TensorDynamic: Bridging Application- and Instruction-Level Fault Injection for DNN Tensor Core Execution

Yuxiao Jia*, Euijun Chung*, Huanzhi Pu*, Ben Feinberg†, Hyesoon Kim*
*Georgia Institute of Technology
†Sandia National Laboratories

***Abstract*—Deep neural network (DNN) inference relies heavily on Tensor Core operations, which are vulnerable to transient hardware faults in computation pipelines not protected by error-correcting codes (ECC). Prior fault injection work has explored both application-level and instruction-level effects on DNN accuracy. However, existing application-level approaches support only coarse perturbations and do not capture hardware execution details, while instruction-level approaches lack application-level context.**

To address this gap, we propose TensorDynamic, an application-aware instruction-level dynamic fault injection tool for Tensor Core execution in DNN workloads. TensorDynamic enables fine-grained fault injection into MMA (matrix-multiply-accumulate) instructions during DNN execution. Across multiple models, we show that, under the same error injection rate and severity, application-level fault injection can produce substantially different inference outcomes from instruction-level fault injection. This result underscores the need for execution-aware fault injection when evaluating DNN resilience on GPU Tensor Cores.

***Index Terms*—DNN, Tensor Core, Fault Injection, Hardware Resilience**

I. Introduction

Deep neural networks (DNNs) are now widely used across a broad range of applications, including safety-critical environments where inference results directly influence real-world decisions with consequential impacts [32], [37]. In such settings, deviations beyond expected error bounds can have severe consequences, making the robustness of DNN inference a first-order concern. A prominent example is autonomous driving, where Convolutional Neural Networks (CNNs) are extensively used for real-time object detection and recognition [18]. In these systems, perception outputs guide downstream planning and control; therefore, unexpected accuracy degradation can significantly impact system safety, particularly in scenarios involving pedestrians or obstacles.

To meet the strict latency and throughput requirements of DNN model inference, these models are often deployed on high-performance GPUs that accelerate generalized matrix multiplication (GEMM) using specialized hardware such as Tensor Cores. While GPUs offer massive parallelism, they are also susceptible to transient hardware faults, commonly referred to as soft errors, which are exacerbated in automotive operating conditions. Although on-chip memories are often protected using error-correcting codes (ECC), the compute pipelines, including Tensor Core units executing mixed-precision GEMM operations, remain largely unprotected. A single bitflip of an FP32 value stored in a Tensor Core destination register, especially in the exponent field, can drastically perturb GEMM outputs, propagate through subsequent network layers, and ultimately degrade perception accuracy in safety-critical tasks, such as pedestrian detection.

To study the resilience of DNN inference, prior work has introduced a variety of fault injection frameworks spanning both application-level and instruction-level abstractions. **Application-level** or software-level approaches emulate transient or persistent faults at coarse granularity, such as perturbing activation values or modifying weight tensors during CNN inference. Notable examples include PyTorchFI [21], TensorFI [6], PyTorchALFI [11], and MRFI [17]. While such frameworks are flexible and easy to deploy, their reliance on high-level abstractions fundamentally limits fidelity, as they cannot capture the effects of GPU execution semantics.

In contrast, **instruction-level** or hardware-level fault injection frameworks operate at much finer granularity, model faults more closely to the actual execution behavior of GPUs, and enable the study of fault effects at the instruction and register levels, which is difficult or impossible to represent using higher-level software abstractions. Representative tools include SASSIFI [14], NVBitFI [33], and domain-specific NVBitFI extensions designed to study mixed-precision GEMM behavior [9], [29]. However, existing hardware-level tools suffer from important limitations in usability and flexibility, despite their high fidelity. For example, previous work typically supports only a single fault injection per experiment and lacks explicit batch- and layer-level context, making large-scale, targeted fault studies difficult to conduct.

Moreover, the relationship between application-level and instruction-level fault injection for GPU workloads remains insufficiently explored. Commonly used instruction-level fault injection tools operate at the SASS level, lacking awareness of high-level constructs such as ”layers” or ”kernels”, a context that is critical, given the well-documented variance in error sensitivity across different DNN layers [19], [25], [36]. Conversely, while many application-level studies claim to faithfully model hardware effects by introducing nominally comparable bitflips, they do not systematically characterize whether these high-level abstractions actually yield accuracy

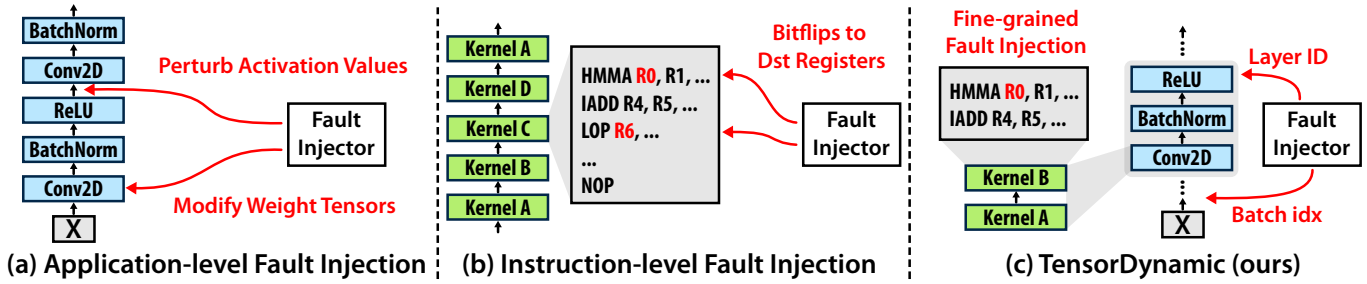

Fig. 1. Comparison of fault injection methodologies. While traditional approaches separate hardware fidelity from semantic awareness, TensorDynamic unifies them by enabling instruction-level faults to be targeted at specific application constructs, such as distinct layers and batches.

degradation profiles consistent with ground-truth microarchitectural faults. Prior CPU-based cross-layer analysis showed that higher-level fault injection can misrepresent the full-system fault behavior [23]. In addition, Esposito et al. [8] compare these abstractions to evaluate the efficacy of software hardening strategies for GPU-based DNN workloads. However, the broader implications of this abstraction mismatch on the baseline failure dynamics of safety-critical GPU workloads remain underexplored. Figure 1 compares existing fault injection methodologies and shows how TensorDynamic bridges semantic awareness and instruction-level fidelity.

Motivated by these gaps, we propose **TensorDynamic**, a dynamic fault injection framework that enables instruction-level error injection on GPU Tensor Cores while providing fine-grained, application-aware control over fault placement. Our work focuses on transient faults in Tensor Core-dominated GEMM workloads, which constitute the performance-critical backbone of modern CNN inference. TensorDynamic is built on a SASS-level instruction instrumentation framework that specializes in profiling and injecting faults into Tensor Core MMA instructions, enabling precise control over kernel identity, instruction instances, and destination registers. Using TensorDynamic, we analyze how hardware-level faults propagate through CNN inference, compare their effects with those of software-level fault models, and evaluate whether high-level injection techniques can reliably approximate low-level transient fault behavior. Our goal is to establish a principled understanding of fault injection abstraction fidelity for CNN workloads on GPUs, thereby guiding the development of accurate and trustworthy resilience evaluation methodologies for safety-critical systems.

The contributions of our paper are summarized as follows:

- We characterize the fidelity gap between application-level and instruction-level fault injection for CNN inference on GPUs, showing that application-level tools consistently overestimate model vulnerability.
- We present **TensorDynamic**, a dynamic fault injection framework that enables fine-grained, instruction-level error injection on GPU Tensor Cores while providing application-aware control over layer, batch, and kernel-level fault injection.
- Based on our findings, we provide insights and guidelines for designing accurate and trustworthy DNN resilience evaluation methodologies for safety-critical GPU-based systems.

II. Background

A. Types of GPU Execution Errors

Modern GPUs are vulnerable to both persistent errors and transient hardware faults. Persistent errors, such as stuck-at faults, arise from permanent hardware malfunctions that remain active once present and are typically associated with long-term degradation mechanisms, including device aging and wear-out [10], [12]. In contrast, transient hardware errors—or soft errors, which are the focus of this work—can arise from physical phenomena such as high-energy particle strikes and electromagnetic perturbations [7], [29], [30].

