GoldenEye: A Platform for Evaluating Emerging Numerical Data Formats in DNN Accelerators

Abdulrahman Mahmoud[1] Thierry Tambe[1] Tarek Aloui[1] David Brooks[1,2] Gu-Yeon Wei [1,3]

[1]Harvard University, [2]Meta, [3]Samsung Electronics

***Abstract*—This paper presents GoldenEye, a functional simulator with fault injection capabilities for common and emerging numerical formats, implemented for the PyTorch deep learning framework. GoldenEye provides a unified framework for numerical format evaluation of DNNs, including traditional number systems such as fixed and floating point, as well as recent DNN-inspired formats such as block floating point and AdaptivFloat. Additionally, GoldenEye enables single- and multi- bit flips at various logical and functional points during a value's lifetime for resiliency analysis, including for the first time attention to numerical values' hardware metadata. This paper describes GoldenEye's technical design and implementation which make it an easy-to-use, extensible, versatile, and fast tool for dependability research and future DNN accelerator design. We showcase its utility with three case studies: a unifying platform for number system comparison and evaluation, a design-space exploration heuristic for data type selection, and fast DNN reliability analysis for different error models. GoldenEye is open-sourced and available at: https://github.com/ma3mool/goldeneye.**

I. Introduction

Computer number formats form the underlying representation of real numbers in modern digital device hardware. Precise specifications and implementations of computer number formats form the basis of hardware arithmetic and functional units in processors. However, the use of binary logic introduces an inherent approximation of real numerical values, and thus introduces the rise of many representations of numbers in hardware. Since the late 1970s, more than 50 different floating point representations have found their way into commercial processors [27]. This diversity of value representations eventually led to the IEEE-754 standard [1], which aimed to insure reliability and portability of numerical values across processors.

With the recent rise of deep learning (DL), there has been a renewed interest in number formats, primarily for their potential performance benefits over traditional IEEE-754 floating point computations. The fundamental property being explored is the tradeoff between *precision* and *range* offered by the underlying number representation, in the context of DL applications. Consequently, many recent formats have diverged from the IEEE-754 single- and half- precision formats in search for number formats with faster arithmetic computational properties, with minimal accuracy loss for their higher-level algorithm objectives (such as classification accuracy in CNNs or specific task accuracy in Transformer).

The exploration of newer number formats in the context of deep learning is challenging, primarily since the majority of compute fabrics available today (namely, CPUs and GPUs) support a limited set of number formats in the hardware. Consequently, software computing platforms (e.g., CUDA [28], OpenCL [21]) and DL frameworks (e.g., PyTorch [30], TensorFlow [2]) are forced to optimize along restrictive dimensions in the space of possible number formats. Many recent performance optimizations in hardware (such as integer quantization) focus on only certain values (such as 8-bit support), which may be not be the optimal bitwidth choice when exploring hardware/software co-design opportunities for future DL accelerators.

Another important design consideration for number format selection in the context of DL accelerator design is the algorithmic reliability of an application in face of transient hardware errors [9]. While precision, dynamic range, arithmetic performance, and area are generally high on the list of design tradeoffs typically explored for DL acceleration, DL robustness to errors (as a function of the underlying number representation) is also a critical component which should inform the number format selection as a first-order parameter. Prior work has found that transient bit flips in exponent bits of IEEE-754 32-bit floating points can lead to erroneous classifications in CNNs [5], [22]. Additionally, recent work has found that even single bit flips in quantized INT8 formats can lead to silent data corruptions (SDCs), especially when the network has lower confidence in an inference [25].

To enable and propel research in this avenue, we present GoldenEye, a functional simulator with fault injection capabilities for common and emerging numerical formats. In addition to being an extensible playground for novel data format exploration, GoldenEye presents a unified framework for evaluating number formats and their effect on DNN classification accuracy. Our implementation is designed to be research-friendly (easy to explore or add new number formats), and is also reasonably fast at both simulating the number format and running error injection campaigns. GoldenEye currently includes 5 configurable number formats (§III), and is engineered in a way to easily incorporate future number formats. Furthermore, GoldenEye's error injection capability uniquely pays attention to a number format's hardware implementation *metadata*. Our objective in performing single-bit flips into different number formats aims to 1) illustrate that the concept of a "single-bit flip" in software can mean many different things in hardware, and 2) provide a starting point for future *hardware-aware* number format exploration and functional error modeling in software. Finally, we explore three uses cases (§IV) to showcase GoldenEye's utility as a research tool.

In summary, the contributions of this work are:

- A rich, open-sourced framework for number format evaluation for DL models.
- A fast and extensible code base, enabling fast prototyping in Python as well as C++/CUDA acceleration potential.
- Fast error injection support for both data values as well as hardware-aware metadata. We study a total of 8 different single-bit injection error sites informed by the number format representations.

- We showcase three use case of GoldenEye for DL accelerator design, including accuracy measurements, design space exploration of number formats, and DL model robustness.

II. Background

We provide a background on common number formats for DNNs (§II-A), their relevance to DNN model resilience (§II-B), and their impact on today's DL accelerator design space (§II-C).

A. Number Formats for DL, Terminology, and Notations

The underlying number format for DNNs has been extensively studied in an attempt to improve computational efficiency for DNN training and inference. Two driving factors for this exploration stem from the application domain of DL. First, DL models have many statistical properties, which allow variable relaxation of precision without significant loss of accuracy. Second, the predominant computation in a DNN, the multiple-and-accumulate operation (or MAC), is mathematically a dot product, which can enable various hardware optimizations for improved speed and/or area.

The IEEE-754 32-bit floating point (or FP32) has historically been the predominant number format in use, due to its wide adoption in CPUs and GPUs, and as a standard for hardware design. FP32 has a bit-width of 32 bits, divided into 3 regions: 1 sign bit, 8 exponent bits, and 23 mantissa bits. We use the classic notation "e8m23" to capture the bit assignments (in this paper, we assume all number formats to be signed unless otherwise noted). Various "named" floating point (FP) formats include half precision (e5m10), bfloat (e8m7) [18], TensorFloat (e8m10) [20], and DLFloat (e6m9) [3].

