via ML libraries allows erasure codes to quickly achieve high performance on current and future architectures.

Additionally, whereas HPC libraries are typically written in low-level languages, such as C and Fortran, many ML libraries enable users to achieve high performance via high-level languages, such as Python. This reduces developer effort.

Finally, HPC libraries have traditionally been optimized for large, mostly-square GEMMs [270, 298], whereas erasure codes involve GEMM-like operations between a small generator matrix and a "short-and-wide" data matrix. In contrast, such irregularly-sized GEMMs are common in ML libraries: this pattern arises in a so-called fully-connected layer of a neural network for which the layer's weights are much smaller than the input/output data. Thus, ML libraries are a better fit for the types of matrix-like operations performed in erasure coding.

10.4.4 Optimizations specific to erasure codes?

A potential downside of our proposal to implement erasure codes via ML libraries by treating them similar to GEMMs is that doing so forgoes opportunities to perform optimizations specific to erasure codes. Recall from §10.2.1 that many current optimized erasure-coding libraries do not treat erasure coding as matrix multiplication so as to employ optimizations specific to erasure coding. Since these optimizations do not apply to GEMM, they are not performed within ML libraries. Thus, our proposed technique cannot employ such optimizations.

However, we find that the benefit of adopting optimizations within ML libraries outweighs any performance lost by forgoing erasure-coding-specific optimizations. This is because the performance of erasure codes is largely determined by those optimizations which are performed in both ML libraries and custom erasure-coding libraries. For example, Zhou et al. [345] perform a comprehensive evaluation of techniques for optimizing bitmatrix erasure codes and find that that vectorization dominates other optimizations in terms of its improvement in encoding/decoding throughput. Since vectorization is a common optimization used for GEMMs in ML libraries, our proposed approach retains this important optimization. Section 10.6 empirically shows that the proposed approach often achieves higher throughput than custom erasure-coding libraries that employ erasure-coding-specific optimizations.

10.5 Implementation using ML libraries

We now describe our prototypes of erasure codes written via two different ML libraries: TVM [97] and CUTLASS [31].

Recall from §10.4.1 that there are multiple classes of ML libraries: high-level frameworks, low-level libraries, and ML compilers. We have chosen to focus our exploration on ML compilers (TVM) and low-level libraries (CUTLASS). We select these classes because these are typically the locations in which high-performance GEMMs are implemented; high-level frameworks, such as PyTorch and TensorFlow, typically call into low-level libraries or use ML compilers to make

```
A = te.placeholder((M, K), name="A")
B = te.placeholder((K, N), name="B")
k = te.reduce_axis((0, K), name="k")

# GEMM
te.compute((M, N),
    lambda i,j: sum(A[i,k] * B[k,j], axis=k))

# Bitmatrix erasure code
xor = te.comm_reducer(lambda i,j: i ^ j, name="xor")
te.compute((M, N),
    lambda i,j: xor(A[i,k] & B[k,j], axis=k))
```

Listing 3: Python code for generating a GEMM and a bitmatrix erasure code in TVM

use of such optimized kernels. We describe within each subsection below why we chose the specific ML library being described.

10.5.1 Erasure coding via Apache TVM

Apache TVM [97] is an open-source compilation framework for optimizing neural networks on a variety of hardware platforms. While TVM also performs cross-kernel optimizations, we focus on its optimization of a single GEMM-like kernel.

TVM takes as input a description of a kernel written in Python using a high-level API called "Tensor Expressions" (represented as `te` in code). Given this specification, TVM performs various autotuning steps: it (1) performs a semantics-preserving transformation on the input expression (e.g., loop unrolling, tiling), (2) generates and compiles code for the platform in question, and (3) measures the performance of the compiled code (e.g., latency). TVM then uses a search procedure (e.g., genetic algorithms) based on measurements obtained so far to determine the next transformation to perform. This process iteratively repeats to find optimized kernels.

The process of declaring a kernel and performing the search and generation procedures above is typically carried out via Python in TVM. The kernel generated by autotuning can be exported to a C++ module for later use. Thus, the erasure codes developed through TVM can be readily used by storage systems developed in lower-level languages such as C++.

Why TVM? We prototype in TVM for multiple reasons:

(1) As an ML compiler, TVM can generate high-performance kernels for multiple hardware platforms. While we focus our evaluation of the TVM-based prototype on CPUs, we consider the cross-platform nature of TVM promising for deploying erasure codes on various platforms.

(2) TVM provides an opportunity to evaluate the benefit of learning-based autotuning in erasure-coding libraries. Zhou et al. [345] showed the potential benefit of using a cost-function-driven (e.g., number of XORs) search to select which optimizations to employ in erasure coding. The search procedure used in autotuning similarly optimizes for a given cost function, but one which involves directly measuring performance on a target platform, rather than through indirect metrics. As we will show in §10.6.2, TVM's autotuning generates code containing complex system-level optimizations without the developer needing any knowledge of these features.

Implementing erasure codes via TVM. We now describe the implementation of TVM-

```
template <typename T, typename LayoutA,
            typename LayoutB, typename LayoutC>
struct Mma<cutlass::gemm::GemmShape<1, 1, 1>, 1,
             T, LayoutA, T, LayoutB,
             T, LayoutC, cutlass::arch::OpMultiplyAdd> {
  using Operand = cutlass::Array<T, 1>;

  void operator()(Operand &d, Operand &a,
                  Operand &b, Operand &c) {
      d[0] = (a[0] * b[0]) + c[0];
  }
};
```

Listing 4: C++ code defining the multiply-accumulate operation of a GEMM in CUTLASS

EC, which leverages TVM to perform bitmatrix erasure coding. To illustrate the simplicity of implementing a bitmatrix erasure code in TVM, Listing 3 compares the Tensor Expressions description for a GEMM in TVM, and that we have added for bitmatrix erasure coding.

In declaring a GEMM, one declares placeholder variables for matrix operands A and B and declares the axis over which a so-called reduction operation (i.e., an operation that reduces multiple elements into a single element, typically via summation) will take place (lines 1–3). One then defines the computation to be performed using a `te.compute` statement. In the case of GEMM, the statement in lines 6 and 7 states that an output of size $M \times N$ is to be generated in which the element at row i and column j is formed by taking the sum of the elementwise multiplication of row i of matrix A and column j of matrix B (each of which contain K elements).

Declaring a bitmatrix erasure code is similar. The only difference comes in in the `te.compute` statement: rather than performing a sum of pairwise multiplications, we wish to perform an XOR of pairwise ANDs. To implement this, one needs only to declare a new reduction operator that performs the XOR of all elements (line 10), and replace the summation performed in GEMM with this XOR, and the multiplication performed in GEMM with AND (lines 11 and 12). Using these declared statements, one can autotune the bitmatrix erasure code using the same process for autotuning a GEMM.

Overall, our TVM-based prototype *did not require any source-level changes to TVM*; all implementation used existing APIs in TVM, requiring only around 40 lines of code.

10.5.2 Erasure coding via NVIDIA CUTLASS

CUTLASS [31] is an open-source library developed by NVIDIA for high-performance ML operations on GPUs. CUTLASS leverages C++ templates to develop building blocks for computing GEMMs at various levels of the GPU hierarchy (e.g., threadblocks, warps). It composes these building blocks alongside techniques such as tiling and double buffering to make efficient use of the GPU memory hierarchy. Finally, CUTLASS leverages the newest hardware features in NVIDIA GPUs for efficient memory access, and achieves performance nearly equal to that of NVIDIA's closed-source ML libraries (e.g., cuDNN). CUTLASS is widely used by other ML libraries, such as within TVM itself, as well as in PyTorch Geometric [42]. CUTLASS has also seen wide adoption in industry, including at Microsoft [24] and Meta [2].

```
1  template <typename T, typename LayoutA,
2               typename LayoutB, typename LayoutC>
3  struct Mma<cutlass::gemm::GemmShape<1, 1, 1>, 1,
4                T, LayoutA, T, LayoutB,
5                T, LayoutC, cutlass::arch::OpAndXor> {
6    using Operand = cutlass::Array<T, 1>;
7    cutlass::bit_xor<T> xor_op;
8    cutlass::bit_and<T> and_op;
9
10   void operator()(Operand &d, Operand &a,
11                   Operand &b, Operand &c) {
12       d[0] = xor_op(and_op(a[0], b[0]), c[0]);
13   }
14 };
```

Listing 5: C++ code defining the AND-XOR operation of a bitmatrix erasure code in CUTLASS

```
using Gemm = cutlass::gemm::device::Gemm<
    /* List of GEMM template parameters */
    cutlass::arch::OpMultiplyAdd>;

using ErasureCode = cutlass::gemm::device::Gemm<
    /* List of GEMM template parameters */
    cutlass::arch::OpAndXor>;
```

Listing 6: GEMM and erasure code declarations in CUTLASS

Why CUTLASS? We prototype in CUTLASS both to provide an example of our approach via a low-level ML library, and also because it provides a platform with which to compare against prior GPU-based erasure-coding libraries. While TVM also could have been used to generate erasure codes targeting GPUs, implementing a prototype via CUTLASS illustrates the generality of our proposed approach beyond a single ML library. Furthermore, TVM is beginning to use CUTLASS under the hood to generate GPU kernels.

Implementing erasure codes via CUTLASS. We now describe the implementation of CUTLASS-EC. Similar to our experience with TVM, we also found implementing CUTLASS-EC to be straightforward, and to closely follow the existing code used for GEMM in CUTLASS.

Listing 4 shows the C++ code used in CUTLASS to support the primitive operation in the innermost loop of Listing 1 for GEMM. This listing defines a partial specialization for the `Mma` (matrix-multiply-accumulate) operation performed by each thread. The struct is specialized for the operation tag `OpMultiplyAdd` which indicates that the primitive operation performed in line 11 of the listing is a multiply-accumulate.

Listing 5 shows the corresponding partial specialization we implemented to support bitmatrix erasure coding. We add a new operation tag `OpAndXor` to mark that the primitive operation to be performed is an AND-XOR. This is defined in line 13 using existing AND and XOR operations in CUTLASS.

This thread-level AND-XOR operation can be immediately used to declare and use warp-level, threadblock-level, or kernel-level bitmatrix erasure codes. As shown in Listing 6, doing so simply involves uses `OpAndXor` in place of `OpMultiplyAdd` in the declarations for these components. This enables our implementation to immediately make use of existing optimizations within CUTLASS for GEMM.

Similar to TVM-EC, our implementation of CUTLASS-EC *does not involve changing existing functionality within CUTLASS*, and required only around 30 lines of code.

10.5.3 Supporting a variety of erasure codes

Our proposed approach supports any erasure code that can be converted to a bitmatrix via the process described in §10.2. This encompasses *all linear* codes, which is a broad class of codes. To the best of our knowledge, all erasure codes used in major storage systems are linear (e.g., Reed-Solomon codes [267], local reconstruction codes (LRCs) [159, 276], and Hitchhiker codes [263]). Due to their linearity property, the encoding/decoding processes for linear codes can be represented as a matrix multiplication, and, thus, can be converted to bitmatrix erasure coding. The proposed framework thus naturally supports a variety of codes.

For example, consider LRCs. LRCs are popular codes used in distributed storage systems to reduce the number of nodes accessed in recovering from single-node failures, while still protecting against multi-node failures. To do so, LRCs typically introduce l "local parities" atop a traditional MDS code, which are formed from encoding only a small subset of data units. For example, a (6, 2, 2) LRC has 6 data units (k), generates 2 "global" parity units (r) via an MDS code by encoding over all 6 of the data units, and generates 2 "local" parity units (l) each formed by encoding 3 of the data units (such that local parities are formed via mutually-exclusive sets of data units). If a single node fails, a local parity and the other 2 data units used in forming the local parity are sufficient for recovery. This requires accessing only 3 nodes, whereas one would need to access 6 nodes in the underlying MDS code.

Since LRCs are linear codes, they can be represented via a bitmatrix, and, thus, can be easily supported in our approach. To represent a (k, r, l) LRC, we create a generator matrix of size $w(r+l) \times kw$ of bits by concatenating lw additional rows to the $rw \times kw$ bitmatrix representing the MDS portion of the LRC. The added rows generate local parities.

10.5.4 Using ML libraries with end-to-end erasure-coding libraries

Erasure-coding libraries, such as Jerasure [253], often provide utilities for erasure coding beyond encoding/decoding (e.g., generating a bitmatrix). Users of these utilities can leverage our approach by simply replacing the encoding/decoding procedures in these libraries with calls to the ML library.

One aspect must be considered when using ML libraries within higher-level erasure-coded systems: ML libraries typically expect that operands to GEMM-like calculations be contiguous in memory, whereas higher-level systems may not guarantee this. For example, Jerasure represents the k data units to be encoded as k pointers to separate allocations in memory. We find that performing `memcpy` operations to reorganize these distinct pointers into a contiguous buffer adds considerable time overhead (up to 84% in our experiments).

In most cases, representing the data to be encoded/decoded as a contiguous allocation of memory is natural and can be done easily. For example, encoding is typically performed over fixed-size chunks of data that are smaller than the entire unit they are part of (e.g., 1 MB chunks). The encoder waits for k such chunks to be passed to it before encoding, and keeps each chunk in memory. This memory must be managed by the storage system itself, rather than by the process that passes the data in, to ensure that no chunks are deallocated before the k chunks are encoded. Thus, the encoder must copy data into its own allocation of memory. A system can easily allocate a contiguous region of memory sufficient for hosting k chunks, and copy incoming data chunks to different pointer offsets in this region. The contiguous region of memory can then be passed to the ML library once all k chunks have arrived.

Similarly, decoding requires gathering k data/parity units from k storage devices into a central location. This requires allocating memory on the central location to hold these units. A system can easily allocate this memory as a contiguous buffer and assign incoming data/parity units to different pointer offsets within this buffer. This strategy also easily supports cases in which the overall chunk of data to be reconstructed is large, as decoding is typically run on small buffers of the overall data, and repeated many times until the entire chunk of data is reconstructed. Using a single allocation in this manner performs the same number of copies from the storage/network device as a system that uses separate allocations, while retaining the ability to pass a contiguous buffer to an ML library. The same technique can also be used when encoding data that is already stored in the storage system.

10.6 Evaluation

We now evaluate our approach of implementing erasure codes using ML libraries. The highlights are as follows:

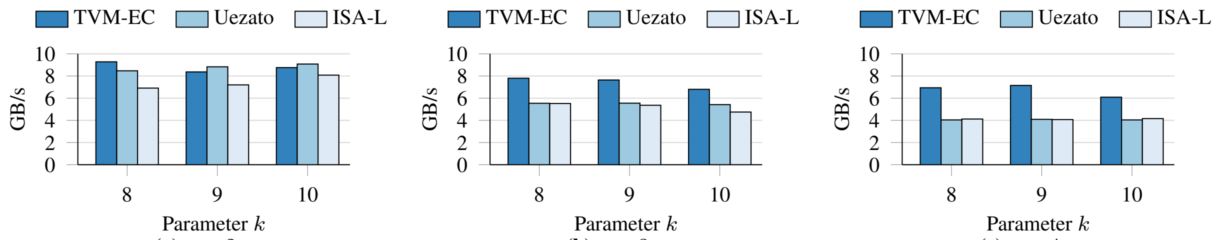
- TVM-EC is up to 1.75× faster than state-of-the-art erasure-coding libraries on CPUs for Reed Solomon codes, and up to 1.9× faster for local reconstruction codes.
- CUTLASS-EC is up to 2.2× faster than state-of-the-art erasure-coding libraries on GPUs.
- Our prototypes require only tens of lines of code to implement an erasure code, and TVM-EC in particular enables one to develop using languages that are typically considered more user-friendly (Python) than those used in current optimized erasure-coding libraries (e.g., C, Rust).
- TVM-EC automatically discovers complex optimizations that would otherwise need to be implemented by hand.