Although these events do not inflict permanent physical damage, they can induce data corruption at distinct stages of the instruction pipeline. A fault may manifest during operand fetch, where a bitflip causes a value to be read incorrectly from the register file or memory system before computation. Alternatively, the disturbance may affect the combinational logic of the execution units (e.g., ALUs or Tensor Cores), causing a logic upset during the arithmetic operation. Finally, errors can occur during the write-back stage, where a correctly computed result is corrupted as it is latched into the destination register. Regardless of the specific pipeline stage, these faults allow incorrect values to propagate silently through subsequent execution.

The manifestation of transient faults depends on both their location and timing, leading to different observable outcomes. Silent data corruption (SDC) occurs when corrupted values produce incorrect outputs or degrade application-level accuracy without triggering exceptions or crashes. Detected unrecoverable errors (DUEs) occur when hardware or software mechanisms detect faults, leading to program or system termination. In other cases, faults may be masked by microarchitectural or algorithmic effects and produce no observable impact on the final output. These outcome categories are widely used in fault injection studies, as formalized in the prior work [14]. We focus on faults that visibly impact inference accuracy, rather than benign errors masked by the system. Assuming that ECC protects the memory and registers, we limit our scope to transient execution faults. These occur specifically within the unprotected combinational logic (e.g., ALUs, Tensor Cores)

or pipeline latches during active computation, corrupting data "in flight" where storage-based protection mechanisms cannot intervene.

B. Tensor Cores and HMMA Instructions

NVIDIA Tensor Cores are specialized hardware units designed to accelerate dense linear algebra operations, particularly generalized matrix multiplication (GEMM), which accounts for a significant portion of the execution time in many deep learning workloads. Tensor Cores implement matrix-multiply-accumulate (MMA) operations using a systolic-array-like microarchitecture that enables high-throughput computation. At the assembly level, higher-level Tensor Core operations are decomposed into warp-level MMA instructions, such as HMMA in NVIDIA SASS. These instructions are executed collectively by a warp to perform fixed-size matrix multiplication and accumulation.

Large GEMM operations are decomposed into smaller tiles, distributed across thread blocks, and scheduled to multiple streaming multiprocessors (SMs). Each thread block cooperatively loads input tiles from memory, and warps within the block execute MMA instructions to compute partial results that are accumulated into destination registers. Due to this hierarchical tiling and accumulation process, faults occurring in Tensor Core execution, particularly within MMA destination registers, can propagate across multiple output elements and subsequent layers, making Tensor Core computation a critical target for resilience analysis.

III. Motivation

A wide range of fault injection techniques has been proposed to emulate errors in GPU execution to study the resilience of DNN workloads. Existing approaches broadly categorize into *application-level* fault injection, which perturbs values exposed at the ML framework level, and *instruction-level* fault injection, which targets hardware-visible operations and values using low-level instrumentation. While both approaches aim to approximate the effects of transient hardware faults, they differ significantly in fidelity, controllability, and scalability, leading to an incomplete and sometimes misleading understanding of DNN robustness on modern GPUs.

A. Application-Level Fault Injection

Application-level fault injection frameworks, such as PyTorchFI [21], TensorFI [6], PyTorchALFI [11], and MRFI [17], emulate hardware soft errors by perturbing neural network tensors during execution. These tools are typically built on top of popular deep learning frameworks, such as PyTorch [24] and TensorFlow [1], and often modify neuron activations or intermediate tensors dynamically during the forward pass. A common approach attaches forward hooks to selected layers, intercepting output tensors at runtime and corrupting values according to predefined error models (e.g., bitflips or random value replacement), without altering the network structure. Alternatively, layers may be wrapped to recompute their outputs using uncorrupted inputs and parameters, followed by controlled perturbations applied directly to the resulting tensors. Despite architectural differences, both approaches inject faults at the same granularity by modifying complete tensor values after layer computation completes.

Although application-level fault injection is convenient and well-suited for large-scale experimentation, it abstracts away critical details of GPU execution. Because faults are introduced only after a layer's computation has finished, these approaches can ignore the fragmented accumulation behavior, warp-level execution semantics, and data layout characteristics of Tensor Core MMA operations. In particular, injection occurs after high-precision accumulation rather than during mixed-precision arithmetic within the Tensor Cores. As a result, application-level perturbations may amplify error effects and produce failure modes that do not accurately reflect how transient faults propagate during actual Tensor Core execution, potentially overstating model vulnerability.

B. Instruction-Level Fault Injection

Instruction-level fault injection approaches aim to model transient hardware faults more faithfully by operating directly on GPU machine instructions. Tools such as SASSIFI [14], built on the SASSI [31] framework, and NVBitFI [33], based on the NVBit [34] dynamic instrumentation framework, inject faults at the SASS level during execution. More recent extensions of NVBitFI [29] enable warp-level value modification and fault injection into Tensor Core operations, while MPGemmFI [9] focuses specifically on injecting errors into GEMM workloads.

Despite their higher fidelity, existing instruction-level tools exhibit important limitations. Most inject only a single transient fault per program execution, typically targeting one instruction instance per run to avoid fault interactions. However, CNN inference workloads execute hundreds of GEMM or convolution kernels per inference, requiring an equivalent number of full program executions to observe statistically meaningful output deviations. For example, MPGemmFI must repeatedly re-execute the application with different injection sites to achieve sufficient fault coverage. This approach results in high runtime overhead, making large-scale resilience studies prohibitively expensive.

Furthermore, despite the repetitive structure of CNN inference, where the same Tensor Core kernels are executed across layers and batches, existing tools provide limited application-level control. Users cannot directly specify semantically meaningful injection targets, such as a particular DNN layer or batch index, even though prior work has shown that fault sensitivity varies across different layers [19], [25], [36]. Instead, fault injection requires manually identifying the corresponding kernel and instruction identifiers, a labor-intensive process that complicates layer-wise vulnerability analysis and limits usability.

For example, targeting a semantically meaningful layer with NVBitFI requires users to translate high-level intent manually (e.g., "Batch 2, Layer 4") into a low-level kernel invocation

```
## Manual NVBitFI workflow for injecting
## fault into Batch 2, Layer 4
# 1. Profile execution:
#    volta_hmma_gemm executes 50 times per batch
# 2. Compute global invocation index:
#    (batch_id * kernels_per_batch) + layer_id
#    = (2 * 50) + 4 = 104
# 3. Configure NVBitFI with the raw index
export NVBITFI_KERNEL_NAME="volta_hmma_gemm"
export NVBITFI_INJECTION_KERNEL_INSTANCE=104
```

Fig. 2. Illustration of the multi-stage effort needed to apply instruction-level GPU fault injection to DNN execution, from high-level semantic selection (layer/batch) to low-level injection site identification.

index. This process, as illustrated in Figure 2, involves profiling the workload to determine the global execution order of kernels and then calculating the exact instance number corresponding to the desired layer. The resulting configuration is both labor-intensive and sensitive: any change in model structure, batch size, or kernel fusion alters the execution order, invalidating previously computed indices and forcing users to re-profile and recompute injection parameters from scratch.

C. Bridging the Gap between Application- and Instruction-Level Fault Injection

Application-level and instruction-level fault injections, which operate at different abstraction levels, are both used to evaluate the reliability of DNNs. Ideally, both approaches should yield similar results, as they aim to model the same hardware faults. However, there is a clear difference between modifying mathematical tensors and flipping bits in hardware registers. It is underexplored whether these distinct methods yield consistent failure assessments or whether the choice of abstraction level affects observed model resilience.

The current literature largely treats these abstraction levels as independent domains, failing to systematically characterize their correlations. This separation creates a dangerous trade-off between fidelity and interpretability. Application-level injection provides high semantic clarity, enabling researchers to target specific logical structures (e.g., layers, channels). Still, it often relies on hardware-agnostic fault models that may not accurately reflect how microarchitectural faults manifest. Conversely, instruction-level injection provides high hardware fidelity but suffers from *contextual blindness*; without a mapping to high-level structures, it is difficult to determine whether a corrupted register corresponds to a critical weight, an insignificant activation, or a control-flow variable. Consequently, relying exclusively on either approach risks producing misleading robustness evaluations—potentially overestimating safety in critical systems, such as autonomous driving, by missing vulnerabilities that manifest only at the intersection of hardware execution and software logic.