Fixed point (FxP) formats do not have exponent bits, and instead are split into an integer portion and a fractional portion. FxP have a reduced dynamic range, but that simplifies the hardware for FxP arithmetic units. Similar to FP, FxP formats have been explored for both DL training and inference [13], [31]. For both FP and FxP, we use the term *radix* to denote the bit position (from the LSB) which separates the exponent/integer from the mantissa/fraction.

Integer Quantization (INT) is a form of a fixed point format, where there are no fractional bits (only integer values). Unlike a native FxP value, integer quantization typically involves a *scaling factor*, which uniformly maps values from one number format (such as FP32) to a lower precision integer format (i.e., INT8). Integer quantization has shown immense potential in DL algorithm design and acceleration, due to its simple hardware structures in addition to the small impact on overall model accuracy [12], [14].

While FP, FxP, and INT have traditionally shown great promise for DL efficiency, newer formats derived from these have recently been proposed as improvements. Block Floating Point (BFP) [19], for example, leverages the concept of a *shared* exponent, allowing a tensor to significantly reduce its memory footprint by only saving one exponent (e.g., 8 bits) for the entire tensor. BFP has had recent traction in DL and accelerator design, due to this potential hardware performance optimization [10], [33], [40]

AdaptivFloat (AFP) [37] is another FP derivative, which uses a shared *exponent bias*, adaptively shifting the range of representable values on the floating point scale to where it is most needed for a set of tensor values. It has also shown potential for transformers, and has been adapted for recent DL accelerator designs [35], [36].

Figure 1 summarizes and illustrates the conceptual and implementation differences of the 5 number formats studied in this paper.

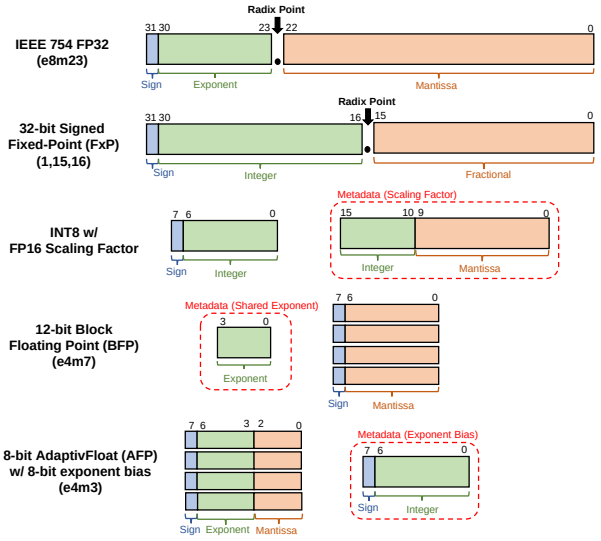


Fig. 1: Illustration of number formats explored in this paper.

B. Reliability of DL Models

Understanding the impact of a hardware error on the resilience of software is intrinsically tied to the number format. Prior work has traditionally explored the "single-bit flip" error model, which many-times implicitly assumes an IEEE-754 FP32 implementation under the hood [5], [15], [23], [32]. However, as new number formats emerge each with different underlying hardware implementations, the software construct of a single-bit flip requires adjusting for hardware-aware resiliency analysis.

For example, as described in §II-A, BFP uses the concept of a shared exponent. While BFP value-wise is similar to FP, the hardware implementation is sufficiently different targeting performance and bandwidth optimization. Thus, a single bit flip in the shared exponent bit of BFP is actually equivalent to a multi-bit flip across the entire tensor of a traditional FP format. Such hardware-aware modeling is imperative for software-directed error resilience and analysis. To that end, a primary contribution of this work is to elevate the hardware implementation of a number format to the software, to enable more accurate hardware error modeling for DL models.

C. DL Accelerator Design

A major motivation for the design and open-source release of GoldenEye is the need for better modeling tools for DL accelerators. Current computing architectures (such as GPUs) have shaped and defined neural network design in recent years. For example, TensorRT [39] provides a (closed-source) automatic quantization optimization for GPU acceleration. However, since GPUs might only support 8-bit precision for INT, this may not be the optimal configuration for the DL model topology, and potentially leaves both performance and energy efficiency opportunity untouched. Since designing new optimized general-purpose hardware for different number systems is both costly and impractical, it is important to have good modeling tools for this space exploration. Furthermore, maintaining resilience as a first-order design constraint is also crucial, particularly as many DL domains are safety critical, such as autonomous driving and medical devices. To the best of our knowledge, GoldenEye is the first open-source tool providing both these attributes, in addition to being easy-to-use, fast, and extensible as a research tool.

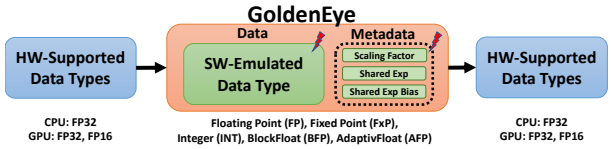


Fig. 2: GoldenEye design overview. GoldenEye is applied at a layer-granularity, supporting different data type emulations and allows error perturbations in both the data values and hardware-aware metadata.

III. GoldenEye Design

GoldenEye is a functional simulator of number formats for DL exploration, implemented in the popular PyTorch [30] framework. GoldenEye is designed in such a way that enables the study of current and emerging number formats for accuracy, precision, and reliability. This section describes the underlying implementation details of GoldenEye, which allow it to be fast, extensible, and a versatile tool for dependability research.

A. GoldenEye Implementation Overview

Modern hardware such as CPUs and GPUs support a limited set of number formats. In addition to the standard IEEE-754 32-bit floating point (e8m23), we also find lower precision formats such as Half (e5m10), BFloat (e8m7), TensorFloat (e8m10), and INT-8. Evaluating a number format that is not already supported in hardware requires simulating the number system in software [3], [18], [37].

