Bridging the Scalability Gap by Exploiting Error Tolerance for Emerging Applications

by

Parker Hill

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2018

Parker Hill

parkerhh@umich.edu

ORCID iD: 0000-0002-6114-6033

ACKNOWLEDGEMENTS

I thank my dissertation committee – Lingjia Tang, Jason Mars, Scott Mahlke, and Hun Seok Kim – for their feedback while writing this dissertation. I particularly appreciate the guidance and support I have received by having Lingjia, Jason, and Scott as co-advisors. My dissertation would not be nearly as exciting or robust if not for Jason's ability to make a compelling case for seemingly every research direction that I've pursued and Lingjia's ability to pinpoint key questions throughout my research. In addition to those in formal advisory roles, I thank Michael Laurenzano for always being available and helping refine every aspect of my research.

I am grateful to all of the members of Clarity Lab. Their contagious work ethic and ambition provided me with the motivation to conduct my work with equally high determination and standards. Specifically, I owe Animesh Jain gratitude for invaluable collaboration and enduring countless practice talks.

I thank my parents, Brent and Caroline, for their love and support before and throughout this process. Architecting the foundation of a computer architecture researcher is certainly more difficult than being one. You truly instilled the characteristics in me that make me a happy and productive researcher. Finally, I express gratitude towards my brothers – Spencer, Mason, and Connor – for the innumerable discussions and debates, providing me with the fortitude to thrive in graduate school.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

In recent years, there has been a surge in demand for intelligent applications. These emerging applications are powered by algorithms from domains such as computer vision, image processing, pattern recognition, and machine learning. Across these algorithms, there exist two key computational characteristics. First, the computational demands they place on computing infrastructure is large, with the potential to substantially outstrip existing compute resources. Second, they are necessarily resilient to errors due to their inputs and outputs being inherently noisy and imprecise.

Despite the staggering computational requirements and resilience of intelligent applications, current infrastructure uses conventional software and hardware methodologies. These systems needlessly consume resources for every bit of precision and arithmetic. To address this inefficiency and help bridge the performance gap caused by intelligent applications, this dissertation investigates exploiting error tolerance across the hardware-software stack. Specifically, we propose (1) statistical machinery to guarantee that accuracy is not compromised when removing work or precision, (2) a GPU optimization framework for work skipping and bottleneck mitigation, and (3) exploration of unconventional numerical representations to steer future hardware designs.

CHAPTER I

Introduction

In the past few years, there has been a surge demand for intelligent applications, as companies like Apple, Google, Microsoft and Amazon devise increasingly wide-reaching and sophisticated software offerings [3, 5, 37, 73]. Underlying these applications are algorithms from domains such as computer vision, image processing, pattern recognition and machine learning. Two characteristics of these domains have become evident in recent years. First, the computational demands they place on computing infrastructure are large, with the potential to outstrip available compute resources by a large margin [45]. Second, their inputs and outputs are inherently noisy and imprecise. Despite these staggering computational requirements and imprecise inputs and outputs, current infrastructure uses conventional methodologies resulting in redundant computations and overly precise arithmetic. However, to remove insignificant work and precision three key challenges must be addressed.

First, developers and service operators do not trust that accuracy will not be compromised when computation and precision is removed. To build trust in these approaches, new runtime systems must be designed that can guarantee consistent output quality. Second, the architecture must be considered in order to exploit error tolerance to improve performance. For example, cutting raw computation (e.g., dynamic floating-point instructions) does not directly map to performance on commodity accelerators due to the wide vector units present in the architecture. To improve performance, the microarchitectural nuances of such accelerators must be carefully considered. Finally, to improve performance by developing customized accelerators for these emerging application, we must explore a large trade-off space between performance and accuracy. Specifically, the accelerator's underlying numerical representation, a critical factor in performance, has not been investigated due to the difficultly in simulating all of these representations.

This dissertation addresses these challenges to promote the widespread use of exploiting error tolerance. In the first chapter, it proposes a runtime system that provides statistical output quality guarantees, allowing approximated computations to be trusted in deployment. We achieve this by casting the problem of finding an accurate approximation to a statistical one and design a framework around robust statistical methods to guarantee output quality. In the next chapter, this dissertation introduces techniques to efficiently exploit error tolerance for DNNs being executed on GPUs. It does this by tailoring work skipping and memory compression to the GPU architecture, requiring (1) skipping work in a very structured manner to fully utilized GPU vector units and (2) decompressing values with very little overhead to avoid undermining the benefits of mitigating the on-chip memory bandwidth architectural bottleneck. Finally, this dissertation proposes exploring the numerical representation design space to achieve even higher performance benefits for custom DNN accelerators.

1.1 Motivation

In this section, we motivate the need of runtime systems and approximation techniques that exploit error tolerance in emerging applications.

1.1.1 Output Quality Guarantees

There has recently been a surge of popularity in intelligent webservices, as companies like Apple, Google, Microsoft and Amazon devise increasingly wide-reaching and sophisticated software offerings [3, 5, 37, 73]. The underlying computational components that are central to such services are often computations that are amenable to approximation from domains that include computer vision, image processing, pattern recognition and machine learning. Two characteristics of these domains have become evident in recent years. First, the computational demands they place on computing infrastructure are large, with the potential to outstrip available compute resources by a large margin [45]. Second, their inputs and outputs are inherently noisy and imprecise.

The confluence of these two characteristics make such applications prime candidates for *approximate computing*, where small, often imperceptible degradations in output quality can be traded for large improvements in performance or energy. While there are a number of techniques that have been devised for both hardware and software to make this tradeoff [4, 46, 51, 63, 98, 99], one of the key challenges that remains in this area and holds back the adoption of approximate computing lies in providing guarantees as to how much accuracy will be lost when applying approximation.

Conventional approximation systems use empirical demonstrations to validate that their approaches to steering approximation provide the accuracy levels intended by the designers, often based on devising training sets resembling the input sets that will be seen in production [29,51,72,102], a notoriously difficult problem to solve [26]. Other systems leverage training-free quality control mechanisms [63, 97, 98], often by assuming that recent inputs will resemble future inputs and adjusting the level of approximation accordingly. In all cases, however, this class of systems cannot provide guarantees of result quality on difficult cases that the system is not trained or designed to handle.

Prior works have taken steps to address the more fundamental problem of providing analytical or statistical guarantees of result quality. These works fall into two classes:

- **Static Analysis –** This class of techniques uses formal static analysis to reason about program accuracy under approximation [10,11,16,74,76], an approach that has two limitations. First, it is difficult to scale such approaches to problems beyond a limited level of complexity (dozens of lines of code). They can be applied to simple programs, for particular kinds of approximation techniques, but have difficulty scaling beyond these narrow cases. Second, these approaches are driven by the worst-case accuracy, and typically leave significant untapped performance opportunities on the table when the worst case fails to materialize.
- **Customized Solutions –** This class of techniques leverage quality guarantee

mechanisms that are specific to a set of assumptions around the application(s) or approximation technique(s), such as database aggregation operations [15] or reductions in a MapReduce framework [36,84]. Such techniques are important, but cannot be applied beyond the scope of applications and approximation techniques they are designed to cover.

Instead of designing application-specific techniques, this paper begins with the observation that many applications amenable to approximation have similar computational patterns, coming from the domains of machine learning, pattern recognition, computer vision, image processing and data mining. Exploiting this commonality, we describe a solution that provides statistical accuracy guarantees when applying approximation to a broad class of applications characterized with the property we describe as *map-based*. We describe an application as being map-based if parts of the application can be executed in parallel, and can also be executed *out-of-order* and *incrementally*. We leverage these properties to first compute a randomly-selected subset of the application output both with and without approximation applied. The fact that random sampling can be applied to the problem allows us to leverage statistical techniques to build statistical guarantees about the accuracy of the output when applying approximation.

1.1.2 Removing Insignificant Computation

This section motivates the use of exploiting error tolerance in emerging applications, specifically deep neural networks. In the following two subsections, we find this need for both commodity and customized hardware.

1.1.2.1 Commodity Hardware

As user demand for state-of-the-art technologies in the domains of computer vision, speech recognition, and natural language processing (NLP) continues to increase, system designers are tasked with supporting increasingly sophisticated machine learning capabilities [44, 45]. An important trend that impacts the design of current and future intelligent systems is the convergence of industry toward *deep learning* as the computational engine providing these services. Large companies, including Google [104], Facebook [33], and Microsoft [89], among others [65], are using deep neural networks (DNNs) as the primary technique underpinning machine learning for vision, speech, and NLP tasks.

With an increasing number of queries requiring DNN computation on the critical path, a significant challenge emerges vis-à-vis the large gap between the amount of computation required to process a DNN-based query relative to a traditional browser centric query such as web search [45].

This work is driven by key insight that, much like biological neural networks, DNNs are intrinsically resilient to both minor numerical adjustments [27, 53, 92, 120] and eliding spurious neurons and synapses [40, 41]. This characteristic can be leveraged to achieve performance improvement. However, as we show later in the paper, reduction of computation and data movement does not directly translate to performance improvement. Techniques from prior work either create a mismatch between the algorithm and underlying architecture, or are not designed to address the real hardware bottlenecks, leaving two open challenges in the way of realizing performance benefits:

- **Limitation 1**: *Irregular Computation* – Network pruning [40, 41], a state-of-the-art machine learning technique that reduces the DNN topology focuses on reducing the memory footprint. However, their methodology fails to realize performance benefits on GPUs. Although this technique significantly reduces the amount of raw computation (i.e. floating-point operations), we show that the hardware-inefficient irregular DNN topology outweighs the benefits and results in substantial slowdown (up to 61×) due to increased branch divergence and uncoalesced memory access on GPUs. We present details on this limitation in §4.1.1. To achieve performance benefits, the challenge of reducing computation while aligning the reduced computation with underlying hardware must be addressed.
- **Limitation 2**: *Not Optimized for Bottleneck* – Our investigation identifies on-chip memory bandwidth to be the key bottleneck for DNN execution on GPUs. However, prior works focus on improving off-chip memory bandwidth using compression [98], removing non-contributing bits to increase the effective bandwidth. This technique, however, fails to provide significant speedups for DNNs (details in §4.1.2). We evaluate off-chip data packing and observe a speedup of less than 4%. On the other hand, compared to off-chip techniques, it is more challenging to perform on-chip compression because frequently reformatting data is difficult to achieve without introducing significant overhead.

1.1.2.2 Customized Hardware

Recently, deep neural networks (DNNs) have yielded state-of-the-art performance on a wide array of AI tasks, including image classification [60], speech recognition [42], and language understanding [114]. In addition to algorithmic innovations [79, 113, 116], a key driver behind these successes are advances in computing infrastructure that enable large-scale deep learning—the training and inference of large DNN models on massive datasets [24, 30]. Indeed, highly efficient GPU implementations of DNNs played a key role in the first breakthrough of deep learning for image classification [60]. Given the ever growing amount of data available for indexing, analysis, and training, and the increasing prevalence of ever-larger DNNs as key building blocks for AI applications, it is critical to design computing platforms to support faster, more resource-efficient DNN computation.

A set of core design decisions are common to the design of these infrastructures. One such critical choice is the numerical representation and precision used in the implementation of underlying storage and computation. Several recent works have investigated the numerical representation for DNNs [12, 19, 27, 78]. One recent work found that substantially lower precision can be used for training when the correct numerical rounding method is employed [38]. Their work resulted in the design of a very energy-efficient DNN platform.

This work and other previous numerical representation studies for DNNs have either limited themselves to a small subset of the customized precision design space or drew conclusions using only small neural networks. For example, the work from Gupta et al. 2015 evaluates 16-bit fixed-point and wider computational precision on LeNet-5 [66] and CIFARNET [58]. The fixed-point representation (Figure 5.1) is only one of many possible numeric representations. Exploring a limited customized precision design space inevitably results in designs lacking in energy efficiency and computational performance. Evaluating customized precision accuracy based on small neural networks requires the assumption that much larger, production-grade neural networks would operate comparably when subjected to the same customized precision.

1.2 Exploiting Error Tolerance

In this section, we outline the proposed techniques for exploiting error tolerance for emerging in this work.

1.2.1 NinjaProx: Achieving Accuracy Guarantees for Approximate Computing

Central to our approach of maintaining output quality is a *guarantee engine* for efficiently providing accuracy guarantees when applying approximation to map-based applications. Our approach leverages the well-known central limit theorem (CLT) as the starting point for generating an accuracy guarantee. However, as its name implies, the CLT provides guarantees that hold only *in the limit* (that is, with infinite samples). However, taking infinite (or even very large numbers of) samples is impractical, particularly when the goal is to improve time-to-solution with approximation. Instead, we leverage recent advances in the Berry-Esseen theorem [107, 108, 119] — a framework for adjusting the results of the CLT in the face of finite numbers of samples — to produce accuracy guarantees that are robust to arbitrary inputs that have not been anticipated or already seen by the system.

Our approach is in contrast to standard practice in approximate computing which is to use heuristics to drive their approximation strategies [4, 6, 29, 51, 63, 72, 97, 98, 102, 109, 112]. Central to these approaches are intangible accuracy constraints that lack completely defined and enforceable semantics. Such accuracy constraints take the form of statements such as *"The accuracy target is 99%"*, including no requirements about how often or on what inputs the accuracy target will be met. Accuracy constraints of this form lead to systems that produce acceptably-accurate approximations in the common case, but cannot provide robust guarantees of accuracy for any particular input or set of inputs. Instead, this work introduces the *Accuracy Service Level Agreement (ASLA)*, a contract between the programmer or end-user of an application and the runtime system that computation accuracy will meet a well-defined set of constraints. Intuitively, these constraints take the form of *"Accuracy must be above 99% with 95% confidence"*, which will be met by the runtime system regardless of the characteristics of the input, the desired accuracy level, and the confidence level specified. This work defines, specifies semantics for, and shows runtime support for ASLAs.

1.2.2 DeftNN: Addressing Bottlenecks for DNN Execution on GPUs via Synapse Vector Elimination and Near-compute Data Fission

This work introduces *DeftNN*, a GPU DNN execution framework that leverages error tolerance of DNN executions by tailoring the removal of precision and work to the GPU architecture. Firstly, *synapse vector elimination* reduces the total problem size by automatically locating and discarding non-contributing synapses in the DNN – those synapses having negligible impact on the output results – to improve performance. To address the limitation of irregular computation, our insight is that it is necessary to preserve existing architectural optimizations in original GPU-efficient applications. Utilizing this insight, synapse vector elimination applies a novel transformation to the DNN data layout, producing computations that efficiently leverage GPU hardware.

The second optimization, *near-compute data fission*, mitigates the GPU on-chip memory bandwidth bottleneck by optimizing the utilization of integer units during DNN execution. To address the prior work's limitation of providing only off-chip bandwidth optimization [98], as on-chip memory is closer to the functional units, we design novel techniques that can support low-overhead very fine-grained data conversion. The key insight that makes near-compute data fission feasible is that the focus of data conversion must be shifted from high compression ratio to low decompression overhead. In addition to the benefits achieved by our carefully optimized near-compute data fission technique on commodity hardware, we describe a small additional unit called the *Data Fission Unit* (DFU) that can be added to existing GPU hardware to obviate data fission overhead to realize additional benefits on future generations of GPU hardware.

1.2.3 Rethinking Numerical Representations for DNNs

In this work, we propose exploring the accuracy-efficiency trade-off made available via specialized custom-precision hardware for inference and present a method to efficiently traverse this large design space to find an optimal design. Specifically, we propose evaluating the impact of a wide spectrum of customized precision settings for fixed-point and floating-point representations on accuracy and computational performance. We propose evaluating these customized precision configurations on large, state-of-the-art neural networks. By evaluating the full computational precision design space on a spectrum of these production-grade DNNs, we want to determine:

1. Whether precision requirements generalize across all neural networks or not. The answer to this question could prompt designers of future DNN infrastructures to carefully consider the applications that will be executed on their platforms, contrary to works that design for large networks and evaluate accuracy on small networks [12, 19].
2. Whether many large-scale DNNs require more precision for arithmetic than previously found from small-scale evaluations [12, 19, 27]. For example, it is unclear if a large network such as GoogLeNet requires the same number of bits as opposed to small networks such as LeNet-5.
3. Whether floating-point representations are more or less efficient than fixed-point representations when selecting optimal precision settings. For example, a lower precision floating-point representation may be acceptable, when compared to a fixed-point representation. Current platform designers may need to reconsider the use of the floating-point representations for DNN computations in place of the commonly used fixed-point representations [12, 19, 27, 78].

To make the answers to these questions of customized precision readily actionable for DNN infrastructure designers, we propose designing a technique to quickly search the large customized precision design space. This technique should leverage critical values in the computation to capture the propagation of numerical error to build a model to predict accuracy. Using such a method on deployable DNNs, should provide infrastructure designers a near-optimal customized precision, without requiring an exhaustive search of all inputs and configurations.

1.3 Summary of Contributions

This work proposes runtime systems and techniques for exploiting error tolerance in emerging applications. The specific contributions are as follows:

- **Accuracy Service Level Agreements** – this work introduces the concept of the Accuracy Service Level Agreement (ASLA). We describe the specification and enforcement mechanisms for making use of ASLAs in approximate computing (described in §3.1).
- **End-to-end Approximation** – we introduce NinjaProx, an end-to-end system for ASLA-enabled approximate computing. NinjaProx presents the user with a flexible set of knobs to trade off performance and accuracy while providing statistical accuracy guarantees (§3.2). Through a thorough evaluation that covers 8 map-based applications employing 4 approximation techniques proposed in the literature, we show that it is the first system to provide both high performance approximation and robust guarantees of accuracy.
- **Robust Accuracy Guarantees** – we enable ASLAs for approximate map-based applications using a set of robust statistical techniques for generating accuracy

guarantees (§3.3). These techniques use no offline training or assumptions about the distribution of input or output data. Nor do they require reasoning about the semantics of the approximation or the exact computation, allowing them to easily be leveraged on complex applications for a wide range of approximation techniques and accuracy metrics.

- **DeftNN.** We introduce DeftNN, a state-of-the-art GPU DNN execution framework. This framework automatically applies synapse vector elimination and near-compute data fission optimizations to existing DNN software applications to dramatically improve performance on today's GPUs.
- **Synapse Vector Elimination.** We introduce a DNN optimization technique for GPUs, synapse vector elimination, that shrinks the topology of the neural network. This method is guided by the insight that network pruning techniques in DNN systems must have computational regularity to achieve significant speedups. Our experiments show that synapse vector elimination achieves 1.5× average end-to-end speedups on a set of 6 state-of-the-art DNNs on real GPU hardware.
- **Near-compute Data Fission.** We introduce near-compute data fission, which improves performance by efficiently packing on-chip memory. To realize speedup, we find that the focus must be shifted from minimizing data size to minimizing unpacking overhead. We find that near-compute data fission provides 1.6× end-to-end speedup on a set of 6 DNNs on real GPU hardware available today by performing unpacking in software. We also introduce a lightweight hardware extension (<0.25% area overhead) to facilitate efficient unpacking, achieving an additional 1.4× speedup over software-only near-compute data fission.

CHAPTER II

Background and Related Work

This chapter presents and compares work related to this dissertation. First, we examine prior work that investigates runtime systems for parameterizing approximation techniques. Next, we present prior work that has optimized DNNs on platforms other than the GPU, which is not directly applicable to GPU acceleration due to not considering the architecture. Finally, we look at prior work for optimizing DNN execution with customized numerical representations.

2.1 Output Quality Guarantees

Many approaches have been proposed in prior work to trade accuracy for improvements in execution time or energy, ranging from modifying the underlying hardware [100,121], the ISA [28], compiler [47,97,98], programming language [4,8,9,101], database [1], runtime system [6, 95] or multiple layers of the hardware/software stack [29,46,71,77,99].

Various techniques have been proposed to develop models of approximation accuracy. Offline training and profiling has been used extensively to guide the choice of how aggressively to approximate [4, 47, 48, 75, 109, 112]. NinjaProx uses no training phase, instead building accuracy guarantees dynamically for each input to an application. Dynamic approaches to maintaining high approximation accuracy have also been proposed in the literature. IRA [63] dynamically tunes the parameters to software-based approximations based on input, while others tune these parameters using intermittent accuracy checks [6, 97, 98]. Uncertain<T> [8] takes advantage of Bayesian networks and sampling to propagate uncertainty through calculations. Unlike NinjaProx, such approaches make no guarantee of runtime accuracy. Mayhap [101] is a tool to validate that a probabilistic property exists in a program by sampling Bayesian networks, but does not impact how approximation is applied to a program.

ApproxHadoop [36] and other recent works [15, 84] devise techniques to bound the error propagated to the single output value when applying a particular set of reduction operators in approximate MapReduce computations. These techniques are applicable to specific reduction operators – sum, count, average, ratio, minimum, and maximum – and are difficult to extend to more complex scenarios. Instead, NinjaProx provides accuracy guarantees for a host of common error metrics when approximating arbitrary map-based computations.

Static analysis of application and approximation semantics has been used to reason about approximation accuracy [10, 11, 16, 103, 111, 124]. Specifically, some have used semantic analysis to reason about the accuracy of perforated programs [76, 94]. Unlike static analysis, NinjaProx uses no semantic analysis of the computation or assumptions about the distribution of the input, instead focusing directly on the results of the computation, which allows it to be applied to applications and approximations that have complex semantics.

2.2 Commodity Hardware

The computational requirements and applicability of deep neural networks [62] and convolutional neural networks [66] have prompted researchers to design novel DNN hardware [2, 13, 14, 31, 39, 56, 69, 90, 105, 117]. Some of these hardware designs specifically target memory bandwidth [17, 19]. Although these works can provide substantial speedup upon fabrication, our techniques can operate on current commodity hardware.

On the software side, there has been a lot of effort to efficiently implement DNNs on GPUs [20, 52, 61, 64, 82]. In addition to optimizing for GPUs, some work has looked at optimizing DNNs at the cluster level [21, 24, 42, 44, 45, 54, 91]. Further software approaches consider using different types of neural networks to improve performance [35]. Optimized algorithms can be applied in concert with our optimization techniques.

Many prior works improve performance by exploiting reduced precision [27, 53, 68, 92, 120]. Reducing precision is possible for both floating-point and fixed-point formats [22, 38]. These works all require substantial hardware modifications to operate. ACME [50], although requiring less modifications to hardware by design, still requires substantial overhead when applied to a high-throughput accelerator such as a GPU. The DFU in DeftNN requires ¡0.25% overhead to continuously provide the functional units with data, while scaling ACME to the same number of units would require over 19% overhead.

2.3 Customized Hardware

To the best of our knowledge, our work is the first to examine the impact of numeric representations on the accuracy-efficiency trade-offs on large-scale, deployed DNNs with over half a million neurons (GoogLeNet, VGG, AlexNet), whereas prior work has only reported results on much smaller networks such as CIFARNET and LeNet-5 [12,19,23,27,38,78]. Many of these works focused on fixed-point computation due to the fixed-point representation working well on small-scale neural networks. We find very different conclusions when considering production-ready DNNs.

Other recent works have looked at alternative neural network implementations such as spiking neural networks for more efficient hardware implementation [21, 25]. This is a very different computational model that requires redevelopment of standard DNNs, unlike our proposed methodologies. Other works have proposed several approaches to improve performance and reduce energy consumption of deep neural networks by taking advantage of the fact that DNNs usually contain redundancies [18, 32].

CHAPTER III

Realizing Service Level Agreements on Result Accuracy for Approximate Data-parallel Programs

A major challenge in approximate computing lies in providing guarantees of result quality on arbitrary inputs that have not been anticipated or already seen by the system. It is widely believed that this is one of the key obstacles that has prevented the adoption of approximate computing in commercial and production environments.

This chapter presents an approach to this challenging problem for a broad class of computational problems, developing the statistical machinery to provide accuracy guarantees when approximating applications regardless of the input content and desired accuracy level. This mechanism builds on the insight that the computation in applications that are amenable to approximation can be performed both out-of-order and incrementally. We leverage this fact to (1) cheaply sample the application output with and without approximation, then (2) dynamically build a statistical description of the accuracy characteristics, leveraging statistical methods to generate accuracy guarantees on the approximation.

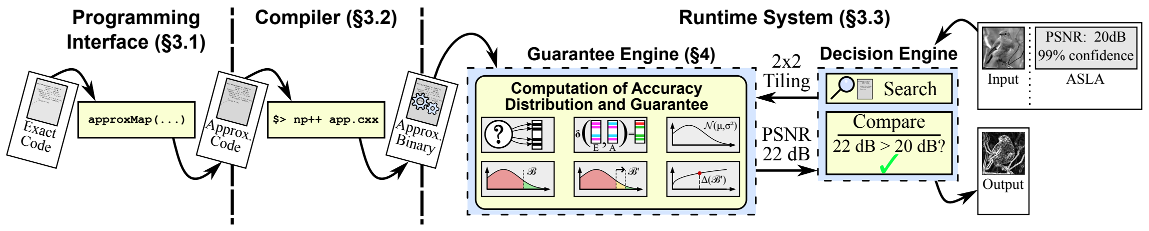


Figure 3.1: End-to-end NinjaProx: after compiling an approximation-amenable application written using the NinjaProx programming interface, the runtime system searches the set of approximations based on the accuracy guarantees supplied by the guarantee engine to select an approximation.