10.6.1 Evaluation setup

Hardware platforms evaluated. We evaluate on CPU and GPU hardware platforms. We focus the majority of our evaluation on CPUs because they represent the current standard platform for erasure coding. We also evaluate on GPU to show the applicability of our approach to a variety of hardware.

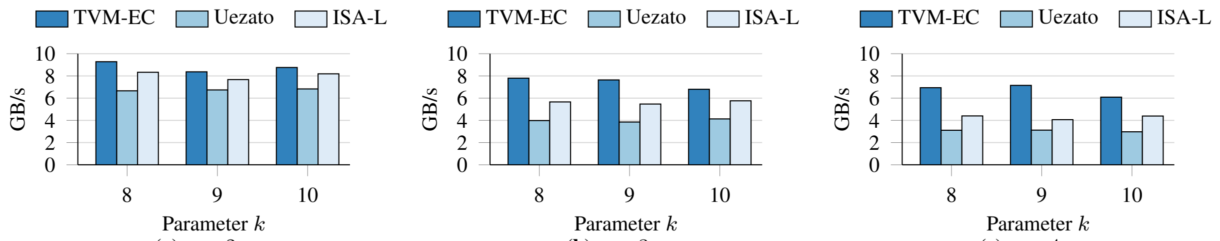
For CPU, we use an eight-core Intel Xeon D-1548 at 2.0 GHz with 64 GB of memory. For GPU, we use an NVIDIA A10 GPU, using a `g5.2xlarge` AWS instance, which uses the state-of-the-art Ampere microarchitecture.

Prototypes and baselines. We evaluate TVM-EC on the CPU platform, and compare it to two state-of-the-art custom erasure-coding libraries optimized for CPUs: (1) the work of



(a) $r = 2$ **(b)** $r = 3$ **(c)** $r = 4$

Figure 10.2: Encoding throughput in GB/s of TVM-EC and CPU-optimized erasure-coding libraries on the CPU platform



(a) $r = 2$ **(b)** $r = 3$ **(c)** $r = 4$

Figure 10.3: Decoding throughput in GB/s of TVM-EC and CPU-optimized erasure-coding libraries on the CPU platform

Uezato [302], a recent approach to optimizing erasure codes that leverages classic techniques from compiler theory alongside the optimizations described in §10.2.1. This work has been shown to outperform other optimized erasure-coding libraries (e.g., [345]), which themselves outperform popular libraries such as Jerasure [253]. (2) Intel's ISA-L library [19], which is a production-grade erasure-coding library optimized for CPUs. ISA-L does not convert an erasure code to a bitmatrix, and thus provides a competitive baseline showcasing the performance of erasure coding with higher finite field sizes. We note that the relative performance of Uezato's work and ISA-L is different from that reported by Uezato [302]. After discussing with the author, we believe that this can be attributed to a difference in evaluation platform: whereas we evaluate on a server-grade CPU, Uezato [302] evaluates on a MacBook.

We evaluate CUTLASS-EC on the GPU platform and compare it against G-CRS [210], a state-of-the-art approach to bitmatrix erasure coding on GPUs. G-CRS employs best-practices in using the GPU memory hierarchy along with new techniques to exploit parallelism on GPUs.

Metrics. The primary metric used in evaluating erasure-coding libraries is encoding throughput (in GB/s) (similarly, decoding throughput). This is calculated by dividing the size of the data matrix by the time it takes to encode. This is also the primary metric used in the works to which we compare. We follow the procedure for measuring throughput used by these prior works: throughput is measured with all data to be read residing in memory (host memory for CPU experiments, GPU memory for GPU experiments), and that the outputs of encoding/decoding are written to memory. This setup helps to isolate the performance of the erasure-coding library from other aspects of the storage system.

Erasure codes and parameters. We primarily evaluate with Reed Solomon codes, but also evaluate with local reconstruction codes (LRCs) [159, 276] to show the generality of our approach. We use the parameters k, r, and w that encompass those used in the prior works to which we compare. For experiments on CPU, we use k of 8–10, r of 2–4 with w fixed at 8, drawing

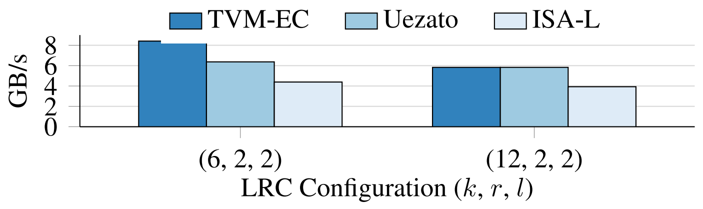


Figure 10.4: Encoding throughput of TVM-EC and CPU-optimized erasure-coding libraries on LRCs.

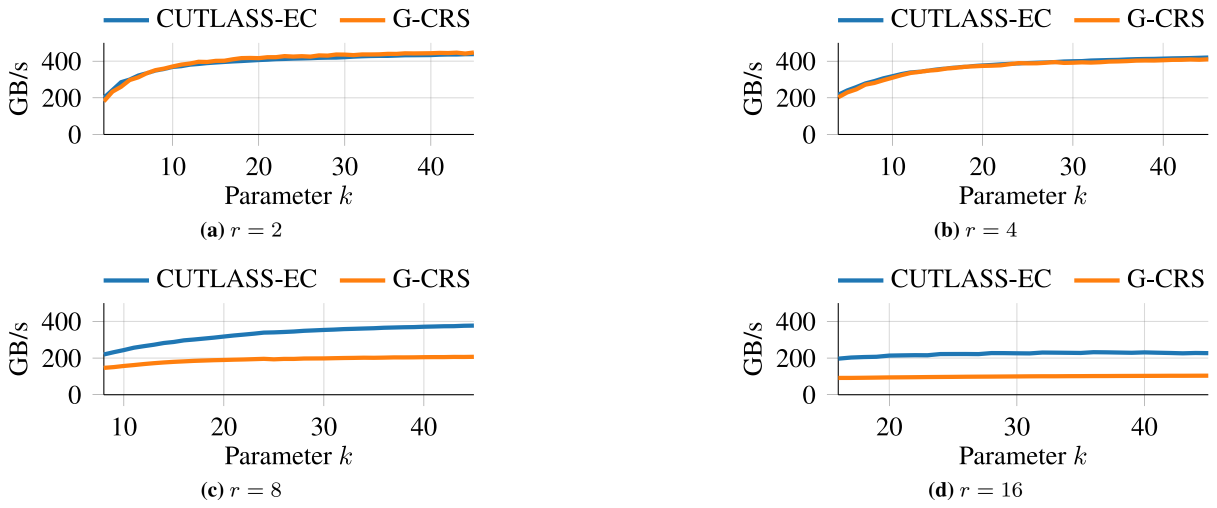


Figure 10.5: Encoding and decoding throughput of CUTLASS-EC and G-CRS [210] on an NVIDIA A10 GPU

these values from the work of Uezato [302]. Each data unit is 128 KB. For the work of Uezato, we evaluate various cache blocking factors, but typically find the performance using a blocking factor of 2 KB to provide the highest performance, and thus report results only for this factor. On GPU, we perform sweeps of the values of k and r, with the minimum value of w needed to support each case, drawing this from G-CRS [210]. For GPU experiments, each data unit is 4 MB. These larger data units than the CPU experiments are used to better saturate GPU memory bandwidth. We use the default Reed Solomon encoding/decoding matrices for a given setting of k and r available drawn from ISA-L.

Measurement setup. Our prototypes use autotuning provided by TVM/CUTLASS to find a high-performing implementation for a particular set of erasure-coding parameters k, r, and w (and l for LRCs).

TVM-EC uses TVM's learning-based Autoscheduler [344]. This is standard within TVM for achieving high performance among any kernel. TVM-EC tunes for 20000 trials, and uses the best configuration found in final evaluation. We report the mean of 1000 executions of encoding/decoding.

CUTLASS-EC uses the `cutlass_profiler` utility to select the fastest amongst various configurations for a given kernel running on the GPU. This is a standard technique used for kernels in CUTLASS. The profiler runs each configuration for a number of iterations and selects the fastest among them. We report the average of 100 runs of the selected kernel, after ten warmup runs, as is standard in the `cutlass_profiler`.

10.6.2 Results

Figure 10.2 compares the encoding throughput of TVM-EC with Uezato's erasure-coding library [302] and ISA-L [19] for the values of k and r used by Uezato, running on the CPU platform. TVM-EC achieves similar or higher throughput than these custom erasure-coding libraries for all values of k and r considered. In particular, TVM-EC achieves up to $1.75\times$ higher throughput than these custom-built libraries.

Effect of parameter r. All erasure-coding libraries in Figure 10.2 achieve lower encoding throughput with higher values of parameter r (i.e., when encoding more parities). Holding k constant, a higher value of r increases the computational intensity of the erasure code. Thus, it takes longer to encode a given amount of data, which results in lower encoding throughput.

We also find that TVM-EC attains its most significant speedups over the baselines with higher values of r: whereas the performance of the libraries is closer for $r = 2$, TVM-EC is up to $1.4\times$ and $1.75\times$ faster for parameter r of 3 and 4, respectively. We attribute this improved performance with a higher value of r to the significant optimization of ML libraries for more-computationally-intense GEMMs. With a higher value of parameter r, and thus higher intensity, TVM-EC is able to leverage these optimizations to achieve higher performance improvement over custom erasure-coding libraries.

Decoding performance. Figure 10.3 compares the decoding performance of TVM-EC and the CPU baselines: TVM-EC modestly improves performance over the baselines with r of 2, and significantly outperforms the baselines with higher values of r. Recall from §10.2 that, similar to encoding, decoding can be represented by the multiplication of an $r \times k$ "decoding matrix" by an $k \times d$ matrix consisting of available data/parity units. Because TVM-EC performs encoding and decoding directly as GEMM-like operations, encoding and decoding throughput in TVM-EC are identical. In contrast, because many custom erasure-coding libraries depend on the number and position of ones within the encoding/decoding matrix, the performance of ISA-L and the work of Uezato varies.

Local reconstruction codes (LRCs). We also evaluate TVM-EC and the CPU-optimized baselines on LRCs from Huang et al. [159] said to be employed in Azure Storage. These codes have configuration (k, r, l) of (6, 2, 2) and (12, 2, 2) (see §10.5.3 for a description of these parameters). We implement LRCs in TVM-EC using the process described in §10.5.3, and follow the suggestions for representing LRCs from the ISA-L GitHub [18] for ISA-L and the work of Uezato.

Figure 10.4 compares the encoding throughput of TVM-EC and the baselines on these LRCs. TVM-EC significantly increases encoding throughput in this setting compared to ISA-L: TVM-EC is 91% faster than ISA-L for the (6, 2, 2) LRC and 48% faster for the (12, 2, 2) LRC. Compared to the work of Uezato, TVM-EC is 30% faster for the (6, 2, 2) LRC and achieves equal encoding throughput for the (12, 2, 2) LRC. These results show the generality of leveraging ML libraries to achieve high performance for a broad class of erasure codes.

Investigating autotuning optimizations. Recall from §10.5.1 that TVM uses a learning-based approach to optimize kernels. To further illustrate the reduction of developer effort enabled by our approach, we analyze the intermediate representation generated by TVM in autotuning a bitmatrix erasure code with $k = 10$ and $r = 4$. Listing 7 shows a part of this intermediate representation performing a parallel layout transformation on the input operands. This is an optional

```
for (ax0.ax1.fused.ax2.fused int32, 0, 400) parallel
 for (ax4 int32, 0, 20)
  for (ax5 int32, 0, 5)
   for (ax6 int32, 0, 4)
    for (ax7 int32, 0, 64)
     auto_scheduler_layout_transform[
      (((((ax0.ax1.fused.ax2.fused*25600)
      + (ax4*1280)) + (ax5*256)) + (ax6*64)) + ax7)]
        = (uint8)B_2[(((((ax4*512000) + (ax6*128000))
            + (ax0.ax1.fused.ax2.fused*320))
            + (ax5*64)) + ax7)]
```

Listing 7: TVM intermediate representation showing parallel layout transformation optimization performed by autotuning

step designed to enable sequential memory accesses in the loops that perform encoding/decoding. It is clear from the listing that there are many design decisions at play: the number and extent of loops, their order, which loops to parallelize, and whether to perform this optimization at all. The complexity of the listing itself speaks to the challenge in performing this optimization by hand. This challenge is only exacerbated considering that this is only one of many optimizations performed by TVM-EC for the entire encoding kernel.

Such optimizations are entirely automated in TVM-EC: the *developer writes only Python code*, and TVM generates an optimized kernel for the target hardware. This frees the developer from understanding such optimizations, while still allowing the erasure-coding library to use them. Noting that the optimizations used in autotuning already existed in TVM reinforces the benefit of exploiting the similarities between operations in ML libraries and bitmatrix erasure coding.

While autotuning does require time and resources (around 6 hours in our experiments), it is important to note that this is a one-time, up-front cost: the configuration found via autotuning is simply loaded by the storage system on system initialization. This autotuning time is negligible compared to the lifetime of use of an erasure code (which, in many cases, is "always on" for production systems). Furthermore, because TVM-EC does not perform optimizations based on the actual values of the entries of the encoding/decoding matrices, *autotuning does not need to be repeated when different encoding/decoding matrices of the same size are used*. Autotuning only needs to be performed for every combination of k and r that will be used in a system, as these will affect the size of the matrices used. Most storage systems today use only a handful of values for these parameters and do not change them over time. Even recently-proposed disk-adaptive redundancy approaches that tune the parameters over the lifetime of storage devices use few values of k and r [174, 175, 176]. Thus, autotuning for each combination once is of negligible cost.

Finally, we expect that the time taken by autotuning is far less than the time required to discover and write such optimizations by hand, and that ongoing work in accelerating autotuning for ML kernels will reduce autotuning time [136].

Results on GPU. Figure 10.5 compares the encoding/decoding throughput of CUTLASS-EC to that of G-CRS [210] across various settings of k and r, chosen to match the evaluation of G-CRS.[3] For erasure codes with smaller values of r, CUTLASS-EC achieves roughly equal performance as the hand-optimized G-CRS library. The benefits of CUTLASS-EC primarily come to fore with parameter r of eight and sixteen, for which CUTLASS-EC outperforms G-CRS by up to 2.2×. Similar to TVM-EC, we also attribute the larger improvement of CUTLASS-EC with higher parameter r to the many optimizations that ML libraries such as CUTLASS contain for more computationally-intensive kernels.