Figure 2 shows an overview of GoldenEye. While the *compute fabric* of the hardware may support FP32, we design a full system on top of it in software to emulate any arbitrary number system. GoldenEye leverages PyTorch's `hook` functionality to perform number format emulation at the layer granularity. This requires reading the original value (e.g., FP32), converting it to the nearest supported value in the number format being emulated (e.g., AdaptivFloat), and then writing the number back at the nearest numerical value in the HW-supported number system (e.g., FP32). In the process, we can also extract hardware specific metadata (such as shared exponents or scaling factors), which may be abstracted in software but will typically be stored in a dedicated register or storage structure in hardware. This allows us to decouple the hardware *implementation* of the number from the *numeric value* it represents.

In order to make GoldenEye an extensible research tool, we define an API requirement for any number system implementation (§III-B). This API gives users the flexibility to implement an arbitrary number system, while preserving the end-to-end semantics to allow the DL model to run seamlessly on top of the desired compute fabric (e.g., a GPU). Furthermore, it allows the user to expose any hardware structures as metadata, which can consequently allow bit flips during a resiliency analysis to functionally represent hardware structures as desired.

B. GoldenEye API

We provide a hierarchical number format class, which enables inheritance and pure virtual methods for implementation. The initialization of the class provides base knobs for the number system (such as `bit_width` and `radix`), and can be extended as needed (such as adding `exp_bias` for ADP).

We define four pure virtual methods in GoldenEye, whose implementation needs to be provided for an arbitrary number system. The four methods are:

1) ***tensor*** **`real_to_format_tensor`**(***tensor***)
2) ***tensor*** **`format_to_real_tensor`**(***tensor***)
3) ***bitstring*** **`real_to_format`**(***value***)
4) ***value*** **`format_to_real`**(***bitstring***)

Method 1 reads in a tensor of values in the number format of the compute fabric (e.g., FP32), and performs the necessary software conversions to implement the desired number format. Method 2 performs the reverse computation, going from the number format to the "real" value in hardware. We provide a default implementation for method 2, as it can simply be a cast operation to `torch.float32`.

Since Methods 1 and 2 are implemented on an entire tensor, these operations are very fast, especially if leveraging PyTorch's built-in methods and operators for arithmetics. Furthermore, they can be accelerated by leveraging C++/CUDA calls underneath the hood (as discussed in §III-C). This differs from Methods 3 and 4, which are scalar operations and are much slower, but provide fine-grained error injection support.

Method 3 converts a value into its *bitstream* equivalent as a list, adhering to the number format's interpretation, while Method 4 performs the reverse operation (bitstream to value). These latter two functions streamline error injection operations in a value's data, one of the features provided by GoldenEye. The API can thus be invoked in an abstract routine for error injections, by calling Method 3, flipping a bit, then calling Method 4 sequentially. We use the PyTorchFI [24] tool to accelerate this routine for data value injections where appropriate.

While the scalar operations (and, more generally, the error injection routine) is slower due to bit manipulations, we find that the overhead is negligible since this happens infrequently (i.e., a single bit flip during a DNN inference). GoldenEye's API aims to provide both the speed (via tensor manipulation) and the granular control of values (via scalar operation) to the researcher, while abstracting the complexity of PyTorch's `hook`'s implementation to improve user productivity.

Metadata support, both for number format emulation and error injections, is supported at the `class` level. For example, the shared exponent for BFP is computed and stored in the number format class, and can directly be manipulated during an error injection via scalar operations (similar to Methods 3 and 4). Since the metadata can differ across number systems, we found this implementation the most natural, and it allows the flexibility to model the hardware to the level of detail required.

Currently, we provide support for 5 number formats as described in §II-A, along with their tunable parameters. These generalizations allow us to support many previous number formats (such as bfloat and TensorFloat) as a parameter tuning of the base class (FP). Further, new formats can be designed and incorporated as described above by implementing the four pure virtual functions with hardware-aware metadata support. We provide support for single-bit injections across 8 different data types: data value bit flips for all 5 number formats, in addition for metadata error injections for INT, BFP, and AFP.

C. Tool Evaluation and Validation

We evaluate GoldenEye's implementation overhead relative to a non-instrumented, native execution of DL models in IEEE-754

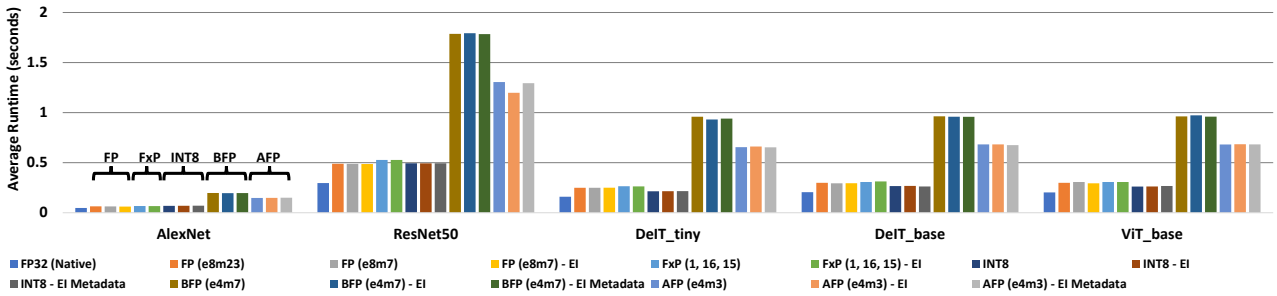


Fig. 3: Runtime performance of GoldenEye, using different number formats and with error injection (EI) on/off.

FP32. We perform our measurements on an NVIDIA RTX 3060 GPU with 12 GB of memory with CUDA 11.4, using PyTorch v1.10 and PyTorchFI v0.6.0. The rest of our system includes an 8-core Ryzen 5800x CPU with 64 GB of DDR4 RAM. We measure the average inference runtime across 100 runs and using a flat batch size of 32 across models. To showcase GoldenEye's versatility for model support, we include both CNNs and Transformers, and measure vision classification accuracy on the ImageNet dataset [8].