Building on this mechanism, we introduce and describe enforcement techniques for a new class of service level agreement (SLA) called an *Accuracy SLA (ASLA)*. Like an SLA on tail latency that bounds the likelihood of a computation falling below some latency target, an ASLA is a bound on the likelihood of computational accuracy falling below some target accuracy. We describe NinjaProx, a language, compiler, and runtime for enforcing ASLAs in approximate computing. We evaluate NinjaProx, applying 4 approximation techniques from the recent literature to 8 applications in the domains of machine learning, image processing, and data mining. Among thousands of individual experiments leveraging ASLAs that cover a number of common accuracy metrics and a range of confidence levels, we find that, in practice with representative approximate applications, NinjaProx never fails to meet the specified ASLA and achieves speedups that average 2.5×.

3.1 Accuracy SLAs

We begin by introducing the concept of the Accuracy Service Level Agreement (ASLA), motivating their use and defining their semantics.

3.1.1 Motivation

A conventional webservice Service Level Agreement (SLA) is an agreement specifying a set of constraints on performance. A common approach in specifying such SLAs is to describe constraints on the latency, such as *"the latency of 99% of queries is under 50ms."* This performance constraint on tail latency has two components – a performance target and a point in the tail of the latency distribution. Tail latency is a common metric because it reflects one of the realities of webservice operation – users observing long latency in their requests may leave for the competition and never return – thus constraining tail latency minimizes the likelihood of this event.

Similarly, users of an approximate application may seek alternatives if they are presented with results that are unsatisfactorily poor, and thus a convention similar to the conventional SLA is needed in dealing with approximation accuracy. There is a wealth of prior work in approximate computing that demonstrates satisfactory result accuracy by example, showing on a set of benchmarks that result accuracy usually satisfies some accuracy target(s) [4,46,51,63,98,99]. These approaches have resulted in useful systems for enacting approximation, but they do not solve the more fundamental problem, providing no assurances on result accuracy for difficult cases that may not have been envisioned by the system designers.

3.1.2 Defintion

The simplest form of an ASLA is shown in Equation 3.1.

$$P[A \geq T] \geq CL \tag{3.1}$$

This expression says that for an approximate application, the result accuracy A must not fall below the target accuracy T with probability of at least CL. Note that exact execution of the application (i.e. 100% accuracy with 100% confidence) can be expressed as an instance of this style of probabilistic guarantee, taking the form $P[A \geq 100\%] \geq 1.0$. Multiple simple ASLAs can be composed to form a compound ASLA. Consider the k-statement compound ASLA in Equation 3.2, which would require that all k expressions are satisfied.

$$\begin{gathered} P[A \geq T_0] \geq CL_0 \\ P[A \geq T_1] \geq CL_1 \\ \dots \\ P[A \geq T_k] \geq CL_k \end{gathered} \tag{3.2}$$

A compound ASLA may be useful, for example, in expressing a willingness to drop to 90% accuracy nearly all the time alongside a strict requirement to rarely drop below 80% accuracy. Such a requirement could be expressed as the 2-clause compound ASLA shown in Equation 3.3.

$$\begin{aligned} &P[A \geq 90\%] \geq 95\% \\ &P[A \geq 80\%] \geq 99.999\% \end{aligned} \tag{3.3}$$

3.1.3 ASLA Specification

The ASLA definitions described above are probabilistic constraints. In contrast to prior work that has focused on validating probabilistic assertions [101], an ASLA is not a stopping condition of the program, but is instead a set of constraints that must be met when computing approximate results.

For an ASLA clause with target accuracy T and confidence level α, an accuracy metric M that gives meaning to the accuracy constraint must also be specified. Our current implementation provides turnkey support for a variety of common accuracy metrics that include mean absolute percentage, mean absolute error, mean square error and peak signal-to-noise ratio (PSNR), as well as the ability to supply additional customized metrics. We describe these accuracy metrics and how they are supported in detail in §3.3.4.2.

3.2 NinjaProx Overview

This section introduces NinjaProx, a language, compiler and runtime system for enforcing ASLAs in map-based applications. An overview of NinjaProx is presented in Figure 3.1. NinjaProx has two main components: an offline component to compile support for approximation and the ASLA into the application, and a runtime to select how to approximate in a way that meets the ASLA.

3.2.1 Programming Interface

NinjaProx provides a simple interface that allows programmers to employ accuracy guarantees in map-based applications. The programmer defines an ASLA, a map-based computational kernel, and the dimensions of the problem being solved to utilize NinjaProx. We use `binarize`, an application that segments an image based on pixel intensity, as an example of the programming interface. For comparison, the original implementation is provided in Figure 3.2, while the NinjaProx implementation is found in Figure 3.3. NinjaProx currently supports C++11, used in the example code, but the NinjaProx statistical framework is language independent.

First (lines 5-6 of Figure 3.3), the programmer defines an ASLA, comprising an accuracy metric, the target accuracy level, and a statistical confidence level. The example code shows a concrete specification of the target accuracy level (0.90) and confidence level (0.99), however we envision that a typical NinjaProx deployment will defer the specification of one or both parameters until runtime. In the example, a simple 1-statement ASLA is defined. However, an array of ASLA objects can used together to build a compound ASLA as described in §3.1.2. Next (lines 8-10), an approximation parameter space is specified. This denotes the set of approximation parameters that NinjaProx should investigate, when it searches for an optimal approximation strategy.

The subsequent code (lines 12-22) is written specifically for the `binarize` application. Approximated `binarize` is made compatible with NinjaProx by refactoring the code so that each output is computed independently. In code, the operation is defined by the programmer by using a closure that produces a single output from loop indices (`mapIdx`) and approximation parameters (`apxParam`) (lines 18-22). Using the defined application kernel, output array, output dimensions, ASLA, and approximation parameter space, the programmer calls the `mapNinjaProx` function (lines 24-27). This function searches the approximation parameter space for the fastest approximation that meets the ASLA for the application, and then, using this approximation, writes an approximated value for each element in the output array.

3.2.2 Compilation

The NinjaProx compiler allows assumptions about the computational kernel to be checked and mitigates overhead introduced by the high-level programming model. For example, the NinjaProx compiler checks that the map-based computation provided to `mapNinjaProx` is pure, since NinjaProx will execute the map-based operation many times and in random order during its search for an optimal approximation strategy. When the map-based function is not pure, the compiler does not apply approximation and produces a warning.

To improve search runtime performance, the NinjaProx compiler orders the approximation parameters by how fast they are. This ordering is usually an obvious consequence of the nature of the approximation. For example, in skipping loop iterations for loop perforation [47], the technique becomes faster and less accurate as more iterations are skipped. The NinjaProx compiler applies another key optimization that benefits approximations that skip work by reusing output values, as in

```
//Input declarations (initialization omitted)
int width, height, threshold;
uint8_t input[width*height];

//Original binarize kernel
uint8_t output[width*height];
for(int x = 0; x < width; x++){
  for(int y = 0; y < height; y++){
    output[x+y*width] =
        input[x+y*width] > threshold ? 255 : 0;
  }
}
```

Figure 3.2: Original implementation of `binarize`, an application that segments pixels based on intensity.

tiling approximation [97]. In this case, the NinjaProx compiler identifies that multiple output values derive from identical computation, splitting the computation of the final approximate result into two phases: a first phase to compute each unique element in the output, and a second phase to copy those results to the remainder of the output.

3.2.3 Runtime Support

Given the set of available approximations and the ASLA, the goal of the NinjaProx runtime is to select the approximation that is as fast as possible and complies with the ASLA. NinjaProx leverages the ordered list of approximations from compilation to perform a binary search over the approximations, using the NinjaProx guarantee engine (described shortly in §3.3) to determine whether each of the searched approximations meets the ASLA. NinjaProx determines whether or not an approximation meets the ASLA by comparing the target accuracy with an *accuracy guarantee* generated by the guarantee engine. If the generated accuracy guarantee is more accurate than the target accuracy, the approximation is deemed to meet the accuracy constraint. In the case of compound ASLAs, an accuracy guarantee is generated for each component and the ASLA is said to be satisfied if all of the components of the guarantee are met.

For example, in Figure 3.1, the user-specified accuracy metric is peak signal-to-noise ratio (PSNR), the target accuracy is 20dB, and the confidence level is 99%. The decision engine tests the accuracy of several approximation strategies, guided by the decision engine. In the example in the figure, a 2×2 tiling approximation [97] results in an accuracy guarantee of 22dB. Since 22dB represents higher accuracy than the target accuracy of 20dB, the decision engine deems the 2×2 tiling approximation to be acceptable. If no faster approximation meeting the accuracy constraint is found, the 2×2 tiling method would be applied to the input to produce the final output of NinjaProx.

3.3 Guarantee Enforcement

In this section we provide an overview of the NinjaProx guarantee engine, the core mechanism designed to enforce ASLAs in approximate map-based applications.

3.3.1 Map-based Model

Central to providing ASLA guarantees lies the insight that computationally intensive applications frequently lend themselves to the map-based pattern, a computational paradigm that pervades the domains of image processing, machine learning,

```
//Include NinjaProx (at top of file)
#include "ninjaprox.hpp"
using namespace ninjaprox;

//ASLA(accuracyMetric,accuracyTarget,confidenceLevel)
ASLA asla(metrics::MissRate),0.9,0.99);

//Approximation parameter space definition
int tileSizes[] = {1,2,4,8,16};
int approxParameters[][] = {tileSizes,tileSizes};

//Input declarations (initialization omitted)
int width, height, threshold;
uint8_t input[width*height];
int dimensions[] = {width,height};

//Approximation-parameterized binarize
auto binarize = [&](int* mapIdx, int* apxParam){
  int x -= mapIdx[0] % apxParam[0];
  int y -= mapIdx[1] % apxParam[1];
  return input[x+y*width] > threshold ? 255 : 0;
};

//Approximate binarize using NinjaProx
uint8_t output[width*height];
approxMap(binarize, output, dimensions,
          asla, approxParameters);
```

Figure 3.3: Example of NinjaProx applied to a `binarize` implementation. The `ninjaprox::approxMap` function writes a value to each element in the output, after automatically finding the approximation parameters that fit the ASLA.

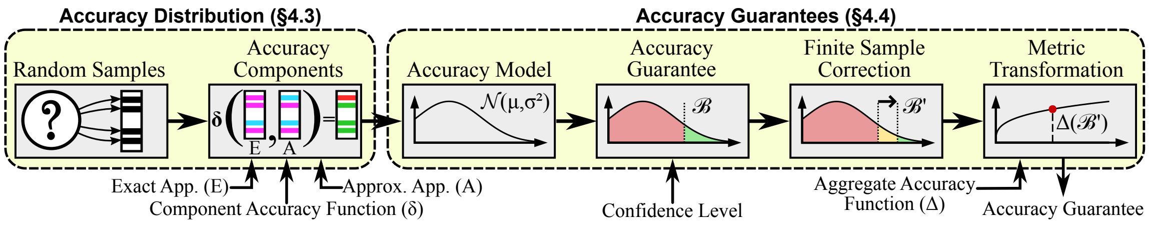


Figure 3.4: Guarantee engine overview comprised of two steps: accuracy inference and guarantee generation.

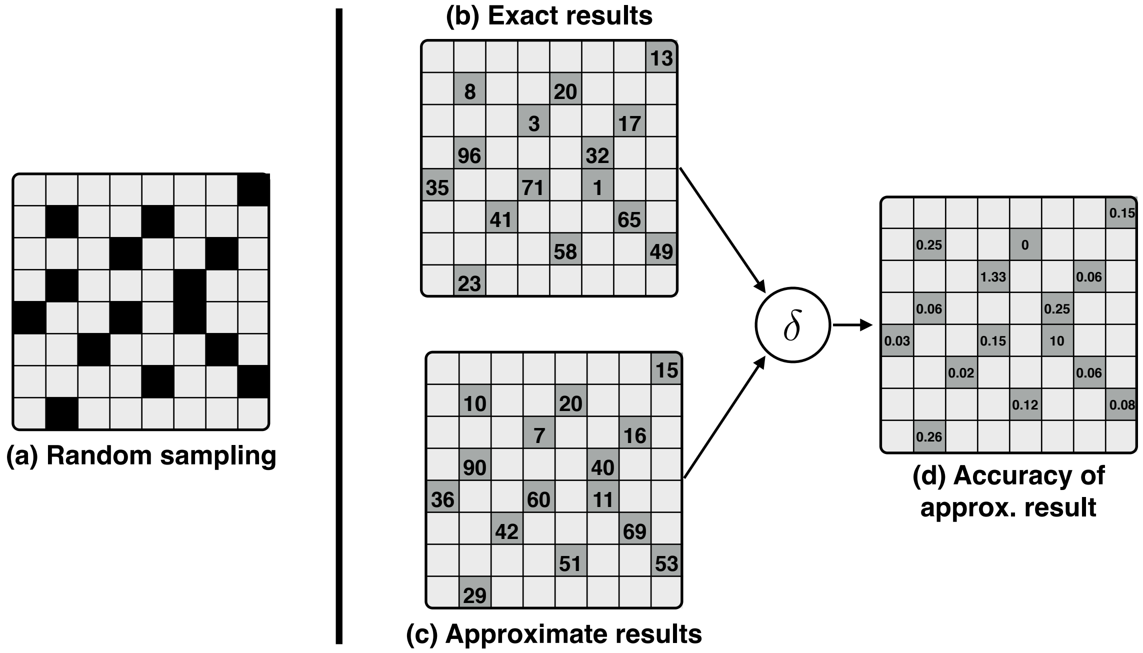


Figure 3.5: Example accuracy distribution derivation.

and computer vision. Specifically, applications holding this pattern have the characteristic that the application output can be divided into components that can be efficiently computed, independently of the others. An example of such an application is image convolution [106], where output pixel computations do not depend on one another and the computation of each adds a fixed number of computational operations.

The two key properties of the map-based model in enforcing ASLAs for approximate computing are its ability to provide both *out-of-order* and *incremental* computation. These properties mean that an arbitrary subset of the outputs can be computed efficiently, which facilitates performing efficient statistical random sampling of the outputs. By applying this random sampling, we get an independent and identically distributed (IID) distribution of the outputs. Even when the application output is not IID, the distribution produced by randomly sampling from the output population is IID. For example, adjacent pixels in an image are very likely to be correlated, but the first randomly selected pixel from the entire image is independent of the second randomly selected pixel. In the case of enforcing ASLAs, an IID distribution of the outputs allows us to apply the central limit theorem to derive an accuracy guarantee.

The second property of the map-based model, incremental computation, allows efficient computation of a subset of the output components. A randomly sampled subset of the output components allows us to reason about the statistical accuracy of the approximation without computing the entire output. This property is critical to building a system that enforces ASLAs for approximate computing, since all of the time spent enforcing accuracy guarantees directly undermines the benefits of approximation.

3.3.2 Guarantee Engine Overview

We apply the methodology outlined in Figure 3.4 to an approximate computation to enforce ASLAs. These ASLAs are guaranteed based on a user-specified map-based exact and approximate computation, an accuracy metric, and a confidence level. In addition to the exact and approximate computations being map-based, the approximate computation must be deterministic. This constraint is required to ensure that the behavior of approximation while analyzing the accuracy matches that of the final approximation applied to the application input. Given these conforming guarantee engine inputs, the guarantee engine hands back an accuracy guarantee. For example, with an ASLA confidence level $CL \in (0, 1]$ and an approximation X, this guarantee is a single number Y that takes the form of a statement such as *"with confidence level CL, approximation X has an accuracy of at least Y."*

The two main steps to enforcing ASLAs with NinjaProx are computing an accuracy distribution and generating an accuracy guarantee. The accuracy distribution is a randomly chosen set of accuracy samples, while the accuracy guarantee specifies an approximation accuracy that will meet the ASLA. In the following sections, we describe the workings of the guarantee engine in detail. For simplicity, throughout the description of the guarantee engine we assume that the accuracy metric is mean absolute percent error (MAPE). Subsequently, in §3.3.4.2 we describe support for a number of other common accuracy metrics supported by NinjaProx.

3.3.3 Computing an Accuracy Distribution

The guarantee engine first constructs an accuracy distribution for an approximation through the following steps. We show an example of these steps in Figure 3.5, illustrating an accuracy distribution computation for mean absolute percent error (MAPE) in the context of a small 8×8 result matrix.

1. **Random Sampling in the Result Space** – we first use simple random sampling (SRS) to select a subset of size n of the results to compute, yielding a set of indices in the result space. For notational convenience, we enumerate these randomly selected indices $1, 2, ..., n$. In Figure 3.5(a), the dark cells illustrate the components of the result that have been randomly selected.

2. **Exact Result Components** – we next compute results for the exact version of the problem for the subset of the result space chosen in the previous step, denoted $E_1, E_2, ..., E_n$. In Figure 3.5(b), the exact result components are filled in for the shaded cells.

3. **Approximate Result Components** – similar to the previous step, we enact the approximation to compute approximate result components for the same subset of the result space, denoted $A_1, A_2, ..., A_n$ and shown as shaded cells in Figure 3.5(c).

4. **Accuracy Components** – finally, given the exact and approximate result components for identical component indices of the result space, we compute the component accuracy of each such (exact, approximate) component pair $\epsilon_i = \delta(E_i, A_i)$, where δ is defined according to the accuracy metric being employed. For MAPE, the component accuracy function is shown in Equation 3.4. Figure 3.5(d) shows accuracy components as shaded cells.

$$\delta(x, y) = \left|\frac{x - y}{x}\right| \tag{3.4}$$

The result of these steps is the sampled accuracy distribution $S = \{\epsilon_1, \epsilon_2, ..., \epsilon_n\}$ of the approximation, telling us how accurate the approximate computation is on a randomly selected subset of the results.

3.3.4 Deriving an Accuracy Guarantee

The goal of our approach is to enforce the ASLA for an approximation. We cast this as the problem of deriving a statistical guarantee on the mean of some population

$\{X_i\}$ (we shorthand $\{X_i\}$ going forward as X), which maps directly to an ASLA for approximation accuracy.

A set random samples S taken from a set of variables $X_1, X_2, ..., X_n$ is treated as the component accuracy distribution of a particular approximation. This distribution is independent and identically distributed (IID) due to the statistical random sampling made possible by the out-of-order property of the map-based model (§3.3.1). To derive a guarantee on the mean of X, we employ the central limit theorem (CLT). The CLT states that for sufficiently large values of n, a set of random samples $S = \{\epsilon_1, \epsilon_2, ..., \epsilon_n\}$ taken from a set of variables $\{X_1, X_2, ..., X_n\}$, the expected value of the mean of S approaches a normal distribution parameterized by $\mu_X = \text{mean}(X)$, $\sigma_X^2 = \text{var}(X)$, and n. We summarize the definition of the CLT in Equation 3.5.

$$\text{mean}(S) = \lim_{n\to\infty} \frac{1}{n}\sum_{i}^{n} \epsilon_i \ \sim \ N(\mu_X, \frac{\sigma_X^2}{n}) \tag{3.5}$$

The CLT has two properties that are critical for our purposes. First, it provides a principled way to reason about the mean of *any* distribution. Second, it formulates the probability distribution of the mean as a normal distribution (the so-called "bell curve"), which has a number of well-understood features. Namely, we can derive a one-sided confidence interval for the mean of the accuracy distribution using standard statistical practices, yielding a probabilistic upper bound on its mean. Computing such a confidence interval only requires the accuracy distribution of the approximation to compute its sample mean $\widehat{\mu}_S$ and sample standard deviation $\widehat{\sigma}_S$, along with an ASLA confidence level CL supplied by the user, and can be computed using the standard formulation shown in Equations 3.6 and 3.7.

$$CI_{UB} = \widehat{\mu}_S + \frac{z * \widehat{\sigma}_S}{\sqrt{n}} \tag{3.6}$$

$$z := Pr(N(0,1) \leq z) = 1 - \text{CL} \tag{3.7}$$

This upper bound, CI_{UB}, is a statistical guarantee on the mean of S, telling us that with probability CL the mean of S is no higher than than CI_{UB}. This statistical guarantee translates directly to an accuracy guarantee that satisfies an ASLA – with confidence level CL, the accuracy of the approximation is no worse than CI_{UB}.

3.3.4.1 Finite Sample Correction

To minimize overhead of the guarantee engine, it is desirable to use a small number of samples to produce accuracy guarantees. However, the definition of the CLT states that the mean of the samples converges to a normal distribution as the number of samples approaches infinity. Instead of increasing the sampling rate until the model is precise, we correct for the difference in distribution between the theoretically-sound infinite number of samples and the pragmatically-usable finite number of samples. To make this correction, we use the Berry-Esseen theorem [7].

The Berry-Esseen theorem places a limit on the maximum distance between the cumulative density functions (CDF) of the theoretical normal distribution that results from applying the CLT and the distribution resulting from a finite set of samples. The formula expressing this maximum difference is provided in Equation 3.8.

$$|F_n(x) - \Phi(x)| \leq \frac{C\rho}{\sigma^3\sqrt{n}} \tag{3.8}$$

Here, $F_n(x)$ is the CDF of a finite-sample distribution, $\Phi(x)$ is the CDF of the infinite-sample distribution, ρ is the skewness of the component accuracy distribution, σ is the standard deviation, and n is the number of accuracy component samples applied to the CLT for the finite-sample distribution. The constant C in the Berry-Esseen theorem is part of ongoing research in the statistics community [57, 107, 108, 119]. In this work, we use $C = 0.4748$, the tightest theoretical bound known at the time of this writing for the upper limit of this constant [108]. Since the Berry-Esseen theorem provides an upper bound on the difference between the theoretical accuracy distribution and the actual sampled accuracy distribution, we employ this theorem to correct our derivation of the accuracy guarantee.

To make this finite sample correction, the confidence level is offset by the value specified by the Berry-Esseen theorem. Essentially, the confidence level represents the target quantile of the accuracy distribution, so adding $|F_n(x) - \Phi(x)|$ to the confidence level creates an accuracy guarantee that will enforce the ASLA in the worst-case random sampling given the finite number of samples used to construct the initial accuracy guarantee. Therefore, the accuracy guarantee with finite sample correction is provided in Equations 3.9 and 3.10.

$$CI'_{UB} = \widehat{\mu}_S + \frac{z' * \widehat{\sigma}_S}{\sqrt{n}} \tag{3.9}$$

$$z' := Pr(N(0,1) \leq z') = 1 - \left(\text{CL} + \frac{C\rho}{\widehat{\sigma}_S^3 \sqrt{n}} \right) \tag{3.10}$$

3.3.4.2 Other Accuracy Metrics

The NinjaProx guarantee engine includes out-of-the-box support for a number of commonly used accuracy metrics detailed in the remainder of this section – mean absolute percentage error, mean absolute error, mean square error, peak signal-to-noise ratio, and miss rate. NinjaProx can also be supplied with a user-defined accuracy metric of the form shown in Equation 3.11, where E_i and A_i are the exact and approximate output components.

$$\Delta \left(\sum \delta(E_i, A_i) \right) \tag{3.11}$$

This formulation of accuracy is general enough to support most commonly used metrics, since most accuracy metrics are a mean or sum of a comparison of output components, captured by the component accuracy function δ. In some cases, for example, in peak signal-to-noise ratio, a transformation is applied to the mean of component accuracies. This transformation is supported by the aggregate accuracy function, Δ. Here we describe the specific formulations used to underpin each of the supported error metrics.

Mean Absolute Percentage Error (MAPE). The detailed derivation of accuracy guarantees that was shown in Section 3.3.3 is based on MAPE.

Application	**Description**	**Domain(s)**	**Input Set**	**Approx. Technique**	**Accuracy Method**	**Accuracy Target**
`binarize`	Convert image to black and white	computer vision, OCR	Images	2D tiling	Miss rate	10%
`crosscorr`	Measure signal/image similarity over sliding window	Image processing, pattern recognition, cryptanalysis, neurophysiology	Images	2D tiling	Mean abs. error (MAE)	26
`gamma`	Apply gamma correction to an image	Image processing	Images	2D tiling	PSNR	20
`gaussian`	Apply a Gaussian filter to an image	computer vision, image smoothing	Images	2D tiling	PSNR	20
`inversek2j`	Inverse kinematics for 2-joint arm	Robotics	Arm positions	Truncated Taylor series	Mean abs. % error (MAPE)	10%
`jmeint`	Triangle intersection detection	3D gaming	Triangles	Perforation	Miss Rate	10%
`matmult`	Matrix-matrix multiply	Machine learning, scientific computing, game theory	Matrices	Extrapolated perforation	Mean abs. % error (MAPE)	10%
`sobel`	Sobel edge detection	Computer vision, edge detection	Images	2D tiling	PSNR	20

Input Set	**Description**
Images	A database of 100 assorted 1024x1024 images
Matrices	102 1024x1024 matrices; **Sparsity (3):** 25%, 50%, 100%; **Shapes (2):** full, upper triangular **Probability distributions (17):** 2x beta, 1x binomial, 2x chi-squared, 1x exponential, 2x f, 1x gamma, 1x geometric, 1x hyper, 1x log-normal, 1x normal, 2x poisson, 1x uniform, 1x weibull
Triangles	1×10^6 pairs of triangles randomly placed into unit cube; **Sizing:** 99 different size distributions
Arm positions	1×10^6 arm positions at different angles; **Angles:** drawn from 98 probability distributions

Table 3.1: Applications and input sets used in the evaluation.