Hence, an accurate reliability analysis requires a *unified fault injection framework* that reconciles instruction-level realism with application-level semantics. By enabling hardware-level faults to be injected within the execution boundaries of specific application constructs (e.g., a target layer, batch, or tensor operations), researchers can isolate the hardware origins of application-level failures and move toward a principled understanding of how faults propagate through the software stack.

IV. TensorDynamic: SASS-Level Tensor Core Fault Injection

We introduce **TensorDynamic**, a comprehensive framework for analyzing Tensor Core reliability via direct SASS-level instrumentation. To address the trade-off between profiling precision and experimental throughput, the framework supports two distinct workflows: a conventional static injection flow and a dynamic injection flow. The dynamic workflow is used as the primary evaluation methodology in this work because it enables high-throughput fault injection without requiring offline profiling. A detailed comparison of the two workflows is provided in Table I.

In the **static workflow**, we provide two discrete tools: `tensor_profiler` and `tensor_injector`. The profiler first executes the application to locate dynamic `HMMA` (Half Precision Matrix Multiply-Accumulate) instructions and record their execution coordinates. Subsequently, the injector uses these coordinates to perturb the destination registers of specific `HMMA` instances during a replay run, applying configurable fault models such as bitflips or value scaling.

To overcome the latency inherent in multi-pass profiling, we introduce the **dynamic workflow**, which unifies profiling and injection into a single kernel execution. Unlike the static approach, which requires an offline global instruction profile, the dynamic mode identifies `HMMA` kernels at runtime. It utilizes a *frequency-guided kernel sampling* mechanism to select representative kernels and targets the first N thread instances of relevant instructions. Faults are injected into the destination registers immediately after identifying the injection sites. This design eliminates the overhead of pre-profiling, enabling high-coverage injection campaigns that are tightly aligned with the real-time execution patterns of Tensor Core-accelerated CNN inference.

A practical use case enabled by the workflow in Figure 3 is layer-wise vulnerability analysis [25], [36]. A researcher first maps DNN layers to specific kernel ranges (e.g., `layer1.0.conv1` to [210, 220]). Using the boundary configuration, TensorDynamic restricts fault injection exclusively to kernels within the target layers and performs multi-fault injection across different batches within a single execution. This mapping allows researchers to rapidly sweep through layers—injecting faults into ”Layer X” in one run and ”Layer Y” in the next—to quantify how error sensitivity varies by depth. TensorDynamic automates the workflow shown in Figure 2, enabling efficient sensitivity analysis that correlates specific architectural features of layers with the final accuracy degradation.

A. Design Objectives

TensorDynamic was developed to satisfy four methodological requirements essential for the trustworthy analysis of transient hardware errors in Tensor Cores. Our primary goal is

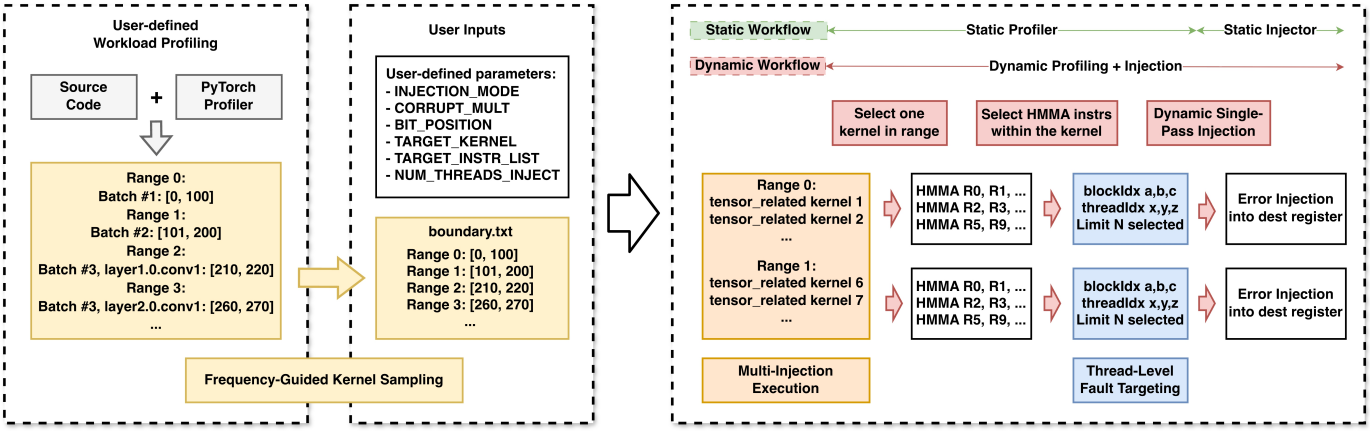

Fig. 3. Overview of TensorDynamic's static and dynamic workflows. The diagram illustrates the execution flow, highlighting how Frequency-Guided Sampling selects kernels within specific layers, Multi-Injection aggregates faults, and Thread-Level Targeting isolates specific threads for precise error injection.

TABLE I
COMPARISON BETWEEN DYNAMIC AND STATIC FAULT INJECTION FLOWS IN TENSORDYNAMIC.

Feature	Dynamic Flow	Static Flow
Profiling	Online (Runtime identification)	Offline (Full enumeration)
Execution	**Single-pass** (Inject immediately)	**Two-pass** (Profile → Inject)
Targeting	Implicit (First N available threads)	Precise (User-controlled randomization)
Profile–Injection Misalignment	None (Single-pass guarantee)	Possible (Due to potential scheduling non-determinism)
Overhead	**Low** (No storage or prerun needed)	**High** (Storage and multi-pass latency)

to enable large-scale experiments on physical GPU hardware while maintaining the high fidelity of SASS-level corruption. To achieve this, the framework is built around four core design objectives and capabilities:

- **Frequency-Guided Kernel Sampling:** To account for the highly repetitive nature of CNN inference, we implement a boundary-based selection strategy. By partitioning the execution timeline into user-defined intervals, the strategy ensures balanced coverage across diverse kernel behaviors, avoiding redundant analysis of identical execution phases while performing selection entirely online.
- **Multi-Injection Execution:** In contrast to standard methodologies that restrict analysis to a single fault per trial, we aggregate multiple injection events into one execution. This aggregation overcomes the latency of repetitive CNN inference runs, thereby accelerating the observation of statistically significant error propagation.
- **Thread-Level Fault Targeting:** To better reflect Tensor Core execution behavior, the injector targets individual threads, enabling observation of HMMA accumulation effects that are not visible at coarser abstraction levels.
- **Dynamic Single-Pass Injection:** To facilitate end-to-end reliability analysis and avoid kernel replays between profiling and injection phases, TensorDynamic performs profiling and fault injection within a single execution pass.

B. Frequency-Guided Kernel Sampling

CNN inference is characterized by the highly repetitive execution of Tensor Core kernels across numerous layers and batches. Due to this repetition, a naive random fault injection strategy is statistically inefficient: it risks oversampling the most frequent kernels (producing redundant data) while potentially missing rare but critical operations. To ensure balanced coverage, TensorDynamic implements a frequency-guided sampling strategy. Instead of selecting instructions purely at random, the framework partitions the execution timeline into distinct frequency segments. The injector can therefore distribute faults evenly across different types of kernel behavior, preventing the analysis from being dominated by the most repetitive operations.

To implement this, users define execution segments via a configuration file containing closed intervals of kernel invocation indices (e.g., $[0, 100], [101, 200]$), as illustrated in Figure 3. These intervals serve as temporal masks, allowing users to map contiguous regions of execution to specific semantic units, such as individual layers, batches, or custom phases. Typically, users derive these boundaries by first using the PyTorch profiler to correlate kernel invocation order with the model's high-level structure, and then partitioning the timeline into meaningful bins.