Figure 3 shows the results. We execute each DNN model on 14 different number format configurations as shown. For each number system, we include a GoldenEye implementation without any error injection (purely number system emulation), an implementation with a random, single-bit *data value* error injection (denoted EI), and finally an implementation with a random, single-bit metadata errors (denoted EI-metadata) for INT, BFP, and AFP. This setup allows us to explore the overhead of both the number format emulation as well as EI overheads.

Overall, we find the wall-clock runtime average for a batch size of 32 to always be under 2 seconds on our system, with a standard deviation across the 100 runs between 1%-4.5%. The native FP32 runtime is consistently the fastest (our baseline), as expected due to its hardware acceleration. Furthermore, we find that the average runtime for our emulated FP, FxP, and INT number systems are extremely similar to the native speed of silicon. For all number formats explored, the overhead of error injections in values and metadata are negligible, as the cost is amortized due to infrequent calls.

BFP and AFP show increased runtime overhead, sometimes as high as a 5× slowdown. The primary reason for this slowdown is that our current implementation for BFP and AFP is entirely in Python. On the other hand, GoldenEye's implementations for the more traditional number formats (FP, FxP, and INT) uses the open-source QPytorch [41] implementations, which is accelerated using C++/CUDA. Porting BFP and AFP to C++/CUDA to take advantage of hardware parallelization is on-going work, but we note that we consider this dichotomy in performance a feature of GoldenEye: our API is general and abstracts the underlying coding implementation. Thus, new number formats can be prototyped and debugged quickly in Python before eventual acceleration in C++/CUDA.[1] Overall, we found that all implementations are reasonably fast in practice, due to the fast real-world performance of under 2 second inferences.

[1]PyTorch [30] went through a similar transformation in its history, where early adoption was enabled due to its Pythonic nature, whereas today most of PyTorch is written in C++/CUDA under the hood for performance.

TABLE I: Dynamic Range of Data Types

Data Type	Absolute Max Value	Absolute Min Value	Range in dB (20 log(Max/Min))
FP32 w/ DN	3.40e+38	1.40e-45	1667.71
FP32 w/o DN	3.40e+38	1.18e-38	1529.23
FxP (1, 15,16)	32768	1.53e-05	186.64
FP16 w/ DN	65504	5.96e-08	240.82
FP16 w/o DN	65504	6.10e-05	180.61
BFloat16 w/ DN	3.39e+38	9.18e-41	1571.34
BFloat16 w/o DN	3.39e+38	1.18e-38	1529.20
INT16 (symetric)	32767	0	90.31
INT8 (symetric)	127	0	42.08
FP8 (e4m3) w/ DN	240	1.95e-03	101.79
FP8 (e4m3) w/o DN	240	1.56e-02	83.73
AFP8 (e4m3) w/o DN	240	1.56e-02	83.73 (movable range)

We validate GoldenEye by employing a test suite to check that conversions are implemented according to each number format's specification, including denormals (DN) where applicable. For "traditional" formats such as FP32/FP16, we also compare our emulated data formats (with and without injections) against non-emulated inferences provided by PyTorchFI [24]. Table I lists the dynamic range of the various data types studied, highlighting that the varying dynamic ranges explored and emulated produce results of statistical significance.

IV. Use Cases

In this section, we expand on three use cases for GoldenEye. The first use case is a functional simulator for accuracy evaluation of different number formats (§IV-A). Second, given our flexible design and user interface, we introduce a basic heuristic for design space exploration (DSE) of number format selection that is tuned to a particular DNN model (§IV-B). Third, we use GoldenEye to perform a fast resiliency analysis study, by leveraging recent work on metric selection and exploring various error models (§IV-C). Our goal here is to demonstrate multiple uses of the tool, and not fully address the research challenges covered by each use case.

A. Functional Simulator for Accuracy

GoldenEye provides a unifying platform for measuring a DL model's accuracy as a function of the underlying number format. Prior work in this space has typically required isolated (i.e., proprietary) simulation of number systems within different environments (e.g., bfloat [18], efloat [4]). Minute implementation details could lead to different accuracies, especially since DL models themselves have lots of statistical variation. Thus, by open-sourcing GoldenEye and providing a generic API for number format representation, we envision future researchers and developers comparing models using

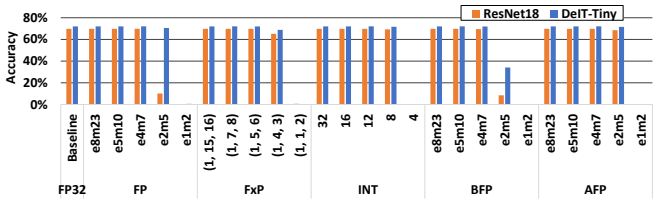


Fig. 4: Accuracy measurements with GoldenEye.

a single underlying number format implementation. Furthermore, GoldenEye provides tunable knobs to adjust a number system's parameters, such as bit-width and radix hyperparameters.

Figure 4 provides an example of this use case, comparing ResNet18 [17] and DeIT-tiny [38] across different bitwidths (32, 16, 12, 8, and 4). One observation we find is that certain models react differently to the same number format. For example, we find that DeIT-tiny (a transformer) can maintain high accuracy at a decreased bitwidth (FP e2m5) while ResNet18 cannot. This suggests that tuning the number format to the DL model can provide improved performance (via bitwidth and area reduction in the hardware) better than a flat parameter choice (such as forcing an INT quantization across the board as prescribed by TensorRT [39]). We also observe that unlike FP, ResNet18 is able to maintain its original accuracy using AdaptivFloat (AFP) at e2m5 suggesting that there are opportunities to achieve accuracy robustness by considering an alternative number format, especially one augmented with metadata. Note that we do not perform any fine-tuning or re-training in any of these experiments, and the results are purely from changing the number format.

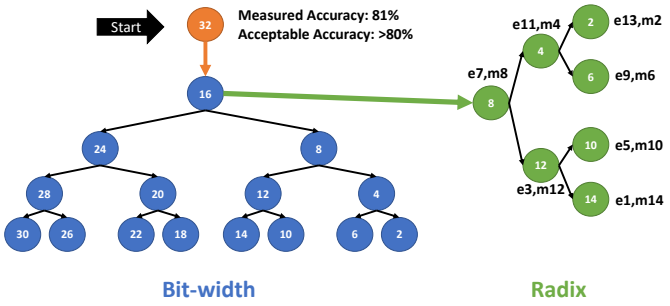


Fig. 5: Recursive binary search heuristic for DSE of number formats.