10.7 Additional related work

Optimized erasure coding. Section 10.2 described work related to optimizing erasure-coding libraries. Our proposal differs from these works by adopting ML libraries to implement high-performance erasure codes with little development effort.

Significant effort has gone into optimizing other aspects of erasure-coded systems, such as reducing network bandwidth used in decoding [159, 263, 276], scheduling decoding operations to optimize network transmission [209], and efficiently changing the level of redundancy used in an erasure code [174, 175, 322]. These techniques complement our work, as each benefit from using high-performance erasure-coding libraries.

Performance portability. A growing body of work focuses on maintaining high performance regardless of the hardware platform being used (e.g., Kokkos [301] and RAJA [79]). However, implementing erasure codes within such platforms still requires a deep understanding of erasure codes, as well as some basis of understanding how to write high-performance code. In contrast, our approach of leveraging ML libraries to implement erasure-coding libraries requires little understanding of erasure codes or hardware due to the significant similarities between erasure codes and matrix multiplication.

Broader use of ML libraries. A small set of work uses ML libraries to accelerate applications that are not the targets of ML libraries [103, 111, 234]. For example, Hummingbird [234] casts traditional ML algorithms, such as decision trees, into dense tensor problems and uses deep learning libraries to accelerate them. These works are similar in nature to our proposal in that they exploit similarities between ML libraries and the algorithm in question. However, to the best of our knowledge, the work described in this chapter is the first to exploit the similarities between erasure codes and ML libraries.

10.8 Conclusion

Erasure codes are critical to many production storage systems, but developing high-performance erasure-coding libraries currently requires expertise in both the mathematical underpinnings of erasure codes as well as of computer architecture. This leaves the development of custom, optimized erasure-coding libraries to a select few individuals equipped with this unique skillset. This situation will only be exacerbated with the increasing trends of hardware heterogeneity.

[3]Both CUTLASS-EC and G-CRS perform nearly-identical operations for encoding and decoding, so encoding and decoding throughput are the same.

To ease the development of current and future optimized erasure-coding libraries, we make the case that erasure codes should be implemented using ML libraries. We show that there is significant similarity between erasure codes and operations common to ML libraries. This enables one to implement erasure codes in few lines of code via existing, well-optimized ML libraries, and thus immediately adopt the many optimizations within these libraries. We develop two prototypes of our proposed approach in different ML libraries, targeting CPUs and GPUs. Compared to custom, optimized erasure-coding libraries, our prototypes are up to 2.2$\times$ faster.

These results show the promise of using ML libraries to implement optimized erasure codes, and have the potential to usher in the next generation of erasure-coded systems.

Chapter 11

Concluding remarks and future directions

As neural networks continue to deliver value to a growing number of applications, the ML systems that deploy them will increase in scale, leverage more expensive and power-hungry specialized accelerators, and be subject to more-stringent latency and safety standards. Thus, current and future ML systems must strive to both operate reliably, despite running atop unreliable infrastructure, while ensuring that they use infrastructure efficiently.

This thesis has studied the interplay between practical coding-theoretic tools and ML systems to address these concerns in ML systems, as well as to improve coding-theoretic tools themselves. In doing so, we have leveraged a combination of insights from computer systems, machine learning, and coding theory.

First, we showed how properties of neural networks and the GPUs on which they execute can be leveraged to more efficiently apply traditional coding-theoretic tools to detect silent data corruptions during neural network inference. The resultant technique, arithmetic-intensity-guided coded computation, significantly reduces the execution-time overhead of detecting faults during inference. This enables safety-critical applications of neural networks to run with lower overhead in error-prone environments and also opens the door for performing "always-on" fault-detection in datacenters to monitor for malfunctioning hardware.

Second, we showed that erasure codes have the potential to significantly reduce the overhead of fault tolerance for distributed recommendation model training. We show that the unique characteristics of recommendation model training call for careful use of redundancy-based fault tolerance, but that doing so can reduce training-time overhead compared to checkpointing-based approaches. The results from this thrust are particularly promising, as they most significantly reduce training-time overhead for large recommendation models (which are expected to continue to grow in the future).

Third, we illustrated how co-designing coding-theoretic tools with ML systems can expand the reach of coding-theoretic tools themselves. In particular, we illustrated how the fundamental challenge of performing coded computation over non-linear functions can be overcome by leveraging machine learning. Our proposed approach, learning-based coded computation, encompasses a broad framework for applying machine learning to coded computation. We have specifically illustrated the effectiveness of learning encoders and decoders for coded computation, as well as learning a computation that operates over inputs encoded via traditional erasure codes. When integrated into a prediction serving system, the resultant approaches enable significant reductions in tail latency in the presence of transient slowdowns, and accurately reconstruct the predictions from slow servers.

Fourth, this thesis has illustrated how ideas inspired by coding theory can be used to improve the efficiency of ML systems even in the absence of reliability concerns. Specifically, we developed FoldedCNNs, a new approach to CNN design that increases the throughput and GPU utilization of specialized CNN inference beyond the limits of increased batch size. Motivated by insights obtained from other parts of this thesis, FoldedCNNs increase the arithmetic intensity of specialized CNNs through a combination of applying coding-theory-inspired transformations to a batch of input images, while increasing the size of each layer of the CNN. The insights from this work open new avenues for leveraging ideas inspired by coding theory beyond reliability purposes.

Finally, this thesis has explored how advancements in ML systems can improve the performance of erasure-coded systems themselves. We showed that similarities between computations performed in erasure codes and those commonly performed in ML libraries enables one to easily implement erasure codes using ML libraries. Implementing erasure codes via ML libraries enables the erasure-coded system to immediately benefit from the many optimizations already within ML libraries, including their ability to make best use of the latest hardware features and to target various hardware platforms.

Through these thrusts, this thesis demonstrates the promise of using coding-theoretic tools in ML systems and ideas from ML and ML systems in coding-theoretic tools to bring about the next generation of efficient and reliable systems.

To conclude, we highlight lessons learned in this thesis and directions for future research.

11.1 Lessons learned

We now briefly describe the lessons learned through the contributions made by this thesis.

Imbalances in system architecture and application properties call for diverse redundancy. This thesis has challenged the notion that a one-size-fits-all approach to reliability is sufficient for both modern computer systems and emerging applications, due to imbalances therein.

First, our work on intensity-guided coded computation (§4) identified opportunities to better tailor redundant execution for the imbalance in compute bandwidth and memory bandwidth on modern GPUs. We showed that, due to this imbalance, different layers of neural networks operate most efficiently under different forms of coded computation, with compute-bound layers preferring coded-computation schemes that add minimal computation, and memory-bandwidth-bound layers preferring coded-computation-schemes that minimize extra memory traffic.

Second, our work on ECRM (§5) leveraged imbalance in application-level characteristics to optimize redundancy in DLRM training. Specifically, we showed that embedding tables in DLRMs occupy the majority of a DLRM's memory capacity, but require little network bandwidth for updates, whereas neural network parameters in DLRMs occupy little memory capacity, but significant memory bandwidth for updates. We showed in §4 that this imbalance calls for erasure coding DLRM parameters, while keeping neural network parameters replicated.

We expect that both current and future systems may benefit from considering imbalances in both the resources available in a system and imbalances in resource usage by applications when determining the best approach to redundancy-based fault tolerance for a given setting.

Combining inputs to CNNs opens multiple opportunities. Multiple research thrusts within this thesis have involved some form of combining the inputs to a CNN. First, learning-based coded computation (§6– §8) leveraged different forms of combining inputs to a CNN to enable coded computation over a broader class of functions. Second, FoldedCNNs (§9) leveraged inputs combined in a style similar to that in learning-based coded computation to increase the throughput and GPU utilization of specialized CNN inference.

The reader with background in training CNNs may find this ability somewhat unsurprising: combinations of input images have been successfully used to improve the generalization of CNNs via data-augmentation techniques such as mixup [339]. However, to the best of our knowledge, this thesis contains the first work to leverage combined inputs for the purposes of coded computation and improved throughput of CNNs. It is interesting to consider broader applications of combining inputs to CNNs.

Forgoing custom optimizations for specific routines to cast such routines as ML-like operations can improve performance and ease development. Our experience with developing erasure-coding libraries by leveraging ML libraries in §10 illustrated that treating erasure codes as matrix-multiplication-like operations can significantly improve performance, even though doing so forgoes years of research in optimizations specific to erasure codes.

It is interesting to consider the extent to which other applications that are heavily domain specific, similar to erasure codes, may benefit from being cast in a form that enables them to be run by the given popular optimized library of the day. It may be the case that other applications will not reap the same benefits we have found in writing erasure codes via ML libraries: perhaps erasure codes occupy a unique "sweet spot" of containing enough similarity with ML operators such that ML libraries can provide significant benefit without needing to introduce custom operations within them to support erasure codes. Exploring the balance between using popular high-performance libraries for a non-target application and bespoke optimized libraries for that application will likely benefit future applications.

11.2 Future directions

To conclude, we highlight future opportunities at the interplay of ML systems and coding-theoretic tools.

11.2.1 Learning-based detection of silent data corruptions.

This thesis has investigated optimizations to traditional coding-theoretic tools for detecting silent data corruptions in neural network inference (§4), and investigated learning-based approaches for tolerating fail-stop failures and slowdowns (§6–§8). Future research may consider investigating learning-based approaches to detecting silent data corruptions in neural networks.

Leveraging learning-based approaches for the detection of silent data corruptions in neural network inference has a number of potential benefits: First, doing so may enable one to differentiate between silent data corruptions that are likely to result in misprediction and those that are not. In addition, learning-based approaches to detecting silent data corruptions have the potential to reduce the overhead of silent-data-corruption detection for neural networks due to the ability to focus on only misprediction-causing errors.

Multiple works have explored the use of learning-based approaches to detect silent data corruptions in neural networks [204, 277, 278]. Here, we consider ways in which the techniques proposed within this thesis could be used for learning-based detection of silent data corruptions.

Detecting silent data corruptions via learning-based coded computation. A natural follow-on to our work on learning-based coded computation is to consider ways in which a similar approach could be used to detect silent data corruptions. For example, consider how the framework surrounding parity models could be adapted to detect silent data corruptions: inference is performed on k separate images on separate instances of a model, and inference is performed over a "parity input" using a parity model. Upon receiving the predictions from all $k+1$ models, the decoder uses all images to attempt to determine whether a silent data corruption occurred in any one of the inferences. While this framework appears to easily support detecting silent data corruptions, additional research is needed. A particular challenge to be addressed is in being able to differentiate between mispredictions made by a model in the absence of silent data corruption and those that occurred due to silent data corruption.

Detecting silent data corruptions via FoldedCNNs. A second, and perhaps less obvious, potential avenue for learning-based detection of silent data corruptions stemming from this thesis is through the use of FoldedCNNs to detect silent data corruptions. Recall form §9 that a FoldedCNN performs inference over f *distinct* images concatenated along the channels dimension.

Consider what would take place if, at inference time, the FoldedCNN was instead fed as input f *copies of the same input image*, concatenated along the channels dimension. Because the FoldedCNN has been trained to make distinct inferences for each of the f images in a single input, it should similarly treat the f replicas of the same image as separate. Thus, it may be the case that a silent data corruption affecting one of the replicas may not affect the other replicas. Should this be the case, the FoldedCNN is essentially performing f-modular redundancy, which is a classic approach to fault tolerance.

Additional research is necessary to determine the feasibility of this approach. For example, recall from §9 that a FoldedCNN occasionally achieves lower accuracy than the original CNN. It would be interesting to determine to what extent this potential drop in accuracy may increase the susceptibility of a FoldedCNN to mispredicting in the event of a silent data corruption. If this were the case, one would need to carefully balance between the potential fault tolerance added by "f-modular redundancy" in a FoldedCNN and the potential increase in number of critical silent data corruptions experienced by a FoldedCNN.

11.2.2 Learning-based coded computation beyond neural networks

This thesis has illustrated the potential for learning-based approaches to expand the reach of coded computation beyond linear computations. Thus far, we have primarily evaluated learning-based coded computation in settings in which the non-linear computation in question is a neural network. However, the basic framework surrounding learning-based coded computation has the potential to be applicable for more-general nonlinear functions.

There are many domains which could benefit from the resource-efficient reliability of coded computation. More-traditional examples include applications such as stream processing or data-parallel analytics. Integrating learning-based coded computation into these applications would raise new challenges, as many of the computations performed in these applications are not strictly numerical (e.g., `SELECT`ing data in stream-processing systems). It is not immediately obvious how to map such computations to the framework of learning-based coded computation.

Additionally, distributed neural-network training could benefit from coded-computation schemes. Alleviating stragglers in training classical ML models was, in fact, one of the popular examples of using traditional coded-computation schemes over linear computations (e.g., [126, 296]). Using learning-based coded computation for distributed training could potentially enable one to perform coded computation over a neural network as a whole—without the associated overhead of splitting the neural network into its constituent linear and non-linear components, as done by prior work [126]. However, using learning-based coded computation for training raises interesting future challenges, such as how to keep the learned components of learning-based coded computation up to date as the neural network being trained evolves. To some extent, this could be abstracted to a coded-computation problem in which the function $\mathcal{F}$ to be protected evolves over time.

11.2.3 Arithmetic-intensity-guided fault tolerance beyond neural networks

Chapter 4 showed that trends in GPU hardware and in the design of neural networks call for taking an approach to fault tolerance that is driven by the arithmetic intensity of a computation in question as well as the compute-to-memory-bandwidth ratio of the target hardware.

While we have primarily explored arithmetic-intensity-guided fault tolerance for linear layers of neural networks, the same basic principles are likely to benefit other applications. This may be of particular interest with the growing trend of running non-deep-learning applications atop hardware that has been specialized for neural networks (e.g., [103, 111, 131, 146]). Furthermore, exploring the use of arithmetic-intensity-guided fault tolerance to ensure the correctness of erasure codes themselves in the presence of silent data corruption may be interesting, particularly when employing the strategies we propose in §10 to leverage ML libraries to implement erasure codes.

11.2.4 Arithmetic-intensity-guided optimizations for coding-theoretic tools

This thesis has explored multiple ways in which arithmetic intensity should guide the design of ML systems and approaches to fault tolerance. It may be fruitful to consider leveraging these insights to develop the next generation of efficient coding-theoretic tools.

Consider, for example, erasure codes. At present, many erasure codes have been designed and optimized with the goal of reducing the number of operations performed in encoding and decoding. However, as described in §4 and §9, and well known in other domains, reducing the number of operations performed is not always sufficient for reducing the execution time of a computation. Taking arithmetic intensity into account in the development of future erasure codes would be particularly interesting, given that many erasure codes are typically memory-bandwidth-bound to begin with. Leveraging data-movement-centric optimization techniques developed primarily for neural networks [162] may prove fruitful for designing the next generation of erasure codes for modern hardware.