Mean Absolute Error (MAE). For MAE it suffices to supply the following component accuracy function in place of the definition given for MAPE in Equation 3.4. This change to the component accuracy function places the accuracy components in the sum of the MAE, allowing use of the CLT.

$$\delta(x, y) = |x - y| \tag{3.12}$$

Mean Square Error (MSE). Similarly, for MSE we swap the following component accuracy function for Equation 3.4.

$$\delta(x, y) = (x - y)^2 \tag{3.13}$$

Peak Signal-to-noise Ratio (PSNR). Peak signal-to-noise ratio is a common metric used to measure image quality. We provide the definition of PSNR in Equation 3.14 for reference.

$$\text{PSNR} = 10 * log(MAX^2) - 20 * log(MSE) \tag{3.14}$$

Note that in the definition of PSNR, MAX is a domain-specific parameter, which describes the maximum possible value that can be assumed in that domain (e.g., 255 for an image with 8-bit color). In cases where PSNR is the accuracy metric specified to NinjaProx, a value for MAX be specified along with the selection of this metric.

To apply our methodology when using PSNR, we use the component accuracy function for MSE shown in Equation 3.13, since the accuracy components that are averaged match those of the MSE metric. We then directly derive an upper bound on the confidence interval for the normal distribution derived from the CLT, just as is done in Equations 3.6 and 3.7. However, instead of using that bound directly, we plug it into the definition of PSNR to yield a lower bound on the confidence interval of PSNR[1]. As shown in Equation 3.15, the aggregate accuracy function, in our formulation of accuracy metrics, becomes the definition of PSNR.

$$\Delta(x = \widehat{\mu}_S + \frac{z' * \widehat{\sigma}_S}{\sqrt{n}}) = 10\ log\left(MAX^2\right) - 20\ log\left(x\right) \tag{3.15}$$

Miss Rate. Formulations of accuracy based on miss rate capture an important class of problems where the component accuracy is binary. This is the case for

[1]An upper bound on MSE yields a lower bound on PSNR due to the subtraction operation in the definition of PSNR. This is the desired result, as higher values of PSNR are better.

applications with binary output components or, more generally, where the output components are either entirely correct or incorrect depending on a component accuracy threshold. The former scenario applies to two of the applications used in our evaluation: `binarize` and `jmeint`. The `binarize` image filter, used in optical character recognition, converts a color or grayscale image into a black and white image. Approximations to `binarize` are, at the pixel level, either correct or incorrect. Similarly, `jmeint` computes whether pairs of triangles overlap one another, and approximations to that computation yield an answer that is either correct or incorrect for each triangle pair. We use a generalized formulation of miss rate, where each component is correct (0) if the absolute difference is less than some threshold T, and incorrect (1) otherwise. NinjaProx allows for the parameter T to be specified and defaults to using 0 when T is not specified.

$$\delta(x, y, T) = \begin{cases} 0 & \text{if } |x - y| \leq T \\ 1 & \text{otherwise} \end{cases} \tag{3.16}$$

Replacing the component accuracy calculation shown in Equation 3.4 with the function shown in Equation 3.16 suffices to allow our CLT-based techniques to apply to miss rate.

Weighted Accuracy Metrics. For certain applications, the position of the output components within the output space may be important for computing accuracy. For example, it may be the case that the pixels near the borders of an image are less important than those that are centrally located. To address this situation, the NinjaProx guarantee engine exposes the position of the output component to the component accuracy function. This position-aware formulation, δ_p, is provided in Equation 3.17, where the added parameters, w, p, and δ, represent the position weight function, the position within the output, and the position-unaware component accuracy function, respectively.

$$\delta_p(w, p, \delta, x, y) = w(p) * \delta(x, y) \tag{3.17}$$

3.4 Evaluation

In this section we evaluate the accuracy and performance characteristics of NinjaProx.

3.4.1 Experimental Methodology

Applications and Approximations We employ four approximation techniques from the literature upon eight test applications as follows. A summary of the approximations and applications can be found in Table 3.1.

- **Tiling** [97] is based on the assumption that, in application domains such as image and video processing, elements nearby one another (e.g., image pixels) are likely to have similar values. Instead of computing each element of the output, a tiling approximation computes a single output element and projects that output value onto the surrounding elements to form a tile. Tiling can be tuned to be more *aggressive* – trading off lower accuracy for better performance – by increasing the size of the tile. In this work, we employ 64 different tiling approximations with tile sizes $\{2, 4, 8, ..., 256\} \times \{2, 4, 8, ..., 256\}$ on `binarize`, `crosscorr`, `gamma`, `gaussian` and `sobel`.

- **Perforation** [47] discards iterations in a loop, and can be used to trade improved performance for lower accuracy by dropping more iterations. In this work we use *modulo* perforation, which discards iterations at regular intervals. Modulo perforation is parameterized by a rate r, where $r > 0$ indicates that every rth iteration is computed and a value of $r < 0$ indicates that every rth iteration is skipped. We use 10 such rates on `jmeint`, where $r = \{-8, -4, 2, 4, 8, ..., 256\}$.

- **Extrapolated perforation** [93] discards iterations in a loop while extrapolating results to correct for skipped iterations. In the `matmult` application, we use the following to compute each result `C[i][j]` in the exact implementation:

  ```
  for (k = 0; k < N; k++)
     C[i][j] += A[i][k] * B[k][j];
  ```

  We use the following extrapolated loop perforation as an approximation:

  ```
  for (k = 0; k < N; k += r)
     C[i][j] += A[i][k] * B[k][j];
  C[i][j] *= r;
  ```

  This approximation is made more aggressive by using larger values of `r`. We use $r = \{2, 4, 8, ..., 256\}$ on `matmult`.

- **Truncated Taylor series** [43] uses a small number of terms from the Taylor series of a mathematical expression in place of the more expensive and accurate computational method in `glibc`. For `inversek2j`, in place of calls to the `sin` and `cos` routines in `glibc` with, we use the following approximations for the sine and cosine functions, respectively:

```
float apx_sin(float x){ return x; }
float apx_cos(float x){ return 1-x*x/2; }
```

These approximations provide high accuracy for small angles, but become increasingly poor as the input angle gets larger. Therefore we apply the approximation for angles smaller than a threshold t and increasingly aggressive approximations by using larger thresholds. On `inversek2j` we use 10 such thresholds: $\{\frac{\pi}{10}, \frac{2\pi}{10}, ..., \frac{9\pi}{10}, \pi\}$.

Accuracy SLAs. In approximating applications with NinjaProx in the evaluation, we specify a 1-statement ASLA for each application consisting of an accuracy metric, an accuracy target and a confidence level. We employ four different error metrics across the applications, many of which cover different accuracy ranges. For example, peak signal-to-noise ratio (PSNR) can assume a value in $[0,\infty)$, while miss rate assumes a value in [0,1]. For three of our metrics, including miss rate, the metric can assume a particular fixed range in a given application. In such cases, we choose an accuracy target corresponding to 90% of perfect accuracy. For PSNR we define the accuracy target based on prior work that has identified acceptable accuracy for wireless transmission protocols [70, 118]. Table 3.1 describes the accuracy metrics and targets for each application. We configure NinjaProx to use a flat 0.1% of the input when generating accuracy guarantees and use confidence levels of 90%, 95%, 99%, and 99.9% when constructing guarantees; the particular confidence levels used in each experiment are discussed in the context of the particular experiments.

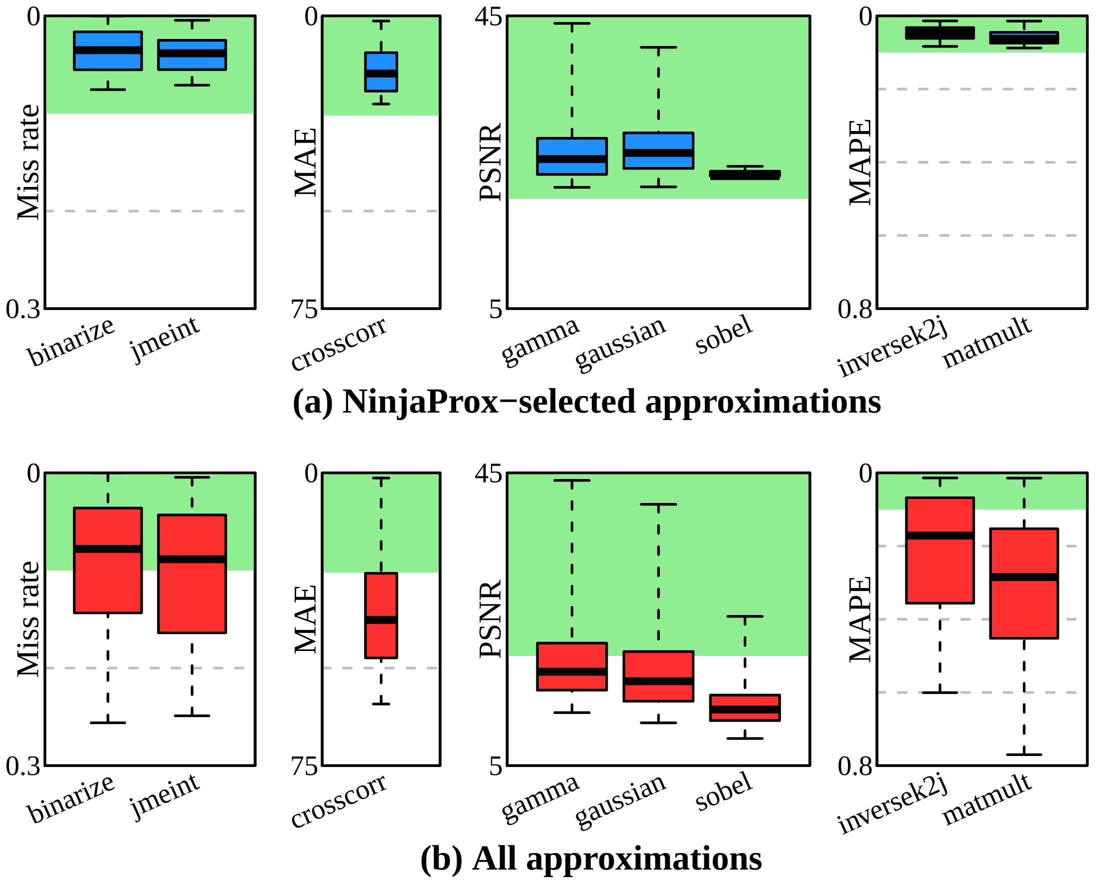


Figure 3.6: A comparison between the accuracy resulting from approximations that pass the NinjaProx guarantee engine (CL=90%) to the accuracy of all approximations, showing that the guarantee engine catches all approximations that do not conform to the specified ASLA.

Miscellaneous. All of our experimental results are collected on an Intel Xeon E3-1240v3 Haswell machine running Ubuntu Linux 14.04 (Linux kernel 3.19.0-59). The performance results presented are averages across 10 runs.

For a fixed input and approximation, NinjaProx makes a statistical accuracy guarantee. These statistical accuracy guarantees are produced using a random sampling

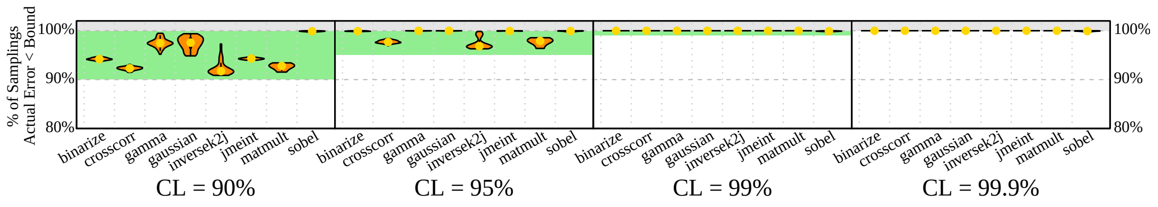


Figure 3.7: Distributions of how frequently the accuracy guarantee meets the accuracy target across all applications, inputs and approximations and 4 confidence levels of 90%, 95%, 99% and 99.9%. By observing that all such experiments are above the specified CL (i.e., that the entire violin is within the shaded region), we can verify that the statistical guarantees made by the NinjaProx guarantee engine work as expected.

of the output space for the given input and approximation. The generated guarantee accuracy for a specific random sampling will be above the actual accuracy with a probability greater than the user-specified confidence level, so each fixed input and approximation pair must be evaluated many times to determine whether or not the statistical accuracy guarantee is met. To capture the variability of random sampling, the accuracy measurements use 1000 runs (often presented as distributions).

3.4.2 Accuracy Guarantees

We first evaluate the *guarantee engine* within NinjaProx. The goal of the guarantee engine is, given a confidence level CL specified in the ASLA and a candidate approximation, to provide a statistical accuracy guarantee of the approximation.

Accuracy Guarantee Quality. We start evaluating the NinjaProx guarantee engine by investigating the accuracy of approximations that the guarantee engine deems to be acceptable with a confidence level of 90%. In this experiment, we run each (input, approximation) pair once to measure the actual accuracy of the approximation, then run the (input, approximation) through the guarantee engine 1000 times to construct an accuracy guarantee distribution. Figure 3.6(a) presents the accuracy guarantee distribution for the approximations that the guarantee engine deems acceptable, in the form of a box plot with whiskers at the 0th and 90th percentiles. The green shaded region represents the cases where the generated guarantee meets the accuracy. Since the confidence level is set to 90%, we expect, and find, that the lower whisker in Figure 3.6(a) (the 90th percentile) is above the output accuracy target.

For comparison, we present Figure 3.6(b), which is produced using the same methodology as the previous experiment, except all (input, approximation) pairs are shown, representing a baseline scenario where no guarantee engine or other mechanism is present to determine whether each approximation meets the accuracy target. In this figure, we see that a wide range of output accuracies are produced across the set of (input, approximation) pairs. The key takeaway from this comparison is that NinjaProx's guarantee engine catches *all cases* within the 90% confidence level where the target accuracy cannot be met by an approximation, showing that the NinjaProx guarantee engine provides high quality accuracy guarantees.

Enforcing ASLAs. As the ASLAs enforced by NinjaProx are probabilistic in nature, we evaluate the ability of the guarantee engine to meet the ASLAs by analyzing how frequently the approximations that are deemed to result in acceptable accuracy fall below the accuracy target. Our experimental setup to do this evaluation comprises running each (input, approximation) pair through the NinjaProx guarantee engine 1000 times. Each of these 1000 experiments for a given pair represent dif-

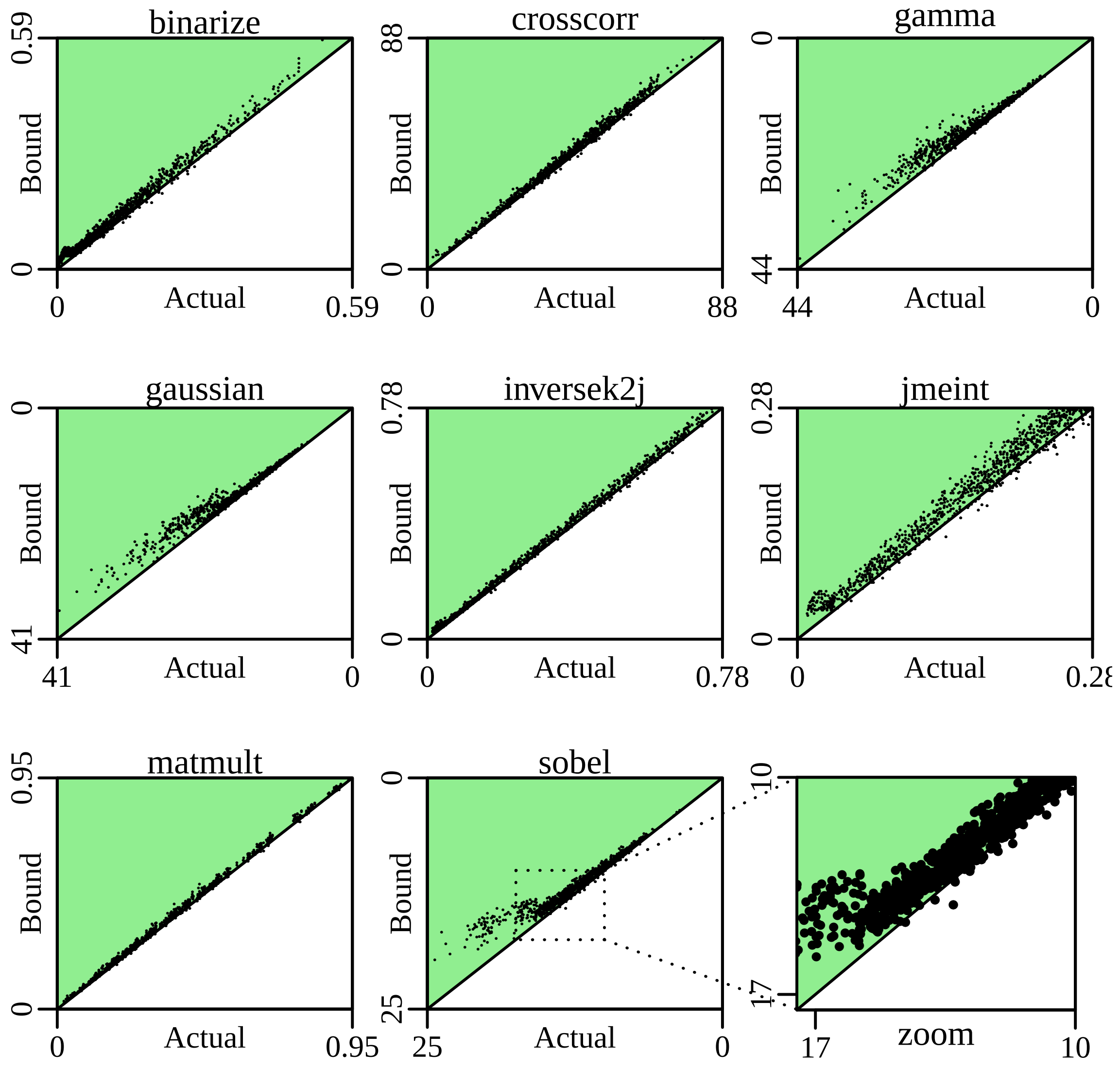


Figure 3.8: Scatter plots comparing actual accuracy (x-axis) to guarantee accuracy (y-axis) for a confidence level of 90%. A point in the green shaded region indicates a guarantee that achieves more accuracy than the target. The white regions should, and do, contain less than 10% of the points in each plot, representing a maximum of 10% outside of the `CL=90%` confidence interval. The accuracy guarantees heavily skew toward the shaded region and closely track the actual accuracy.

ferent random samplings of the result space, capturing the distribution of possible accuracy guarantees produced by the NinjaProx guarantee engine. The metric is then the percentage of accuracy guarantees that falls short of the actual accuracy.

Figure 3.7 presents a violin plot that shows how frequently the accuracy target is met as a distribution across each of the approximation methods for confidence level $CL \in \{90\%, 95\%, 99\%, 99.9\%\}$. In all cases, we are looking to verify that all points lie above CL (i.e. the entire box sits in the shaded green region in the figure), since the accuracy guarantee is a statistical guarantee with confidence of CL. In all cases, the output quality violation rate is above the CL specified in the ASLA, showing that the NinjaProx guarantee engine can strictly enforce ASLAs for a range of confidence levels.

Accuracy Guarantee vs. Actual Accuracy. Figure 3.8 shows a detailed illustration of how the accuracy guarantee compares to actual accuracy for all (input, application) pairs at a confidence level of 90%. In these plots, each point represents one approximation on one input, where the position on the x-axis is the actual accuracy of the approximation and the position on the y-axis is the accuracy guarantee generated by the guarantee engine. The shaded area in each plot shows points for which the actual accuracy is lower than the accuracy guarantee. Two important observations can be made from these plots. First, the points skew heavily toward the shaded region, corresponding with the large fraction of tests where the accuracy guarantee holds. The NinjaProx statistical accuracy guarantee requires that 90% of these points are in the shaded region, since the system is configured for `CL=90%`, which is shown to be the case in the `CL=90%` portion of Figure 3.7.

The second observation is that the accuracy guarantee generated by the NinjaProx guarantee engine tracks the actual accuracy of the approximation very closely. This is important because an accuracy guarantee that is not closely linked to the actual accuracy will be more likely to cause the NinjaProx decision engine to (incorrectly) identify the approximation as being so inaccurate as to not be viable to meet the accuracy target specified in the ASLA. We examine how closely the guarantee and actual accuracy track each other by calculating the coefficient of determination (also called R^2) of the accuracy guarantee versus the actual accuracy for the experiments shown in Figure 3.8. The coefficient of determination is a unitless indicator of how well the accuracy guarantee models the actual accuracy, and can assume a value between 0 and 1, with 1 being a perfect model. The average value across all 8 applications is 0.987, while the lowest value is for `sobel`, which has a value of 0.962 and indicates a superb fit between the generated guarantee and the actual accuracy.

3.4.3 End-to-end Approximation System

Utilizing the guarantees produced by the guarantee engine, NinjaProx employs a decision engine to dynamically choose from among the available approximations in a way that complies with the ASLA. To evaluate NinjaProx, we perform 10 runs of each application with each input and present the average results. Recall that the ASLA consists of both an accuracy target and a confidence level. Together, these parameters instruct NinjaProx's decision engine to choose an approximation that meets a particular accuracy target with a particular confidence. In our evaluation

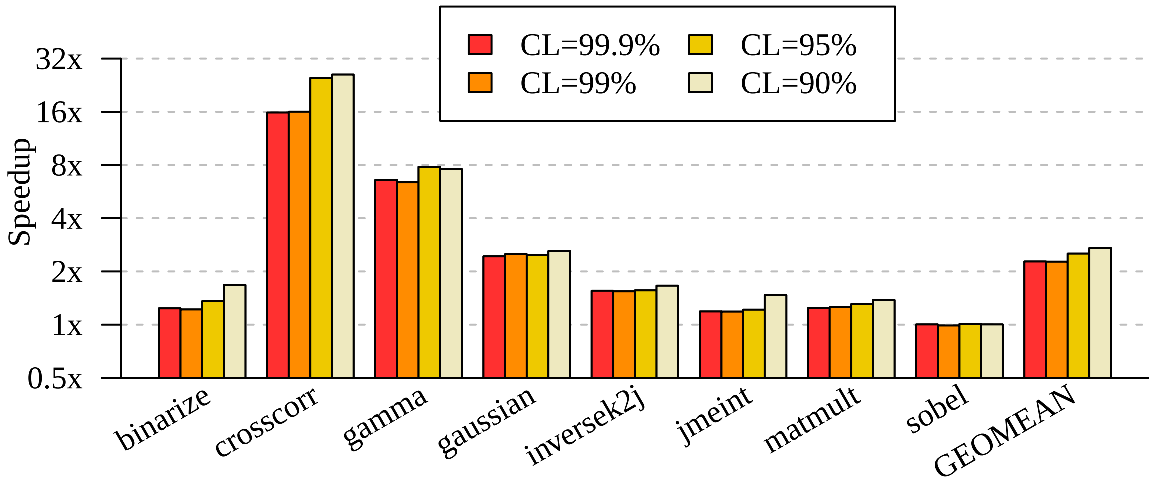


Figure 3.9: Speedup achieved per application across all inputs for the NinjaProx runtime system with $\texttt{CL} = \{90\%, 95\%, 99\%, 99.9\%\}$, showing that less strict ASLAs allow for higher performance.

of the NinjaProx runtime, we use the accuracy targets described in Table 3.1 for each application, along with $CL \in \{90\%, 95\%, 99\%, 99.9\%\}$. That is, we supply constraints that instruct the runtime to target an aggressive approximation roughly equivalent to the accuracy targets employed in prior work [47,97,98] and to hit those targets with varying confidence.