At runtime, TensorDynamic monitors kernel launches against these defined boundaries. When a kernel's invocation index falls within a targeted range, the tool identifies it as a valid candidate for HMMA profiling and injection. This approach performs sampling entirely online, eliminating the significant storage and processing overhead associated with offline trace collection or static binary analysis. By enforcing

this frequency-guided selection, the framework ensures that fault injection campaigns achieve balanced coverage across diverse execution behaviors while avoiding redundant analysis of statistically identical operations.

C. Multi-Injection Execution Model

TensorDynamic employs a multi-injection execution model, enabling a single workload run to support multiple independent fault injection events at distinct dynamic execution points. This design addresses the efficiency bottlenecks inherent in CNN inference workloads, where the high latency of individual runs makes traditional "one-fault-per-run" campaigns prohibitively expensive. By aggregating multiple injections into a single execution, TensorDynamic significantly reduces total experimental runtime while generating statistically significant data on model-level degradation.

In this model, the targeted HMMA instruction occurrences and the thread count N are specified via configuration parameters. During execution, the framework tracks the selected HMMA instruction positions within each kernel invocation and injects faults into the first N dynamic thread instances observed for each targeted instruction. This online first-N selection is used because the instrumentation tool does not know the full set of executing threads or their exact block/thread coordinates before kernel execution, so direct selection of specific threads is not practical at runtime. Since GPU thread scheduling is non-deterministic, capturing the first N available threads inherently captures a randomized subset of physical hardware threads across different runs. For users requiring exact target control, TensorDynamic also provides a higher-overhead static workflow based on offline profiling.

Once selected, faults are immediately injected into the destination registers of these N targeted instances, while all subsequent or non-targeted instructions proceed unmodified. This approach ensures instruction-level isolation, as faults are applied to specific, discrete destination registers that do not structurally interfere with the execution pipeline of neighboring threads. For faults injected across different input samples, errors affecting one sample do not propagate to others because each sample is classified independently. However, if users inject faults into multiple layers, error propagation is an intentional feature for modeling cumulative degradation. Consequently, TensorDynamic enables the efficient accumulation of fault effects within a single inference pass, exposing observable accuracy shifts without the overhead of repetitive single-fault trials.

D. Thread-level Fault Targeting

Existing fault injection frameworks vary in the granularity and mechanism used to choose injection targets. For Tensor Core HMMA operations, TensorDynamic supports thread-level fault targeting, allowing faults to be injected into destination registers associated with individual thread instances. This design is intended to better approximate perturbations to individual output elements and enables element-level sensitivity analysis, revealing fine-grained failure behavior.

E. Dynamic Single-Pass Injection

Existing fault injection tools like NVBitFI [33] and MPGemmFI [9] operate in a two-phase execution model, where a profiling run enumerates dynamic instructions and identifies target points, followed by multiple separate injection runs that perturb one chosen dynamic event at a time. While effective for general workloads, two-phase designs are problematic for Tensor Core kernels invoked by deep learning frameworks: the dynamic instruction stream and kernel execution order may differ across runs due to optimization and algorithm changes. As a result, offline profiles may not reflect the exact dynamic instance at injection time, leading to mismatches in instruction index and incorrect instruction positions, and eventually causing unsuccessful error injection or injection into unexpected places.

TensorDynamic adopts a single-pass execution model, where profiling and injection occur within the same kernel execution. The tool instruments both pre- and post-HMMA callbacks and transitions between profiling and injection modes once the selected dynamic instruction is reached. TensorDynamic therefore avoids trace storage, eliminates mismatches between profiling and injection runs, and ensures the injected fault corresponds to the exact instruction instance observed at runtime. Combined, TensorDynamic gathers dynamic kernel frequency information and performs multi-injection during the same execution to avoid errors arising from evolving kernel configurations. Algorithm 1 summarizes the runtime workflow of TensorDynamic.

F. Fault model

TensorDynamic provides two fault models, both inspired by hardware-level fault mechanisms in modern accelerators and fully parameterized at runtime. First, the tool supports a single bitflip: one selected bit in the 32-bit destination register of an HMMA instruction is inverted. The second fault model is error multiplication, which abstracts the effect of large-magnitude perturbations that arise when faults hit high-order bits. Prior work [36] shows that faults near the MSB dominate SDC outcomes because they introduce exponentially larger deviations. Instead of flipping multiple high-order bits, TensorDynamic directly multiplies the destination register by a user-defined scaling factor, allowing controlled exploration of extreme positive or negative deviations. Both injection modes operate at the instruction level and directly modify Tensor Core outputs, enabling a continuous range of perturbation magnitudes and controlled sensitivity analysis of CNN inference.

TensorDynamic injects faults into the destination register values of HMMA instructions, modeling errors that may occur during operand reads or during the Tensor Core computation. We assume that the register file contents are correct before execution; that is, our fault model targets transient errors within the Tensor Core pipeline rather than storage-level corruption. Extending the fault model to cover register file or SRAM faults is straightforward in principle. Still, since our focus is on modeling faults in Tensor Core execution, we do not provide this capability in the current implementation.

Algorithm 1 TensorDynamic Runtime Execution Flow

Parameters:

- Boundary intervals $\mathcal{B}$ (user-defined via profiling)
- Target kernel k (the k-th HMMA kernel launch in each interval)
- Target HMMA instruction set $\mathcal{H}$
- Thread count N (number of injected threads per instruction)
- Fault model $m \in \{\text{BITFLIP}, \text{ERRORMULT}\}$

for all kernel launch with ID kid **do**
  {Frequency-Guided Sampling}
  if $kid \in \mathcal{B}_i$ **and** is the targeted k-th kernel in $\mathcal{B}_i$ **then**
    {Multi-Injection: $|\mathcal{H}| \times N$ faults per kernel}
    for all HMMA site $h \in \mathcal{H}$ **do**
      {Thread-level Profiling}
      $\mathcal{T}_h \leftarrow$ Sample N active threads executing instruction h
      {Thread-level Injection}
      for all thread $t \in \mathcal{T}_h$ **do**
        Apply BITFLIP or ERRORMULT to destination register
      end for
    end for
  end if
end for

Dynamic Single-Pass Injection

G. Static profiling and injection

TensorDynamic also provides a static two-phase workflow for users who need deterministic, repeatable fault experiments, following the classical NVBitFI [33] structure. The `tensor_profile` tool runs the program once to record all HMMA instruction instances. It produces a user-selected target list (by explicit specification of kernel ID, instruction ID, block/thread coordinates). The `tensor_injector` then replays the program and injects faults only at those recorded points. By separating profiling and injection, this workflow provides stable reproducibility. Once a target list is produced, the same execution points can be perturbed repeatedly under different fault models or parameter settings, enabling controlled comparisons across injections without relying on runtime sampling.

V. EXPERIMENT SETUP

To systematically study the impact of fault injection abstraction on deep learning inference, we explore a design space defined by model architecture, fault characterization, injection granularity, and evaluation workloads. Our goal is to enable controlled comparison between software-level and hardware-level fault injection while minimizing confounding factors.

A. Models, Datasets, and Metrics

We evaluate a diverse set of convolutional neural network architectures that vary in depth, structural complexity, and application domain. Specifically, we consider ResNet20 [15], ShuffleNetV2 [20] and MobileNetV2 [28], [5] for image classification, and YOLOv9 [35], [4] for object detection, as summarized in Table II. ResNet20, ShuffleNetV2 and MobileNetV2 [5] are evaluated on the CIFAR-100 dataset, and YOLOv9 [35], [4] is evaluated with the selected images from MS COCO validation set (val2017). For classification tasks, we report top-1 accuracy over the full evaluation dataset, while for object detection, we report mean Average Precision (mAP). All experiments are conducted in inference mode with fixed model parameters, inputs, and batch sizes to reduce nondeterminism.

TABLE II
DNN MODELS AND DATASETS USED IN THE EVALUATION.