11.2.5 Accelerating other aspects erasure-coded-systems via ML systems

Chapter 10 illustrated the promise of using ML libraries to implement erasure codes both to accelerate the computation of encoding and decoding functions as well as to ease development and optimization effort. However, performing encoding and decoding operations are only one part of erasure-coded systems. It is interesting to consider whether advancements in other aspects of ML systems could provide similar benefits to other aspects of erasure-coded systems.

Consider, for example, reconstructing lost data in an erasure-coded distributed storage system. As described in §5 and §10, reconstruction involves reading available data units over the network and performing decoding operations on this data. Among these components, transferring available data over the network is often a key bottleneck. This has led to the development of a number of algorithm- and system-level optimizations to reconstruction that optimize data transfer (e.g., [209, 263]).

Similar to reconstruction in an erasure-coded system, training neural networks in a distributed fashion also requires significant network communication (e.g., to perform a reduction over gradients computed by data-parallel workers). This has been the subject of a significant amount of research, such as developing new collective-communication techniques (e.g., [36, 104, 279, 307, 347]) and taking advantage of emerging programmable network hardware [192, 275]. These advancements could potentially also be used to accelerate the reconstruction process in an erasure code, due to similarities between the reduction operation performed in data-parallel neural network training and the XOR-based "reductions" performed in decoding. In particular using ML-based communication libraries for erasure coding could ease the transition of erasure-coded systems onto emerging hardware platforms, such as GPUs and programmable switches, for which these ML-based libraries have already been optimized.

Bibliography

[1] Air Traffic By The Numbers. https://www.faa.gov/air_traffic/by_the_numbers/. Last accessed 02 February 2022.

[2] AITemplate. https://github.com/facebookincubator/AITemplate. Last accessed 05 December 2022.

[3] Amazon Alexa. https://developer.amazon.com/alexa. Last accessed 01 September 2019.

[4] Amazon EC2 C5 Instances. https://aws.amazon.com/ec2/instance-types/c5/. Last accessed 01 September 2019.

[5] Amazon EC2 Spot Instances https://aws.amazon.com/ec2/spot/. Last accessed 24 January 2022.

[6] AMD MIOpen. https://github.com/ROCmSoftwarePlatform/MIOpen. Last accessed 15 September 2022.

[7] Azure Machine Learning Studio. https://azure.microsoft.com/en-us/services/machine-learning-studio/. Last accessed 01 September 2019.

[8] CANDLE: Exascale Deep Learning and Simulation Enabled Precision Medicine for Cancer. https://candle.cels.anl.gov/. Last accessed 23 August 2021.

[9] Ceph Erasure Code. https://docs.ceph.com/en/latest/rados/operations/erasure-code/. Last accessed 20 August 2022.

[10] Criteo Labs: Download Terabyte Click Logs https://labs.criteo.com/2013/12/download-terabyte-click-logs/. Last accessed 19 April 2022.

[11] CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Last accessed 23 August 2021.

[12] Edge TPU. https://cloud.google.com/edge-tpu. Last accessed 28 September 2021.

[13] Europa Clipper Mission. https://europa.nasa.gov/mission/about/. Last accessed 28 January 2022.

[14] Google Cloud AI. https://cloud.google.com/products/machine-learning/. Last accessed 01 September 2019.

[15] Google lens: real-time answers to questions about the world around you. https://bit.ly/2MHAOLq. Last accessed 01 September 2019.

[16] HDFS Erasure Coding. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html. Last accessed 20 August 2022.

[17] Intel Galois Field New Instructions (GFNI) Technology Guide. https://tinyurl.com/35w8pebx. Last accessed 12 December 2022.

[18] Intel Intelligent Storage Acceleration Library GitHub Question on LRCs https://github.com/intel/isa-l/issues/28. Last accessed 09 December 2022.

[19] Intel Intelligent Storage Acceleration Library. https://www.intel.com/content/www/us/en/developer/tools/isa-l/overview.html. Last accessed 17 September 2022.

[20] Introducing NVIDIA Merlin HugeCTR: A Training Framework Dedicated to Recommender Systems. https://tinyurl.com/yy82pd2l. Last accessed 19 April 2022.

[21] iOS Siri. https://www.apple.com/ios/siri/. Last accessed 01 September 2019.

[22] ISO-26262 Road vehicles – Functional safety. https://www.iso.org/standard/68383.html. Last accessed 23 August 2021.

[23] Machine Learning on AWS. https://aws.amazon.com/machine-learning/. Last accessed 01 September 2019.

[24] Microsoft Translator enhanced with Z-code Mixture of Experts models. https://tinyurl.com/npnzk4t5. Last accessed 17 September 2022.

[25] MLPerf Inference Github Repository. https://github.com/mlperf/inference. Last accessed 19 April 2022.

[26] Model Server for Apache MXNet. https://github.com/awslabs/mxnet-model-server. Last accessed 01 September 2019.

[27] NVIDIA A100 GPU. https://www.nvidia.com/en-us/data-center/a100/. Last accessed 23 August 2021.

[28] NVIDIA CUB. https://nvlabs.github.io/cub/. Last accessed 23 August 2021.

[29] NVIDIA cuDNN. https://developer.nvidia.com/cudnn. Last accessed 23 August 2021.

[30] NVIDIA CUTLASS T4 clock frequency setting https://github.com/NVIDIA/cutlass/issues/154#issuecomment-745426099. Last accessed 23 August 2021.

[31] NVIDIA CUTLASS. https://github.com/NVIDIA/cutlass. Last accessed 23 August 2021.

[32] NVIDIA Deep Learning Performance Guide. https://docs.nvidia.com/deeplearning/performance/index.html. Last accessed 23 August 2021.

[33] NVIDIA DRIVE - Autonomous Vehicle Development Platforms. https://developer.nvidia.com/drive. Last accessed 23 August 2021.

[34] NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect. Last accessed 17 September 2022.

[35] NVIDIA Jetson. https://developer.nvidia.com/embedded/jetson-modules. Last accessed 23 August 2021.

[36] NVIDIA NCCL. https://developer.nvidia.com/nccl. Last accessed 21 September 2022.

[37] NVIDIA Parallel Thread Execution ISA Version 7.2: Matrix Fragments for mma.m16n8k8 https://docs.nvidia.com/cuda/parallel-thread-execution/index.

html#warp-level-matrix-fragment-mma-1688. Last accessed 23 August 2021.

[38] NVIDIA Tensor Cores https://www.nvidia.com/en-us/data-center/tensor-cores/. Last accessed 23 August 2021.

[39] NVIDIA TensorRT. https://developer.nvidia.com/tensorrt. Last accessed 23 August 2021.

[40] NVIDIA Tesla P4 GPU Datasheet. https://images.nvidia.com/content/pdf/tesla/184457-Tesla-P4-Datasheet-NV-Final-Letter-Web.pdf. Last accessed 23 August 2021.

[41] OpenCV. https://opencv.org/. Last accessed 01 September 2019.

[42] PyTorch. https://pytorch-geometric.readthedocs.io/en/latest/. Last accessed 15 September 2022.

[43] PyTorch. https://pytorch.org/. Last accessed 01 September 2019.

[44] Qualcomm Neural Processing SDK for AI. https://developer.qualcomm.com/software/qualcomm-neural-processing-sdk. Last accessed 28 September 2021.

[45] Speculative Execution in Hadoop MapReduce. https://data-flair.training/blogs/speculative-execution-in-hadoop-mapreduce/. Last accessed 01 September 2019.

[46] Tensorflow Checkpointing. https://tinyurl.com/2vsc779t. Last accessed 17 September 2022.

[47] TensorFlow. https://www.tensorflow.org/. Last accessed 15 September 2022.

[48] Torchvision Models. https://pytorch.org/vision/stable/models.html. Last accessed 23 August 2021.

[49] XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla. Last accessed 15 September 2022.

[50] Asirra: A CAPTCHA That Exploits Interest-aligned Manual Image Categorization. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 07)* (2007).

[51] CUDA Warps and Occupancy. https://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_WarpsAndOccupancy.pdf, 2011. Last accessed 23 August 2021.

[52] Single event effects - a comparison of configuration upsets and data upsets. Tech. Rep. WP0203, Microsemi Corporation, 2015.

[53] Akamai Online Retail Performance Report: Milliseconds are Critical. https://www.ir.akamai.com/news-releases/news-release-details/akamai-online-retail-performance-report-milliseconds-are, 2017. Last accessed 25 April 2022.

[54] NVIDIA Tesla V100 GPU Architecture. Tech. Rep. WP-08608-001_v1.1, 2017.

[55] NVIDIA Turing GPU Architecture. Tech. Rep. WP-09183-001_v01, 2018.

[56] Developing CUDA Kernels to Push Tensor Cores to the Absolute Limit on NVIDIA A100. https://tinyurl.com/53ex6v8s, 2020. Last accessed 05 September 2022.

[57] Hewlett Packard Enterprise accelerates space exploration with first ever in-space commercial edge computing and artificial intelligence capabilities. https://tinyurl.com/544853we, 2021.

Last accessed 04 April 2023.

[58] Supplemental material for: Boosting the Throughput and Accelerator Utilization of Specialized CNN Inference Beyond Increasing Batch Size. `http://proceedings.mlr.press/v139/kosaian21a/kosaian21a-supp.pdf`, 2021.

[59] ABADI, M., AND ANDERSEN, D. G. Learning to protect communications with adversarial neural cryptography. *arXiv preprint arXiv:1610.06918* (2016).

[60] ACUN, B., MURPHY, M., WANG, X., NIE, J., WU, C.-J., AND HAZELWOOD, K. Understanding Training Efficiency of Deep Learning Recommendation Models at Scale. In *2021 IEEE International Symposium on High Performance Computer Architecture (HPCA 21)* (2021).

[61] AGARWAL, D., LONG, B., TRAUPMAN, J., XIN, D., AND ZHANG, L. LASER: A Scalable Response Prediction Platform for Online Advertising. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining (WSDM 14)* (2014).

[62] AGIAKATSIKAS, D., FOUTRIS, N., SARI, A., VLAGKOULIS, V., SOUVATZOGLOU, I., PSARAKIS, M., LUJÁN, M., KASTRIOTOU, M., AND CAZZANIGA, C. Evaluation of Xilinx Deep Learning Processing Unit under Neutron Irradiation. *arXiv preprint arXiv:2206.01981* (2022).

[63] ALEX KRIZHEVSKY AND VINOD NAIR AND GEOFFREY HINTON. The CIFAR-10 and CIFAR-100 Datasets. `https://www.cs.toronto.edu/~kriz/cifar.html`.

[64] ALIPOURFARD, O., LIU, H. H., CHEN, J., VENKATARAMAN, S., YU, M., AND ZHANG, M. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (2017).

[65] ANANTHANARAYANAN, G., GHODSI, A., SHENKER, S., AND STOICA, I. Effective Straggler Mitigation: Attack of the Clones. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (2013).

[66] ANANTHANARAYANAN, G., KANDULA, S., GREENBERG, A. G., STOICA, I., LU, Y., SAHA, B., AND HARRIS, E. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)* (2010).

[67] ANDERSON, E., BAI, Z., BISCHOF, C., BLACKFORD, L. S., DEMMEL, J., DONGARRA, J., DU CROZ, J., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A., ET AL. *LAPACK Users' Guide*. SIAM, 1999.

[68] ANTHONY, Q., AND DAI, D. Evaluating Multi-Level Checkpointing for Distributed Deep Neural Network Training. In *2021 SC Workshops Supplementary Proceedings (SCWS 21)* (2021).

[69] AOUDIA, F. A., AND HOYDIS, J. End-to-End Learning of Communications Systems Without a Channel Model. *arXiv preprint arXiv:1804.02276* (2018).

[70] ARORA, M., MANNE, S., PAUL, I., JAYASENA, N., AND TULLSEN, D. M. Understanding Idle Behavior and Power Gating Mechanisms in the Context of Modern Benchmarks on CPU-GPU Integrated Systems. In *2015 IEEE 21st international symposium on high performance computer architecture (HPCA 15)* (2015).

[71] AWS OUTPOSTS. `https://aws.amazon.com/outposts/`. Last accessed 08 June 2021.

[72] AZURE STACK EDGE. `https://azure.microsoft.com/en-us/products/azure-stack/edge/`. Last accessed 08 June 2021.

[73] BA, J., AND CARUANA, R. Do Deep Nets Really Need to be Deep? In *Advances in Neural Information Processing Systems (NIPS 14)* (2014).

[74] BAEK, I., CHEN, W., ZHU, Z., SAMII, S., AND RAJKUMAR, R. FT-DeepNets: Fault-Tolerant Convolutional Neural Networks With Kernel-Based Duplication. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV 22)* (2022).

[75] BARROSO, L., MARTY, M., PATTERSON, D., AND RANGANATHAN, P. Attack of the Killer Microseconds. *Communications of the ACM 60*, 4 (2017), 48–54.

[76] BARROSO, L. A., CLIDARAS, J., AND HÖLZLE, U. The Datacenter as a Computer: An Introduction to the Design of Warehouse-scale Machines. *Synthesis Lectures on Computer Architecture 8*, 3 (2013), 1–154.

[77] BARTLETT, W., AND SPAINHOWER, L. Commercial Fault Tolerance: A Tale of Two Systems. *IEEE Transactions on Dependable and Secure Computing 1*, 1 (2004), 87–96.

[78] BAYLOR, D., BRECK, E., CHENG, H.-T., FIEDEL, N., FOO, C. Y., HAQUE, Z., HAYKAL, S., ISPIR, M., JAIN, V., KOC, L., ET AL. TFX: A Tensorflow-Based Production-Scale Machine Learning Platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 17)* (2017).

[79] BECKINGSALE, D. A., BURMARK, J., HORNUNG, R., JONES, H., KILLIAN, W., KUNEN, A. J., PEARCE, O., ROBINSON, P., RYUJIN, B. S., AND SCOGLAND, T. R. RAJA: Portable Performance for Large-Scale Scientific Applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)* (2019).

[80] BENGIO, Y., LOURADOUR, J., COLLOBERT, R., AND WESTON, J. Curriculum Learning. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML 09)* (2009).

[81] BENTOUTOU, Y. A Real Time EDAC System for Applications Onboard Earth Observation Small Satellites. *IEEE Transactions on Aerospace and Electronic Systems 48*, 1 (2012), 648–657.

[82] BHARDWAJ, R., XIA, Z., ANANTHANARAYANAN, G., JIANG, J., KARIANAKIS, N., SHU, Y., HSIEH, K., BAHL, V., AND STOICA, I. Ekya: Continuous Learning of Video Analytics Models on Edge Compute Servers. *arXiv preprint arXiv:2012.10557* (2020).

[83] BLALOCK, D., ORTIZ, J. J. G., FRANKLE, J., AND GUTTAG, J. What is the State of Neural Network Pruning? In *The Third Conference on Systems and Machine Learning (MLSys 20)* (2020).

[84] BLAUM, M., AND ROTH, R. M. On Lowest Density MDS Codes. *IEEE Transactions on Information Theory 45*, 1 (1999), 46–59.