NinjaProx Performance. We begin by showing the overall speedups realized when approximating our test applications with NinjaProx across four ASLA confidence levels in Figure 3.9. These speedups encompass the full execution time of the application running with NinjaProx, including the time taken by the decision engine to search for the best approximation, which involves calling the guarantee engine and constructing accuracy guarantees for a number of approximations. We find that NinjaProx is able to choose aggressive approximations for all of the supplied confidence levels, with a geometric mean ranging from 2.3× for CL=99.9% to 2.7× for

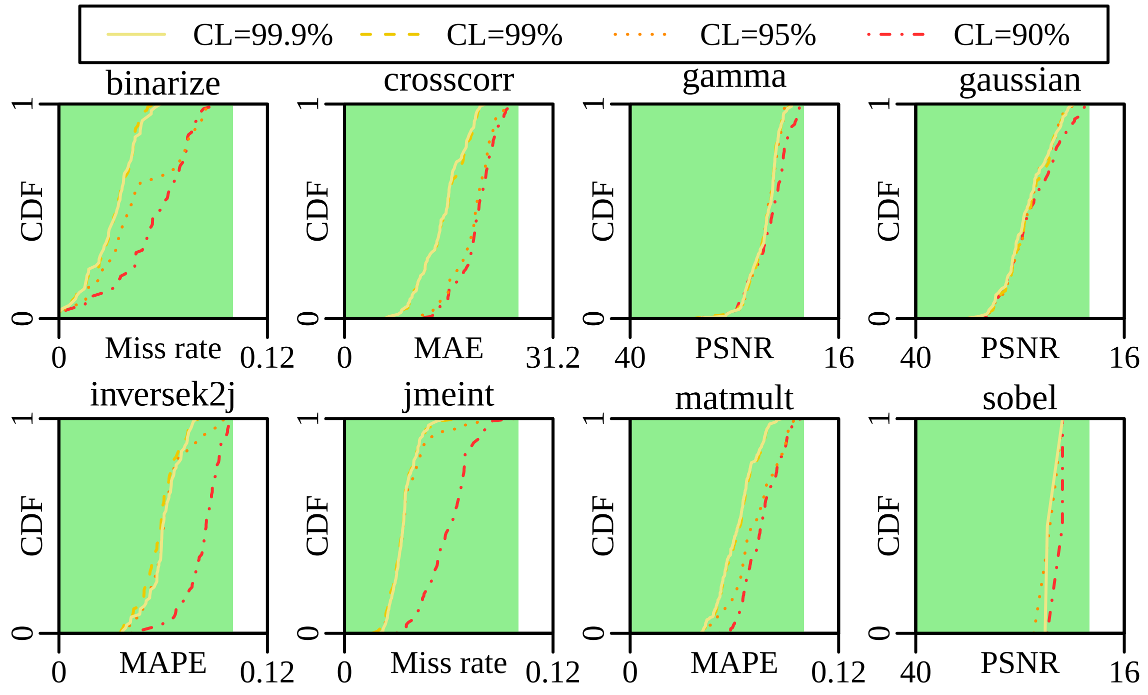


Figure 3.10: CDFs of accuracy achieved for all application inputs when allowing NinjaProx to choose approximations, showing that accuracy targets are always met and that the accuracy approaches the target output quality limit as a less strict confidence level is used in the ASLA.

CL=90%.

Runtime Accuracy. We now evaluate the accuracy of the NinjaProx runtime system when configured for various confidence levels. Figure 3.10 shows a CDF of the accuracy achieved for each of an application's inputs when running the application approximately with NinjaProx for a sweep of confidence levels. The accuracy of each applications is provided on the x-axis, with the shaded green region indicating those final accuracy levels that are at least as good as the accuracy target specified in the ASLA. In all cases, the approximation chosen by the NinjaProx runtime system meets the target accuracy. However, the worst-case accuracy approaches the output quality boundary because of the aggressive approximations that NinjaProx is able to choose. We can also see in the figure that, as the confidence level becomes more strict, the accuracy CDF becomes further from violating the output quality constraint. Therefore, NinjaProx is able to choose aggressive approximations while providing a tunable ASLA.

3.4.4 Runtime Analysis

In Figure 3.11, we divide the runtime spent when approximating applications with the NinjaProx system into 3 categories: time spent running the selected approximation, time spent searching for the approximation (including the time spent computing the accuracy distributions and guarantees) and the amount of time saved by approximating the application. Our results show the average time of 10 runs across all inputs for each application. From the yellow region of the stacked bar graph, we observe that, in all cases, the overhead of generating statistical accuracy guarantees is very low. Even with statistical guarantees, we find that NinjaProx substantially reduces the total execution time (orange region). We conclude that the low overhead of generating accuracy guarantees provides a compelling mechanism for selecting approximations.

3.4.5 Comparison to Static Oracle

We next compare NinjaProx to a Static Oracle approach to approximation. This oracle is constructed first by measuring the speedup and accuracy of all available

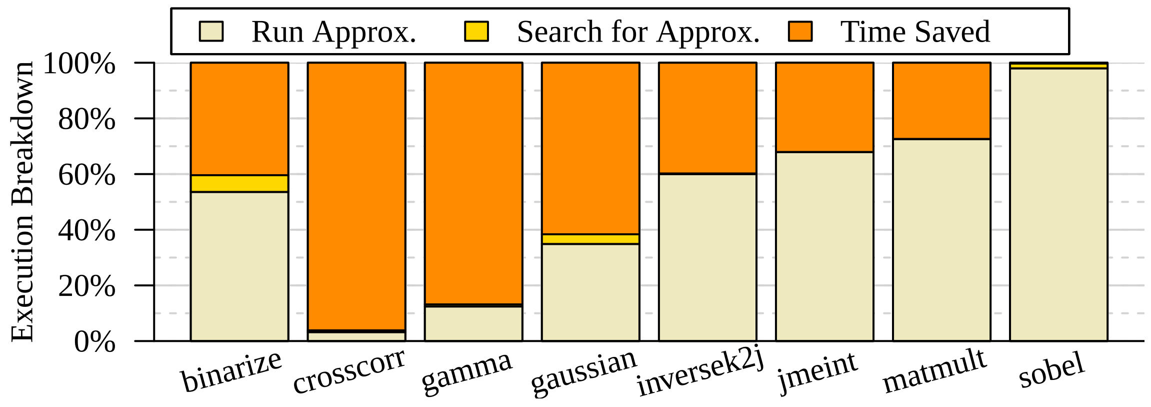


Figure 3.11: Time breakdown of the NinjaProx runtime system, showing very low overhead for selecting approximations and significant time-to-solution improvements.

approximations across all inputs for each application, then by allowing the oracle to choose the single approximation technique that results in no violations of the accuracy target, thereby achieving an identical number of accuracy violations as NinjaProx. This oracle represents a highly idealized scenario, in which all inputs are known a priori and can be tractably measured before choosing how to approximate the application.

Figure 3.12 presents the speedups achieved by this oracle. In some cases, as in `inversek2j` and `sobel`, the speedup is similar to the speedup achieved by NinjaProx. However, for a number of applications – `binarize`, `gamma`, `gaussian`, `jmeint` and `matmult` – the speedups achieved by NinjaProx are significantly larger and for several applications – `binarize`, `gamma`, `gaussian` and `matmult` – no speedup is achieved by the oracle. This highlights the benefit of *dynamically* choosing how to approximate: different inputs behave differently, and some can be drastically harder to approximate than others. Dynamic techniques can take advantage of this

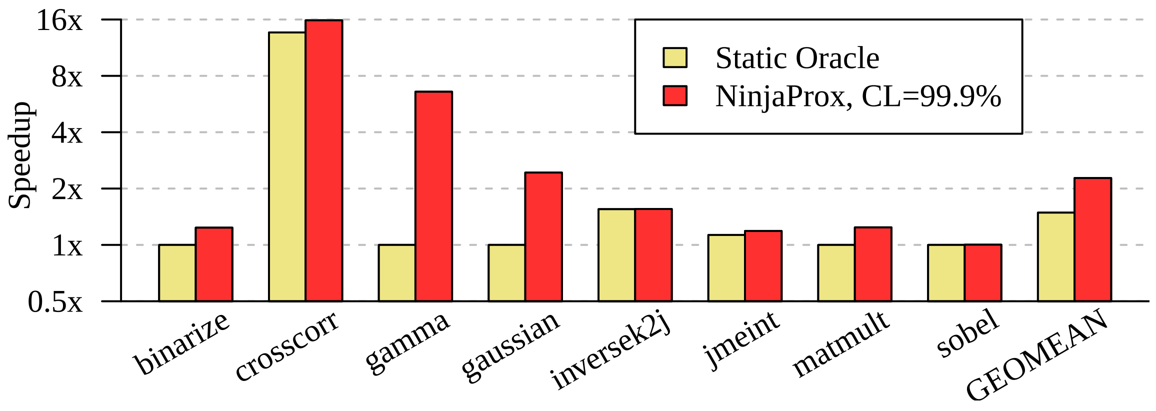


Figure 3.12: Speedup achieved by NinjaProx for `CL` = 99.9% compared to a static oracle, showing that NinjaProx outperforms the static oracle. The accuracy is above the target accuracy in all cases for both techniques.

fact to aggressively approximate certain inputs and conservatively approximate others.

3.5 Summary

This chapter introduces the first approach to address the challenging problem of guaranteeing the accuracy of approximation results for a broad class of map-based computational problems, developing the statistical machinery to provide accuracy guarantees when approximating such applications regardless of the input content and desired accuracy level. This mechanism builds on the insight that the computation in map-based applications can be performed both out-of-order and incrementally. Building on this mechanism, we introduce and describe enforcement techniques for a new class of service level agreement (SLA) called an *Accuracy SLA (ASLA)*. We show that NinjaProx can achieve significant application performance improvements that average 2.5× while providing strong probabilistic guarantees of high result quality.

CHAPTER IV

DeftNN: Addressing Bottlenecks for DNN Execution on GPUs via Synapse Vector Elimination and Near-compute Data Fission

Deep neural networks (DNNs) are key computational building blocks for emerging classes of web services that interact in real time with users via voice, images and video inputs. Although GPUs have gained popularity as a key accelerator platform for deep learning workloads, the increasing demand for DNN computation leaves a significant gap between the compute capabilities of GPU-enabled datacenters and the compute needed to service demand.

The state-of-the-art techniques to improve DNN performance have significant limitations in bridging the gap on real systems. Current network pruning techniques remove computation, but the resulting networks map poorly to GPU architectures, yielding no performance benefit or even slowdowns. Meanwhile, current bandwidth optimization techniques focus on reducing off-chip bandwidth while overlooking on-chip bandwidth, a key DNN bottleneck.

To address these limitations, this chapter introduces DeftNN, a GPU DNN execution framework that targets the key architectural bottlenecks of DNNs on GPUs to automatically and transparently improve execution performance. DeftNN is composed of two novel optimization techniques – (1) synapse vector elimination, a technique that identifies non-contributing synapses in the DNN and carefully transforms data and removes the computation and data movement of these synapses while fully utilizing the GPU to improve performance, and (2) near-compute data fission, a mechanism for scaling down the on-chip data movement requirements within DNN computations. Our evaluation of DeftNN spans 6 state-of-the-art DNNs. By applying both optimizations in concert, DeftNN is able to achieve an average speedup of 2.1× on real GPU hardware. We also introduce a small additional hardware unit per GPU core to facilitate efficient data fission operations, increasing the speedup achieved by DeftNN to 2.6×.

4.1 Challenges

In this section, we describe the key ideas and challenges in applying real-system, GPU-based optimizations to DNNs.

4.1.1 Computation Elimination

Network pruning [40, 41] has been proposed to remove non-contributing synapses and neurons by removing those with near-zero values. These removed computations

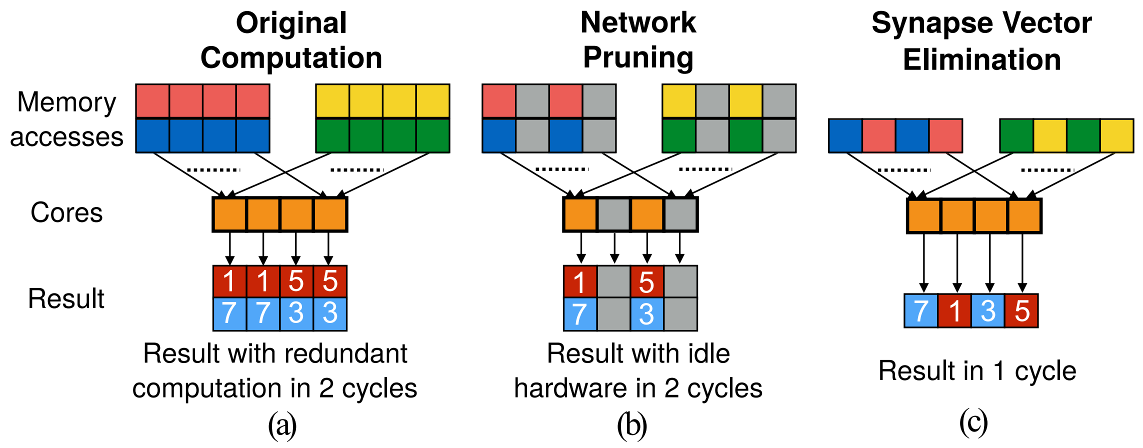


Figure 4.1: (a) Original DNN computation resulting in redundant computation, (b) network pruning [40, 41] resulting in underutilized hardware, and (c) synapse vector elimination showing efficient use of resources.

occur sporadically throughout the DNN topology, limiting benefits on commodity architectures.

GPU hardware, requiring contiguous data structures for efficient execution, presents a significant challenge when omitting arbitrary neurons or synapses. Specifically, for GPUs, branch divergence [34] and uncoalesced memory access [49] present two performance pitfalls for execution on noncontiguous data structures:

1. **Branch divergence** is where some of the threads, partitioned into groups by hardware (e.g., warps in CUDA), need to execute different instructions than the other threads in its group [80]. The hardware is designed such that all of the threads in a group execute instructions in lockstep. This requires that divergent sections of code are executed sequentially, so omitted computation that occurs irregularly due to noncontiguous data structures results in idle hardware rather than more efficient execution.

2. **Uncoalesced memory access** is a similar issue in the memory subsystem [49]. When multiple threads in a thread group issue memory instructions, requests to consecutive addresses, a result of contiguous data structures, can be grouped together to utilize a wide memory bus. Values stored in noncontiguous data structures are unlikely to have consecutive addresses, causing substantial under-utilization of the memory bus.

The challenge faced by network pruning is illustrated in Figure 4.1. An example baseline computation is presented in Figure 4.1(a). The original computation takes two cycles to complete because there are eight inputs with four being completed each cycle. Figure 4.1(b) shows the computational pattern produced by network pruning where uncoalesced memory accesses, due to the sparse, noncontiguous data structure, prevent high utilization of the arithmetic cores. Therefore, it still takes two cycles to process four inputs, since only two inputs are being processed per cycle. In practice, this kind of sparse computation results in very poor hardware utilization because it fails to take advantage of the GPU's very wide vector units. To validate the necessity of addressing this challenge, we compare the performance of DNN inference subjected to contiguous and noncontiguous data structures. In our real-system GPU experiments, we observe that applying noncontiguous data structures, produced by network pruning, to DNN inference results in a slowdown of 61× (§4.4.8).

To efficiently reduce the DNN topology, we propose synapse vector elimination, which improves DNN performance during inference by discovering and removing performance-exploitable non-contributing synapses. Synapse vector elimination overcomes the sparsity challenge by applying dynamically transformed input data to the

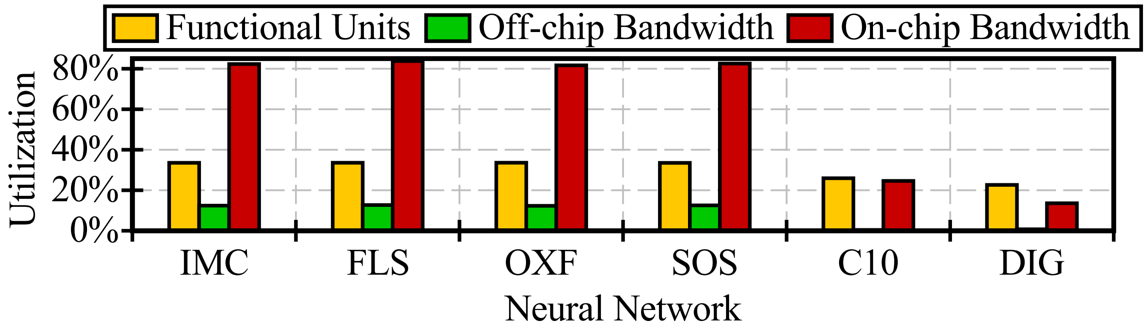


Figure 4.2: GPU utilization when processing DNNs, showing the on-chip memory bandwidth bottleneck.

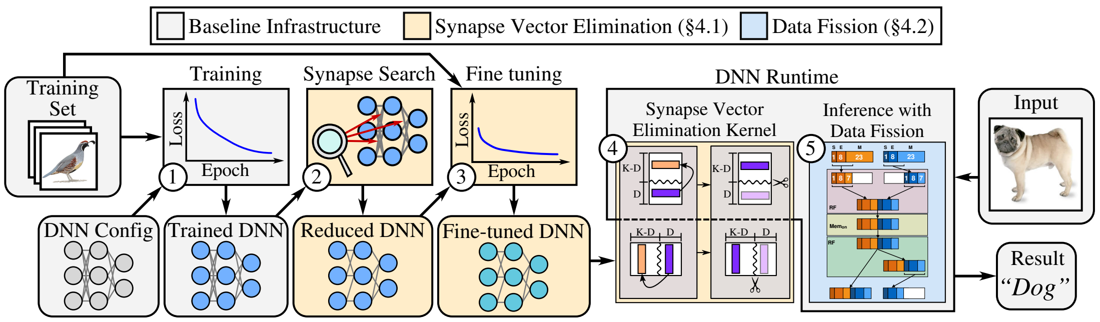


Figure 4.3: Overview of the DeftNN framework.

original hardware-efficient computation. As shown in Figure 4.1(c), synapse vector elimination reduces the total execution time by efficiently utilizing hardware resources on the transformed input. As shown in the diagram, our methodology results in only one cycle to process four inputs, since the removed and retained data is not interleaved. More details on our synapse vector elimination technique are presented in §4.3.1.

4.1.2 On-chip Memory Bandwidth

In contrast to reducing the amount of work with synapse vector elimination, another approach to achieve speedup is to alleviate the DNN processing bottleneck on GPUs by effectively exchanging one hardware resource for another. There are three main hardware resources on a GPU that are susceptible to becoming a bottleneck: the functional units, the off-chip memory bandwidth, and the on-chip memory bandwidth. We present kernel-weighted average utilization metrics of these three components in Figure 4.2, which were produced by profiling 6 state-of-the-art DNNs (application details are presented in §5.1), using Caffe [52] and running on an Nvidia Titan X (Pascal) GPU.

The key takeaway from the figure is that the system is greatly limited by on-chip memory bandwidth. This utilization profile is a result of optimized matrix multiplication, the main underlying GPU kernel for DNN inference, which makes use of loop tiling [20]. Loop tiling optimization allows on-chip memory storage and registers to be traded for off-chip memory bandwidth and on-chip memory bandwidth, respectively. While there is sufficient on-chip memory storage to sufficiently reduce off-chip memory bandwidth, the on-chip memory bandwidth remains a bottleneck due to the limited number of registers available for loop tiling.

As an example, the state-of-the-art Titan X (Pascal) GPU provides 11 single-precision TFLOPS (i.e. 44 TB/s), but its on-chip memory bandwidth is limited to 3.6 TB/s ($frequency \times \#\ shared\ memory\ banks \times bus\ width = 1$ GHz $\times$ 28 banks $\times$ 128 bytes) [87]. While loop tiling at the register level mitigates this throughput gap, on-chip memory bandwidth is still the limiting resource due to the limited number of registers available for tiling.

To alleviate the on-chip memory bandwidth bottleneck, unused functional unit cycles can be leveraged to compress on-chip memory. Unfortunately, the most recent GPU memory compression technique only applies to off-chip memory [98]. This technique works by compressing the data in off-chip memory, while storing the decompressed data in on-chip memory. Although this can reduce off-chip memory bandwidth, it provides no benefit for DNNs because on-chip memory bandwidth is the performance bottleneck.

Moving existing memory compression techniques closer to the functional units is more complex than simply applying the compression technique at a different place in the memory hierarchy. The central challenge when moving the compressed data closer to the compute is that the decompression overhead can outweigh the gains of reduced memory bandwidth and storage. The bandwidth for on-chip memory, however, is much greater than that of off-chip memory, making the size of the compressed data format less critical. The differences in proximity to functional units and available bandwidth cause a fundamental shift in the compression design space. While off-chip data packing focuses on larger reductions in memory bandwidth, a solution to this problem for DNNs must focus on minimizing decompression overhead.

Our near-compute data fission technique mitigates the GPU bottleneck in the system by targeting on-chip memory bandwidth. It realizes speedup by treating fission overhead as the paramount characteristic of the design. More details on our near-compute data fission technique are presented in §4.3.2.

4.2 System Overview

In this section, we present an overview of the DeftNN system, a GPU DNN execution framework for optimizing DNN inference by tailoring it to the underlying architecture. The two optimizations, synapse vector elimination and near-compute data fission, are built upon a standard DNN software framework, comprising an offline training phase for optimizing the DNN topology and a runtime system. Together, the offline and runtime systems work in concert to apply optimizations automatically and transparently to unmodified DNN applications. An overview diagram of the DeftNN framework is presented in Figure 4.3.

① **Initial Training** – First, as in all DNN execution frameworks, a set of training inputs are used along with a DNN configuration that specifies the topology of the DNN. Using the training inputs, the DNN parameters are adjusted iteratively until the classification loss function converges. This process produces a trained DNN model.

② **Synapse Search** – After producing the baseline trained DNN model, DeftNN automatically performs a synapse search to find the non-contributing synapse vectors – groups of synapses that are architecturally efficient to eliminate on the GPU. This process, as detailed in §4.3.1.2, locates and removes non-contributing synapse vectors, which we define as any vector highly correlated with another vector. As illustrated in the figure, the synapse search results in a DNN model that has some set of synapse vectors eliminated from the computation.

③ **Fine Tuning** – Although the retained synapse vectors are chosen to be representative of those that were eliminated, the nuanced impact of the missing, eliminated synapse vectors can result in accuracy degradation if used directly. To remedy this, DeftNN uses fine tuning, a well-known technique used to refine the DNN weights by applying a small number of DNN training iterations [122]. This process allows the DNN model to fully recover accuracy that is lost from minor perturbations of the weights or topology. By using fine tuning after applying synapse vector elimination, DeftNN produces a DNN model with negligible loss in inference accuracy.

④ **Synapse Vector Elimination** – Beyond the training mechanism used by DeftNN to produce an efficient DNN model, DeftNN services DNN applications using a runtime system that seamlessly allows inputs formatted for the unoptimized DNN model to be applied to the DNN model from synapse vector elimination. The synapse vector elimination kernel, as shown in the figure, reorganizes input activation values prior to inference so that they can be applied to the optimized DNN model. A detailed description of the architecture-efficient synapse vector elimination kernel are provided in §4.3.1.1, though we note that the reorganization takes a maximum of 5% of kernel execution time (see Figure 4.6), an overhead dwarfed by the reduction in computation facilitated by synapse vector elimination.

⑤ **Near-compute Data Fission** – In addition to reducing the size of the DNN using synapse vector elimination, DeftNN optimizes the key GPU bottleneck, on-chip memory bandwidth, within the inference kernel using near-compute data fission. Near-compute data fission packs DNN weights and activations into on-chip memory by removing non-contributing bits from the numerical representation. Since this technique resides in the low-level computational DNN kernels at run-time, no further changes are required to the baseline infrastructure to utilize this optimization. A detailed description of near-compute data fission is presented in §4.3.2.

4.3 Optimization Techniques

In this section, we describe two novel optimization techniques, synapse vector elimination and near-compute data fission. We present the key insights and challenges of both techniques when implemented on real GPU systems executing DNN workloads. In addition to our real-system implementation, we observe that near-compute data fission is amenable to acceleration and design a lightweight GPU hardware extension to mitigate overhead. Synapse vector elimination has minimal overhead when implemented in software, not warranting the costs of additional hardware.