Model	Task	Dataset	Batch Size
ResNet20	Image Classification	CIFAR-100	100
ShuffleNetV2	Image Classification	CIFAR-100	100
MobileNetV2	Image Classification	CIFAR-100	100
YOLOv9	Object Detection	MS COCO	2

B. Hardware Platform

All experiments are performed on an NVIDIA A100 GPU based on the Ampere architecture (SM80) with NVBit version 1.7.6. We focus exclusively on transient faults affecting Tensor Core–accelerated MMA execution and assume that ECC protects memory systems and register files. This setup enables focused study of transient faults affecting Tensor Core execution, minimizing the impact of other architectural components.

C. Fault Injection Methodology

Fault characteristics are explored along two dimensions: injection intensity and fault severity. Injection intensity is controlled by varying the number of injected faults per inference run, allowing us to study the effects of cumulative corruption. Fault severity is varied using different error factors, including value perturbations and targeted bitflips. In particular, we focus on bitflips in higher-order exponent bits of floating-point values, which are known to induce large numerical deviations and are representative of severe transient computational faults.

We compare two fault injection abstractions. At the application level, faults are injected by directly modifying intermediate tensors produced by PyTorch operators, modeling corruption at operator boundaries. We use both PyTorchFI [21] and MRFI [17] to modify selected output tensor elements of the target layers. Consistent with the neuron-perturbation method used in PyTorchFI, these runtime injections are implemented through hook-based instrumentation, perturbing layer output activation values at user-specified locations.

At the instruction level, we use TensorDynamic to inject faults into the destination registers of dynamic Tensor Core HMMA instructions at the SASS level, modeling transient faults during execution. To compare the two levels, we approximate layer-to-kernel correspondence by selecting kernels and application-level convolutional layers at similar relative

TABLE III
AVERAGE FAULT OBSERVABILITY AND NUMERICAL IMPACT UNDER HARDWARE-LEVEL TENSOR CORE FAULT INJECTION.

Metric	Convolution	GEMM
Total output elements	1,638,400	1,638,400
Average insertion rate (%)	0.169	0.169
Observability (%)	85.2	100.2
Mean Squared Error	0.051	7.206
Mean Abs. Mult. Scale	1.160	1.423

positions within each boundary interval, and inject faults at the corresponding targets in that interval. We also enforce matched fault injection parameters (e.g., error count, corruption multiplier, and target bit position for the bitflip fault model) across all tools to ensure a fair comparison.

D. Operator-Level Characterization

In addition to end-to-end inference evaluation, we instrument hardware-level experiments to record the fraction of zero-valued destination registers for all HMMA instructions after instruction completion. This measurement characterizes the sparsity and value distribution of thread-level HMMA outputs and provides insight into how injected faults interact with underlying execution behavior.

Finally, we use simple matrix-multiplication and convolution workloads rather than full neural networks to study fault propagation in isolation. By injecting errors only into standalone matrix and convolution computations at both the PyTorch and SASS levels, we avoid interference from complex network layers and control logic. This decoupling allows us to directly observe how injected errors affect numerical outputs in simple computations and compare this behavior with error propagation in full models.

VI. EVALUATION

A. End-to-End Accuracy Trends Across Injection Methods

Figure 4 presents end-to-end inference accuracy under increasing fault injection counts for all evaluated models, comparing instruction-level SASS injection (TensorDynamic) against application-level tensor perturbation (PyTorchFI [21] and MRFI [17]). We use the layer-to-kernel correspondence methodology described in Section V to align application-level and instruction-level injection sites. For each boundary interval, the fault count $N \in \{10, 50, 100, 500, 1000\}$ and corruption multiplier $\alpha \in \{10, 50, 100, 1000\}$ are swept identically across TensorDynamic and the application-level fault injection frameworks. For the bitflip model, the injected bit position is fixed at bit 30 across all tools, corresponding to the most significant exponent bit of the FP32 value, so that each injected fault produces a large numerical perturbation.

Across four models and two fault modes, we observe a consistent pattern: instruction-level injection generally results in smaller accuracy degradation than application-level injection at comparable injection counts and fault severity, and the discrepancy grows as the number of injected faults and the error magnitude increase. This result suggests that tensor-level perturbations introduce stronger effective disturbances than transient instruction-level faults during hardware execution. Consequently, application-level fault injection methods tend to overestimate model vulnerability relative to instruction-level fault injection, particularly as the number of injected faults per inference increases.

To further investigate the disparity between application-level and instruction-level fault injection, we progressively increase the number of injected faults at the SASS level to induce comparable model degradation. In our experiments, the application-level baseline corresponds to injecting 500 errors with an error multiplication factor of 1000, which is sufficient to produce substantial accuracy degradation. In contrast, as shown in Figure 5, SASS-level injection requires considerably more fault events to achieve a similar end-to-end impact. Across all four evaluated models, approaching the effect of application-level injection typically requires increasing the error count by 20×–200×, often accompanied by an additional error-magnitude amplification of 10×–100×. Notably, for MobileNetV2 [28], even this level of amplification is often insufficient to fully match the degradation observed under application-level injection. As a result, achieving comparable model-level impact at the hardware level requires substantially more aggressive fault injection.

B. Results of Operator-Level Characterization

At the operator level, analysis of Tensor Core HMMA destination registers reveals moderate sparsity in post-execution values, with a non-trivial fraction of registers evaluating to zero. As shown in Fig 6, the x-axis denotes the fraction of destination registers that evaluate to zero for each Tensor Core kernel, while the y-axis reports the number of kernel invocations falling into each sparsity bin. The resulting distribution shows that most kernels produce relatively dense outputs, with many kernels exhibiting only 0–10% of destination registers set to 0, while others exhibit moderately higher sparsity levels. Averaged across all kernels, destination-register sparsity ranges from 8.3% to 14.8% across different models. Although the overall sparsity is moderate, the presence of zero-valued destination registers introduces opportunities for fault masking. A single bitflip on a zero-valued register can introduce a perturbation with a maximum absolute value of 2.0 in both FP16 and FP32, achieved by flipping the most significant exponent bit. Small perturbations like this can be masked by subsequent computation. As a result, such errors are less likely to propagate to final outputs, reducing fault observability in Tensor Core–accelerated workloads.

To isolate fault propagation mechanisms, we evaluate standalone Tensor Core-accelerated convolution computation and matrix multiplication (GEMM) workloads under identical hardware-level fault injection configurations. Both workloads produce output tensors of 1,638,400 elements, enabling direct comparison. Faults are injected into Tensor Core MMA destination registers, with the number of injected events swept across 10, 100, 500, 1,000, 5,000, and 10,000, corresponding to insertion rates ranging from 6.1×10^{-6} to 6.1×10^{-3}.