[85] BLOEMER, J., KALFANE, M., KARP, R., KARPINSKI, M., LUBY, M., AND ZUCKERMAN, D. An XOR-Based Erasure-Resilient Coding Scheme. Tech. Rep. TR-95-048, University of California, Berkeley, 1995.

[86] BOSILCA, G., DELMAS, R., DONGARRA, J., AND LANGOU, J. Algorithm-Based Fault Tolerance Applied to High Performance Computing. *Journal of Parallel and Distributed Computing 69*, 4 (2009), 410–416.

[87] BRAUN, C., HALDER, S., AND WUNDERLICH, H. J. A-ABFT: Autonomous Algorithm-Based Fault Tolerance for Matrix Multiplications on Graphics Processing Units. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 14)* (2014).

[88] BRONEVETSKY, G., AND DE SUPINSKI, B. Soft Error Vulnerability of Iterative Linear Algebra Methods. In *Proceedings of the 22nd Annual International Conference on Supercomputing (ICS 08)* (2008).

[89] BRUTLAG, J. Speed Matters for Google Web Search, 2009.

[90] CAI, H., GAN, C., WANG, T., ZHANG, Z., AND HAN, S. Once for All: Train One Network and Specialize it for Efficient Deployment. In *International Conference on Learning Representations (ICLR 20)* (2020).

[91] CAI, H., ZHU, L., AND HAN, S. ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. In *Proceedings of the 6th International Conference on Learning Representations (ICLR 18)* (2018).

[92] CAMPBELL, A., MCDONALD, P., AND RAY, K. Single Event Upset Rates in Space. *IEEE Transactions on Nuclear Science 39*, 6 (1992), 1828–1835.

[93] CANEL, C., KIM, T., ZHOU, G., LI, C., LIM, H., ANDERSEN, D. G., KAMINSKY, M., AND DULLOOR, S. R. Scaling Video Analytics on Constrained Edge Nodes. In *Proceedings of the 2nd Conference on Systems and Machine Learning (SysML 19)* (2019).

[94] CHANG, C.-K., YIN, W., AND EREZ, M. Assessing the Impact of Timing Errors on HPC Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 19)* (2019).

[95] CHEN, G., ZHOU, H., SHEN, X., GAHM, J., VENKAT, N., BOOTH, S., AND MARSHALL, J. OpenCL-Based Erasure Coding on Heterogeneous Architectures. In *2016 IEEE 27th International Conference on Application-Specific Systems, Architectures and Processors (ASAP 16)* (2016).

[96] CHEN, J., LIANG, X., AND CHEN, Z. Online Algorithm-Based Fault Tolerance for Cholesky Decomposition on Heterogeneous Systems with GPUs. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS 16)* (2016).

[97] CHEN, T., MOREAU, T., JIANG, Z., ZHENG, L., YAN, E., SHEN, H., COWAN, M., WANG, L., HU, Y., CEZE, L., GUESTRIN, C., AND KRISHNAMURTHY, A. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (2018).

[98] CHEN, Y., LIU, Z., REN, B., AND JIN, X. On Efficient Constructions of Checkpoints. In *Proceedings of the International Conference on Machine Learning (ICML 20)* (2020).

[99] CHEN, Z. Online-ABFT: An Online Algorithm Based Fault Tolerance Scheme for Soft Error Detection in Iterative Methods. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 13)* (2013).

[100] CHEN, Z., LI, G., AND PATTABIRAMAN, K. A Low-cost Fault Corrector for Deep Neural Networks through Range Restriction. *arXiv preprint arXiv:2003.13874v4* (2021).

[101] CHEN, Z., LI, G., PATTABIRAMAN, K., AND DEBARDELEBEN, N. BinFI: An Efficient Fault Injector for Safety-Critical Machine Learning Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 19)* (2019).

[102] CHEN, Z., NARAYANAN, N., FANG, B., LI, G., PATTABIRAMAN, K., AND DEBARDELEBEN, N. TensorFI: A Flexible Fault Injection Framework for TensorFlow Applications. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE 20)* (2020).

[103] CHERN, F., HECHTMAN, B., DAVIS, A., GUO, R., MAJNEMER, D., AND KUMAR, S. TPU-KNN: K Nearest Neighbor Search at Peak FLOP/s. *arXiv preprint arXiv:2206.14286* (2022).

[104] CHO, M., FINKLER, U., KUNG, D., AND HUNTER, H. BlueConnect: Decomposing All-Reduce for Deep Learning on Heterogeneous Network Hierarchy. *Proceedings of Machine Learning and Systems (MLSys 19)* (2019).

[105] CHUNG, E., FOWERS, J., OVTCHAROV, K., PAPAMICHAEL, M., CAULFIELD, A., MASSENGILL, T., LIU, M., LO, D., ALKALAY, S., HASELMAN, M., ET AL. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro 38*, 2 (2018), 8–20.

[106] COOPER, N. G. The Invisible Neutron Threat. *National Security Science* (February 2012), 12–16.

[107] COVINGTON, P., ADAMS, J., AND SARGIN, E. Deep Neural Networks for YouTube Recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems* (2016).

[108] CRANKSHAW, D., SELA, G.-E., ZUMAR, C., MO, X., GONZALEZ, J. E., STOICA, I., AND TUMANOV, A. InferLine: ML Inference Pipeline Composition Framework. *arXiv preprint arXiv:1812.01776* (2018).

[109] CRANKSHAW, D., WANG, X., ZHOU, G., FRANKLIN, M. J., GONZALEZ, J. E., AND STOICA, I. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (2017).

[110] CURRY, M. L., SKJELLUM, A., LEE WARD, H., AND BRIGHTWELL, R. Gibraltar: A Reed-Solomon Coding Library for Storage Applications on Programmable Graphics Processors. *Concurrency and Computation: Practice and Experience 23*, 18 (2011), 2477–2495.

[111] DAKKAK, A., LI, C., XIONG, J., GELADO, I., AND HWU, W.-M. Accelerating Reduction and Scan Using Tensor Core Units. In *Proceedings of the ACM International Conference on Supercomputing (ICS 19)* (2019).

[112] DALY, J. T. A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps. *Future Generation Computer Systems 22*, 3 (2006), 303–312.

[113] DEAN, J., AND BARROSO, L. A. The Tail at Scale. *Communications of the ACM 56*, 2 (2013), 74–80.

[114] DELL, T. J. A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory. *IBM Microelectronics division 11* (1997), 1–23.

[115] DENBY, B., AND LUCIA, B. Orbital Edge Computing: Nanosatellite Constellations as a New Class of Computer System. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 20)* (2020).

[116] DHAKAL, A., KULKARNI, S. G., AND RAMAKRISHNAN, K. GSLICE: Controlled Spatial Sharing of GPUs for a Scalable Inference Platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SOCC 20)* (2020).

[117] DI, S., GUO, H., PERSHEY, E., SNIR, M., AND CAPPELLO, F. Characterizing and Understanding HPC Job Failures Vver the 2k-day Life of IBM Bluegene/Q System. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 19)* (2019).

[118] DIEDERIK P. KINGMA AND JIMMY BA. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations (ICLR 15)* (2015).

[119] DIMITROV, M., MANTOR, M., AND ZHOU, H. Understanding Software Approaches for GPGPU Reliability. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 09)* (2009).

[120] DIXIT, H. D., BOYLE, L., VUNNAM, G., PENDHARKAR, S., BEADON, M., AND SANKAR, S. Detecting Silent Data Corruptions in the Wild. *arXiv preprint arXiv:2203.08989* (2022).

[121] DIXIT, H. D., PENDHARKAR, S., BEADON, M., MASON, C., CHAKRAVARTHY, T., MUTHIAH, B., AND SANKAR, S. Silent Data Corruptions at Scale. *arXiv preprint arXiv:2102.11245* (2021).

[122] DOS SANTOS, F. F., LUNARDI, C., OLIVEIRA, D., LIBANO, F., AND RECH, P. Reliability Evaluation of Mixed-Precision Architectures. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA 19)* (2019).

[123] DOS SANTOS, F. F., MALDE, S., CAZZANIGA, C., FROST, C., CARRO, L., AND RECH, P. Experimental Findings on the Sources of Detected Unrecoverable Errors in GPUs. *IEEE Transactions on Nuclear Science* (2022).

[124] DOS SANTOS, F. F., PIMENTA, P. F., LUNARDI, C., DRAGHETTI, L., CARRO, L., KAELI, D., AND RECH, P. Analyzing and Increasing the Reliability of Convolutional Neural Networks on GPUs. *IEEE Transactions on Reliability 68*, 2 (2018), 663–677.

[125] DUCHI, J., HAZAN, E., AND SINGER, Y. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research 12*, 7 (2011).

[126] DUTTA, S., BAI, Z., LOW, T. M., AND GROVER, P. CodeNet: Training Large Scale Neural Networks in Presence of Soft-Errors. *arXiv preprint arXiv:1903.01042* (2019).

[127] DUTTA, S., CADAMBE, V., AND GROVER, P. Short-dot: Computing Large Linear Transforms Distributedly Using Coded Short Dot Products. In *Advances In Neural Information Processing Systems (NIPS 16)* (2016).

[128] DUTTA, S., CADAMBE, V., AND GROVER, P. Coded Convolution for Parallel and Distributed Computing Within a Deadline. In *Proceedings of the 2017 IEEE International Symposium on Information Theory (ISIT 17)* (2017).

[129] EISENMAN, A., MATAM, K. K., INGRAM, S., MUDIGERE, D., KRISHNAMOORTHI, R., NAIR, K., SMELYANSKIY, M., AND ANNAVARAM, M. Check-N-Run: A Checkpointing System for Training Deep Learning Recommendation Models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022).

[130] EISENMAN, A., NAUMOV, M., GARDNER, D., SMELYANSKIY, M., PUPYREV, S., HAZELWOOD, K., CIDON, A., AND KATTI, S. Bandana: Using Non-Volatile Memory for Storing Deep Learning Models. In *The Second Conference on Systems and Machine Learning (SysML 19)* (2019).

[131] FENG, B., WANG, Y., CHEN, G., ZHANG, W., XIE, Y., AND DING, Y. EGEMM-TC: Accelerating Scientific Computing on Tensor Cores with Extended Precision. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 21)* (2021).

[132] FISHER YU AND VLADLEN KOLTUN. Multi-Scale Context Aggregation by Dilated Convolutions. In *International Conference on Learning Representations (ICLR 16)* (2016).

[133] GARDNER, K., ZBARSKY, S., DOROUDI, S., HARCHOL-BALTER, M., AND HYYTIA, E. Reducing Latency via Redundant Requests: Exact Analysis. *ACM SIGMETRICS Performance Evaluation Review 43*, 1 (2015), 347–360.

[134] GEIST, A. Supercomputing's Monster in the Closet. *IEEE Spectrum 53*, 3 (2016), 30–35.

[135] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP 03)* (2003).

[136] GIBSON, P., AND CANO, J. Transfer-Tuning: Reusing Auto-Schedules for Efficient Tensor Program Code Generation. In *The 31st International Conference on Parallel Architectures and Compilation Techniques (PACT 22)* (2022).

[137] GINART, A., NAUMOV, M., MUDIGERE, D., YANG, J., AND ZOU, J. Mixed Dimension Em-

beddings with Application to Memory-Efficient Recommendation Systems. In *2021 IEEE International Symposium on Information Theory (ISIT 21)* (2021).

[138] GLOROT, X., AND BENGIO, Y. Understanding the Difficulty of Training Deep Feedforward Neural Networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS 10)* (2010).

[139] GOBIESKI, G., LUCIA, B., AND BECKMANN, N. Intelligence Beyond the Edge: Inference on Intermittent Embedded Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 19)* (2019).

[140] GOTO, K., AND GEIJN, R. A. V. D. Anatomy of High-Performance Matrix Multiplication. *ACM Transactions on Mathematical Software (TOMS) 34*, 3 (2008), 1–25.

[141] GROSVENOR, M. P., SCHWARZKOPF, M., GOG, I., WATSON, R. N. M., MOORE, A. W., HAND, S., AND CROWCROFT, J. Queues Don't Matter When You Can JUMP Them! In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (2015).

[142] GUJARATI, A., ELNIKETY, S., HE, Y., MCKINLEY, K. S., AND BRANDENBURG, B. B. Swayam: Distributed Autoscaling to Meet SLAs of Machine Learning Inference Services with Resource Efficiency. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware 17)* (2017).

[143] GUJARATI, A., KARIMI, R., ALZAYAT, S., HAO, W., KAUFMANN, A., VIGFUSSON, Y., AND MACE, J. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (2020).

[144] GUPTA, U., HSIA, S., SARAPH, V., WANG, X., REAGEN, B., WEI, G.-Y., LEE, H.-H. S., BROOKS, D., AND WU, C.-J. DeepRecSys: A System for Optimizing End-to-End At-Scale Neural Recommendation Inference. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA 20)* (2020).

[145] GUPTA, U., WU, C.-J., WANG, X., NAUMOV, M., REAGEN, B., BROOKS, D., COTTEL, B., HAZELWOOD, K., HEMPSTEAD, M., JIA, B., ET AL. The Architectural Implications of Facebook's DNN-based Personalized Recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA 20)* (2020).

[146] HAIDAR, A., TOMOV, S., DONGARRA, J., AND HIGHAM, N. J. Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC 18)* (2018).

[147] HAO, M., LI, H., TONG, M. H., PAKHA, C., SUMINTO, R. O., STUARDO, C. A., CHIEN, A. A., AND GUNAWI, H. S. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP 17)* (2017).

[148] HARCHOL-BALTER, M. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013.

[149] HARI, S. K. S., SULLIVAN, M., TSAI, T., AND KECKLER, S. W. Making Convolutions Resilient via Algorithm-Based Error Detection Techniques. *IEEE Transactions on Dependable and Secure Computing* (2021).

[150] HARLAP, A., CUI, H., DAI, W., WEI, J., GANGER, G. R., GIBBONS, P. B., GIBSON, G. A., AND XING, E. P. Addressing the Straggler Problem for Iterative Convergent Parallel ML. In

Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC 16) (2016).

[151] HAUSWALD, J., KANG, Y., LAURENZANO, M. A., CHEN, Q., LI, C., MUDGE, T., DRESLINSKI, R. G., MARS, J., AND TANG, L. DjiNN and Tonic: DNN as a Service and Its Implications for Future Warehouse Scale Computers. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA 15)* (2015).

[152] HAZELWOOD, K., BIRD, S., BROOKS, D., CHINTALA, S., DIRIL, U., DZHULGAKOV, D., FAWZY, M., JIA, B., JIA, Y., KALRO, A., ET AL. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA 18)* (2018).

[153] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 16)* (2016).

[154] HO, Q., CIPAR, J., CUI, H., LEE, S., KIM, J. K., GIBBONS, P. B., GIBSON, G. A., GANGER, G., AND XING, E. P. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *Advances in Neural Information Processing Systems (NIPS 13)* (2013).

[155] HOCHSCHILD, P. H., TURNER, P., MOGUL, J. C., GOVINDARAJU, R., RANGANATHAN, P., CULLER, D. E., AND VAHDAT, A. Cores that don't count. In *Proceedings of the 18th Workshop on Hot Topics in Operating System (HotOS 21)* (2021).