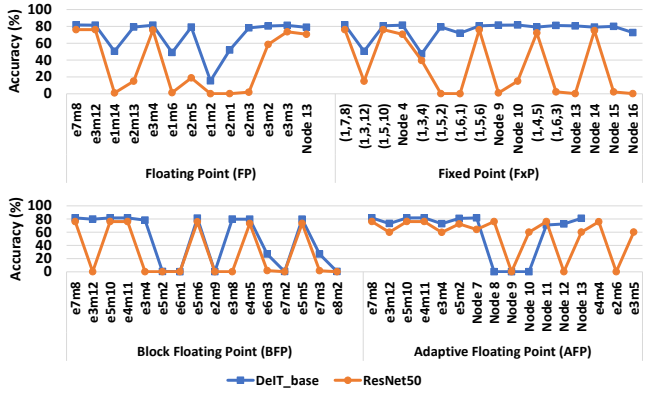


Fig. 6: Design points explored by heuristic for DeIT-base and ResNet50. Node-N denotes a divergence in particular number format exploration between the two DL models.

B. Domain Space Exploration

The second use case we study with GoldenEye is a domain space exploration (DSE) for selecting number formats. While certain number formats may "intuitively" work, the space of possible formats grows extremely large due to parameter choices such as bitwidth, radix, bias value, and others. With GoldenEye, we provide a set of command line arguments for hyperparameter tuning, which we extend with wrapper scripts to perform a DSE for number format selection for a model.

We introduce an *approximate* and *accuracy-preserving* heuristic for number format DSE using a recursive binary tree search. Figure 5 illustrates the algorithm. We first measure the baseline accuracy by profiling the dataset using native FP32 support. Then, using GoldenEye, we explore different number representations following a tree structure for each number format parameter. The key idea is to traverse the tree path (left vs right) based on the measured accuracy (across the entire dataset) at each node, where our heuristic aggressively chooses a shorter bitwidth and radix as long as the accuracy is above a predefined threshold (e.g., 1% accuracy loss from baseline). By logarithmically exploring the space, we can drastically reduce the number of nodes explored, while producing multiple, *approximately* accuracy-preserving nodes at lower precision.

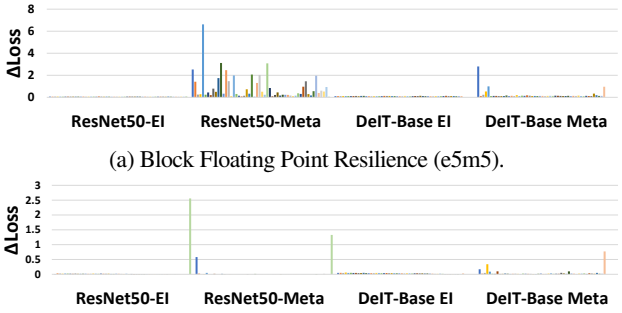
Figure 6 illustrates our results, where the x-axis is ordered according the nodes visited by our heuristic. We find that the heuristic is complete after covering a maximum of 16 nodes (or less), of which more than half suggest design points with an accuracy above our acceptable threshold. Our results indicate that design points differ based on the model being studied, and that different number formats (e.g., BFP vs ADP) may traverse different nodes and have different optimal configurations compared to their originally inspired design (e.g., FP). BFP, for example, shows an accuracy drop at various points because of a large shared block size across an entire layer. Thus, the resolution of low magnitude numbers may suffer, by being essentially rounded to zero. On the other hand, FxP at lower bitwidths (e.g., FxP(1,4,4) at Node 13) seems to preserve accuracy well for transformers, but its accuracy preservation differs dramatically for CNN-based models (such as ResNet50). Our heuristic further illustrates the approximate computing potential of number formats, where a small degradation in model accuracy can potentially lead to a large reduction in bit-width choice for a number format (and, in turn, improve performance for an accelerator design).

Other potential DSE heuristics can be explored via GoldenEye's interface (e.g., a genetic search algorithm). By providing the right level of abstraction, knobs, and runtime speed, this opens the door for more research into choosing the best number format for a DL model and/or hardware accelerator, especially as we envision the number of DL number formats to increase over time.

C. Resiliency Analysis

Our third use cases explores GoldenEye as a resiliency analysis tool. We focus on analyzing the resilience of BFP and AFP due to space limitations, as multiple prior works have already explored the reliability properties of DL models using FP [6], [22], FxP [13], [31], and INT [25].

To measure model resilience to single bit perturbations, two metrics have been recently proposed. The primarily used metric of *mismatches* captures how many error-injected inferences resulted in



(a) Block Floating Point Resilience (e5m5).

(b) Adaptive Floating Point Resilience (e5m2).

Fig. 7: Resilience of BFP and AFP, by layer.

an output misclassification compared to an error-free inference [26]. On the other hand, the recently proposed ΔLoss metric [25] suggests measuring the average absolute difference of the cross entropy loss between the faulty and the error-free inferences for resilience measurements. The work shows statistically that the two metrics produce the same final result, however ΔLoss asymptotically converges much faster due its continuous value comparison (as opposed to the binary outcome comparison of mismatch). GoldenEye supports both metrics, but we primarily use ΔLoss for the faster error injection campaigns, which complements GoldenEye's fast number format emulation.

We perform 1000 unique single-bit flip injections for each of data and metadata at a layer-granularity using GoldenEye, measuring the ΔLoss for each layer. Figure 7 shows the results for ResNet50 and DeIT-base, using BFP (e5m5) and AFP (e5m2). We find that layers on average exhibit similar vulnerability in BFP with value injections, since exponents are no longer part of the equation (Fig. 7a). Metadata error injections, however, are much more egregious across the board, particularly for BFP. This was expected, as a single bit flip behaves as a multiple bit flip due to many reads of the faulty value. Through additional analysis, we also find that the sign bit in BFP is more vulnerable than in FP, since the bitwidth of the data value is now shorter (by removing the exponent bits). Essentially, BFP magnifies the importance of the sign bit via the shared exponent design.