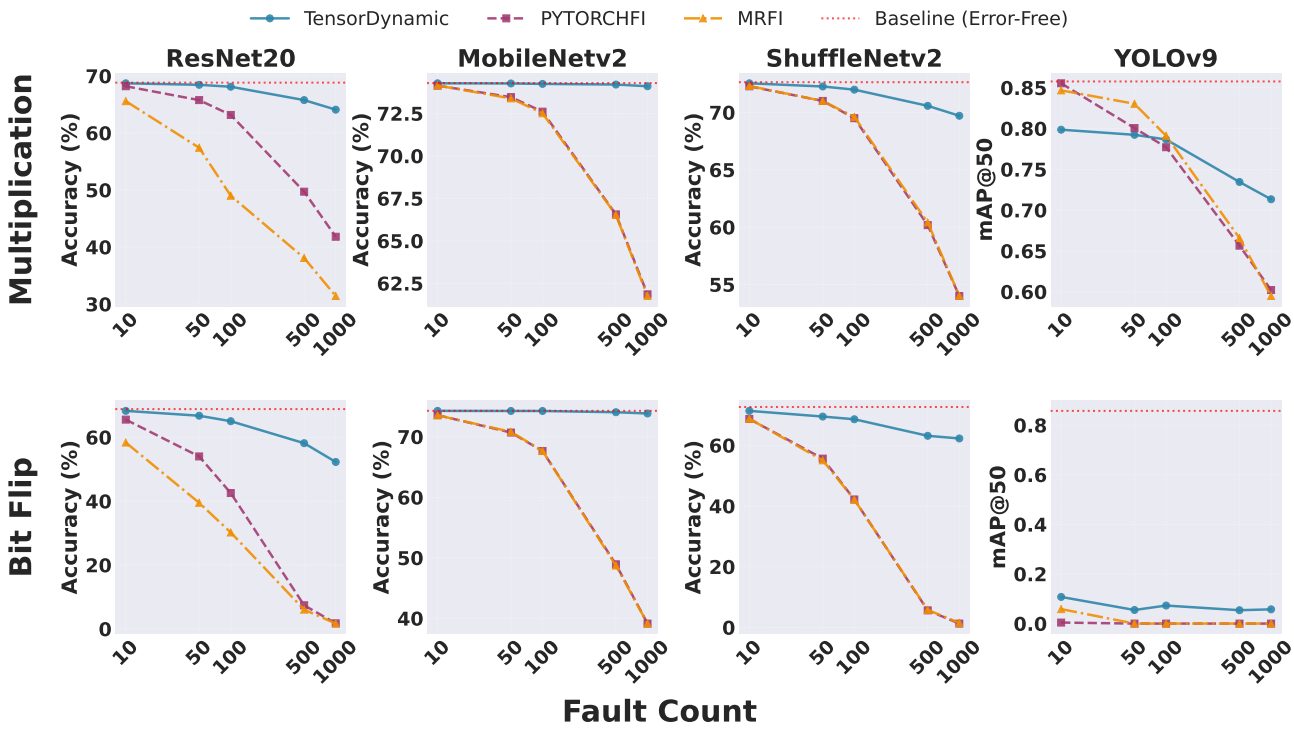

Fig. 4. Fault injection analysis across four deep learning models (ResNet20, ShuffleNetV2, MobileNetV2, and YOLOv9) comparing three fault injection methods: TensorDynamic, PyTorchFI, and MRFI.

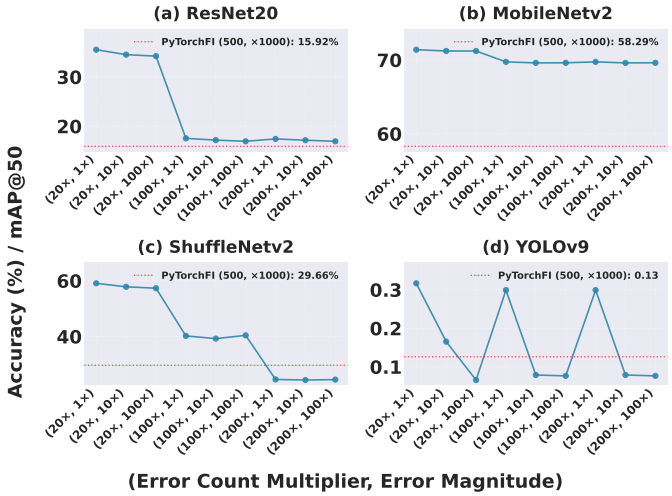

Fig. 5. Accuracy and mAP degradation under increasing instruction-level fault injection intensity across four models. The dashed red line shows the application-level injection baseline (PyTorchFI: 500 errors, magnitude 1000).

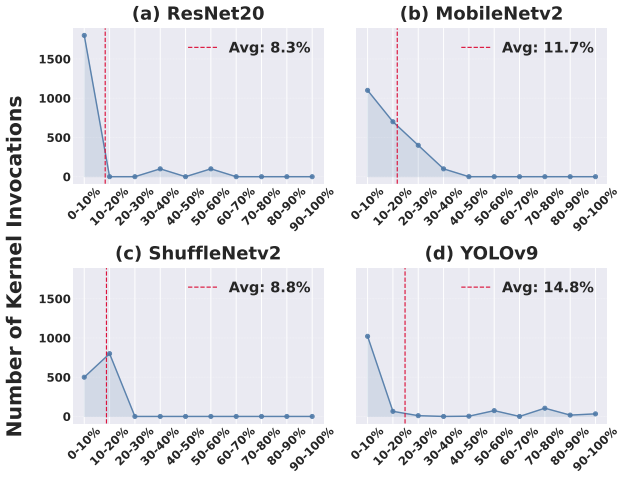

Fig. 6. Distribution of zero-valued Tensor Core MMA destination registers across CNN architectures. The x-axis shows the fraction of destination registers that are zero per Tensor Core kernel, and the y-axis shows the number of Tensor Core kernel invocations in each bin.

Injected faults are modeled by scaling the affected destination register values by $50\times$. For each configuration, we measure the *observability* (defined as the ratio of changed outputs to injected events), the mean squared error (MSE) over the full output tensor, and the mean absolute multiplicative scale factor, defined as the absolute ratio between corrupted and golden output values, to capture how these large-magnitude perturbations propagate to architectural outputs.

As summarized in Table III, GEMM exhibits very high fault observability, with an average observability of 100.2%, indicating that some injected faults propagate to multiple output elements. In contrast, convolution shows reduced observability of 85.2%, indicating that approximately 15% of injected faults are masked before reaching the architectural output. Moreover, GEMM experiences substantially higher numerical impact: the average MSE is significantly larger, and the mean absolute multiplicative scale factor is consistently higher than those observed for convolution. Overall, these averaged results

TABLE IV
SUMMARY OF RELATED TOOLS.

Work	Batch-level injection	Layer-level injection	Instr-level injection	Thread-level injection
PyTorchFI [21]	◐	✓	✗	✗
MRFI [17]	◐	✓	✗	✗
SASSIFI* [14]	✗	✗	✓	✓
NVBitFI [33]	✗	✗	✓	✓
MPGemmFI [9]	✗	✗	✓	✓
TensorDynamic (ours)	✓	✓	✓	✓

*Deprecated; lack of support for Tensor Cores

Legend: ✓ native; ◐ partial/indirect; ✗ not the focus.

reinforce that operator structure fundamentally governs both the observability and severity of hardware-level faults—effects that software-level fault injection models do not capture.

VII. Related Work

Research on DNN reliability commonly relies on physical experiments [29], [30], such as neutron beam testing, as well as software-based fault injection to derive fault models [26], [27]. Prior characterization studies have explored fault behaviors across different system layers [13], [16], [36]. While these efforts provide valuable insights, existing cross-layer and microarchitectural approaches often lack the fine-grained control necessary to analyze the complex execution behavior of Tensor Cores.

Fault injection frameworks span multiple abstraction levels, ranging from high-level software-based perturbation, to intermediate representations such as IR-level injection, and low-level injection at the assembly- or instruction-level [3] [22]. Application-level fault injection [21], [17] offers high throughput but sacrifices accuracy, frequently overestimating vulnerability compared to hardware-aware, instruction-level injection [8], [23]. Tools such as SASSIFI [14] and NVBitFI [33] leverage binary instrumentation to enable low-level fault injection. However, they struggle to associate injected faults with high-level application semantics. Moreover, most existing tools inject only a single fault per execution, limiting their ability to capture realistic and complex failure scenarios. Table IV summarizes the differences.

Specialized accelerators, such as Tensor Cores, introduce additional reliability challenges due to mixed-precision arithmetic and limited ECC coverage. While prior work has examined the impact of faults on specific matrix operations [2], [9], there is still no dedicated tool that directly injects faults into HMMA instructions while preserving the high-level DNN context. Our work addresses this gap by enabling targeted fault injection at the HMMA instruction level, enabling precise, context-aware reliability analysis of Tensor Core execution.

VIII. Conclusion

In this work, we demonstrated that the choice of fault injection abstraction fundamentally alters the observed reliability profile of GPU-accelerated deep learning workloads. By comparing TensorDynamic against application-level methodologies (PyTorchFI, MRFI), we revealed that high-level tensor perturbations consistently overestimate model vulnerability. This divergence stems from the inability of application-level models to capture critical microarchitectural masking effects, specifically the sparsity of Tensor Core destination registers and the distinct error propagation dynamics between convolution and GEMM operations. Consequently, relying solely on software-level abstractions risks generating misleading failure characterizations that may lead to inefficient or unsafe system designs. Our findings underscore the need for fine-grained, instruction-level analysis to address realistic fault scenarios in safety-critical environments.