[156] HSIEH, K., ANANTHANARAYANAN, G., BODIK, P., VENKATARAMAN, S., BAHL, P., PHILIPOSE, M., GIBBONS, P. B., AND MUTLU, O. Focus: Querying Large Video Datasets with Low Latency and Low Cost. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (2018).

[157] HU, H., DEY, D., BAGNELL, J. A., AND HEBERT, M. Learning Anytime Predictions in Neural Networks via Adaptive Loss Balancing. *arXiv preprint arXiv:1708.06832* (2018).

[158] HUANG, C., LI, J., AND CHEN, M. On Optimizing XOR-Based Codes for Fault-Tolerant Storage Applications. In *IEEE Information Theory Workshop (ITW 07)* (2007).

[159] HUANG, C., SIMITCI, H., XU, Y., OGUS, A., CALDER, B., GOPALAN, P., LI, J., AND YEKHANIN, S. Erasure Coding in Windows Azure Storage. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)* (2012).

[160] HUANG, K.-H., AND ABRAHAM, J. A. Algorithm-Based Fault Tolerance for Matrix Operations. *IEEE Transactions on Computers 100*, 6 (1984), 518–528.

[161] IORGULESCU, C., AZIMI, R., KWON, Y., ELNIKETY, S., SYAMALA, M., NARASAYYA, V., HERODOTOU, H., TOMITA, P., CHEN, A., ZHANG, J., AND WANG, J. PerfIso: Performance Isolation for Commercial Latency-Sensitive Services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018).

[162] IVANOV, A., DRYDEN, N., BEN-NUN, T., LI, S., AND HOEFLER, T. Data Movement is All You Need: A Case Study on Optimizing Transformers.

[163] JACOB, B., KLIGYS, S., CHEN, B., ZHU, M., TANG, M., HOWARD, A., ADAM, H., AND KALENICHENKO, D. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. *arXiv preprint arXiv:1712.05877* (2017).

[164] JAIN, P., MO, X., JAIN, A., SUBBARAJ, H., DURRANI, R. S., TUMANOV, A., GONZALEZ, J., AND STOICA, I. Dynamic Space-Time Scheduling for GPU Inference. In *NeurIPS Workshop on*

Systems for Machine Learning (2018).

[165] JEON, H., AND ANNAVARAM, M. Warped-DMR: Light-Weight Error Detection for GPGPU. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 12)* (2012).

[166] JEON, M., VENKATARAMAN, S., PHANISHAYEE, A., QIAN, J., XIAO, W., AND YANG, F. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (2019).

[167] JIA, Z., MAGGIONI, M., SMITH, J., AND SCARPAZZA, D. P. Dissecting the NVidia Turing T4 GPU via Microbenchmarking. *arXiv preprint arXiv:1903.07486* (2019).

[168] JIANG, A. H., WONG, D. L.-K., CANEL, C., TANG, L., MISRA, I., KAMINSKY, M., KOZUCH, M. A., PILLAI, P., ANDERSEN, D. G., AND GANGER, G. R. Mainstream: Dynamic Stem-Sharing for Multi-Tenant Video Processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018).

[169] JIANG, B., DENG, C., YI, H., HU, Z., ZHOU, G., ZHENG, Y., HUANG, S., GUO, X., WANG, D., SONG, Y., ET AL. XDL: An Industrial Deep Learning Framework for High-Dimensional Sparse Data. In *Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data* (2019).

[170] JIANG, W., HE, Z., ZHANG, S., PREUSSER, T. B., ZENG, K., FENG, L., ZHANG, J., LIU, T., LI, Y., ZHOU, J., ET AL. MicroRec: Accelerating Deep Recommendation Systems to Microseconds by Hardware and Data Structure Solutions. In *The Fourth Conference on Systems and Machine Learning (MLSys 21)* (2021).

[171] JOSHI, G., LIU, Y., AND SOLJANIN, E. On the Delay-Storage Trade-Off in Content Download From Coded Distributed Storage Systems. *IEEE JSAC*, 5 (2014), 989–997.

[172] JOUPPI, N. P., YOUNG, C., PATIL, N., PATTERSON, D., AGRAWAL, G., BAJWA, R., BATES, S., BHATIA, S., BODEN, N., BORCHERS, A., ET AL. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA 17)* (2017).

[173] JULIAN, K. D., KOCHENDERFER, M. J., AND OWEN, M. P. Deep Neural Network Compression for Aircraft Collision Avoidance Systems. *Journal of Guidance, Control, and Dynamics 42*, 3 (2019), 598–608.

[174] KADEKODI, S., MATURANA, F., ATHLUR, S., MERCHANT, A., RASHMI, K., AND GANGER, G. R. Tiger: Disk-Adaptive Redundancy Without Placement Restrictions. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (2022).

[175] KADEKODI, S., MATURANA, F., SUBRAMANYA, S. J., YANG, J., RASHMI, K., AND GANGER, G. R. PACEMAKER: Avoiding HeART Attacks in Storage Clusters with Disk-Adaptive Redundancy. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (2020).

[176] KADEKODI, S., RASHMI, K., AND GANGER, G. R. Cluster Storage Systems Gotta Have HeART: Improving Storage Efficiency by Exploiting Disk-Reliability Heterogeneity. In *17th USENIX Conference on File and Storage Technologies (FAST 19)* (2019).

[177] KALAMKAR, D., GEORGANAS, E., SRINIVASAN, S., CHEN, J., SHIRYAEV, M., AND HEINECKE, A. Optimizing Deep Learning Recommender Systems' Training On CPU Cluster Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 20)* (2020).

[178] Kang, D., Bailis, P., and Zaharia, M. BlazeIt: Optimizing Declarative Aggregation and Limit Queries for Neural Network-Based Video Analytics. *Proceedings of the VLDB Endowment 13*, 4 (2020).

[179] Kang, D., Emmons, J., Abuzaid, F., Bailis, P., and Zaharia, M. NoScope: Optimizing Neural Network Queries over Video at Scale. *Proceedings of the VLDB Endowment 10*, 11 (2017), 1586–1597.

[180] Karakus, C., Sun, Y., Diggavi, S., and Yin, W. Straggler Mitigation in Distributed Optimization Through Data Encoding. In *Advances in Neural Information Processing Systems (NIPS 17)* (2017).

[181] Karakus, C., Sun, Y., Diggavi, S., and Yin, W. Redundancy Techniques for Straggler Mitigation in Distributed Optimization and Learning. *arXiv preprint arXiv:1803.05397* (2018).

[182] Kim, H., Jiang, Y., Rana, R., Kannan, S., Oh, S., and Viswanath, P. Communication Algorithms via Deep Learning. In *International Conference on Learning Representations (ICLR 18)* (2018).

[183] Kim, H., Zeng, J., Liu, Q., Abdel-Majeed, M., Lee, J., and Jung, C. Compiler-Directed Soft Error Resilience for Lightweight GPU Register File Protection. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 20)* (2020).

[184] Koo, R., and Toueg, S. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on software Engineering*, 1 (1987), 23–31.

[185] Koppula, S., Orosa, L., Yağlikçi, A. G., Azizi, R., Shahroodi, T., Kanellopoulos, K., and Mutlu, O. EDEN: Enabling Energy-Efficient, High-Performance Deep Neural Network Inference Using Approximate DRAM. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 19)* (2019).

[186] Kosaian, J., Phanishayee, A., Philipose, M., Dey, D., and Vinayak, K. V. Boosting the Throughput and Accelerator Utilization of Specialized CNN Inference Beyond Increasing Batch Size. In *Proceedings of the 38th International Conference on Machine Learning (ICML 21)* (2021).

[187] Kosaian, J., and Rashmi, K. V. Arithmetic-Intensity-Guided Fault Tolerance for Neural Network Inference on GPUs. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC 21)* (2021).

[188] Kosaian, J., Rashmi, K. V., and Venkataraman, S. Parity Models: Erasure-Coded Resilience for Prediction Serving Systems. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 19)* (2019).

[189] Kosaian, J., Rashmi, K. V., and Venkataraman, S. Learning-Based Coded Computation. *IEEE Journal on Selected Areas in Information Theory* (2020).

[190] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems (NIPS 12)* (2012).

[191] LaBel, K. A. NASA and COTS Electronics: Past Approach and Successes–Future Considerations.

[192] Lao, C., Le, Y., Mahajan, K., Chen, Y., Wu, W., Akella, A., and Swift, M. ATP: In-network Aggregation for Multi-tenant Learning. In *18th USENIX Symposium on Networked*

Systems Design and Implementation (NSDI 21) (2021).

[193] LECUN, Y. The MNIST database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`.

[194] LECUN, Y., BOTTOU, L., BENGIO, Y., AND HAFFNER, P. Gradient-based Learning Applied to Document Recognition. *Proceedings of the IEEE 86*, 11 (1998), 2278–2324.

[195] LEE, K., LAM, M., PEDARSANI, R., PAPAILIOPOULOS, D., AND RAMCHANDRAN, K. Speeding Up Distributed Machine Learning Using Codes. *IEEE Transactions on Information Theory* (July 2018).

[196] LEE, Y., SCOLARI, A., CHUN, B.-G., SANTAMBROGIO, M. D., WEIMER, M., AND INTERLANDI, M. PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (2018).

[197] LEE, Y., SCOLARI, A., INTERLANDI, M., WEIMER, M., AND CHUN, B.-G. Towards High-Performance Prediction Serving Systems. *NIPS ML Systems Workshop* (2017).

[198] LENG, J., BUYUKTOSUNOGLU, A., BERTRAN, R., BOSE, P., AND REDDI, V. J. Safe Limits on Voltage Reduction Efficiency in GPUs: a Direct Measurement Approach. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 15)* (2015).

[199] LI, G., HARI, S. K. S., SULLIVAN, M., TSAI, T., PATTABIRAMAN, K., EMER, J., AND KECKLER, S. W. Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 17)* (2017).

[200] LI, G., PATTABIRAMAN, K., HARI, S. K. S., SULLIVAN, M., AND TSAI, T. Modeling Soft-Error Propagation in Programs. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 18)* (2018).

[201] LI, S., HUANG, J., TANG, P. T. P., KHUDIA, D., PARK, J., DIXIT, H. D., AND CHEN, Z. Efficient Soft-Error Detection for Low-Precision Deep Learning Recommendation Models. *arXiv preprint arXiv:2103.00130* (2021).

[202] LI, S., LI, H., LIANG, X., CHEN, J., GIEM, E., OUYANG, K., ZHAO, K., DI, S., CAPPELLO, F., AND CHEN, Z. FT-iSort: Efficient Fault Tolerance for Introsort. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 19)* (2019).

[203] LI, S., MADDAH-ALI, M. A., AND AVESTIMEHR, A. S. A Unified Coding Framework for Distributed Computing With Straggling Servers. In *2016 IEEE Globecom Workshops (GC Wkshps)* (2016).

[204] LI, Y., LI, M., LUO, B., TIAN, Y., AND XU, Q. DeepDyve: Dynamic Verification for Deep Neural Networks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS 20)* (2020).

[205] LI, Y., LIU, Y., LI, M., TIAN, Y., LUO, B., AND XU, Q. D2NN: A Fine-Grained Dual Modular Redundancy Framework for Deep Neural Networks. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC 19)* (2019).

[206] LI, Z. L., LIANG, C.-J. M., HE, W., ZHU, L., DAI, W., JIANG, J., AND SUN, G. Metis: Robustly Tuning Tail Latencies of Cloud Systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018).

[207] LIANG, G., AND KOZAT, U. C. FAST CLOUD: Pushing the Envelope on Delay Performance of

Cloud Storage with Coding. *arXiv:1301.1294* (2013).

[208] LIBANO, F., RECH, P., NEUMAN, B., LEAVITT, J., WIRTHLIN, M., AND BRUNHAVER, J. How Reduced Data Precision and Degree of Parallelism Impact the Reliability of Convolutional Neural Networks on FPGAs. *IEEE Transactions on Nuclear Science 68*, 5 (2021), 865–872.

[209] LIN, S., GONG, G., SHEN, Z., LEE, P. P., AND SHU, J. Boosting Full-Node Repair in Erasure-Coded Storage. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (2021).

[210] LIU, C., WANG, Q., CHU, X., AND LEUNG, Y.-W. G-CRS: GPU Accelerated Cauchy Reed-Solomon Coding. *IEEE Transactions on Parallel and Distributed Systems 29*, 7 (2018), 1484–1498.

[211] LIU, K., KOSAIAN, J., AND RASHMI, K. ECRM: Efficient Fault Tolerance for Recommendation Model Training via Erasure Coding. *arXiv preprint arXiv:2104.01981* (2021).

[212] LIU, Q., JUNG, C., LEE, D., AND TIWARI, D. Compiler-Directed Lightweight Checkpointing for Fine-Grained Guaranteed Soft Error Recovery. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 16)* (2016).

[213] LIU, Y., WANG, Y., YU, R., LI, M., SHARMA, V., AND WANG, Y. Optimizing CNN Model Inference on CPUs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (2019).

[214] LOTFI, A., HUKERIKAR, S., BALASUBRAMANIAN, K., RACUNAS, P., SAXENA, N., BRAMLEY, R., AND HUANG, Y. Resiliency of Automotive Object Detection Networks on GPU Architectures. In *2019 IEEE International Test Conference (ITC 19)* (2019).

[215] LU, D. J. Watchdog Processors and Structural Integrity Checking. *IEEE Transactions on Computers 31*, 07 (1982), 681–685.

[216] LUI, M., YETIM, Y., ÖZKAN, Ö., ZHAO, Z., TSAI, S.-Y., WU, C.-J., AND HEMPSTEAD, M. Understanding Capacity-Driven Scale-Out Neural Recommendation Inference.

[217] LUO, J., SHRESTHA, M., XU, L., AND PLANK, J. S. Efficient encoding schedules for XOR-based erasure codes. *IEEE Transactions on Computing* (May 2013).

[218] MA, N., ZHANG, X., ZHENG, H.-T., AND SUN, J. ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design. In *Proceedings of the 15th European Conference on Computer Vision (ECCV 18)* (2018).

[219] MACE, J., BODIK, P., MUSUVATHI, M., FONSECA, R., AND VARADARAJAN, K. 2DFQ: Two-Dimensional Fair Queuing for Multi-Tenant Cloud Services. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM 16)* (2016).

[220] MACWILLIAMS, F. J., AND SLOANE, N. J. A. *The Theory of Error Correcting Codes*, vol. 16. Elsevier, 1977.

[221] MAENG, K., BHARUKA, S., GAO, I., JEFFREY, M. C., SARAPH, V., SU, B.-Y., TRIPPEL, C., YANG, J., RABBAT, M., LUCIA, B., ET AL. CPR: Understanding and Improving Failure Tolerant Training for Deep Learning Recommendation with Partial Recovery. In *The Fourth Conference on Systems and Machine Learning (MLSys 21)* (2021).