4.3.1 Synapse Vector Elimination

Synapse vector elimination removes non-contributing synapses from DNNs, thereby reducing the total computation required for the DNN to process its inputs. Previous network pruning techniques produce an inefficient mapping of operations to hardware. Specifically, these techniques modify the computational kernel to be irregular, limiting performance benefits due to branch divergence and uncoalesced memory

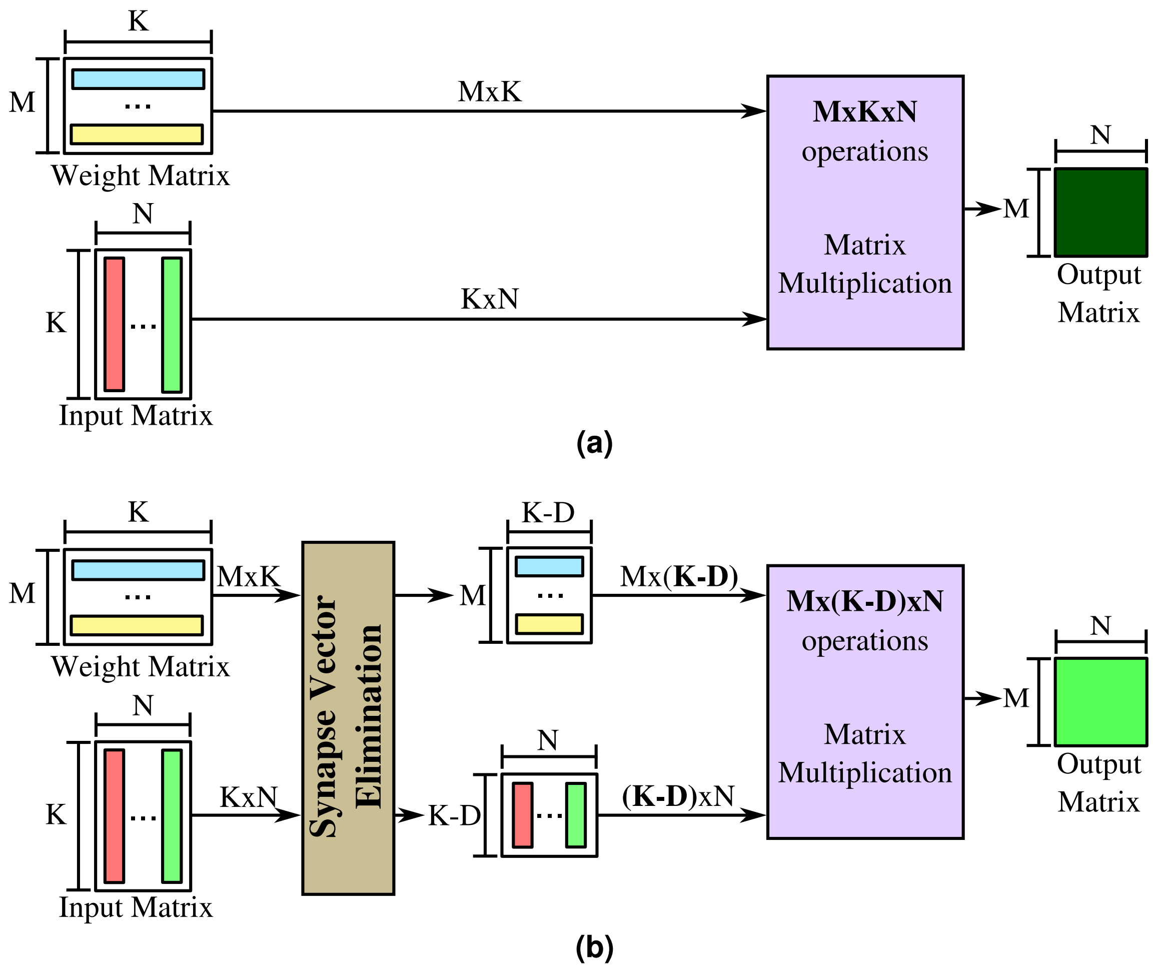


Figure 4.4: High-level view of synapse vector elimination, showing that (a) the exact computation is an $M \times K$ by $K \times N$ matrix multiplication, while (b) synapse vector elimination preprocesses the input to make the computation an $M \times (K - D)$ by $(K - D) \times N$ matrix multiplication.

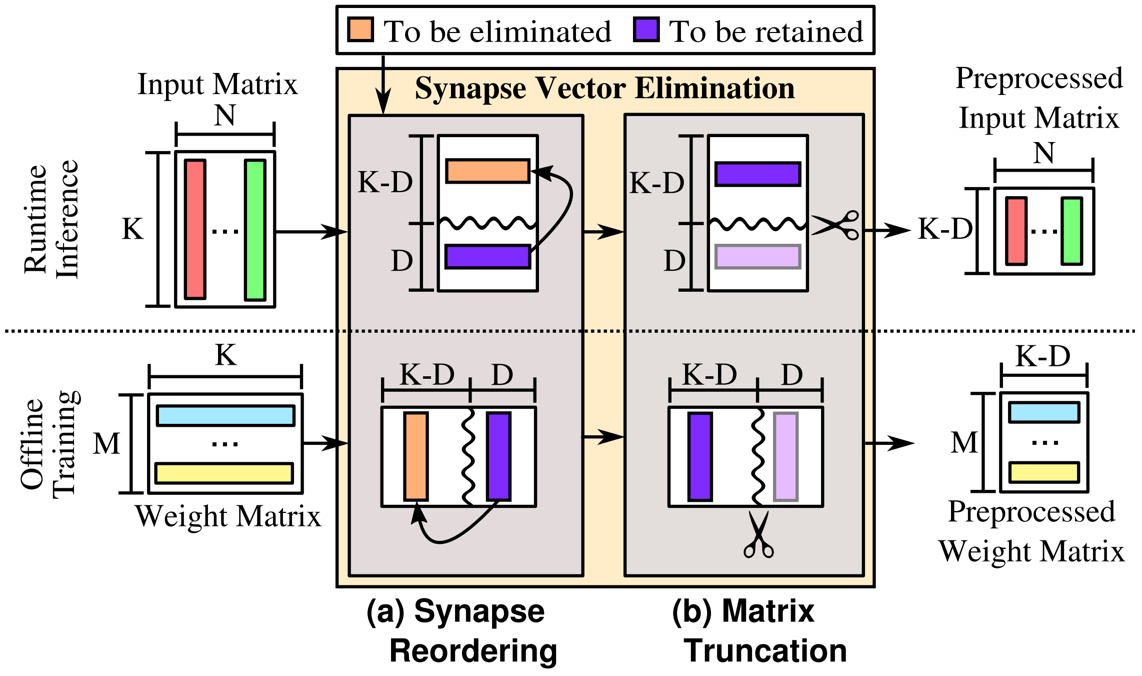


Figure 4.5: Internal workings of synapse vector elimination, showing compacting retained synapses so that the matrix can be trivially resized.

access. Instead, synapse vector elimination retains a hardware-efficient design by transforming DNN inputs for similarly-structured but smaller DNN computations.

Many DNNs have a large number of synapses that can potentially be eliminated. Without considering the underlying architecture, the selection of non-contributing synapses is fairly straightforward: network pruning techniques simply select the synapses with the lowest weights [40, 41]. One of the insights motivating this work is that the granularity of synapses that should be removed is constrained by the architecture, thus the selection of synapses becomes a multi-dimensional optimization problem. We devise a novel search technique to solve this problem based on the correlation matrix formed by the architectural groups of synapse weights.

4.3.1.1 Architecture-efficient Design

Here, we present the GPU architecture-efficient design of synapse vector elimination, our technique that avoids the performance pitfalls associated with network pruning, as described in §4.1.1, by applying a preprocessing step to efficiently rearrange computation. First, we show a high-level view of the approach in Figure 4.4. The original neural network computation (Figure 4.4(a)) is carried out by multiplying the $M \times K$ weight matrix by the $K \times N$ output matrix of the previous layer. The synapse vector elimination variant of this operation (Figure 4.4(b)) preprocesses the input and weight matrices to reduce the total problem size. The weight matrix is preprocessed offline since it is reused many times, while the input matrix is preprocessed during runtime to allow seamless switching between the original computation and the synapse vector elimination optimized computation. These smaller matrices are then given to a standard matrix multiplication algorithm. The performance benefits are realized by applying an inexpensive transformation that reduces the size of the inner dimension (i.e. K in the figure) of the matrix multiplication.

There are two main steps for our synapse vector elimination transformation, as shown in Figure 4.5: synapse reordering and matrix truncation. First, synapse reordering efficiently repositions rows and columns of the neural network matrices so that they are easier to manipulate. Next, matrix truncation reduces the amount of computation required for matrix multiplication while preserving its uniform data structure. Finally, a correction factor is applied to the matrices to retain the scale of the output values.

Synapse Reordering. First, we reorder synapses to simplify the task of discarding unwanted synapses. The central goal of reordering is to preserve the data structure's uniformity without diminishing the gains of skipping synapses, so a quick method of grouping the retained and discarded synapses is necessary. We group the synapses (rows in the weight matrix and columns in the input matrix) together based on whether they will be discarded or retained. For brevity, we only describe the reordering of the weight matrix, but an equivalent reordering is applied to the transpose of the input matrix. Since the number of discarded synapse weights, D, is known before we start reordering synapses, we partition the matrix at column $K-D$ so that the $K-D$ columns on the left represent the retained synapse group and the D columns on the right represent the discarded one.

After defining this partition point, some of the synapses that are to be retained are already contained in the retained synapse partition. In fact, there is an equal number of synapses to be retained as discarded that are in the incorrect partition. By using this observation, we create a pairing between misplaced retained synapses and misplaced discarded synapses. The matrix is then reordered by swapping the two columns for each of these pairs. After swapping all of the misplaced columns, the retained and discarded synapses are strictly separated at column $K-D$.

To determine the benefits of our synapse reordering method compared to naively copying all retained synapses into a separate buffer, we evaluate the overhead of these two methods in Figure 4.6 as a percentage of end-to-end execution time for AlexNet (the same trend is present for all of our evaluated applications). In our experiments, we found that more than 50% of synapses are needed to retain accuracy. Using this discarding rate, we observe that synapse reordering is at least $1.4\times$ faster than copy-

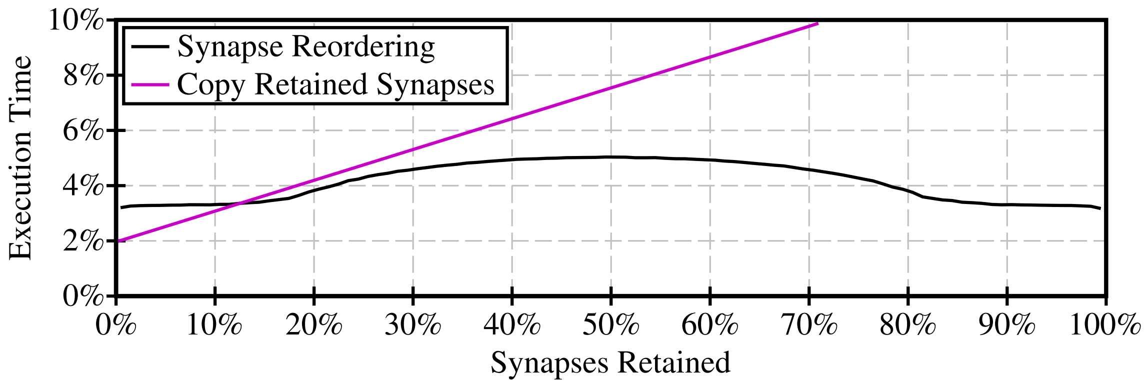


Figure 4.6: Overhead from synapse reordering compared to copying all retained synapses, showing that DeftNN substantially reduces the overhead of input transformation by reordering synapses at useful design points (¿50% retained).

ing retained synapses. We find that it is impractical to design hardware for synapse reordering, since there is little overhead involved in synapse vector elimination.

Matrix Truncation. Next, we reduce the dimensions of the input matrices to reduce the required amount of computation. To describe the matrix truncation step, we supply the formula for computing the value of a neuron (i.e. a cell of the output matrix) in Equation 4.1, where Out is the output matrix, W is the weight matrix, In is the previous layer matrix, i is the input index (e.g., the convolution kernel index or the fully connected input vector index), and j represents the input neuron index.

$$Out_{i,j} = \sum_{k=1}^{K} W_{i,k} In_{k,j} \tag{4.1}$$

Note that, after reordering the matrices, the output of matrix multiplication is the same as it would be without reordering; only the order of the weighted sum is changed. Thus, the output is equivalently shown in Equation 4.2, where the $K - D$

retained synapses in the reordered matrices, W' and In', are summed first and then the D discarded synapses are summed.

$$Out_{i,j} = \sum_{k=1}^{K-D} W'_{i,k} In'_{k,s} + \sum_{k=K-D+1}^{K} W'_{i,k} In'_{k,j} \tag{4.2}$$

In order to remove the computation for the discarded synapses, the summation is stopped at $K - D$ instead of at K. In terms of the DNN matrix multiplication, we cut the last D columns from the input neuron matrix and the last D rows from the weight matrix.

Scale Adjustment. To compensate for the discarded synapses in each summation, we increase the magnitude of the retained synapses so that the expected value of the original and optimized results match. If we assume that the synapses are all drawn from a similar distribution, then the expected value is equal to the expected value of any single synapse multiplied by the number of synapses, shown in Equation 4.3.

$$\mathbf{E}(Out_{i,j}) = \mathbf{E}(\sum_{k=1}^{K} W_{i,k} In_{k,s}) = K\mathbf{E}(W_{i,x} In_{x,j}) \tag{4.3}$$

The expected value of this sum, after removing the discarded synapses, can be represented similarly (Equation 4.4).

$$\mathbf{E}(Out'_{i,j}) = \mathbf{E}(\sum_{k=1}^{K-D} W'_{i,k} In'_{k,s}) = (K - D)\mathbf{E}(W_{i,x} In_{x,j}) \tag{4.4}$$

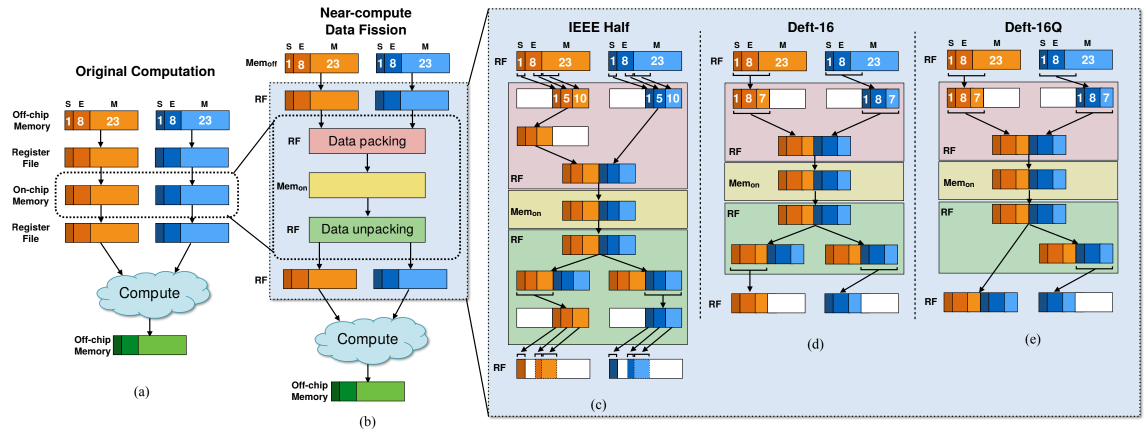


Figure 4.7: (a) The original design, (b) high level view of a design with near-compute data fission, (c) fission using the IEEE 754 single precision floating-point to the half precision variant, (d) fission using the Deft-16 floating-point format, and (e) fission using the optimized Deft-16Q floating-point conversion format.

To match the expected value from synapse vector elimination to the original expected value, the weighted sum is scaled by the ratio between the unadjusted expected value from synapse vector elimination and the original expected value. This produces our final expression for the synapse discarded summation, as shown in Equation 4.5.

$$Out''_{i,s} = \frac{\mathbf{E}(Out_{i,s})}{\mathbf{E}(Out'_{i,s})} \sum_{k=1}^{K-D} W'_{i,k} In'_{k,s} = \frac{K}{K-D} \sum_{k=1}^{K-D} W'_{i,k} In'_{k,s} \tag{4.5}$$

4.3.1.2 Synapse Search

Equipped with a method to efficiently discard synapses, we now describe how we find which synapses are not contributing to the final output. Trying all combinations of synapses is not tractable, since there are thousands of synapses and each synapse is a binary variable that can be either retained or discarded (i.e. there are $2^{\#synapses}$ possibilities). When reducing the DNN at a per-synapse granularity, prior works discard synapses with near-zero weights [40,41], avoiding the need for a sophisticated search mechanism. Although this pruning strategy is effective for pruning sporadic synapses, our insight is that GPU-efficient optimizations must discard synapses in groups to exploit wide vector unit hardware. We evaluate the necessity of this insight by comparing to these prior works that discard sporadic synapses in §5.8. A synapse vector pruning search mechanism must choose to retain or discard each architectural group of synapses, referred to as synapse vectors, rather than single ones. This presents a new challenge, since no synapse vectors have enough near-zero weights to be discarded as is done in these prior works.

Instead of discarding synapses with weights nearest to zero, we aim to retain a subset of the synapse vectors that are representative of the entire set of synapses. To select contributing synapses, synapse vector elimination starts by computing the correlation matrix, ρ, for the synapse vectors, as shown in Equation 4.6, where S_x is the synapse vector for the group of synapses at index x.

$$\rho_{i,j} = \frac{covariance(S_i, S_j)}{\sqrt{var(S_i)var(S_j)}} \tag{4.6}$$

For each synapse vector, S_i, we find the set of synapse vectors that it can represent. We define S_i to be representative of S_j if the correlation between the two synapse vectors ($\rho_{i,j}$) is above the representative correlation threshold, α. This is shown in Equation 4.7.

$$R_i = \sum_{j=1}^{N} \begin{cases} 1 & \rho_{i,j} \geqslant \alpha \\ 0 & \text{otherwise} \end{cases} \tag{4.7}$$

The synapse vector that represents the most synapse vectors, R_i, is selected to be retained in the output DNN from synapse vector elimination, while the non-contributing synapse vectors represented by the retained one are removed. This process is repeated on the remaining synapse vectors, until all synapse vectors are either retained or discarded.

4.3.1.3 Exposing Further Performance Opportunities

In addition to selecting non-contributing synapse vectors, synapse vector elimination can be parameterized to discard marginally-contributing ones by adjusting the representative correlation threshold. As the correlation threshold is lowered, the number of synapse vectors that can be represented by a single synapse increases. This capability can be used to enact approximate computing, essentially shedding small amounts of accuracy to realize improved performance.

The number of such readily available performance-accuracy trade-off configurations is limited due to large DNN memory footprints, if each configuration is stored in memory separately. To greatly increase the flexibility of synapse vector elimination, applied to marginally-contributing synapses, DeftNN dynamically builds DNNs using combinations of layers that were trained with varying correlation thresholds. Each <layer, correlation threshold> pair is fine-tuned independently of the others, allowing arbitrary combinations of these pairs to be composed during runtime without requiring a new DNN model for each combination.

Given a performance or accuracy constraint, DeftNN must quickly select an appropriate set of correlation thresholds for each of the layers. To do this, DeftNN is configured to build a Pareto frontier of configurations during training and to select the configuration that is nearest to the user-specified goal during runtime. We evaluate the idea of using DeftNN for approximation in §4.4.6.

4.3.2 Near-compute Data Fission

In this section, we present our near-compute data fission technique, which fuses multiple values into a single value of lesser size in on-chip memory to improve effective bandwidth. Near-compute data fission directly improves performance, since DNN computation is bottlenecked by on-chip memory bandwidth.

Although on-chip memory bandwidth is the key limitation of DNN performance, fission at this level of the memory hierarchy requires very frequent data reformatting, causing excessive overhead, unless the data format is carefully chosen. We start with a standard CUDA-supported half precision format, but find that it is insufficient for near-compute data fission and devise a new format that results in far better performance due to reduced reformatting overheads. To exploit the non-contributing bits further, by reducing the reformatting overhead, we introduce hardware to allow conversion to narrower numerical representations.

4.3.2.1 Technique

Before addressing the challenge of efficient near-compute data fission, we describe each of the GPU components that are relevant to our near-compute data fission method. Figure 4.7(a) shows a baseline implementation, in a GPU context, with no fission. The data is first loaded from off-chip memory into registers. To improve performance, the values in registers are stored into an on-chip memory scratchpad for future reuse. The application reads from and computes on the data stored in scratchpad memory many times. Finally, the result is written to off-chip memory.

Figure 4.7(b) shows an extension of this baseline with near-compute data fission added to the system. As before, data is loaded from the off-chip memory into the register file. Instead of writing directly to the scratchpad memory, multiple values are fused into a single element. Similarly, each time the application reads from the scratchpad memory, fission is applied to the value before it is computed on. This process removes the non-contributing bits from the numerical representation in the on-chip memory. We do not store fused data into the off-chip memory because, as shown before, the off-chip memory utilization is already very low.

4.3.2.2 Format Optimization

We investigate 3 near-compute data fission formats, as shown in the figure. The formatting process for these types are shown in Figure 4.7(c-e). All 3 of the fused data formats are reduced precision floating-point representations with a sign (S), a mantissa (M) that specifies the precision, and an exponent (E) that denotes dynamic range. In Figure 4.7(c), we start with the IEEE 754 half precision data format. In our evaluation, we find that this format results in excessive reformatting overhead, resulting in slowdown, due to the complex conversion taking several cycles.

The next technique shown in Figure 4.7(d), Deft-16, takes advantage of a special floating-point format defined to be the 16 most significant bits of the IEEE single precision floating-point format. This format is a data type with 8 exponent bits and 7 mantissa bits, which provides sufficient precision and dynamic range for DNN workloads. To apply fission to this value, only inexpensive shift and bitwise operations are necessary.

Finally, we observe that we can reduce another instruction from the fission process by allowing the most significant bits of one value to spill into the least significant bits of the other value. We call this the Deft-16 quick format (Deft-16Q) and show the set of operations for fission in Figure 4.7(e). Despite only reducing the fission process by a single logical AND instruction, we find substantial performance difference between Deft-16Q and Deft-16.

Nevertheless, while this optimized fission process can be applied to today's commodity hardware, its design is specific to 16-bit data and introduces a non-trivial amount of overhead. To address both of these limitations, we next describe a small additional hardware unit to perform the fission operation.

4.3.2.3 Hardware Acceleration

Since the fission operation is on the critical path when applying near-compute data fission, we introduce a lightweight GPU hardware extension called the *Data Fission Unit* (DFU) to accelerate fission. The DFU is replicated for each floating-

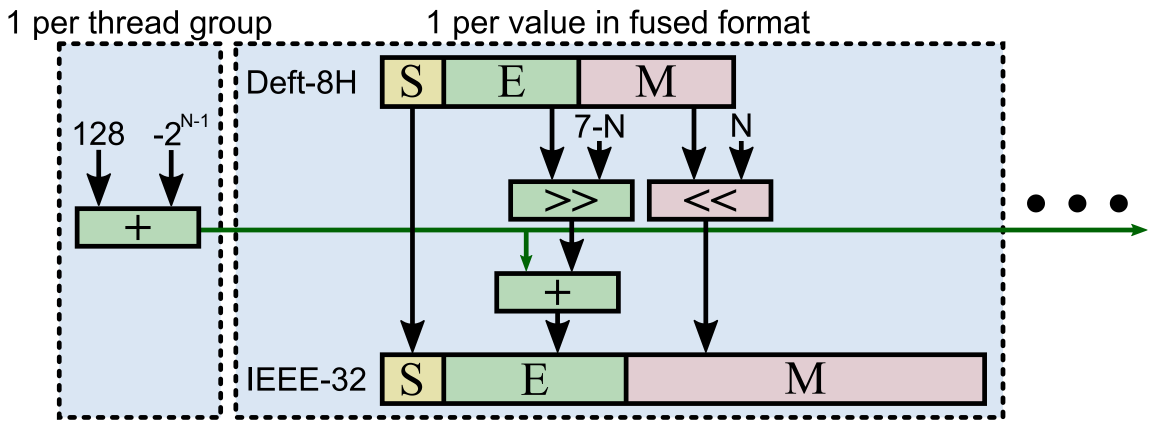


Figure 4.8: Fission in the DFU, showing hardware to apply fission to an 8-bit, variable exponent length (N) floating-point value to an IEEE single-precision value.

point unit to maintain high throughput, so our central design goal of the DFU is minimizing area overhead. For this reason, the DFU is specialized for the data representations that are most likely to be beneficial. The DFU is specifically targeted to accelerate the fission of custom 8-bit floating-point and Deft-16Q representations.

ISA Extension. DFU fission operations are accessed with a parallel thread execution (PTX) [88] ISA extension. We add two new instructions to PTX, `dfu_cvt_16` and `dfu_cvt_8`, which provide the ability to invoke the 16-bit and 8-bit DFUs, respectively. The 16-bit DFU operation is parameterized with a source `.b32` (a 32-bit conversion-only data type in PTX) register and two contiguous `.f32` (a 32-bit floating-point data type in PTX) destination registers. The 8-bit DFU operation is similar, except it is parameterized by four destination registers and an immediate floating-point exponent bitwidth. The flexibility of variable-width exponent allows low-precision 8-bit values to be more versatile, outweighing the negligible area cost (provided in §5.5).

Architecture Integration. The `dfu_cvt` instructions are executed by the DFU, which is integrated into the microarchitecture as an extension of the ALU. This extension adds a DFU to each floating-point unit, so the conversion throughput is sufficiently high to provide enough data for all of the floating-point units. In §5.5, we find that the area-efficient design of the DFU requires only 0.22% area overhead when replicated for each floating-point unit. In addition to allocating a DFU for each floating-point unit, we increase the throughput of the DFU by requiring that the 32-bit floating-point destination registers are contiguous. Using contiguous registers allows the DFU to use 64-bit and 128-bit register write operations when writing two and four 32-bit values, as produced by 16-bit and 8-bit data fission, respectively.

Conversion Details. The hardware design for applying 16-bit fission in the Deft-16Q format is straightforward, as it requires only a single zero-padded bitwise shift to prepare two values for computation. The hardware for 8-bit fission is illustrated in Figure 4.8. The 8-bit floating-point representation is shown at the top of the figure, with the sign bit denoted by "S", the exponent bits denoted by "E", and the mantissa bits denoted by "M".