Acknowledgment

We thank the members of the HPArch group and the anonymous reviewers for their helpful feedback and suggestions. We also acknowledge CRNCH Rouge Gallery for infrastructure support. In addition, we acknowledge the use of generative AI tools, including ChatGPT, Google Gemini, and Claude, to assist in improving the manuscript's readability, refining code and formatting the figures. This article has been authored by an employee of National Technology & Engineering Solutions of Sandia, LLC under Contract No. DE-NA0003525 with the U.S. Department of Energy (DOE). The employee owns all right, title, and interest in and to the article and is solely responsible for its contents. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this article or allow others to do so, for United States Government purposes. The DOE will provide public access to the results of federally sponsored research in accordance with the DOE Public Access Plan (https://www.energy.gov/downloads/doe-public-access-plan).

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: a system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. USA: USENIX Association, 2016, p. 265–283.

[2] P. M. Basso, F. F. d. Santos, and P. Rech, "Impact of tensor cores and mixed precision on the reliability of matrix multiplication in GPUs," vol. 67, no. 7, pp. 1560–1565. [Online]. Available: https://ieeexplore.ieee.org/document/9019610/

[3] C. Bolchini, L. Cassano, A. Miele, and A. Toschi, "Fast and accurate error simulation for CNNs against soft errors," *IEEE Transactions on Computers*, vol. 72, no. 4, pp. 984–997, Apr. 2023. [Online]. Available: https://doi.org/10.1109/TC.2022.3184274

[4] H.-S. Chang, C.-Y. Wang, R. R. Wang, G. Chou, and H.-Y. M. Liao, "YOLOR-based multi-task learning," *arXiv preprint arXiv:2309.16921*, 2023.

[5] Y. Chen, "Pytorch cifar models," https://github.com/chenyaofo/pytorch-cifar-models, accessed: 2025-5-17.

[6] Z. Chen, N. Narayanan, B. Fang, G. Li, K. Pattabiraman, and N. DeBardeleben, "Tensorfi: A flexible fault injection framework for tensorflow applications," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, 2020, pp. 426–435.

[7] A. Dehbaoui, J.-M. Dutertre, B. Robisson, and A. Tria, "Electromagnetic transient faults injection on a hardware and a software implementations of aes," in *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2012, pp. 7–15.

[8] G. Esposito, J.-D. Guerrero-Balaguera, J. E. R. Condia, and M. S. Reorda, "Evaluating different fault injection abstractions on the assessment of dnn sw hardening strategies," in *2024 IEEE 33rd Asian Test Symposium (ATS)*, 2024, pp. 1–6.

[9] B. Fang, X. Li, H. Dam, C. Tan, S. K. S. Hari, T. Tsai, I. Laguna, D. Tao, G. Gopalakrishnan, P. Nair, K. Barker, and A. Li, "Understanding mixed precision GEMM with MPGemmFI: Insights into fault resilience," in *2024 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 166–178, ISSN: 2168-9253. [Online]. Available: https://ieeexplore.ieee.org/document/10740821/

[10] J. Gracia-Morán, J. C. Baraza Calvo, D. A. Gil Tomás, L. Saiz-Adalid, and P. Gil, "Effects of intermittent faults on the reliability of a reduced instruction set computing (risc) microprocessor," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 144–153, 2014.

[11] R. Gräfe, Q. S. Sha, F. Geissler, and M. Paulitsch, "Large-scale application of fault injection into pytorch models: An extension to pytorchfi for validation efficiency," in *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks – Supplemental Volume (DSN-S)*, 2023, pp. 56–62.

[12] J. Guerrero-Balaguera, J. E. R. Condia, F. F. dos Santos, M. S. Reorda, and P. Rech, "Understanding the effects of permanent faults in GPU's parallelism management and control units," in *SC23: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–15.

[13] J. D. Guerrero Balaguera, J. E. Rodriguez Condia, and M. Sonza Reorda, "Effective fault effects evaluation for permanent faults in gpus executing dnns," *ACM Transactions on Design Automation of Electronic Systems*, vol. 30, no. 2, pp. 1–33, 2025.

[14] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, "Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017, pp. 249–258.

[15] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[16] K. Hector, P.-A. Moellic, M. Dumont, and J.-M. Dutertre, "A closer look at evaluating the bit-flip attack against deep neural networks," in *2022 IEEE 28th International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE, pp. 1–5. [Online]. Available: https://ieeexplore.ieee.org/document/9897693/

[17] H. Huang, C. Liu, X. Xue, B. Liu, H. Li, and X. Li, "MRFI: An open-source multiresolution fault injection framework for neural network processing," vol. 32, no. 7, pp. 1325–1335. [Online]. Available: https://ieeexplore.ieee.org/document/10494993/

[18] P. K P, D. M S, P. Chaudhary, A. Dev, N. Patra, and S. Ghosal, "Real-time object detection using convolutional neural networks for autonomous vehicles," in *2025 International Conference on Automation and Computation (AUTOCOM)*, Mar. 2025, pp. 1051–1056.

[19] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. 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*, 2017, pp. 1–12.

[20] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, "Shufflenet v2: Practical guidelines for efficient cnn architecture design," in *Computer Vision – ECCV 2018: 15th European Conference, Munich, Germany, September 8–14, 2018, Proceedings, Part XIV*. Berlin, Heidelberg: Springer-Verlag, 2018, p. 122–138. [Online]. Available: https://doi.org/10.1007/978-3-030-01264-9_8

[21] 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)*, 2020, pp. 25–31.

[22] L. Palazzi, G. Li, B. Fang, and K. Pattabiraman, "A tale of two injectors: End-to-end comparison of ir-level and assembly-level fault injection," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, 2019, pp. 151–162.

[23] 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)*, 2021, pp. 902–915.

[24] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Z. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, Eds., 2019, pp. 8024–8035. [Online]. Available: https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[25] B. Reagen, U. Gupta, L. Pentecost, P. Whatmough, S. K. Lee, N. Mulholland, D. Brooks, and G.-Y. Wei, "Ares: A framework for quantifying the resilience of deep neural networks," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[26] A. Ruospo, G. Gavarini, C. de Sio, J. Guerrero, L. Sterpone, M. S. Reorda, E. Sanchez, R. Mariani, J. Aribido, and J. Athavale, "Assessing convolutional neural networks reliability through statistical fault injections," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, ISSN: 1558-1101. [Online]. Available: https://ieeexplore.ieee.org/document/10136998/

[27] A. Ruospo, M. S. Reorda, R. Mariani, and E. Sanchez, "An effective iterative statistical fault injection methodology for deep neural networks," vol. 74, no. 7, pp. 2431–2444. [Online]. Available: https://ieeexplore.ieee.org/document/10985784/

[28] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.

[29] F. F. d. Santos, A. Kritikakou, J. E. R. Condia, J.-D. Guerrero-Balaguera, M. S. Reorda, O. Sentieys, and P. Rech, "Characterizing a neutron-induced fault model for deep neural networks," *IEEE Transactions on Nuclear Science*, vol. 70, no. 4, pp. 370–380, 2023.

[30] F. F. d. Santos, P. F. Pimenta, C. Lunardi, L. Draghetti, L. Carro, D. Kaeli, and P. Rech, "Analyzing and increasing the reliability of convolutional neural networks on GPUs," vol. 68, no. 2, pp. 663–677. [Online]. Available: https://ieeexplore.ieee.org/document/8536419/

[31] M. Stephenson, S. K. S. Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Connor, and S. W. Keckler, "Flexible software profiling of GPU architectures," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 185–197.

[32] X. Tian, J. Gu, B. Li, Y. Liu, Y. Wang, Z. Zhao, K. Zhan, P. Jia, X. Lang, and H. Zhao, "Drivevlm: The convergence of autonomous driving and large vision-language models," *arXiv preprint arXiv:2402.12289*, 2024.