That said, given that it is easier to protect one register rather than a full tensor, BFP provides an attractive number format for low-cost resilience. Alternatively, we find that AFP on average (Fig. 7b) is more resilient layer-wise than BFP for both value and metadata errors, except for the last layer. This is because the last layer has a wider distribution, which makes it difficult for AFP to fully capture the range of values and, correspondingly, increases the possible faulty value range. While additional insights and data are required to fully characterize the resilience of BFP and AFP, this paper is the first to explore the resilience of these number formats, enabled by our versatile tool.

V. Discussion

A. Tuning Accuracy, Resilience, and Bitwidth

As shown in §IV, at longer bitwidths, a number format may overcompensate for both accuracy and resilience. However, when tuning for hardware performance or efficiency, the number format choice along with corresponding parameters (e.g., bitwidth, radix) is critical, in addition to the algorithmic properties of the DL model topology. As GoldenEye provides both accuracy and resilience

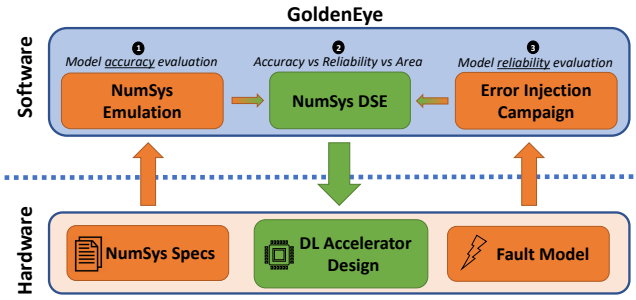


Fig. 8: GoldenEye's utility within a co-designed ML and HW ecosystem.

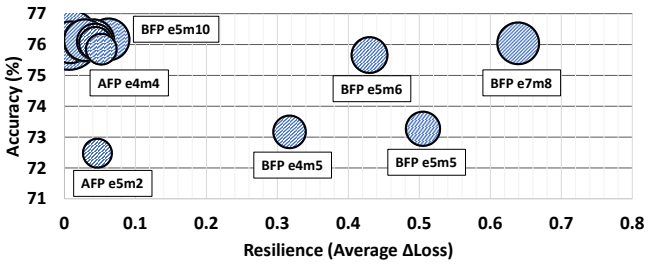


Fig. 9: Accuracy vs Resilience vs Bitwidth exploration for ResNet50.

analysis capabilities, we can combine multiple Use-Cases above to explore the tradeoff between accuracy, resilience, and bit-width (as a proxy for area and performance). This tuning can particularly be helpful during early hardware-software co-design of accelerators in the ML workflow, as depicted in Figure 8.

Figure 9 illustrates a tuning example for ResNet50, focusing on BFP and AFP with a single-bit flip error model as described previously. We plot number formats suggested by our heuristic in §IV-B with reasonable accuracy, and measure resilience as a single value by averaging ΔLoss across all layers in a network (for both value and metadata resilience). This is one possible method to capture resilience/sensitivity with a single value, but more research is required to properly identify this summarizing metric. What we observe from the figure is that there exists various, low-precision, high-accuracy, and low ΔLoss design points in the top, left corner. Thus, depending on the accelerator designers need, they can select the number format which optimizes their budget requirements, potentially opting for newer formats (such as AFP) with lower precision (e.g., e4m4).

B. Additional Features

While §III focuses on describing the technical underpinnings which make GoldenEye fast and extensible, we also provide additional features to make it useful as a broad research tool, including:

- all layer types in PyTorch are supported by GoldenEye for number format emulation and error injection, with CONV and LINEAR as defaults due to their computational intensity [16], [34].
- finer details of a number systems (such as denormal numbers) are provided, and can optionally be disabled by the user for value approximation opportunities [10].
- support for number format conversions and error injections in both weights and neurons. We study neurons in this paper as the more complex case, since weight injections can be performed offline and do not need dynamic runtime support.

- number format emulation is supported for training and inference, as backpropogation is supported.

Additionally, we provide a toggle-able range detector for resiliency analysis (enabled by default), modeled off of recent work [5]. Note that some number formats (such as INT) require a range to be provided for functional correctness [12]), absolving the need for a range detector. More generally, GoldenEye can be used for software-directed protection techniques (such as various forms of duplication), making it a handy tool for additional resilience studies.

C. Current Limitations

GoldenEye is not a cycle-accurate simulator. Performance measurements on real hardware cannot be directly extrapolated and is not the intended environment for using GoldenEye. Users can potentially use proxies such as number of MAC operations and expected MAC area for runtime, but this is out of scope for our design. GoldenEye does not negate the importance of cycle accurate simulation before accelerator design [29].

GoldenEye also does not yet support mixed-precision operations, which would require detailed attention to accumulation error and rounding error during computations across different data types. This is an interesting and valuable future direction we plan to pursue. Additionally, while backpropogation is supported for number system emulation, the current infrastructure does not support error injection on gradients. This is another direction we plan to take GoldenEye for modeling errors during model training, as described below.

D. Additional Use Cases and Future Directions

Beyond the use cases described in §IV, we foresee GoldenEye also being a useful security analysis tool for DL attacks and defenses. For example, GoldenEye can be used to simulate different number formats for a given adversarial attack, and be used to assess the attacks efficacy (or lack thereof). Evaluating a security techniques robustness as a function of the underlying number format and implementation is an interesting future direction.

Additionally, since GoldenEye can perform error injections during the forward-pass in training, it can potentially be used to build resilient models via novel training routines. Similarly, exploring which number formats work well during training (such as advertised by bfloat) versus inference (such as INT8) can push development of a universally applicable number format for use in both. Such explorations are also potential uses cases of GoldenEye.