[222] MAHMOUD, A., AGGARWAL, N., NOBBE, A., VICARTE, J. R. S., ADVE, S. V., FLETCHER, C. W., FROSIO, I., AND HARI, S. K. S. PyTorchFI: A Runtime Perturbation Tool for DNNs. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W 20)* (2020).

[223] MAHMOUD, A., HARI, S. K. S., FLETCHER, C. W., ADVE, S. V., SAKR, C., SHANBHAG, N.,

MOLCHANOV, P., SULLIVAN, M. B., TSAI, T., AND KECKLER, S. W. HarDNN: Feature Map Vulnerability Evaluation in CNNs. *arXiv preprint arXiv:2002.09786* (2020).

[224] MAHMOUD, A., HARI, S. K. S., FLETCHER, C. W., ADVE, S. V., SAKR, C., SHANBHAG, N., MOLCHANOV, P., SULLIVAN, M. B., TSAI, T., AND KECKLER, S. W. Optimizing Selective Protection for CNN Resilience. In *IEEE International Conference on Software Reliability Engineering (ISSRE 21)* (2021).

[225] MAHMOUD, A., HARI, S. K. S., SULLIVAN, M. B., TSAI, T., AND KECKLER, S. W. Optimizing Software-Directed Instruction Replication for GPU Error Detection. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC 18)* (2018).

[226] MAITY, R. K., RAWAT, A. S., AND MAZUMDAR, A. Robust Gradient Descent via Moment Encoding with LDPC Codes. In *2019 IEEE International Symposium on Information Theory (ISIT 19)* (2019).

[227] MALLICK, A., CHAUDHARI, M., SHETH, U., PALANIKUMAR, G., AND JOSHI, G. Rateless Codes for Near-Perfect Load Balancing in Distributed Matrix-Vector Multiplication. *Proceedings of the ACM on Measurement and Analysis of Computing Systems (SIGMETRICS 19) 3*, 3 (2019), 1–40.

[228] MATTSON, P., CHENG, C., COLEMAN, C., DIAMOS, G., MICIKEVICIUS, P., PATTERSON, D., TANG, H., WEI, G.-Y., BAILIS, P., BITTORF, V., ET AL. MLPerf Training Benchmark.

[229] MOHAN, J., PHANISHAYEE, A., AND CHIDAMBARAM, V. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)* (2021).

[230] MOODY, A., BRONEVETSKY, G., MOHROR, K., AND DE SUPINSKI, B. R. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC 10)* (2010).

[231] MUDIGERE, D., HAO, Y., HUANG, J., JIA, Z., TULLOCH, A., SRIDHARAN, S., LIU, X., OZDAL, M., NIE, J., PARK, J., ET AL. Software-Hardware Co-design for Fast and Scalable Training of Deep Learning Recommendation Models. *arXiv preprint arXiv:2104.05158* (2021).

[232] MULLAPUDI, R. T., CHEN, S., ZHANG, K., RAMANAN, D., AND FATAHALIAN, K. Online Model Distillation for Efficient Video Inference. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV 19)* (2019), pp. 3573–3582.

[233] NACHMANI, E., MARCIANO, E., LUGOSCH, L., GROSS, W. J., BURSHTEIN, D., AND BE'ERY, Y. Deep Learning Methods for Improved Decoding of Linear Codes. *IEEE Journal of Selected Topics in Signal Processing 12*, 1 (2018), 119–131.

[234] NAKANDALA, S., SAUR, K., YU, G.-I., KARANASOS, K., CURINO, C., WEIMER, M., AND INTERLANDI, M. A Tensor Compiler for Unified Machine Learning Prediction Serving. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (2020).

[235] NARAYANAN, D., SANTHANAM, K., PHANISHAYEE, A., AND ZAHARIA, M. Accelerating Deep Learning Workloads Through Efficient Multi-Model Execution. In *NeurIPS Workshop on Systems for Machine Learning* (2018).

[236] NARAYANAN, D., SHOEYBI, M., CASPER, J., LEGRESLEY, P., PATWARY, M., KORTHIKANTI, V., VAINBRAND, D., KASHINKUNTI, P., BERNAUER, J., CATANZARO, B., PHANISHAYEE, A., AND ZAHARIA, M. Efficient Large-Scale Language Model Training on GPU Clusters Using

Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 21)* (2021).

[237] NAUMOV, M., MUDIGERE, D., SHI, H.-J. M., HUANG, J., SUNDARAMAN, N., PARK, J., WANG, X., GUPTA, U., WU, C.-J., AZZOLINI, A. G., ET AL. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *arXiv preprint arXiv:1906.00091* (2019).

[238] NICOLAE, B., LI, J., WOZNIAK, J. M., BOSILCA, G., DORIER, M., AND CAPPELLO, F. Deepfreeze: Towards Scalable Asynchronous Checkpointing of Deep Learning Models. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)* (2020).

[239] NICOLAE, B., MOODY, A., GONSIOROWSKI, E., MOHROR, K., AND CAPPELLO, F. VeloC: Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS 19)* (2019).

[240] NORMAND, E., WERT, J. L., QUINN, H., FAIRBANKS, T. D., MICHALAK, S., GRIDER, G., IWANCHUK, P., MORRISON, J., WENDER, S., AND JOHNSON, S. First record of single-event upset on ground, cray-1 computer at Los Alamos in 1976. *IEEE Transactions on Nuclear Science 57*, 6 (2010), 3114–3120.

[241] NVIDIA. NVIDIA Deep Learning Performance Guide. `https://docs.nvidia.com/deeplearning/sdk/dl-performance-guide/index.html`. Last accessed 08 June 2021.

[242] OH, N., SHIRVANI, P. P., AND MCCLUSKEY, E. J. Control-Flow Checking by Software Signatures. *IEEE transactions on Reliability 51*, 1 (2002), 111–122.

[243] OLIVEIRA, D., PILLA, L., DEBARDELEBEN, N., BLANCHARD, S., QUINN, H., KOREN, I., NAVAUX, P., AND RECH, P. Experimental and Analytical Study of Xeon Phi Reliability. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 17)* (2017).

[244] OLSTON, C., FIEDEL, N., GOROVOY, K., HARMSEN, J., LAO, L., LI, F., RAJASHEKHAR, V., RAMESH, S., AND SOYKE, J. TensorFlow-Serving: Flexible, High-Performance ML Serving. *NIPS ML Systems Workshop* (2017).

[245] OSTROUCHOV, G., MAXWELL, D., ASHRAF, R. A., ENGELMANN, C., SHANKAR, M., AND ROGERS, J. H. GPU lifetimes on Titan Supercomputer: Survival Analysis and Reliability. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC 20)* (2020).

[246] OUSTERHOUT, K., RASTI, R., RATNASAMY, S., SHENKER, S., AND CHUN, B.-G. Making Sense of Performance in Data Analytics Frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (2015).

[247] OZEN, E., AND ORAILOGLU, A. Sanity-Check: Boosting the Reliability of Safety-Critical Deep Neural Network Applications. In *2019 IEEE 28th Asian Test Symposium (ATS 19)* (2019).

[248] OZEN, E., AND ORAILOGLU, A. Concurrent Monitoring of Operational Health in Neural Networks Through Balanced Output Partitions. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC 20)* (2020).

[249] OZEN, E., AND ORAILOGLU, A. Just Say Zero: Containing Critical Bit-Error Propagation in Deep Neural Networks with Anomalous Feature Suppression. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD 20)* (2020).

[250] PARK, J., NAUMOV, M., BASU, P., DENG, S., KALAIAH, A., KHUDIA, D., LAW, J., MALANI, P., MALEVICH, A., NADATHUR, S., ET AL. Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications. *arXiv preprint arXiv:1811.09886* (2018).

[251] PATTERSON, D., GONZALEZ, J., HÖLZLE, U., LE, Q., LIANG, C., MUNGUIA, L.-M., ROTHCHILD, D., SO, D., TEXIER, M., AND DEAN, J. The Carbon Footprint of Machine Learning Training Will Plateau, Then Shrink. *arXiv preprint arXiv:2204.05149* (2022).

[252] PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 88)* (1988).

[253] PLANK, J., AND GREENAN, K. Jerasure: A Library in C Facilitating Erasure Coding for Storage Applications _ Version 2.0. Technical Report UT-EECS-14–721. *University of Tennessee* (2014).

[254] PLANK, J. S. The RAID-6 Liberation Codes. In *6th USENIX Conference on File and Storage Technologies (FAST 08)* (2008).

[255] PLANK, J. S., GREENAN, K. M., AND MILLER, E. L. A Complete Treatment of Software Implementations of Finite Field Arithmetic for Erasure Coding Applications. Tech. Rep. UT-CS-13-717, University of Tennessee, October 2013.

[256] PLANK, J. S., GREENAN, K. M., AND MILLER, E. L. Screaming Fast Galois Field Arithmetic Using Intel SIMD Instructions. In *11th USENIX Conference on File and Storage Technologies (FAST 13)* (2013).

[257] PLANK, J. S., SCHUMAN, C. D., AND ROBISON, B. D. Heuristics for Optimizing Matrix-Based Erasure Codes for Fault-Tolerant Storage Systems. In *The Fourty Second International Conference on Dependable Systems and Networks (DSN 12)* (2012).

[258] QIAO, A., ARAGAM, B., ZHANG, B., AND XING, E. Fault Tolerance in Iterative-Convergent Machine Learning. In *International Conference on Machine Learning* (2019), pp. 5220–5230.

[259] QIAO, Y., ZHANG, M., ZHOU, Y., KONG, X., ZHANG, H., XU, M., BI, J., AND WANG, J. NetEC: Accelerating Erasure Coding Reconstruction With In-Network Aggregation. *IEEE Transactions on Parallel and Distributed Systems 33*, 10 (2022), 2571–2583.

[260] QURESHI, Z., MAILTHODY, V. S., GELADO, I., MIN, S. W., MASOOD, A., PARK, J., XIONG, J., NEWBURN, C., VAINBRAND, D., CHUNG, I., ET AL. BaM: A Case for Enabling Fine-grain High Throughput GPU-Orchestrated Access to Storage. *arXiv preprint arXiv:2203.04910* (2022).

[261] RASHMI, K. V., CHOWDHURY, M., KOSAIAN, J., STOICA, I., AND RAMCHANDRAN, K. EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016).

[262] RASHMI, K. V., SHAH, N. B., GU, D., KUANG, H., BORTHAKUR, D., AND RAMCHANDRAN, K. A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster. In *5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 13)* (June 2013).

[263] RASHMI, K. V., SHAH, N. B., GU, D., KUANG, H., BORTHAKUR, D., AND RAMCHANDRAN, K. A Hitchhiker's Guide to Fast and Efficient Data Reconstruction in Erasure-Coded Data Centers. In *Proceedings of the 2014 ACM SIGCOMM Conference (SIGCOMM 14)* (2014).

[264] RECH, P., AGUIAR, C., FERREIRA, R., FROST, C., AND CARRO, L. Neutron radiation test

of graphic processing units. In *IEEE 18th International On-Line Testing Symposium (IOLTS 12)* (2012).

[265] RECH, R. L., MALDE, S., CAZZANIGA, C., KASTRIOTOU, M., LETICHE, M., FROST, C., AND RECH, P. High Energy and Thermal Neutrons Sensitivity of Google Tensor Processing Units. *IEEE Transactions on Nuclear Science* (2022).

[266] RECHT, B., RE, C., WRIGHT, S., AND NIU, F. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems (NIPS 11)* (2011).

[267] REED, I. S., AND SOLOMON, G. Polynomial Codes Over Certain Finite Fields. *Journal of the society for industrial and applied mathematics 8*, 2 (1960), 300–304.

[268] REIS, G. A., CHANG, J., VACHHARAJANI, N., RANGAN, R., AND AUGUST, D. I. SWIFT: Software Implemented Fault Tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO 05)* (2005).

[269] REISIZADEH, A., PRAKASH, S., PEDARSANI, R., AND AVESTIMEHR, S. Coded Computation Over Heterogeneous Clusters. In *Proceedings of the 2017 IEEE International Symposium on Information Theory (ISIT 17)* (2017).

[270] RIVERA, C., CHEN, J., XIONG, N., ZHANG, J., SONG, S. L., AND TAO, D. TSM2X: High-Performance Tall-and-Skinny Matrix-Matrix Multiplication on GPUs. *Journal of Parallel and Distributed Computing 151* (2021), 70–85.

[271] RIZZO, L. Effective Erasure Codes for Reliable Computer Communication Protocols. *ACM SIGCOMM Computer Communication Review 27*, 2 (1997), 24–36.

[272] RUDOW, M., RASHMI, K., AND GURUSWAMI, V. A Locality-Based Lens for Coded Computation. In *2021 IEEE International Symposium on Information Theory (ISIT 21)* (2021).

[273] RUSSAKOVSKY, O., DENG, J., SU, H., KRAUSE, J., SATHEESH, S., MA, S., HUANG, Z., KARPATHY, A., KHOSLA, A., BERNSTEIN, M., BERG, A. C., AND FEI-FEI, L. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV) 115*, 3 (2015), 211–252.

[274] SANDLER, M., HOWARD, A., ZHU, M., ZHMOGINOV, A., AND CHEN, L.-C. MobilenetV2: Inverted Residuals and Linear Bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 18)* (2018).

[275] SAPIO, A., CANINI, M., HO, C.-Y., NELSON, J., KALNIS, P., KIM, C., KRISHNAMURTHY, A., MOSHREF, M., PORTS, D., AND RICHTARIK, P. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)* (2021).

[276] SATHIAMOORTHY, M., ASTERIS, M., PAPAILIOPOULOS, D., DIMAKIS, A. G., VADALI, R., CHEN, S., AND BORTHAKUR, D. XORing Elephants: Novel Erasure Codes for Big Data. *Proceedings of the VLDB Endowment 6*, 5 (2013).

[277] SCHORN, C., AND GAUERHOF, L. FACER: A Universal Framework for Detecting Anomalous Operation of Deep Neural Networks. In *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC 21)* (2020).

[278] SCHORN, C., GUNTORO, A., AND ASCHEID, G. Efficient On-Line Error Detection and Mitigation for Deep Neural Network Accelerators. In *International Conference on Computer Safety,*

Reliability, and Security (SAFECOMP 18) (2018).

[279] SERGEEV, A., AND DEL BALSO, M. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[280] SETHI, G., ACUN, B., AGARWAL, N., KOZYRAKIS, C., TRIPPEL, C., AND WU, C.-J. RecShard: Statistical Feature-Based Memory Optimization for Industry-Scale Neural Recommendation. In *Proceedings of the Twenty-Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 22)* (2022).

[281] SHAH, N. B., LEE, K., AND RAMCHANDRAN, K. When do Redundant Requests Reduce Latency? *IEEE Transactions on Communications 64*, 2 (2016), 715–722.

[282] SHEN, H., CHEN, L., JIN, Y., ZHAO, L., KONG, B., PHILIPOSE, M., KRISHNAMURTHY, A., AND SUNDARAM, R. Nexus: A GPU Cluster Engine for Accelerating DNN-based Video Analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 19)* (2019).

[283] SHEN, H., HAN, S., PHILIPOSE, M., AND KRISHNAMURTHY, A. Fast Video Classification via Adaptive Cascading of Deep Models. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 17)* (2017).