[33] T. Tsai, S. K. S. Hari, M. Sullivan, O. Villa, and S. W. Keckler, "NVBitFI: Dynamic fault injection for GPUs," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 284–291, ISSN: 2158-3927. [Online]. Available: https://ieeexplore.ieee.org/document/9505068/

[34] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, "NVBit: A dynamic binary instrumentation framework for NVIDIA GPUs," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*. Association for Computing Machinery, 2019, pp. 372–383.

[35] C.-Y. Wang and H.-Y. M. Liao, "YOLOv9: Learning what you want to learn using programmable gradient information," 2024.

[36] X. Wei, C. Zhou, H. Yue, and J. T. Zhou, "TC-SEPM: Characterizing soft error resilience of CNNs on tensor cores from program and microarchitecture perspectives," vol. 145, p. 103024. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1383762123002035

[37] X. Zhou, M. Liu, E. Yurtsever, B. L. Zagar, W. Zimmer, H. Cao, and A. C. Knoll, "Vision language models in autonomous driving: A survey and outlook," *IEEE Transactions on Intelligent Vehicles*, 2024.

APPENDIX

A. Abstract

This artifact appendix describes how to set up TensorDynamic and provides scripts and instructions to reproduce the key results presented in the paper. TensorDynamic is an application-aware, instruction-level fault injection framework for NVIDIA Tensor Cores. The artifact includes all necessary steps and configurations to reproduce the results shown in Table 3, Figure 4, Figure 5, and Figure 6. The full repository is publicly available at https://github.com/gthparch/TensorDynamic_AE, and an archived version is available at https://doi.org/10.5281/zenodo.18919761.

B. Artifact check-list (meta-information)

- **Algorithm:** Instruction-level fault injection via NVBit dynamic binary instrumentation targeting HMMA instructions.
- **Compilation:** GCC 12.3.0, CUDA 12.6.1.
- **Data set:** CIFAR-100 and a subset of COCO val2017.
- **Run-time environment:** Linux x86-64, Python 3.10 (Conda).
- **Hardware:** NVIDIA A100 GPU. Other NVIDIA GPUs may also be used; however, differences in ISA and microarchitecture across GPU generations can lead to substantially different fault injection results, so an A100 is strongly recommended for reproducing the reported numbers.
- **Metrics:** Top-1 accuracy (%) for CIFAR-100 models; mAP50 for YOLOv9.
- **Output:** CSV summary files and plots reproducing Table 3 and Figures 4–6.
- **How much disk space required (approximately)?:** ∼20 GB total
- **How much time is needed to prepare workflow (approximately)?:** ∼ 1 hour
- **How much time is needed to complete experiments (approximately)?:** ∼ 30-35 hours
- **Publicly available?:** Yes.
- **Workflow automation framework used?:** Bash shell scripts.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.18919761

C. Description

1) How to access: The artifact is publicly available on GitHub at https://github.com/gthparch/TensorDynamic_AE and Zenodo at https://doi.org/10.5281/zenodo.18919761.

2) Hardware dependencies: Experiments in the paper were conducted on an NVIDIA A100 GPU. Other NVIDIA GPUs with Tensor Core support may run the artifact, but differences in ISA and microarchitecture across GPU generations can lead to substantially different fault injection results.

3) Software dependencies: Linux x86-64, CUDA 12.6.1, GCC 12.3.0, Conda (Miniconda or Anaconda), and Python 3.10. All required Python packages are specified in `requirements.txt`.

4) Data sets: CIFAR-100 is downloaded automatically by torchvision during the first run. A small subset of COCO val2017 (20 images) and YOLOv9 weights are downloaded using `setup_yolo.sh`.

5) Models: The artifact evaluates three CIFAR-100 classification models (ResNet-20, MobileNetV2, and ShuffleNetV2) and one object detection model (YOLOv9). Pretrained model weights are downloaded automatically during setup.

D. Installation

Follow the steps below from the `TensorDynamic/` directory.

1. Download and extract NVBit 1.7.6, then clone this repo:

```
$ wget https://github.com/NVlabs/NVBit/\
releases/download/v1.7.6/\
nvbit-Linux-x86_64-1.7.6.tar.bz2
$ tar xvfj nvbit-Linux-x86_64-1.7.6.tar.bz2
$ cd nvbit_release_x86_64
$ git clone https://github.com/gthparch/\
TensorDynamic_AE
$ cd TensorDynamic
```

2. Create conda environment and install dependencies (∼10 min):

```
$ conda create -n myenv python=3.10 pip -y
$ conda activate myenv
$ pip install -r requirements.txt
```

3. Set up YOLOv9 repository, weights, and dataset (∼10 min):

```
$ bash setup_yolo.sh
```

E. Experiment workflow

All experiments are run from the `TensorDynamic/` directory using self-contained Bash scripts. Each script performs compilation (if needed), model inference, fault injection, result parsing, and plotting. Table 3 and Figures 4–6 can be reproduced independently by running the corresponding scripts described below.

F. Evaluation and expected results

Because fault injection is inherently stochastic, reproduced results may differ slightly from those reported in the paper. However, the overall patterns and trends should closely match.

Boundary files defining the kernel ranges for each evaluated model are included in the repository. Since `BOUNDARY_PATH` is a user-defined parameter in TensorDynamic, these files allow the artifact to run using the same kernel ranges as in the paper.

Figure 6 — HMMA Register Sparsity (∼1 h)

Recommended first step to verify the environment before running longer sweeps.

```
$ bash figure6/figure6.sh
```

Output is saved to `figure6/reg_dist_plots/combined_sparsity_dest_zero_4panel.pdf`, corresponding to Figure 6 in the paper.

Table 3 — Fault Injection Observability (∼3 h)

```
$ bash table3/table3.sh
```

Output is saved to `table3/results/table3_summary.csv`, corresponding to Table 3 in the paper.

Figure 4 — Accuracy Comparison Across Fault Injection Methods (∼16–17 h)

```
$ bash figure4/figure4.sh
```

Alternatively, run the steps separately:

```
$ bash figure4/sweep_pytorchfi_mrfi.sh
$ bash figure4/sweep_tensordynamic.sh
$ bash figure4/plot_results.sh
```

Output is saved to `figure4/plots/eight_plots.pdf`, corresponding to Figure 4 in the paper.

Figure 5 — Instruction-Level Fault Injection Scaling (∼8 h)

```
$ bash figure5/figure5.sh
```

Alternatively, run the steps separately:

```
$ bash figure5/sweep_baseline.sh
$ bash figure5/plot_figure5.sh
```

Output is saved to `figure5/plots/nine_point_plot.pdf`, corresponding to Figure 5 in the paper.

G. Experiment customization

TensorDynamic is configured at runtime through environment variables, eliminating the need for recompilation between experiments.

- `INJECTION_MODE`: `0`=single bit-flip, `1`=FP16 multiplication, `2`=FP32 multiplication
- `NUM_THREADS_RECORD`: number of HMMA thread executions to profile and inject per kernel
- `CORRUPT_MULT_F32`: corruption multiplier applied as FP32 to the full 32-bit register (mode 2 only)
- `CORRUPT_MULT_LOW` / `CORRUPT_MULT_HIGH`: multipliers for low and high FP16 halves of each register (mode 1 only)
- `BIT_POSITION`: bit index to flip in mode 0, range 0–31 (mode 0 only)
- `TARGET_KERNEL_POS`: which HMMA kernel within a frequency range to inject
- `TARGET_INSTR_LIST`: comma-separated 1-indexed HMMA instruction positions to inject, e.g. `"1,3,5"`
- `BOUNDARY_PATH`: path to the boundary file defining kernel ID ranges for one forward pass

Example command using FP32 multiplication injection mode:

```
NUM_THREADS_RECORD=500 INJECTION_MODE=2 \
CORRUPT_MULT_F32=1000 TARGET_KERNEL_POS=1 \
TARGET_INSTR_LIST=1,3 \
BOUNDARY_PATH=<path_to_boundary_file> \
LD_PRELOAD=tensor_dynamic/tensor_dynamic.so \
python3 <evaluation_script>.py
```

H. Methodology

Submission, reviewing and badging methodology:

- https://ieeexplore.ieee.org/Xplorehelp/overview-of-ieee-xplore/about-content#reproducibility-badges
- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://ctuning.org/ae