VI. Related Work

QPyTorch [41] is an open source library which provides support for variable hyperparameter exploration for FP, FxP, and BFP. GoldenEye leverages QPyTorch's C++/CUDA implementations for FP and FxP, which translates to native runtime performance during inference. Further, QPyTorch supports backpropogation, which can be extended to training with different number systems. We found that the BFP implementation of QPyTorch, however, had two primary drawbacks: 1) it did not allow for variable exponent sizes (it was pegged at 8 bits), and 2) it had multiple implementation issues in the number format implementation of BFP. Thus, in GoldenEye, we used our own implementation for BFP, while simulataneously ensuring it was generalizable and enable hyperparameter tuning of the number system.

	QPyTorch [41]	Ares [32]	TensorFI [7]	PyTorchFI [24]	GoldenEye (This work)
Floating Point (FP)	✓	✓	✓	✓	✓
Fixed Point (FxP)	✓	✓			✓
Integer Quantization (INT)	✓	✓	✓	✓	✓
Block Floating Point (BFP)	∼				✓
AdaptivFloat (AFP)					✓
Future Number Format Support	✓				✓
Support Error injections in Values		✓	✓	✓	✓
Support Error injections in Metadata					✓
Error Metric: Mismatch		✓	✓	✓	✓
Error Metric: ΔLoss				✓	✓

TABLE II: Open-source tool comparison of related work.

BFP and AFP have recently been explored across multiple recent works for hardware accelerator design [11], [35], [36], [40]. While the concept of BFP is old (from the 1970s), its recent resurgence is relevant as a performance optimization which also necessitates a thorough resiliency analysis. With GoldenEye, we present a framework for its assessment, and scratch the surface with basic insights as presented in §IV. Similarly, AFP is a new number system with major performance insights for Transformers. This work also explores AFP resilience properties.

Many recent fault injection tools have been proposed for DL resilience evaluation, particularly as DL perception, planning, and control play a large and important part of safety-critical system design [7], [24], [32]. However, most techniques leverage the slow metric of mismatch counting for resiliency evaluation. By using the concept of ΔLoss in GoldenEye, we aim to accelerate resiliency analysis, while making sure the underlying number system is properly represented. Finally, with the emergence of newer number systems, the hardware-aware metadata is important to take into consideration, which prior work has not explored. This unique addition to GoldenEye is important for future reliability studies, and in this work we provide a fast and seamless way to continue modeling future number systems in a co-designed manner.

A qualitative comparison with related work is shown in Table II, highlighting GoldenEye's versatility. Notably, GoldenEye is the only tool that provides wide support for multiple legacy and emerging number formats along with a comprehensive framework to evaluate their resilience at different numerical locations, including the often neglected metadata.

VII. Conclusion

We introduce GoldenEye, an open-source, number format functional simulator and error injection framework for DL models, implemented in the PyTorch framework. Our feature-rich tool handles many popular and emerging number formats, and provides tunable hyperparameters for the exploration of number format design points for DL model accuracy and resilience analysis. GoldenEye provides an extensible and versatile toolbox for error injection support, including for the first time, hardware-aware metadata modeling. Overall, GoldenEye provides a unifying framework for number system exploration and resilience studies, and has multiple use cases in the domain of resilient accelerator design for deep learning models.

Acknowledgements

We would like to thank our shepherd, Ningfang Mi, and the anonymous reviewers for their detailed feedback which improved this paper. A special thanks to Siva Hari, whose incredibly helpful feedback helped set the stage for this work, and Josh Park for his assistance in improving the code. This work is supported in part by the National Science Foundation (NSF) grant CCF-1704834, and by the Application Driving Architectures (ADA) Research Center, a JUMP Center cosponsored by SRC and DARPA. Thierry Tambe is supported by an NVIDIA Graduate Fellowship.

References

[1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] Ankur Agrawal, Silvia M. Mueller, Bruce M. Fleischer, Xiao Sun, Naigang Wang, Jungwook Choi, and Kailash Gopalakrishnan. DLFloat: A 16-b Floating Point Format Designed for Deep Learning Training and Inference. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pages 92–95, 2019.

[4] Rajesh Bordawekar, Bülent Abali, and Ming-Hung Chen. EFloat: Entropy-coded Floating Point Format for Deep Learning. *CoRR*, abs/2102.02705, 2021.

[5] Zitao Chen, Guanpeng Li, and Karthik Pattabiraman. Ranger: Boosting Error Resilience of Deep Neural Networks through Range Restriction. *CoRR*, abs/2003.13874, 2020.

[6] Zitao Chen, Guanpeng Li, Karthik Pattabiraman, and Nathan DeBardeleben. BinFI: An Efficient Fault Injector for Safety-Critical Machine Learning Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[7] Zitao Chen, Niranjhana Narayanan, Bo Fang, Guanpeng Li, Karthik Pattabiraman, and Nathan DeBardeleben. TensorFI: A Flexible Fault Injection Framework for TensorFlow Applications. *CoRR*, abs/2004.01743, 2020.

[8] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[9] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. Silent Data Corruptions at Scale. *CoRR*, abs/2102.11245, 2021.

[10] Mario Drumond, Tao Lin, Martin Jaggi, and Babak Falsafi. Training DNNs with Hybrid Block Floating Point. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 451â461, Red Hook, NY, USA, 2018. Curran Associates Inc.

[11] Mario Paulo Drumond, Tao Lin, Martin Jaggi, and Babak Falsafi. Training DNNs with Hybrid Block Floating Point. In *NeurIPS*, 2018.

[12] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. A Survey of Quantization Methods for Efficient Neural Network Inference. *CoRR*, abs/2103.13630, 2021.

[13] Rishabh Goyal, Joaquin Vanschoren, Victor van Acht, and Stephan Nijssen. Fixed-point Quantization of Convolutional Neural Networks for Quantized Inference on Embedded Platforms. *CoRR*, abs/2102.02147, 2021.

[14] Rishabh Goyal, Joaquin Vanschoren, Victor van Acht, and Stephan Nijssen. Fixed-point Quantization of Convolutional Neural Networks for Quantized Inference on Embedded Platforms. *CoRR*, abs/2102.02147, 2021.