The size of the exponent can be adjusted from 7 bits to 1 bit, denoted by N in the figure, depending on the exponent length encoded into the DFU instruction. Floating-point representations encode the exponent with a fixed offset, the bias, based on the bit width of the representation. Since this bias is different between the 32-bit representation and the 8-bit DeftNN representations, the DFU finds the difference between the two biases (adder on the left side of the figure), and then adds this difference to the exponent bits of the fused values. Because the GPU architecture

Name	**Neural Network**	**# Classes**	**Dataset**
IMC	AlexNet [61]	1000	ImageNet [96]
FLS	Flickr Style [55]	20	FS-20 [55]
OXF	Flower Species [83]	102	Flower-102 [83]
SOS	SOS CNN [123]	5	SOSDS [123]
C10	CifarNet [59]	10	CIFAR-10 [59]
DIG	LeNet-5 [66]	10	MNIST [67]

Table 4.1: The set of benchmarks used to evaluate DeftNN.

executes threads in each thread group in lockstep, we reuse the bias difference when applying fission to all of the fused values in a given thread group.

The mantissa bits, also of variable length, of the fused representation are shifted to the left, so that the most significant bit is aligned with the most significant bit of the 32-bit value. After alignment, the value is zero-padded to 23 bits and used as the mantissa of the 32-bit representation. The sign bit is directly transferred from the 8-bit representation to the 32-bit one. Leveraging the DFU, which provides single-cycle fission operations, we can significantly reduce the cost of performing DeftNN near-compute data fission.

4.4 Evaluation

We evaluate DeftNN to determine its efficacy on improving DNN performance. We examine each of the key components of the system: synapse vector elimination, near-compute data fission, and the complete DeftNN runtime system.

4.4.1 Methodology

DeftNN is evaluated using a real-system GPU implementation with robust open source frameworks. Our implementation is built upon Caffe [52], a neural network framework developed by the Berkeley Vision and Learning Center. Caffe provides the high-level neural network functionality and offloads the computational kernels to BLAS. We use MAGMA [81] as the BLAS implementation, a fast open source CUDA implementation. We take measurements on a machine containing a Xeon E5-2630 v3 CPU and an Nvidia Titan X (Pascal) GPU, a configuration representative of state-of-the-art commodity hardware.

Table 4.1 summarizes the set of applications used in our evaluation. To gain an accurate representation of real DNN workloads, we use the trained neural network models deployed in Caffe and designed by the machine learning community. For each benchmark, we use the given machine learning task's canonical validation and training datasets. We randomly select 500 inputs from the validation set for speedup and accuracy measurements and use the entire training set for fine tuning.

4.4.2 Overall DeftNN System

We begin by evaluating the end-to-end real-system GPU performance characteristics of DeftNN when applying both synapse vector elimination and near-compute data fission. In these experiments, we follow the steps outlined in §4.2 to automatically and transparently optimize the 6 DNNs covered in the evaluation. Figure 4.9 shows the results of these experiments. We first observe that applying each of the

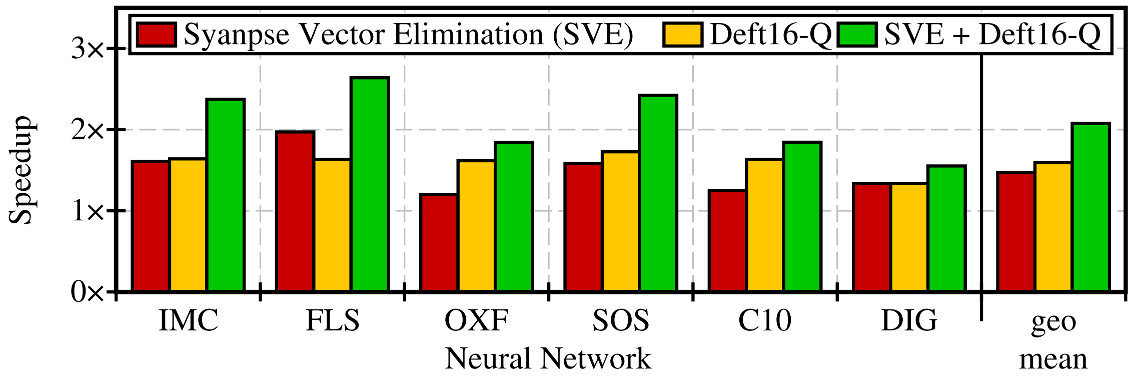


Figure 4.9: Speedup achieved by DeftNN when applying synapse vector discarding, data fission, and the combination of the two, showing the significant benefits of each technique and their efficacy when applied in concert.

two optimization techniques in isolation provides significant speedup, 1.5× and 1.6× geometric means across the applications for synapse vector elimination and near-compute data fission, respectively. When both techniques are applied, DeftNN provides an average speedup of 2.1×, showing the substantial performance benefit of deploying DeftNN.

4.4.3 Synapse Vector Elimination

We next evaluate synapse vector elimination in isolation to provide insight into its workings and characteristics, focusing on the layer-by-layer speedup achieved by synapse vector elimination. We present the per-layer performance improvements in Figure 4.10, showing that synapse vector elimination is capable of optimizing nearly every individual layer across the DNNs. We note that C10 and DIG observe the smallest performance improvements from synapse vector elimination. These networks are the smallest of our evaluated applications in terms of the number of layers as

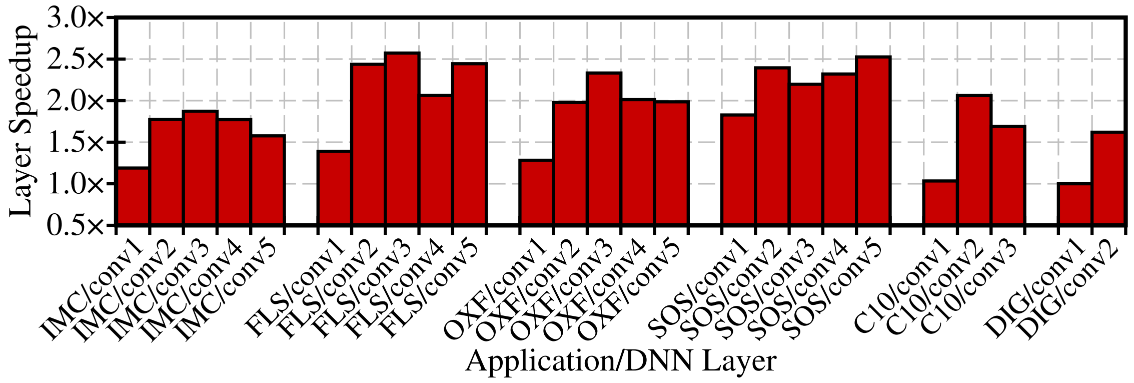


Figure 4.10: Per-layer speedup when applying synapse vector elimination, showing large performance improvements, particularly for the large DNNs (IMC, FLS, OXF, and SOS).

well as the size of each layer. Small DNN topologies limit the available parallelism on GPU architectures, causing lower utilization of hardware. As a consequence, substantial reductions in the topology of the DNN may result in under-utilization of GPU resources and limit the speedup that can occur. Nevertheless, synapse vector elimination is able to improve the performance by at least 1.5× for all but the smallest layers (the input layers).

4.4.4 Near-compute Data Fission

To evaluate our near-compute data fission technique, we compare the three fission formats described in §4.3.2.2, the baseline computation, and the computation with 16-bit compute and storage. The baseline computation is produced without fission, which uses the IEEE 754 single precision 32-bit floating-point format for computation and storage throughout the memory hierarchy. Comparisons of these near-compute data fission strategies are shown in Figure 4.11.

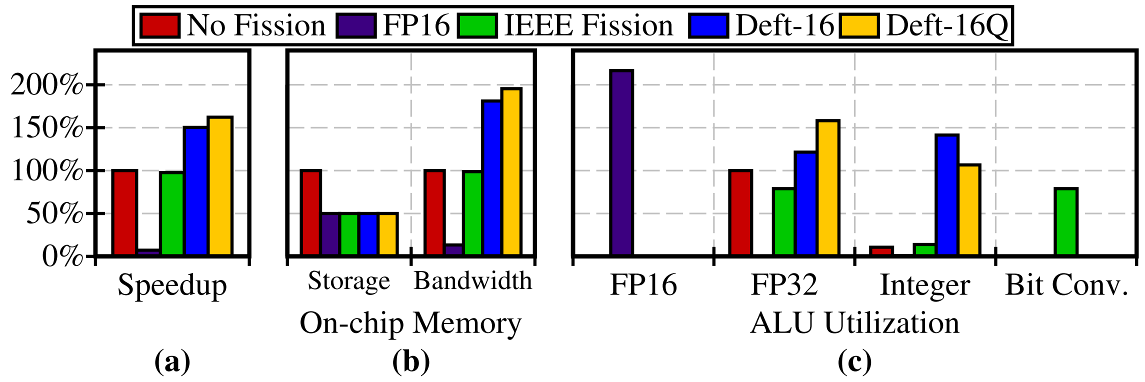


Figure 4.11: (a) Speedup from using 16-bit compute (FP16) and fission in three different formats (IEEE Fission, Deft-16, and Deft-16Q) compared to 32-bit storage and computation (No Fission) along with (b,c) pertinent profiling metrics, showing that Deft-16Q, achieves the highest performance because of improved effective on-chip memory bandwidth.

Speedup. Figure 4.11(a) presents the speedup achieved during end-to-end DNN inference for each of the three fission formats. First, we consider the 16-bit computation and storage configuration, FP16 in the figure. We observe that 16-bit computation results in a 14× slowdown due to state-of-the-art GPUs having many more 32-bit ALUs than 16-bit ALUs.

Next, we consider the IEEE half precision format. The IEEE Fission results represent a CUDA mechanism to convert the fused IEEE half precision values into single precision values, which leverages specialized bit-convert hardware. Due to the limited amount of hardware allocated for these conversion instructions, the IEEE conversion process imposes significant overhead, resulting in slight slowdown rather than speedup. Our novel fission formats, Deft-16 and Deft-16Q, result in the same change in data size, but both achieve over 50% improvement in end-to-end performance showing that the complexity of data type conversion is critical.

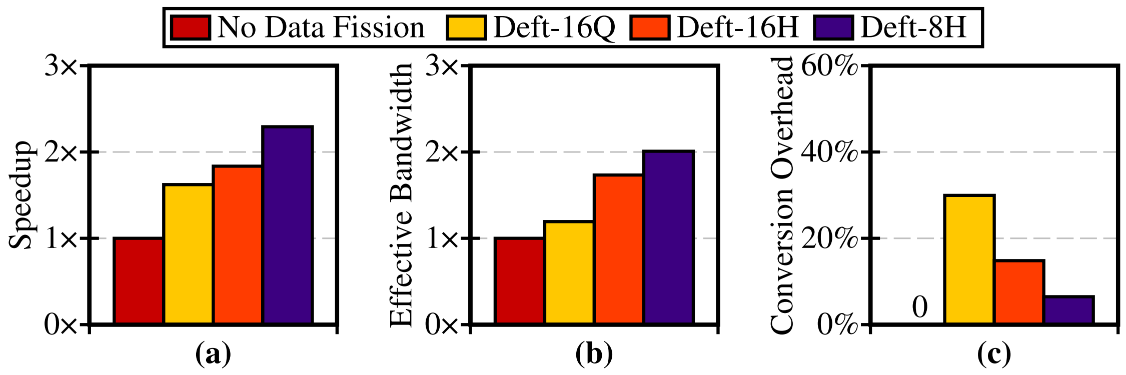


Figure 4.12: Comparison of no near-compute data fission, software-only fission (Deft-16Q), and hardware accelerated 16-bit (Deft-16H) and 8-bit (Deft-8H) fission, showing that (a) performance is improved as (b) effective on-chip bandwidth is increased with smaller representations, without (c) restrictive conversion overhead.

Profiling Details. All three of the near-compute data fission formats yield half of the storage requirements for on-chip memory, shown in Figure 4.11(b), since the values in each format occupy 16 bits instead of 32 bits. The key benefit of fusing data into on-chip memory is the increased effective on-chip memory bandwidth. In Figure 4.11(b), we note that the on-chip memory bandwidth is not equivalent to speedup, since the increase in register pressure required for format conversion causes registers to spill to on-chip memory. Spilling registers increases the total amount of data that must traverse the on-chip memory bus, so the measured effective bandwidth will exceed the speedup. As expected from the speedup of the other two fission formats, we see substantially increased on-chip memory effective bandwidth.

The ALU utilization, presented in Figure 4.11(c), shows the utilization of the four relevant functional units. The utilization of the functional units is normalized to the floating-point unit utilization of the configuration without fission. The single precision floating-point unit (FP32 in the figure), which is used for the core computation of the neural network, only serves as a rough proxy for performance due to fission using the floating-point unit. The integer and bit conversion functional units provide more insight into the speedup differences, representing overhead of data fission.

4.4.5 Hardware-accelerated Data Fission

We now evaluate DeftNN with the addition of the DFU, a lightweight GPU architectural extension to accelerate near-compute data fission. We implemented and synthesized the DFU for an Nvidia Titan X (Pascal) using the ARM Artisan IBM SOI 45 nm library, showing that the DFU has an area overhead of $1.20mm^2$ (0.25% area overhead), and an active power consumption of 2.48W (0.99% power overhead).

We evaluate end-to-end performance of DeftNN atop a DFU-enabled Titan X using an in-house GPU simulation tool. This tool emulates end-to-end execution by modifying the GPU kernel to mimic the performance behavior of the modified hardware. Specifically, the fission instructions are automatically replaced by a set of instructions that have the same register dependencies, but throughput and latency characteristics matching the DFU (i.e. single cycle using single-precision floating-point ALUs).

Benefits Over Software Fission. We first evaluate the efficacy of the DFU by comparing it to software-implemented fission in isolation (i.e., no synapse vector elimination is involved). Figure 4.12(a) shows the speedup when the DNN computation is subjected to fission. Accelerated 16-bit fission (Deft-16H) yields a modest

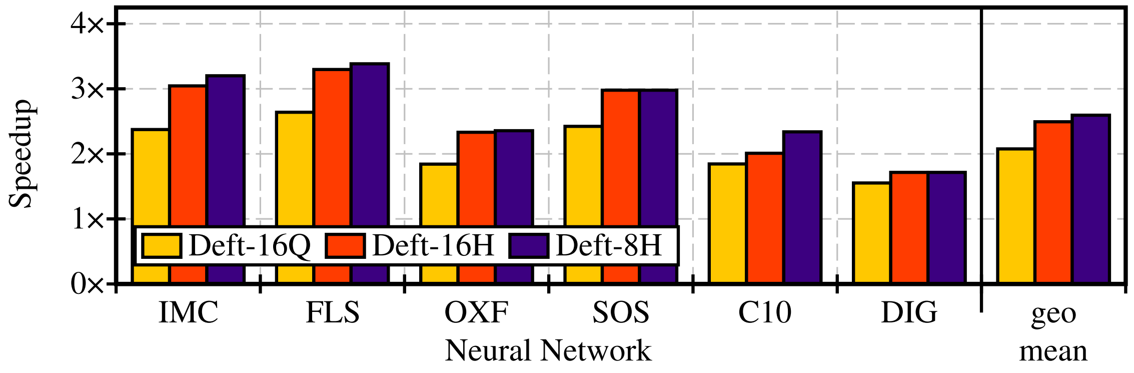


Figure 4.13: DeftNN runtime performance achieved by employing software-only (Deft-16Q) and hardware-accelerated (Deft-16H/8H) fission, showing substantial speedup via hardware-accelerated DeftNN.

performance improvement over software-implemented fission (Deft-16Q), improving speedup from 1.6× to 1.8× by mitigating the overhead of performing the fission operations. The speedup of 8-bit accelerated fission (Deft-8H) is 2.3×, significantly higher than Deft-16H because the amount of data moved is dramatically reduced when using 8-bit values.

These sources of speedup are explored further in Figure 4.12(b) and (c). In (b), we observe comparable decreases in the effective on-chip memory bandwidth among the fission techniques. Moreover, in Figure 4.12(c) we observe that both of the hardware-accelerated configurations use less than half of the data conversion time compared to the software-only configuration. Although the DFU hardware for 16-bit conversion is far simpler than the hardware for 8-bit conversion, we note that it results in more total overhead due to the fact that twice as many conversions are made – only two values are produced per instruction using 16-bit conversions, rather than four values per instruction for 8-bit conversions.

End-to-end Performance. We next examine the impact of leveraging the DFU for accelerated data fission in the end-to-end DeftNN system. The speedup for all applications of the end-to-end system for Deft-16Q, Deft-16H and Deft-8H are presented in Figure 4.13, which shows that the system improving performance substantially when leveraging the DFU to facilitate efficient near-compute data fission. We observe that the end-to-end speedup averages 2.1× with Deft-16Q and that it increases to 2.5× with Deft-16Q and 2.6× with Deft-8H. As Deft-16Q and Deft-16H have the same data movement characteristics, the difference between the two represents the removal of (most of) the overhead of performing data fission in software. The additional speedup achieved by Deft-8H is due to the substantial reduction in the amount of data moved compared to Deft-16Q and Deft-16H.

4.4.6 Performance-Accuracy Tradeoffs

In addition to removing only non-contributing synapses that do not impact accuracy, recall from §4.3.1.3 that synapse vector elimination parameterizations for higher performance, but slightly degraded accuracy, are also possible. The correlation parameter used for synapse vector elimination can be relaxed to allow the system to eliminate synapses that contribute in a small way to the output result. By relaxing the correlation parameter, we can be selective about the resulting DNN density and thus the amount of speedup achieved by synapse vector elimination. This section explores using this feature of synapse vector elimination within a runtime system that facilitates approximate computing, trading off small levels of accuracy

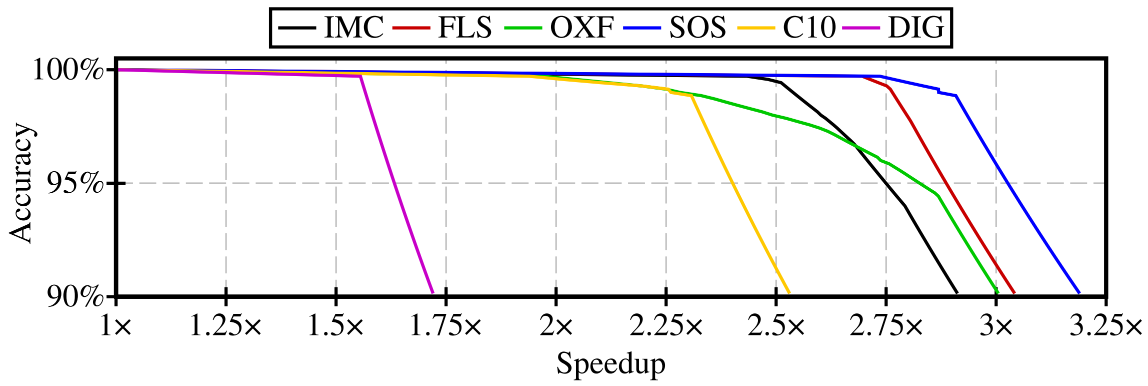


Figure 4.14: DeftNN Pareto frontiers, showing a range of advantageous operating points as the accuracy target is tuned.

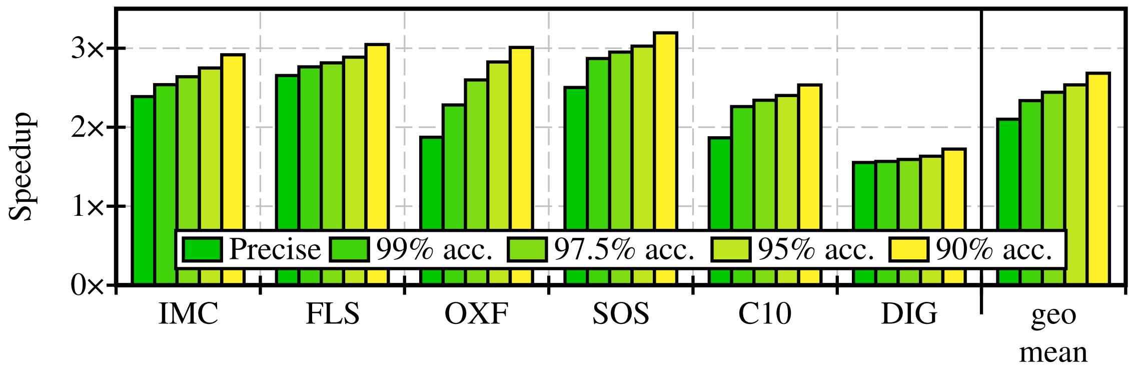


Figure 4.15: Speedup achieved by DeftNN at particular accuracy levels, showing that DeftNN exposes a range of useful design points for approximate computing.

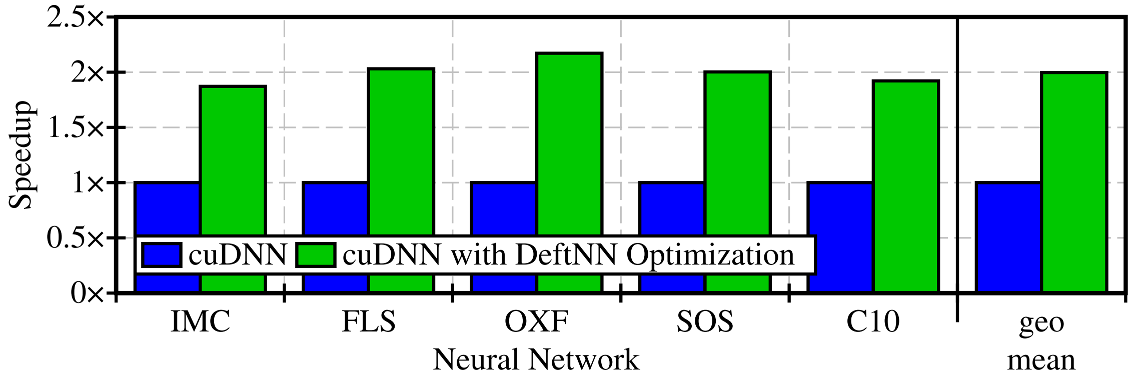


Figure 4.16: Speedup of cuDNN [20] with DeftNN optimizations, showing DeftNN provides similar speedup for cuDNN as it provides for MAGMA.

for larger performance improvements.

Figure 4.14 presents the accuracy versus speedup Pareto frontier for each of the evaluated neural networks. As the correlation parameter is relaxed (going from left to right), the accuracy degradation is initially minimal due to the resilience of the DNN, but as more contributing synapses are removed the accuracy decreases more quickly. Beyond the precise configuration, where no loss in accuracy is permitted, which is used in the other sections of the evaluation, tuning the correlation parameter allows synapse vector elimination to achieve further speedups for small accuracy losses. We observe in Figure 4.15 that a spectrum of useful design points that are commonly focused on in approximate computing are available to the system [36,63,72], allowing DeftNN to service a wide range of use-cases where there is tolerance in end-user accuracy or a more aggressive performance target.

4.4.7 cuDNN with DeftNN Optimization

To demonstrate the applicability of DeftNN and its underlying optimization strategies, we apply DeftNN to cuDNN [85] by implementing the cuDNN convolution algorithm [20]. This algorithm is similar to the standard matrix multiplication algorithm, except that the preprocessing step of translating the DNN input and convolution weights into a matrix (known as `im2col`) is interleaved with the matrix multiplication. This optimization reduces off-chip memory storage requirements by lazily evaluating the contents of the input matrix. The only adjustment in synapse vector elimination to handle cuDNN is that, as the input matrix is being produced in on-chip memory, the synapse vector elimination takes place.

In Figure 4.16, we show the speedup achieved when DeftNN is applied to cuDNN. In applying DeftNN to cuDNN, we observe a geometric mean speedup of 2.0×, a similar speedup to what is achieved when applying DeftNN to MAGMA. Since DeftNN is similarly effective on both MAGMA and cuDNN algorithms, the fundamental GPU DNN bottlenecks being addressed by DeftNN are common to the most popular GPU DNN implementations.

4.4.8 Comparison to Prior Work

We next compare the novel optimizations introduced in this work and leveraged by DeftNN to prior work.

Network Pruning. Network pruning is a technique that iteratively prunes synapses from the neural network [41]. This technique reduces the number of synapses in the DNN, but it produces an irregular sparse matrix because it places no performanceaware constraints on which synapses are removed. In comparison, our synapse vector elimination technique maintains a regular dense matrix by removing entire rows or columns of synapses, allowing synapse vector elimination to map efficiently to GPU hardware.

We compare synapse vector elimination to network pruning for IMC in Figure 4.17, presenting network density achieved versus speedup. Network pruning [40] achieves matrix densities between 9% and 100% across IMC's layers, and a weighted average of 28%. Using the network pruning approach for IMC, we execute the pruned networks using both dense and sparse kernels via cuBLAS [85] and cuSPARSE [86]. With the sparse kernel we observe that network pruning is 60× slower than the dense kernel baseline computation and 91× slower than synapse vector elimination, while with the dense kernel we observe that network pruning results in no speedup over the baseline. We performed a sweep of density levels on the sparse kernel, finding that the density must be reduced to 2.5% before the sparse kernel outperforms the dense kernel baseline. On the other hand, synapse vector elimination achieves 50% density and a 1.5× speedup on this kernel, illustrating the benefit of synapse vector elimination's architecturally-aware design.