[284] SHI, H., AND LU, X. TriEC: Tripartite Graph Based Erasure Coding NIC Offload. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 19)* (2019).

[285] SHI, H., AND LU, X. INEC: Fast and Coherent In-Network Erasure Coding. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC 20)* (2020).

[286] SICELOFF, S. Shuttle Computers Navigate Record of Reliability. `https://www.nasa.gov/mission_pages/shuttle/flyout/flyfeature_shuttlecomputers.html`. Last accessed 28 January 2022.

[287] SIMONYAN, K., AND ZISSERMAN, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations (ICLR 15)* (2015).

[288] SMITH, T. M., VAN DE GEIJN, R., SMELYANSKIY, M., HAMMOND, J. R., AND VAN ZEE, F. G. Anatomy of High-Performance Many-Threaded Matrix Multiplication. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS 14)* (2014).

[289] SMITH, T. M., VAN DE GEIJN, R. A., SMELYANSKIY, M., AND QUINTANA-ORTI, E. S. Toward ABFT for BLIS GEMM. In *Tech. Rep. TR-15-05*. The University of Texas at Austin, 2015.

[290] SULLIVAN, M. B., HARI, S. K. S., ZIMMER, B., TSAI, T., AND KECKLER, S. W. Swapcodes: Error Codes for Hardware-Software Cooperative GPU Pipeline Error Detection. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 18)* (2018).

[291] SULLIVAN, M. B., SAXENA, N., O'CONNOR, M., LEE, D., RACUNAS, P., HUKERIKAR, S., TSAI, T., HARI, S. K. S., AND KECKLER, S. W. Characterizing and Mitigating Soft Errors in GPU DRAM. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 21)* (2021).

[292] SUN, R., QIU, P., LYU, Y., WANG, D., DONG, J., AND QU, G. Lightning: Striking the Secure Isolation on GPU Clouds with Transient Hardware Faults. *arXiv preprint arXiv:2112.03662* (2021).

[293] SURESH, L., CANINI, M., SCHMID, S., AND FELDMANN, A. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (2015).

[294] TAN, L., SONG, S. L., WU, P., CHEN, Z., GE, R., AND KERBYSON, D. J. Investigating the Interplay Between Energy Efficiency and Resilience in High Performance Computing. In *2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS 15)* (2015).

[295] TAN, M., AND LE, Q. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning (ICML 19)* (2019).

[296] TANDON, R., LEI, Q., DIMAKIS, A. G., AND KARAMPATZIAKIS, N. Gradient Coding: Avoiding Stragglers in Distributed Learning. In *International Conference on Machine Learning (ICML 17)* (2017).

[297] TAO, W., JIANG, F., ZHANG, S., REN, J., SHI, W., ZUO, W., GUO, X., AND ZHAO, D. An End-to-End Compression Framework Based on Convolutional Neural Networks. In *2017 Data Compression Conference (DCC 17)* (2017).

[298] THANGARAJ, S., VARAGANTI, K., PUTTUR, K., AND RAO, P. Accelerating Machine Learning using BLIS, 2017.

[299] TODERICI, G., O'MALLEY, S. M., HWANG, S. J., VINCENT, D., MINNEN, D., BALUJA, S., COVELL, M., AND SUKTHANKAR, R. Variable Rate Image Compression with Recurrent Neural Networks. *arXiv preprint arXiv:1511.06085* (2015).

[300] TORRES-HUITZIL, C., AND GIRAU, B. Fault and Error Tolerance in Neural Networks: A Review. *IEEE Access 5* (2017), 17322–17341.

[301] TROTT, C. R., LEBRUN-GRANDIÉ, D., ARNDT, D., CIESKO, J., DANG, V., ELLINGWOOD, N., GAYATRI, R., HARVEY, E., HOLLMAN, D. S., IBANEZ, D., LIBER, N., MADSEN, J., MILES, J., POLIAKOFF, D., POWELL, A., RAJAMANICKAM, S., SIMBERG, M., SUNDERLAND, D., TURCKSIN, B., AND WILKE, J. Kokkos 3: Programming Model Extensions for the Exascale Era. *IEEE Transactions on Parallel and Distributed Systems 33*, 4 (2022), 805–817.

[302] UEZATO, Y. Accelerating XOR-Based Erasure Coding Using Program Optimization Techniques. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 21)* (2021).

[303] VENKATARAMAN, S., YANG, Z., FRANKLIN, M., RECHT, B., AND STOICA, I. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (2016).

[304] VIOLA, P., AND JONES, M. J. Robust Real-Time Face Detection. *International Journal of Computer Vision 57*, 2 (2004), 137–154.

[305] WADDEN, J., LYASHEVSKY, A., GURUMURTHI, S., SRIDHARAN, V., AND SKADRON, K. Real-World Design and Evaluation of Compiler-Managed GPU Redundant Multithreading. In *2014 ACM/IEEE 41st Annual International Symposium on Computer Architecture (ISCA 14)* (2014).

[306] WAN, Z., ANWAR, A., HSIAO, Y.-S., JIA, T., REDDI, V. J., AND RAYCHOWDHURY, A. Analyzing and Improving Fault Tolerance of Learning-Based Navigation Systems. In *2021 58th ACM/IEEE Design Automation Conference (DAC 21)* (2021).

[307] WANG, G., VENKATARAMAN, S., PHANISHAYEE, A., DEVANUR, N., THELIN, J., AND STOICA, I. Blink: Fast and generic collectives for distributed ml. In *Proceedings of Machine Learning and Systems (MLSys 20)* (2020).

[308] WANG, H. Low Precision Inference on GPU. `https://tinyurl.com/1g9e5dpw`, 2019.

Last accessed 08 June 2021.

[309] Wang, S., Liu, J., and Shroff, N. Coded Sparse Matrix Multiplication. In *Proceedings of the International Conference on Machine Learning (ICML 18)* (2018).

[310] Wang, W., Gao, J., Zhang, M., Wang, S., Chen, G., Ng, T. K., Ooi, B. C., Shao, J., and Reyad, M. Rafiki: Machine Learning as an Analytics Service System. *Proceedings of the VLDB Endowment 12*, 2 (2018), 128–140.

[311] Wang, X., Luo, Y., Crankshaw, D., Tumanov, A., Yu, F., and Gonzalez, J. E. IDK Cascades: Fast Deep Learning by Learning not to Overthink. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI 18)* (2018).

[312] Warden, P. Speech commands: A Dataset for Limited-Vocabulary Speech Recognition. *arXiv preprint arXiv:1804.03209* (2018).

[313] Weatherspoon, H., and Kubiatowicz, J. D. Erasure Coding vs. Replication: A Quantitative Comparison. In *International Workshop on Peer-to-Peer Systems (IPTPS 2002)* (2002).

[314] Weiss, T. R. Microsoft, HPE Bringing AI, Edge, Cloud to Earth Orbit in Preparation for Mars Missions. `https://www.hpcwire.com/2021/02/12/microsoft-hpe-bringing-ai-edge-cloud-to-earth-orbit-in-preparation-for-mars-` Last accessed 23 August 2021.

[315] Welinder, P., Branson, S., Mita, T., Wah, C., Schroff, F., Belongie, S., and Perona, P. Caltech-UCSD Birds 200. Tech. Rep. CNS-TR-2010-001, California Institute of Technology, 2010.

[316] Wilkening, M., Gupta, U., Hsia, S., Trippel, C., Wu, C.-J., Brooks, D., and Wei, G.-Y. RecSSD: Near Data Processing for Solid State Drive Based Recommendation Inference. In *Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 21)* (2021).

[317] Williams, S., Waterman, A., and Patterson, D. Roofline: an Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM 52*, 4 (2009), 65–76.

[318] Williams-Young, D. B., De Jong, W. A., Van Dam, H. J., and Yang, C. On the Efficient Evaluation of the Exchange Correlation Potential on Graphics Processing Unit Clusters. *Frontiers in chemistry 8* (2020), 581058.

[319] Wu, B., Dai, X., Zhang, P., Wang, Y., Sun, F., Wu, Y., Tian, Y., Vajda, P., Jia, Y., and Keutzer, K. FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 19)* (2019).

[320] Wu, C.-J., Raghavendra, R., Gupta, U., Acun, B., Ardalani, N., Maeng, K., Chang, G., Behram, F. A., Huang, J., Bai, C., et al. Sustainable AI: Environmental Implications, Challenges and Opportunities. *arXiv preprint arXiv:2111.00364* (2021).

[321] Wu, P., Guan, Q., DeBardeleben, N., Blanchard, S., Tao, D., Liang, X., Chen, J., and Chen, Z. Towards Practical Algorithm Based Fault Tolerance in Dense Linear Algebra. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC 16)* (2016).

[322] Wu, S., Shen, Z., and Lee, P. P. Enabling I/O-Efficient Redundancy Transitioning in Erasure-Coded KV Stores via Elastic Reed-Solomon Codes. In *2020 International Symposium on Reliable*

Distributed Systems (SRDS) (2020), IEEE, pp. 246–255.

[323] XIAO, H., RASUL, K., AND VOLLGRAF, R. Fashion-Mnist: A Novel Image Dataset for Benchmarking Machine Learning Algorithms. *arXiv preprint arXiv:1708.07747* (2017).

[324] XIAO, W., BHARDWAJ, R., RAMJEE, R., SIVATHANU, M., KWATRA, N., HAN, Z., PATEL, P., PENG, X., ZHAO, H., ZHANG, Q., YANG, F., AND ZHOU, L. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (2018).

[325] XU, Y., MUSGRAVE, Z., NOBLE, B., AND BAILEY, M. Bobtail: Avoiding Long Tails in the Cloud. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (2013).

[326] YADWADKAR, N. J., ANANTHANARAYANAN, G., AND KATZ, R. Wrangler: Predictable and Faster Jobs using Fewer Resources. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC 14)* (2014).

[327] YADWADKAR, N. J., HARIHARAN, B., GONZALEZ, J. E., SMITH, B., AND KATZ, R. H. Selecting the Best VM Across Multiple Public Clouds: A Data-Driven Performance Modeling Approach. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC 17)* (2017).

[328] YAN, S., LI, H., HAO, M., TONG, M. H., SUNDARARAMAN, S., CHIEN, A. A., AND GUNAWI, H. S. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *15th USENIX Conference on File and Storage Technologies (FAST 17)* (2017).

[329] YANG, J. A., HUANG, J., PARK, J., TANG, P. T. P., AND TULLOCH, A. Mixed-Precision Embedding Using a Cache. *arXiv preprint arXiv:2010.11305* (2020).

[330] YANG, L., NIE, B., JOG, A., AND SMIRNI, E. Enabling Software Resilience in GPGPU Applications via Partial Thread Protection. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE 21)* (2021).

[331] YIN, C., ACUN, B., LIU, X., AND WU, C.-J. TT-Rec: Tensor Train Compression for Deep Learning Recommendation Models. In *The Fourth Conference on Systems and Machine Learning (MLSys 21)* (2021).

[332] YU, P., AND CHOWDHURY, M. Salus: Fine-Grained GPU Sharing Primitives for Deep Learning Applications. In *The Third Conference on Machine Learning and Systems (MLSys 20)* (2020).

[333] YU, Q., MADDAH-ALI, M., AND AVESTIMEHR, S. Polynomial Codes: An Optimal Design for High-Dimensional Coded Matrix Multiplication. In *Advances in Neural Information Processing Systems (NIPS 17)* (2017).

[334] YU, Q., RAVIV, N., SO, J., AND AVESTIMEHR, A. S. Lagrange Coded Computing: Optimal Design for Resiliency, Security and Privacy. In *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics (AISTATS 19)* (2019).

[335] ZAHARIA, M., KONWINSKI, A., JOSEPH, A. D., KATZ, R., AND STOICA, I. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI 08)* (2008).

[336] ZAMANI, H., LIU, Y., TRIPATHY, D., BHUYAN, L., AND CHEN, Z. GreenMM: Energy Efficient GPU Matrix Multiplication Through Undervolting. In *Proceedings of the ACM International Conference on Supercomputing (ICS 19)* (2019).

[337] ZHAI, Y., GIEM, E., FAN, Q., ZHAO, K., LIU, J., AND CHEN, Z. FT-BLAS: A High Performance

BLAS Implementation With Online Fault Tolerance. *arXiv preprint arXiv:2104.00897* (2021).

[338] ZHANG, H., ANANTHANARAYANAN, G., BODIK, P., PHILIPOSE, M., BAHL, P., AND FREEDMAN, M. J. Live Video Analytics at Scale with Approximation and Delay-Tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (2017).

[339] ZHANG, H., CISSE, M., DAUPHIN, Y. N., AND LOPEZ-PAZ, D. mixup: Beyond Empirical Risk Minimization. In *International Conference on Learning Representations (ICLR 18)* (2018).

[340] ZHANG, J., RANGINENI, K., GHODSI, Z., AND GARG, S. ThUndervolt: Enabling Aggressive Voltage Underscaling and Timing Error Resilience for Energy Efficient Deep Learning Accelerators. In *Proceedings of the 55th Annual Design Automation Conference (DAC 18)* (2018).

[341] ZHANG, M., RAJBHANDARI, S., WANG, W., AND HE, Y. DeepCPU: Serving RNN-based Deep Learning Models 10x Faster. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018).

[342] ZHAO, K., DI, S., LI, S., LIANG, X., ZHAI, Y., CHEN, J., OUYANG, K., CAPPELLO, F., AND CHEN, Z. FT-CNN: Algorithm-Based Fault Tolerance for Convolutional Neural Networks. *IEEE Transactions on Parallel & Distributed Systems 32*, 07 (2021), 1677–1689.

[343] ZHAO, W., ZHANG, J., XIE, D., QIAN, Y., JIA, R., AND LI, P. AIBox: CTR Prediction Model Training on a Single Node. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management (CIKM 2019)* (2019).

[344] ZHENG, L., JIA, C., SUN, M., WU, Z., YU, C. H., HAJ-ALI, A., WANG, Y., YANG, J., ZHUO, D., SEN, K., ET AL. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (2020).

[345] ZHOU, T., AND TIAN, C. Fast Erasure Coding for Data Storage: A Comprehensive Study of the Acceleration Techniques. In *17th USENIX Conference on File and Storage Technologies (FAST 19)* (2019).

[346] ZHOU, Y., EBRAHIMI, S., ARIK, S. Ö., YU, H., LIU, H., AND DIAMOS, G. Resource-Efficient Neural Architect. *arXiv preprint arXiv:1806.07912* (2018).

[347] ZHUANG, S., LI, Z., ZHUO, D., WANG, S., LIANG, E., NISHIHARA, R., MORITZ, P., AND STOICA, I. Hoplite: Efficient and Fault-Tolerant Collective Communication for Task-Based Distributed Systems. In *Proceedings of the 2021 ACM SIGCOMM Conference (SIGCOMM 21)* (2021).

[348] ZOPH, B., AND LE, Q. V. Neural Architecture Search with Reinforcement Learning. *arXiv preprint arXiv:1611.01578* (2016).