[15] Hui Guan, Lin Ning, Z. Lin, Xipeng Shen, Huiyang Zhou, and Seung-Hwan Lim. In-Place Zero-Space Memory Protection for CNN. *ArXiv*, abs/1910.14479, 2019.

[16] Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, and Stephen W Keckler. Making Convolutions Resilient via Algorithm-Based Error Detection Techniques. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1, 2021.

[17] Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[18] Dhiraj D. Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. A Study of BFLOAT16 for Deep Learning Training. *CoRR*, abs/1905.12322, 2019.

[19] K. Kalliojarvi and J. Astola. Roundoff errors in block-floating-point systems. *IEEE Transactions on Signal Processing*, 44(4):783–790, 1996.

[20] P. Kharya. TensorFloat-32 in the A100 GPU Accelerates AI Training, HPC up to 20x. https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/, 2020.

[21] Khronos OpenCL Working Group. *The OpenCL Specification, Version 1.1*, 2011.

[22] Guanpeng Li, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, Karthik Pattabiraman, Joel Emer, and Stephen W. Keckler. Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[23] Guanpeng Li, Karthik Pattabiraman, Siva Kumar Sastry Hari, Michael Sullivan, and Timothy Tsai. Modeling Soft-Error Propagation in Programs. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2018.

[24] A. Mahmoud, N. Aggarwal, A. Nobbe, J. R. S. Vicarte, S. V. Adve, C. W. Fletcher, I. Frosio, and S. K. S. Hari. PyTorchFI: A Runtime Perturbation Tool for DNNs. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 25–31, 2020.

[25] Abdulrahman Mahmoud, Siva Kumar Sastry Hari, Christopher W. Fletcher, Sarita V. Adve, Charbel Sakr, Naresh R. Shanbhag, Pavlo Molchanov, Michael B. Sullivan, Timothy Tsai, and Stephen W. Keckler. Optimizing Selective Protection for CNN Resilience. *International Symposium on Software Reliability Engineering (ISSRE)*, 2021.

[26] Sparsh Mittal. A survey on modeling and improving reliability of DNN algorithms and accelerators. *Journal of Systems Architecture*, 104:101689, 2020.

[27] Robert Munafo. Survey of floating-point formats. https://mrob.com/pub/math/floatformats.html, 2020.

[28] NVIDIA, Peter Vingelmann, and Frank H.P. Fitzek. CUDA, release: 10.2.89. https://developer.nvidia.com/cuda-toolkit, 2020.

[29] G. Papadimitriou and D. Gizopoulos. Demystifying the System Vulnerability Stack: Transient Fault Effects Across the Layers. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 902–915, Los Alamitos, CA, USA, jun 2021. IEEE Computer Society.

[30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[31] Edward Pyne, Lillian Pentecost, Udit Gupta, Gu-Yeon Wei, and David Brooks. Quantifying the impact of data encoding on DNN fault tolerance. *FASTPATH*, 2020.

[32] Brandon Reagen, Udit Gupta, Lillian Pentecost, Paul Whatmough, Sae Kyu Lee, Niamh Mulholland, David Brooks, and Gu-Yeon Wei. Ares: A Framework for Quantifying the Resilience of Deep Neural Networks. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, pages 17:1–17:6, New York, NY, USA, 2018. ACM.

[33] Zhourui Song, Zhenyu Liu, Chunlu Wang, and Dongsheng Wang. Computation Error Analysis of Block Floating Point Arithmetic Oriented Convolution Neural Network Accelerator Design. *CoRR*, abs/1709.07776, 2017.

[34] Zhuoran Song, Bangqi Fu, Feiyang Wu, Zhaoming Jiang, Li Jiang, Naifeng Jing, and Xiaoyao Liang. DRQ: Dynamic Region-Based Quantization for Deep Neural Network Acceleration. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 1010â1021. IEEE Press, 2020.

[35] Thierry Tambe, Coleman Hooper, Lillian Pentecost, Tianyu Jia, En-Yu Yang, Marco Donato, Victor Sanh, Paul N. Whatmough, Alexander M. Rush, David Brooks, and Gu-Yeon Wei. EdgeBERT: Sentence-Level Energy Optimizations for Latency-Aware Multi-Task NLP Inference. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture*. Association for Computing Machinery, 2021.

[36] Thierry Tambe, En-Yu Yang, Glenn G. Ko, Yuji Chai, Coleman Hooper, Marco Donato, Paul N. Whatmough, Alexander M. Rush, David Brooks, and Gu-Yeon Wei. A 25mm2 SoC for IoT Devices with 18ms Noise Robust

Speech-to-Text Latency via Bayesian Speech Denoising and Attention-Based Sequence-to-Sequence DNN Speech Recognition in 16nm FinFET. In *International Solid-State Circuits Conference (ISSCC)*, 2021.

[37] Thierry Tambe, En-Yu Yang, Zishen Wan, Yuntian Deng, Vijay Janapa Reddi, Alexander M. Rush, David Brooks, and Gu-Yeon Wei. Algorithm-Hardware Co-Design of Adaptive Floating-Point Encodings for Resilient Deep Learning Inference. In *Design Automation Conference (DAC'20)*, 2020.

[38] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Herve Jegou. Training data-efficient image transformers amp; distillation through attention. In *International Conference on Machine Learning*, volume 139, pages 10347–10357, July 2021.

[39] Leyuan Wang, Zhi Chen, Yizhi Liu, Yao Wang, Lianmin Zheng, Mu Li, and Yida Wang. A Unified Optimization Approach for CNN Model Inference on Integrated GPUs. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, New York, NY, USA, 2019. Association for Computing Machinery.

[40] Sai Qian Zhang, Bradley McDanel, and H. T. Kung. FAST: DNN Training Under Variable Precision Block Floating Point with Stochastic Rounding. *ArXiv*, abs/2110.15456, 2021.

[41] Tianyi Zhang, Zhiqiu Lin, Guandao Yang, and Christopher De Sa. QPyTorch: A Low-Precision Arithmetic Simulation Framework, 2019.