Recent work also proposed EIE, a custom ASIC to execute network pruned DNNs [39]. While this ASIC achieves impressive results on fully connected layers, those components account for only a fraction of the end-to-end execution time in many modern DNNs, including for the IMC network. Figure 4.17 includes the density and speedup achieved by EIE for the end-to-end IMC network, which achieves a 1.1× speedup. Meanwhile, synapse vector elimination's speedup of 1.5× is achieved

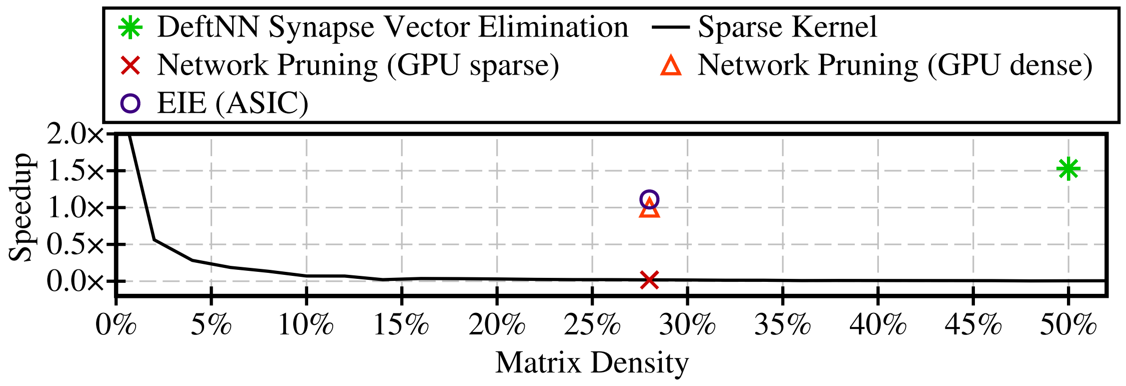


Figure 4.17: End-to-end speedup of DeftNN synapse vector elimination, software executed network pruning [41], and EIE [39] hardware-accelerated network pruning.

on a real GPU system.

Off-chip Data Packing. Off-chip data packing is similar to near-compute data fission, except data is packed into off-chip memory and unpacked into on-chip memory, saving off-chip memory storage and bandwidth [98]. Certain applications are able to benefit substantially from off-chip data packing, but we found that, for DNN applications, off-chip bandwidth is only slightly utilized, while on-chip bandwidth is saturated. We compare the performance improvements of off-chip and near-compute data fission in Figure 4.18. As expected, off-chip data packing yields modest speedups, since off-chip memory is already underutilized and the bottleneck lies elsewhere in DNN execution.

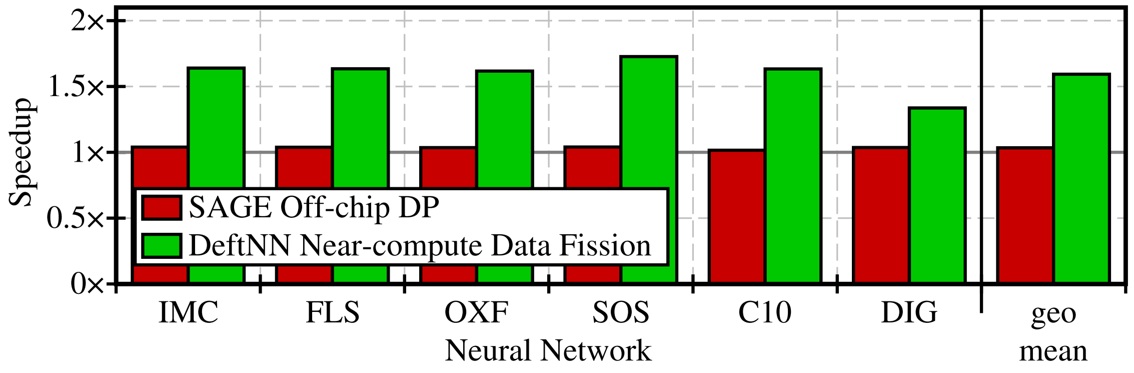


Figure 4.18: Comparison of DeftNN data fission to off-chip data packing [98].

4.5 Summary

This chapter described *DeftNN*, a system for optimizing GPU-based DNNs. DeftNN uses two novel optimization techniques – synapse vector elimination, a technique that drops the non-contributing synapses in the neural network, as well as near-compute data fission, a mechanism for scaling down the data movement requirements within DNN computations. Our experimental results show that DeftNN can significantly improve DNN performance, improving performance by over 2.1× on commodity GPU hardware and over 2.6× when leveraging a small additional hardware unit that accelerates fission.

CHAPTER V

Rethinking Numerical Representations for Deep Neural Networks

With ever-increasing computational demand for deep learning, it is critical to investigate the implications of the numeric representation and precision of DNN model weights and activations on computational efficiency. In this work, we explore unconventional narrow-precision floating-point representations as it relates to inference accuracy and efficiency to steer the improved design of future DNN platforms. We show that inference using these custom numeric representations on production-grade DNNs, including GoogLeNet and VGG, achieves an average speedup of 7.6$\times$ with less than 1% degradation in inference accuracy relative to a state-of-the-art baseline platform representing the most sophisticated hardware using single-precision floating point. To facilitate the use of such customized precision, we also present a novel technique that drastically reduces the time required to derive the optimal precision configuration.

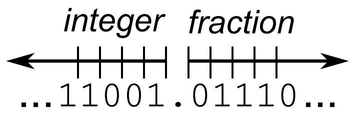


Figure 5.1: A fixed-point representation. Hardware parameters include the total number of bits and the position of the radix point.

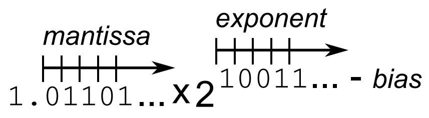


Figure 5.2: A floating-point representation. Hardware parameters include the number of mantissa and exponent bits, and the bias.

5.1 Customized Precision Hardware

We begin with an overview of the available design choices in the representation of real numbers in binary and discuss how these choices impact hardware performance.

5.1.1 Design Space

We consider three aspects of customized precision number representations. First, we contrast the high-level choice between fixed-point and floating-point representations. Fixed-point binary arithmetic is computationally identical to integer arithmetic, simply changing the interpretation of each bit position. Floating-point arithmetic, however, represents the sign, mantissa, and exponent of a real number separately. Floating-point calculations involve several steps absent in integer arithmetic. In particular, addition operations require aligning the mantissas of each operand. As a result, floating-point computation units are substantially larger, slower, and more complex than integer units.

In CPUs and GPUs, available sizes for both integers and floating-point calculations are fixed according to the data types supported by the hardware. Thus, the second aspect of precision customization we examine is to consider customizing the number of bits used in representing floating-point and fixed-point numbers. Third, we may vary the interpretation of fixed-point numbers and assignment of bits to the mantissa and exponent in a floating-point value.

5.1.2 Customized Precision Types

In a fixed-point representation, we select the number of bits as well as the position of the radix point, which separates integer and fractional bits, as illustrated in Figure 5.1. A bit array, x, encoded in fixed point with the radix point at bit l (counting from the right) represents the value $2^{-l} \sum_{i=0}^{N-1} 2^i \cdot x_i$. In contrast to floating point, fixed-point representations with a particular number of bits have a fixed level of precision. By varying the position of the radix point, we change the representable range.

An example floating-point representation is depicted in Figure 5.2. As shown in the figure, there are three parameters to select when designing a floating-point representation: the bit-width of the mantissa, the bit-width of the exponent, and an exponent bias. The widths of the mantissa and exponent control precision and dynamic range, respectively. The exponent bias adjusts the offset of the exponent (which is itself represented as an unsigned integer) relative to zero to facilitate positive and negative exponents. Finally, an additional bit represents the sign. Thus, a floating-point format with N_m mantissa bits, N_e exponent bits, and a bias of b, encodes the value $2^{(\sum_{i=0}^{N_e-1} 2^i \cdot e_i)-b}(1 + \sum_{i=1}^{N_m} 2^{-i} \cdot m_i)$, where m and e are the segments of a bit array representing the mantissa and exponent, respectively. Note that the leading bit of the mantissa is assumed to be 1 and hence is not explicitly stored,

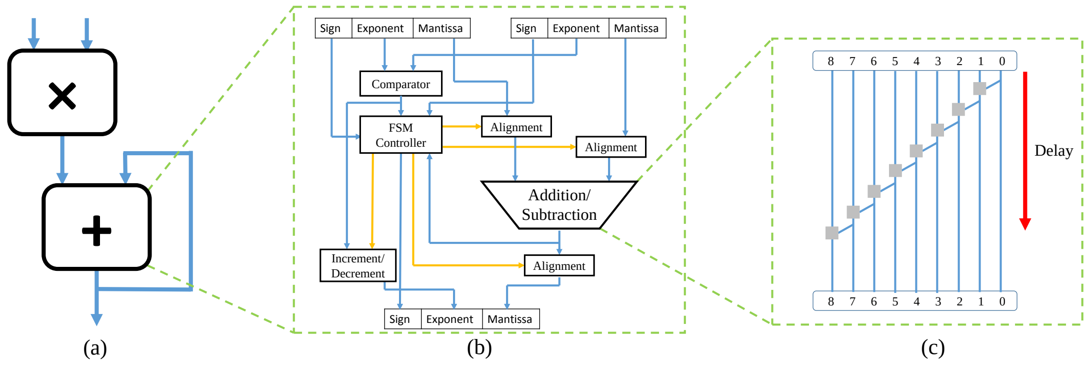


Figure 5.3: Floating point multiply-accumulate (MAC) unit with various levels of detail: (a) the high level mathematical operation, (b) the modules that form a floating point MAC, and (c) the signal propagation of the unit.

eliminating redundant encodings of the same value. A single-precision value in the IEEE-754 standard (i.e. `float`) comprises 23 mantissa bits, 8 exponent bits, and a sign bit. IEEE-754 standardized floating-point formats include special encodings for specific values, such as zero and infinity.

Both fixed-point and floating-point representations have limitations in terms of the precision and the dynamic ranges available given particular representations, manifesting themselves computationally as rounding and saturation errors. These errors propagate through the deep neural network in a way that is difficult to estimate holistically, prompting experimentation on the DNN itself.

5.1.3 Hardware Implications

The key hardware building block for implementing DNNs is the multiply-accumulate (MAC) operation. The MAC operation implements the sum-of-products operation that is fundamental to the activation of each neuron. We show a high-level hardware block diagram of a MAC unit in Figure 5.3 (a). Figure 5.3 (b) adds detail for the addition operation, the more complex of the two operations. As seen in the figure, floating-point addition operations involve a number of sub-components that compare exponents, align mantissas, perform the addition, and normalize the result. Nearly all of the sub-components of the MAC unit scale in speed, power, and area with the bit width.

Reducing the floating-point bit width improves hardware performance in two ways. First, reduced bit width makes a computation unit faster. Binary arithmetic computations involve chains of logic operations that typically grows at least logarithmically, and sometimes linearly (e.g., the propagation of carries in an addition, see Figure 5.3 (c)), in the number of bits. Reducing the bit width reduces the length of these chains, allowing the logic to operate at a higher clock frequency. Second, reduced bit width makes a computation unit smaller and require less energy, typically linearly in the number of bits. The circuit delay and area is shown in Figure 5.4 when the mantissa bit widths are varied. As shown in the figure, scaling the length of the mantissa provides substantial opportunity because it defines the size of the internal addition unit. Similar trends follow for bit-widths in other representations. When a unit is smaller, more replicas can fit within the same chip area and power budget, all of which can operate in parallel. Hence, for computations like those in DNNs, where ample parallelism is available, area reductions translate into proportional performance improvement.

This trend of bit width versus speed, power, and area is applicable to every computation unit in hardware DNN implementations. Thus, in designing hardware

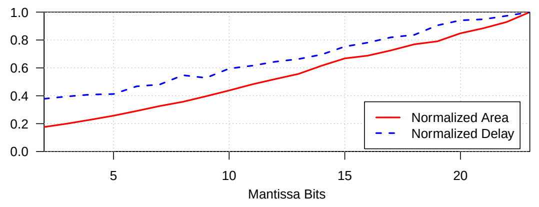


Figure 5.4: Delay and area implications of mantissa width, normalized to a 32-bit Single Precision MAC with 23 mantissa bits.

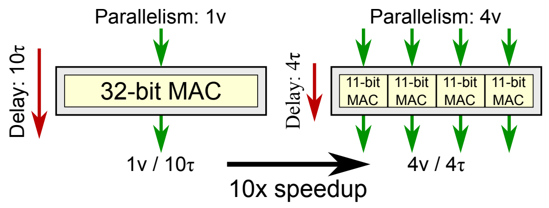


Figure 5.5: Speedup calculation with a fixed area budget. The speedup exploits the improved function delay and parallelism.

that uses customized representations there is a trade-off between accuracy on the one hand and power, area, and speed on the other. Our goal is to use precision that delivers sufficient accuracy while attaining large improvements in power, area, and speed over standard floating-point designs.

5.2 Methodology

We describe the methodology we use to evaluate the customized precision design space, using image classification tasks of varying complexity as a proxy for computer vision applications. We evaluate DNN implementations using several metrics, classification accuracy, speedup, and energy savings relative to a baseline custom hardware design that uses single-precision floating-point representations. Using the results of this analysis, we propose and validate a search technique to efficiently determine the correct customized precision design point.

5.2.1 Accuracy

We evaluate accuracy by modifying the Caffe [52] deep learning framework to perform calculations with arbitrary fixed-point and floating-point formats. We continue to store values as C `floats` in Caffe, but truncate the mantissa and exponent to the desired format after each arithmetic operation. Accuracy, using a set of test inputs disjoint from the training input set, is then measured by running the forward pass of a DNN model with the customized format and comparing the outputs with the ground truth. We use the standard accuracy metrics that accompany the dataset for each DNN. For MNIST (LeNet-5) and CIFAR-10 (CIFARNET) we use top-1 accuracy and for ImageNet (GoogLeNet, VGG, and AlexNet) we use top-5 accuracy. Top-1 accuracy denotes the percent of inputs that the DNN predicts correctly after a single prediction attempt, while top-5 accuracy represents the percent of inputs that DNN predicts correctly after five attempts.

5.2.2 Efficiency

We quantify the efficiency advantages of customized floating-point representations by designing a floating-point MAC unit in each candidate precision and determining its silicon area and delay characteristics. We then report speedup and energy savings relative to a baseline custom hardware implementation of a DNN that uses standard single-precision floating-point computations. We design each variant of the MAC unit using Synopsys Design Compiler and Synopsys PrimeTime, industry standard ASIC design tools, targeting a commercial 28nm silicon manufacturing process. The tools report the power, delay, and area characteristics of each precision variant. As shown in Figure 5.5, we compute speedups and energy savings relative to the standardized IEEE-754 floating-point representation considering both the clock frequency advantage and improved parallelism due to area reduction of the narrower bit-width MAC units. This allows customized precision designs to yield a quadratic improvement in total system throughput.

5.2.3 Training

Given a network trained with full precision, a standard approach for speed-up is to round the weights to a customized precision network. An intriguing question is whether the inevitable loss of classification accuracy can be minimized by training with customized precision. We thus experiment with three training strategies: (1) training with full precision, (2) training with full precision followed by fine-tuning—another round of training with customized precision, and (3) training with customized precision from scratch. Note that in both cases of training with customized precision, only the forward pass is performed in customized precision. The backward pass—weight representation and updates—is still done with full precision.

Intuitively, each of these approaches appear to have features that could cause them to perform more effectively than the others. When a pre-trained model with weights outside of the range of values that can be encoded is immediately approximated, an anomalous percentage of weights at the limits of the numeric representation (i.e. ±*saturation point* and 0) will be present. On the other hand, training entirely with small numeric representations may prevent the model from learning due to insufficient precision. Finally, fine-tuning will avoid the high output density of extreme values by making adjustments to these neurons, but it may also cause degradation if the remaining precision is insufficient to yield meaningful training.

5.2.4 Efficient Customized Precision Search

To exploit the benefits of customized precision, a mechanism to select the correct configuration must be introduced. There are hundreds of designs among floating-point and fixed-point formats due to designs varying by the total bit width and the allocation of those bits. This spectrum of designs strains the ability to select an optimal configuration. A straightforward approach to select the customized precision design point is to exhaustively compute the accuracy of each design with a large number of neural network inputs. This strategy requires substantial computational resources that are proportional to the size of the network and variety of output classifications. We describe our technique that significantly reduces the time required to search for the correct configuration in order to facilitate the use of customized precision.

The key insight behind our search method is that customized precision impacts the underlying internal computation, which is hidden by evaluating only the NN final accuracy metric. Thus, instead of comparing the final accuracy generated by networks with different precision configurations, we compare the original NN activations to the customized precision activations. This circumvents the need to evaluate the large number of inputs required to produce representative neural network accuracy. Furthermore, instead of examining all of the activations, we only analyze the last layer, since the last layer captures the usable output from the neural network as well as the propagation of lost accuracy. Our method summarizes the differences between the last layer of two configurations by calculating the linear coefficient of determination between the last layer activations.

A method to translate the coefficient of determination to a more desirable metric, such as end-to-end inference accuracy, is necessary. We find that a linear model provides such a transformation. The customized precision setting with the highest speedup that meets a specified accuracy threshold is then selected. In order to account for slight inaccuracies in the model, inference accuracy for a subset of configurations is evaluated. If the configuration provided by the accuracy model results in insufficient accuracy, then an additional bit is added and the process repeats. Similarly, if the accuracy threshold is met, then a bit is removed from the customized precision format.

5.3 Experiments

In this section, we evaluate five common neural networks spanning a range of sizes and depths in the context of customized precision hardware. We explore the trade-off between accuracy and efficiency when various customized precision representations are employed. Next, we address the sources of accuracy degradation when customized precision is utilized. Finally, we examine the characteristics of our customized precision search technique.

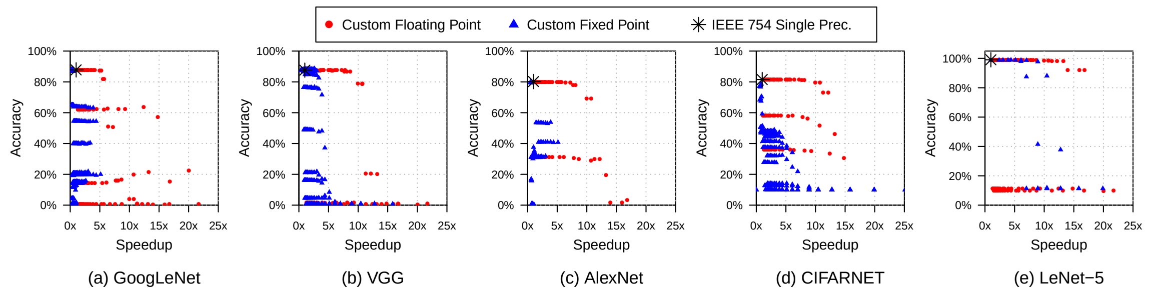


Figure 5.6: The inference accuracy versus speedup design space for each of the neural networks, showing substantial computational performance improvements for minimal accuracy degradation when customized precision floating-point formats are used.

5.3.1 Experimental Setup

We evaluate the accuracy of customized precision operations on five DNNs: GoogLeNet [115], VGG [110], AlexNet [60], CIFARNET [58], and LeNet-5 [66]. The implementations and pre-trained weights for these DNNs were taken from Caffe [52]. The three largest DNNs (GoogLeNet, VGG, and AlexNet) represent real-world workloads, while the two smaller DNNs (CIFARNET and LeNet-5) are the largest DNNs evaluated in prior work on customized precision. For each DNN, we use the canonical benchmark validation set: ImageNet for GoogLeNet, VGG, and AlexNet; CIFAR-10 for CIFARNET; MNIST for LeNet-5. We utilize the entire validation set for all experiments, except for GoogLeNet and VGG experiments involving the entire design space. In these cases we use a randomly-selected 1% of the validation set to make the experiments tractable.

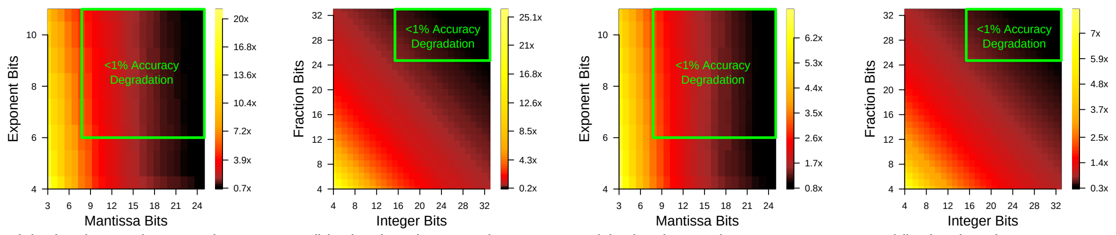
5.3.2 Accuracy versus Efficiency Trade-offs

To evaluate the benefits of customized precision hardware, we swept the design space for accuracy and performance characteristics. This performance-accuracy trade off is shown in Figure 5.6. This figure shows the DNN inference accuracy across the full input set versus the speedup for each of the five DNN benchmarks. The black star represents the IEEE 754 single precision representation (i.e. the original accuracy with $1\times$ speedup), while the red circles and blue triangles represent the complete set of our customized precision floating-point and fixed-point representations, respectively.

For GoogLeNet, VGG, and AlexNet it is clear that the floating-point format is superior to the fixed-point format. In fact, the standard single precision floating-point format is faster than all fixed-point configurations that achieve above 40% accuracy. Although fixed-point computation is simpler and faster than floating-point computation when the number of bits is fixed, customized precision floating-point representations are more efficient because less bits are needed for similar accuracy.

By comparing the results across the five different networks in Figure 5.6, it is apparent that the size and structure of the network impacts the customized precision flexibility of the network. This insight suggests that hardware designers should carefully consider which neural network(s) they expect their device to execute as one of the fundamental steps in the design process. The impact of network size on accuracy is discussed in further detail in the following section.

The specific impact of bit assignments on performance and energy efficiency are illustrated in Figure 5.7. This figure shows the the speedup and energy improvements



(a) Floating−point speedup (b) Fixed−point speedup (c) Floating−point energy (d) Fixed−point energy

Figure 5.7: The speedup and energy savings as the two parameters are adjusted for the custom floating point and fixed-point representations. The marked area denotes configurations where the total loss in AlexNet accuracy is less than 1%.

over the single precision floating-point representation as the number of allocated bits is varied. For the floating-point representations, the number of bits allocated for the mantissa (x-axis) and exponent (y-axis) are varied. For the fixed-point representations, the number of bits allocated for the integer (x-axis) and fraction (y-axis) are varied. We highlight a region in the plot deemed to have acceptable accuracy. In this case, we define acceptable accuracy to be 99% normalized AlexNet accuracy (i.e., no less than a 1% degradation in accuracy from the IEEE 754 single precision accuracy on classification in AlexNet).

The fastest and most energy efficient representation occurs at the bottom-left corner of the region with acceptable accuracy, since a minimal number of bits are used. The configuration with the highest performance that meets this requirement is a floating-point representation with 6 exponent bits and 7 mantissa bits, which yields a 7.2× speedup and a 3.4× savings in energy over the single precision IEEE 754 floating-point format. If a more stringent accuracy requirement is necessary, 0.3% accuracy degradation, the representation with one additional bit in the mantissa can

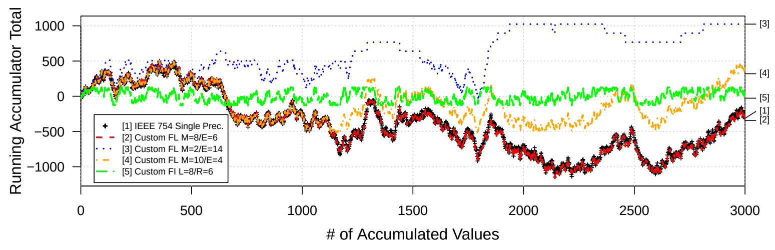


Figure 5.8: The accumulation of weighted neuron inputs for a specific neuron with various customized precision DNNs as well as the IEEE 754 single precision floating point configuration for reference. FL and FI are used to abbreviate floating point and fixed-point, respectively. The format parameters are as follows: M=mantissa, E=exponent, L=bits left of radix point, R=bits right of radix point.

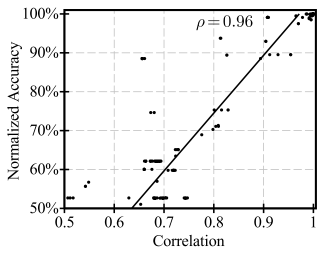


Figure 5.9: The linear fit from the correlation between normalized accuracy and last layer activations of the exact and customized precision DNNs.

be used, which achieves a 5.7× speedup and 3.0× energy savings.

5.3.3 Sources of Accumulation Error

In order to understand how customized precision degrades DNN accuracy among numeric representations, we examine the impact of various reduced precision computations on a neuron. Figure 5.8 presents the serialized accumulation of neuron inputs in the third convolution layer of AlexNet. The x-axis represents the number of inputs that have been accumulated, while the y-axis represents the current value of the running sum. The black line represents the original DNN computation, a baseline for customized precision settings to match. We find two causes of error between the customized precision fixed-point and floating-point representations, saturation and excessive rounding.

In the fixed-point case (green line, representing 16 bits with the radix point in the center), the central cause of error is from saturation at the extreme values. The running sum exceeds 255, the maximum representable value in this representation, after 60 inputs are accumulated, as seen in the figure. After reaching saturation, the positive values are discarded and the final output is unpredictable. Although floating-point representations do not saturate as easily, the floating-point configuration with 10 mantissa bits and 4 exponent bits (orange line) saturates after accumulating 1128 inputs. Again, the lost information from saturation causes an unpredictable final output.

For the next case, the floating-point configuration with 2 bits and 14 bits for the mantissa and exponent (blue line), respectively, we find that the lack of precision for large values causes excessive rounding errors. As shown in the figure, after accumulating 120 inputs, this configuration's running sum exceeds 256, which limits the minimum adjustment in magnitude to 64 (the exponent normalizes the mantissa to 256, so the two mantissa bits represent 128 and 64). Finally, one of the customized precision types that has high performance and accuracy for AlexNet, 8 mantissa bits and 6 exponent bits (red line), is shown as well. This configuration almost perfectly matches the IEEE 754 floating-point configuration, as expected based on the final output accuracy.

The other main cause of accuracy loss is from values that are too small to be encoded as a non-zero value in the chosen customized precision configuration. These values, although not critical during addition, cause significant problems when multiplied with a large value, since the output should be encoded as a non-zero value in the specific precision setting. We found that the weighted input is minimally

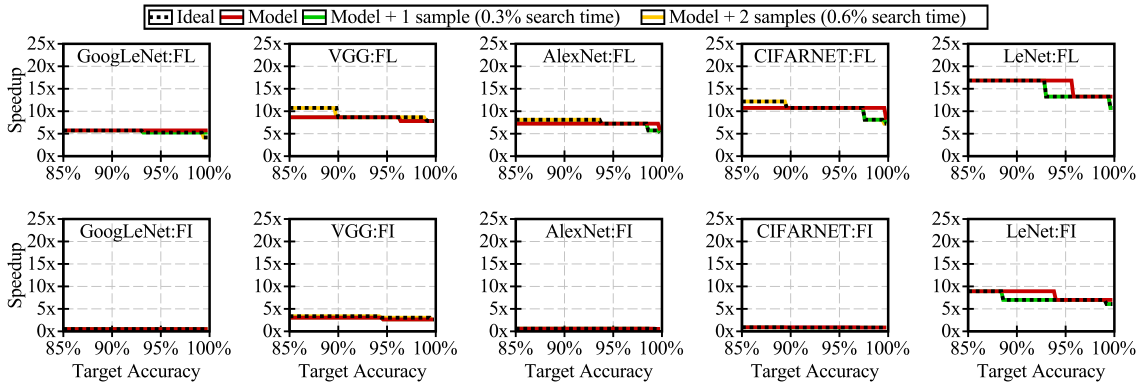


Figure 5.10: The speedup achieved by selecting the customized precision using an exhaustive search (i.e. the ideal design) and prediction using the accuracy model with accuracy evaluated for some number of configurations (model + X samples). The floating-point (FL) and fixed-point (FI) results are shown in the top and bottom rows, respectively. The model with two evaluated designs produces the same configurations, but requires <0.6% of the search time.

impacted, until the precision is reduced low enough for the weight to become zero.

While it may be intuitive based on these results to apply different customized precision settings to various stages of the neural network in order to mitigate the sudden loss in accuracy, the realizable gains of multi-precision configurations present significant challenges. The variability between units will cause certain units to be unused during specific layers of the neural network causing gains to diminish (e.g., 11-bit units are idle when 16-bit units are required for a particular layer). Also, the application specific hardware design is already an extensive process and multiple customized precision configurations increases the difficulty of the hardware design and verification process.

5.3.4 Customized Precision Search

Now we evaluate our proposed customized precision search method. The goal of this method is to significantly reduce the required time to navigate the customized precision design space and still provide an optimal design choice in terms of speedup, limited by an accuracy constraint.

Correlation model. First, we present the linear correlation-accuracy model in Figure 5.9, which shows the relationship between the normalized accuracy of each setting in the design space and the correlation between its last layer activations compared to those of the original NN. This model, although built using all of the customized precision configurations from AlexNet, CIFARNET, and LeNet-5 neural networks, produces a good fit with a correlation of 0.96. It is important that the model matches across networks and precision design choices (e.g., floating point versus fixed point), since creating this model for each DNN, individually, requires as much time as exhaustive search.

Validation. To validate our search technique, Figure 5.10 presents the accuracy-speedup trade-off curves from our method compared to the ideal design points. We first obtain optimal results via exhaustive search. We present our search with a variable number of refinement iterations, where we evaluate the accuracy of the current design point and adjust the precision if necessary. To verify robustness, the accuracy models were generated using cross-validation where all configurations in the DNN being searched are excluded (e.g., we build the AlexNet model with LeNet and CIFARNET accuracy/correlation pairs). The prediction is made using only ten randomly selected inputs, a tiny subset compared that needed for classification accuracy, some of which are even incorrectly classified by the original neural network. Thus, the cost of prediction using the model is negligible.

We observe that, in all cases, the accuracy model combined with the evaluation of just two customized precision configurations provides the same result as the exhaustive search. Evaluating two designs out of 340 is 170× faster than exhaustively evaluating all designs. When only one configuration is evaluated instead of two (i.e. a further 50% reduction is search time), the selected customized precision setting never violates the target accuracy, but concedes a small amount of performance. Finally, we note that our search mechanism, without evaluating inference accuracy for any of the design points, provides a representative prediction of the optimal customized precision setting. Although occasionally violating the target accuracy (i.e. the cases where the speedup is higher than the exhaustive search), this prediction can be used to gauge the amenability of the NN to customized precision without investing any considerable amount of time in experimentation.

Speedup. We present the final speedup produced by our search method in Figure 5.11 when the algorithm is configured for 99% target accuracy and to use two samples for refinement. In all cases, the chosen customized precision configuration meets the targeted accuracy constraint. In most cases, we find that the larger networks require more precision (DNNs are sorted from left to right in descending order based on size). VGG requires less precision than expected, but VGG also uses smaller convolution kernels than all of the other DNNs except LeNet-5.

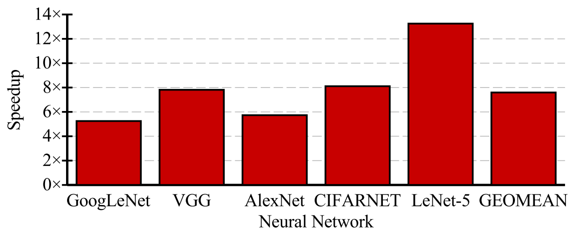


Figure 5.11: The speedup resulting from searching for the fastest setting with less than 1% inference accuracy degradation. All selected customized precision DNNs meet this accuracy constraint.

5.4 Summary

This chapter introduced the importance of carefully considering customized precision when realizing neural networks. We show that using the IEEE 754 single precision floating point representation in hardware results in surrendering substantial performance. On the other hand, picking a configuration that has lower precision than optimal will result in severe accuracy loss. By reconsidering the representation from the ground up in designing custom precision hardware and using our search technique, we find an average speedup across deployable DNNs, including GoogLeNet and VGG, of 7.6× with less than 1% degradation in inference accuracy.

CHAPTER VI

Conclusion

A widespread interest in artificial intelligence in applications has grown in recent years, including from companies such as Apple, Google, Microsoft and Amazon. To provide this intelligence, these applications employ algorithms from domains such as computer vision, image processing, pattern recognition and machine learning. Across these algorithms, two common characteristics have emerged. First, the massive datasets and computationally intensive logic required to execute these algorithms present a significant challenge for existing infrastructure. Second, these datasets are gathered from inherently noisy and imprecise inputs, resulting in computation of outputs that are statistical in nature. Although these algorithms necessarily require resilience to inaccuracies, each output is computed with as much precision as in conventional applications. This substantial fraction of redundant computations leads existing infrastructure to struggle to deliver the throughput for emerging intelligent applications. There remain three key challenges that must be addressed in order to remove this redundant computation and help bridge the performance gap.

First, removal of redundant computation cannot widespread adoption until practitioners can trust that the final output will be sufficiently accurate. Second, many of these emerging applications rely on hardware acceleration in order to meet throughput demands. However, existing approaches to removing redundant computation only consider conventional CPU architectures. We find that the underlying architecture must be carefully considered in order to remove computation without compromising the software's efficiency when executing on accelerator hardware. Finally, even further exploration is required when cutting redundant work from custom accelerators. We find a void in existing work in regards to the numerical representation being employed on custom accelerators, leaving untapped performance gains for these accelerators.

In this dissertation, we address these challenges to increase the adoption of exploiting error tolerance. In the first chapter, we present a statistical mechanism to provide statistical accuracy guarantees. This statistical machinery, integrated into a runtime system, allows practitioners to provide statistical accuracy bounds on the approximations applied in their applications. In the second chapter, we consider how to remove computation efficiently when executed on GPU accelerators. We find that work skipping must be done in a very structured way in order to find benefits due to the wide vectors units present on the GPU. Additionally, we find that a central bottleneck for these computationally intensive algorithms lies in the on-chip memory. Using these insights, we develop two techniques to substantially improve performance on GPUs with negligible loss in accuracy. Finally, in the third chapter, we explore the numerical representation landscape to vastly improve performance.

We find that a customized-precision floating-point representation offers significant performance improvements with little loss in accuracy.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. BlinkDB: Queries with bounded errors and bounded response times on very large data. In *European Conference on Computer Systems (EuroSys*, 2013.

[2] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[3] Amazon. Amazon Echo. `https://www.amazon.com/Amazon-Echo-Bluetooth-Speaker-with-WiFi-Alexa/dp/B00X4WHP5E`, 2016. [Online; accessed 15-August-2016].

[4] Jason Ansel, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. In *Code Generation and Optimization (CGO)*, 2010.

[5] Apple. Siri. `https://www.apple.com/ios/siri`, 2016. [Online; accessed 15-August-2016].

[6] Woongki Baek and Trishul M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *Programming Language Design and Implementation (PLDI)*, 2010.

[7] Andrew C Berry. The accuracy of the gaussian approximation to the sum of independent variates. *Transactions of the American Mathematical Society*, 49(1):122–136, 1941.

[8] James Bornholt, Todd Mytkowicz, and Kathryn S McKinley. Uncertain<T>: A first-order type for uncertain data. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[9] Brett Boston, Adrian Sampson, Dan Grossman, and Luis Ceze. Probability type inference for flexible approximate programming. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2015.

[10] Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *Programming Language Design and Implementation (PLDI)*, 2012.

[11] Michael Carbin, Sasa Misailovic, and Martin C Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2013.

[12] Lukas Cavigelli, David Gschwend, Christoph Mayer, Samuel Willi, Beat

Muheim, and Luca Benini. Origami: A convolutional network accelerator. In *Great Lakes Symposium on VLSI (GLSVLSI)*, 2015.

[13] Lukas Cavigelli, Michele Magno, and Luca Benini. Accelerating real-time embedded scene labeling with convolutional networks. In *Design Automation Conference (DAC)*, 2015.

[14] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. A dynamically configurable coprocessor for convolutional neural networks. In *International Symposium on Computer Architecture (ISCA)*, 2010.

[15] Surajit Chaudhuri, Gautam Das, and Vivek Narasayya. Optimized stratified sampling for approximate query processing. *Transactions on Database Systems.*, 2007.

[16] Swarat Chaudhuri, Sumit Gulwani, Roberto Lublinerman, and Sara Navidpour. Proving programs robust. In *Foundations of Software Engineering (FSE)*, 2011.

[17] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[18] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Q Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. In *32nd International Conference on Machine Learning*, pages 2285–2294, 2015.

[19] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Dadiannao: A machine-learning supercomputer. In *International Symposium on Microarchitecture (MICRO)*, 2014.

[20] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. 2014.

[21] Francesco Conti and Luca Benini. A ultra-low-energy convolution engine for fast brain-inspired vision in multicore clusters. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2015.

[22] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Low precision arithmetic for deep learning. 2014.

[23] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Low precision arithmetic for deep learning. *CoRR*, abs/1412.7024, 2014.

[24] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc extquotesingle aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. Large scale distributed deep networks. In *Neural Information Processing Systems (NIPS)*, 2012.

[25] Peter U Diehl and Matthew Cook. Efficient implementation of stdp rules on spinnaker neuromorphic hardware. In *International Joint Conference on Neural Networks*, 2014.

[26] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman Amarasinghe. Autotuning algorithmic choice for input sensitivity. In *Programming Language Design and Implementation (PLDI)*, 2015.

[27] Zidong Du, Avinash Lingamneni, Yunji Chen, Krishna Palem, Olivier Temam, and Chengyong Wu. Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2014.

[28] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture support for disciplined approximate programming. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[29] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *International Symposium on Microarchitecture (MICRO)*, 2012.

[30] Clement Farabet, Camille Couprie, Laurent Najman, and Yann LeCun. Learning hierarchical features for scene labeling. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 35(8):1915–1929, 2013.

[31] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. Neuflow: A runtime reconfigurable dataflow processor for vision. In *Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2011.

[32] Michael Figurnov, Dmitry Vetrov, and Pushmeet Kohli. Perforatedcnns: Acceleration through elimination of redundant convolutions. 2015.

[33] Klint Finley. Facebook open-sources a trove of AI tools, 2015.

[34] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *International Symposium on Microarchitecture (MICRO)*, 2007.

[35] Ross Girshick. Fast r-cnn. 2015.

[36] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[37] Google. Google Now. `https://www.google.com/landing/now`, 2016. [Online; accessed 15-August-2016].

[38] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. 2015.

[39] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[40] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. 2015.

[41] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. 2015.

[42] Awni Y. Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deepspeech: Scaling up end-to-end speech recognition. 2014.

[43] John F Hart. *Computer approximations*. Krieger Publishing Co., Inc., 1978.

[44] Johann Hauswald, Yiping Kang, Michael A Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G Dreslinski, Jason Mars, and Lingjia Tang. Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers. In *International Symposium on Computer Architecture (ISCA)*, 2015.

[45] Johann Hauswald, Michael A. Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G. Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, and Jason Mars. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[46] Henry Hoffmann, Jonathan Eastep, Marco Santambrogio, Jason Miller, and Anant Agarwal. Application heartbeats for software performance and health. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[47] Henry Hoffmann, Sasa Misailovic, Stelios Sidiroglou, Anant Agarwal, and Martin Rinard. Using code perforation to improve performance, reduce energy consumption, and respond to failures. In *MIT Tech Report (MIT-CSAIL-TR-2009-042)*, 2009.

[48] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Margin Rinard. Dynamic knobs for responsive power-aware computing. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 199–212. ACM, 2011.

[49] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *International Symposium on Computer Architecture (ISCA)*, 2009.

[50] Animesh Jain, Parker Hill, Shih-Chieh Lin, Muneeb Khan, Md E. Haque, Michael A. Laurenzano, Scott Mahlke, Lingjia Tang, and Jason Mars. Concise loads and stores: The case for an asymmetric compute-memory architecture for approximation. In *International Symposium on Microarchitecture (MICRO)*, 2016.

[51] Animesh Jain, Parker Hill, Shih-Chieh Lin, Muneeb Khan, Michael A. Laurenzano, Scott Mahlke, Lingjia Tang, and Jason Mars. Concise loads and stores:

The case for an asymmetric compute-memory architecture for approximation. In *International Symposium on Microarchitecture (MICRO)*, 2016.

[52] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. 2014.

[53] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Proteus: Exploiting numerical precision variability in deep neural networks. In *International Conference on Supercomputing (ICS)*, 2016.

[54] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[55] Sergey Karayev, Matthew Trentacoste, Helen Han, Aseem Agarwala, Trevor Darrell, Aaron Hertzmann, and Holger Winnemoeller. Recognizing image style. 2013.

[56] Joo-Young Kim, Minsu Kim, Seungjin Lee, Jinwook Oh, Kwanho Kim, and Hoi-Jun Yoo. A 201.4 gops 496 mw real-time multi-object recognition processor with bio-inspired neural perception engine. In *Journal of Solid-State Circuits (JSSC)*, 2010.

[57] V Yu Korolev and IG Shevtsova. On the upper bound for the absolute con-

stant in the berry-esseen inequality. *Theory of Probability & Its Applications*, 54(4):638–658, 2010.

[58] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. *Computer Science Department, University of Toronto, Tech. Rep*, 1(4):7, 2009.

[59] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[60] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 2012.

[61] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Neural Information Processing Systems (NIPS)*, 2012.

[62] Hugo Larochelle, Dumitru Erhan, Aaron Courville, James Bergstra, and Yoshua Bengio. An empirical evaluation of deep architectures on problems with many factors of variation. In *International Conference on Machine learning (ICML)*, 2007.

[63] Michael A Laurenzano, Parker Hill, Mehrzad Samadi, Scott Mahlke, Jason Mars, and Lingjia Tang. Input responsiveness: using canary inputs to dynamically steer approximation. In *Programming Language Design and Implementation (PLDI)*, 2016.

[64] Andrew Lavin. maxdnn: An efficient convolution kernel for deep learning with maxwell gpus. 2015.

[65] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. 2015.

[66] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, 1998.

[67] Yann LeCun, Corinna Cortes, and Christopher JC Burges. The mnist database of handwritten digits, 1998.

[68] Yongsoon Lee, Younhee Choi, Seok-Bum Ko, and Moon Ho Lee. Performance analysis of bit-width reduced floating-point arithmetic units in fpgas: a case study of neural network-based face detector. In *EURASIP Journal on Embedded Systems*, 2009.

[69] Boxun Li, Yuzhi Wang, Yu Wang, Yuanfeng Chen, and Huazhong Yang. Training itself: Mixed-signal training acceleration for memristor-based neural network. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2014.

[70] Xiangjun Li and Jianfei Cai. Robust transmission of jpeg2000 encoded images over packet loss channels. In *International Conference on Multimedia and Expo*, 2007.

[71] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. Flikker: Saving DRAM refresh-power through data partitioning. In *Architec-*

tural Support for Programming Languages and Operating Systems (ASPLOS), 2011.

[72] Divya Mahajan, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, and Hadi Esmaeilzadeh. Towards statistical guarantees in controlling quality trade-offs for approximate acceleration. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[73] Microsoft. Microsoft Cortana. `https://www.microsoft.com/en-us/mobile/experiences/cortana`, 2016. [Online; accessed 15-August-2016].

[74] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C Rinard. Chisel: Reliability-and accuracy-aware optimization of approximate computational kernels. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2014.

[75] Sasa Misailovic, Deokhwan Kim, and Martin Rinard. Parallelizing sequential programs with statistical accuracy tests. In *Transactions on Embedded Computing Systems (TECS)*, 2013.

[76] Sasa Misailovic, Daniel M Roy, and Martin C Rinard. Probabilistically accurate program transformations. In *Static Analysis*. 2011.

[77] Thierry Moreau, Mark Wyse, Jacob Nelson, Adrian Sampson, Hadi Esmaeilzadeh, Luis Ceze, and Mark Oskin. Snnap: Approximate computing on

programmable socs via neural acceleration. In *High Performance Computer Architecture (HPCA)*, 2015.

[78] Lorenz K Muller and Giacomo Indiveri. Rounding methods for neural networks with low resolution synaptic weights. *arXiv preprint arXiv:1504.05767*, 2015.

[79] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *International Conference on Machine Learning (ICML)*, 2010.

[80] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. Improving gpu performance via large warps and two-level warp scheduling. In *International Symposium on Microarchitecture (MICRO)*, 2011.

[81] Rajib Nath, Stanimire Tomov, and Jack Dongarra. Accelerating gpu kernels for dense linear algebra. In *High Performance Computing for Computational Science (VECPAR)*, 2010.

[82] Jiquan Ngiam, Adam Coates, Ahbik Lahiri, Bobby Prochnow, Quoc V Le, and Andrew Y Ng. On optimization methods for deep learning. In *International Conference on Machine Learning (ICML)*, 2011.

[83] M-E. Nilsback and A. Zisserman. Automated flower classification over a large number of classes. In *Indian Conference on Computer Vision, Graphics and Image Processing (ICVGIP)*, 2008.

[84] Nitin, Mithuna Thottethodi, TN Vijaykumar, and Milind Kulkarni. Stratified online sampling for sound approximation in MapReduce. 2015.

[85] Nvidia. cuBLAS, 2017.

[86] Nvidia. cuSPARSE, 2017.

[87] Nvidia. GeForce GTX TITAN X, Specifications, 2017.

[88] Nvidia. Parallel Thread Execution ISA Version 5.0, 2017.

[89] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric Chung. Accelerating deep convolutional neural networks using specialized hardware, 2015.

[90] Eustace Painkras, Luis A Plana, Jim Garside, Steve Temple, Simon Davidson, Jeffrey Pepper, David Clark, Cameron Patterson, and Steve Furber. Spinnaker: a multi-core system-on-chip for massively-parallel neural net simulation. In *Custom Integrated Circuits Conference (CICC)*, 2012.

[91] Robert Preissl, Theodore M Wong, Pallab Datta, Myron Flickner, Raghavendra Singh, Steven K Esser, William P Risk, Horst D Simon, and Dharmendra S Modha. Compass: a scalable simulator for an architecture for cognitive computing. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[92] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David

Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[93] Martin Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *International Conference on Supercomputing (ICS)*, 2006.

[94] Martin Rinard. Probabilistic accuracy bounds for perforated programs. In *Workshop on Partial Evaluation and Program Manipulation*, 2011.

[95] Michael Ringenburg, Adrian Sampson, Isaac Ackerman, Luis Ceze, and Dan Grossman. Monitoring and debugging the quality of results in approximate programs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[96] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. 2015.

[97] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. Paraprox: pattern-based approximation for data parallel applications. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[98] Mehrzad Samadi, Janghaeng Lee, D Anoushe Jamshidi, Amir Hormati, and

Scott Mahlke. SAGE: Self-tuning approximation for graphics engines. In *International Symposium on Microarchitecture (MICRO)*, 2013.

[99] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. In *Programming Language Design and Implementation (PLDI)*, 2011.

[100] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. Approximate storage in solid-state memories. *Transactions on Computer Systems (TOCS)*, 2014.

[101] Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S McKinley, Dan Grossman, and Luis Ceze. Expressing and verifying probabilistic assertions. In *Programming Language Design and Implementation (PLDI)*, 2014.

[102] Joshua San Miguel, Jorge Albericio, Natalie Enright Jerger, and Aamer Jaleel. The bunker cache for spatio-value approximation. In *International Symposium on Microarchitecture (MICRO)*, 2016.

[103] John Sartori and Rakesh Kumar. Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications. In *Transactions on Multimedia*. 2013.

[104] Kaz Sato, Cliff Young, and David Patterson. An in-depth look at Google's first Tensor Processing Unit (TPU), 2017.

[105] Johannes Schemmel, Johannes Fieres, and Karlheinz Meier. Wafer-scale integration of analog neural networks. In *International Joint Conference on Neural Networks (IJCNN)*, 2008.

[106] Linda Shapiro and George C Stockman. Computer vision. 2001. *ed: Prentice Hall*, 2001.

[107] Irina Shevtsova. An improvement of convergence rate estimates in the lyapunov theorem. *Doklady Mathematics*, 82(3):862–864, 2010.

[108] Irina Shevtsova. On the absolute constants in the berry-esseen type inequalities for identically distributed summands. *arXiv preprint arXiv:1111.6554*, 2011.

[109] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Foundations of Software Engineering (FSE)*, 2011.

[110] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. 2014.

[111] Joseph Sloan, David Kesler, Rakesh Kumar, and Ali Rahimi. A numerical optimization-based methodology for application robustification: Transforming applications for error tolerance. In *Dependable Systems and Networks (DSN)*, 2010.

[112] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. Eon: A language and runtime system

for perpetual systems. In *Conference on Embedded Networked Sensor Systems*, 2007.

[113] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. 2014.

[114] Ilya Sutskever, Oriol Vinyals, and Quoc VV Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, pages 3104–3112, 2014.

[115] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[116] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lars Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Computer Vision and Pattern Recognition (CVPR)*, pages 1701–1708, 2014.

[117] Olivier Temam. A defect-tolerant accelerator for emerging high-performance applications. In *International Symposium on Computer Architecture (ISCA)*, 2012.

[118] Nikolaos Thomos, Nikolaos V Boulgouris, and Michael G Strintzis. Optimized transmission of jpeg2000 streams over wireless channels. *Transactions on Image Processing*, 2006.

[119] I. S. Tyurin. On the accuracy of the gaussian approximation. *Doklady Mathematics*, 80(3):840–843, 2009.

[120] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop*, 2011.

[121] Thomas Y. Yeh, Petros Faloutsos, Milos Ercegovac, Sanjay J. Patel, and Glenn Reinman. The art of deception: Adaptive precision reduction for area efficient physics acceleration. In *International Symposium on Microarchitecture (MICRO)*, 2007.

[122] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Neural Information Processing Systems (NIPS)*, 2014.

[123] Jianming Zhang, Shuga Ma, Mehrnoosh Sameki, Stan Sclaroff, Margrit Betke, Zhe Lin, Xiaohui Shen, Brian Price, and Radomír Měch. Salient object subitizing. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[124] Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A. Kelner, and Martin Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *Principles of Programming Languages (POPL)*, 2012.