ALPRI-FI: A Framework for Early Assessment of Hardware Fault Resiliency of DNN Accelerators

Karim Mahmoud * and Nicola Nicolici

Department of Electrical and Computer Engineering, McMaster University,
Hamilton, ON L8S 4L8, Canada; nicolici@mcmaster.ca
* Correspondence: mahmoudk@mcmaster.ca

Abstract: Understanding how faulty hardware affects machine learning models is important to both safety-critical systems and the cloud infrastructure. Since most machine learning models, like Deep Neural Networks (DNNs), are highly computationally intensive, specialized hardware accelerators are developed to improve performance and energy efficiency. Evaluating the fault resilience of these DNN accelerators during early design and implementation stages provides timely feedback, making it less costly to revise designs and address potential reliability concerns. To this end, we introduce Architecture-Level Pre-Register-Transfer-Level Implementation Fault Injection (ALPRI-FI), which is a comprehensive framework for assessing the fault resilience of DNN models deployed on hardware accelerators.

Keywords: hardware fault assessment; machine learning hardware accelerators

1. Introduction

Over the past decade, the field of machine learning has experienced a resurgence, which is primarily driven by the availability of large amounts of training data and computational power. These advancements have enabled the validation and practical application of machine learning models in real-world scenarios that were previously impractical to explore. Deep Neural Networks (DNNs), considered the most significant subfield of machine learning in recent years, have played a crucial role in advancing areas such as computer vision, speech recognition, and natural language processing.

As the adoption of DNNs continues to grow, so does the development of custom hardware accelerators [1–4]. DNN models often involve billions of Multiply–Accumulate (MAC) operations per inference, making their computations computationally expensive. Additionally, DNNs are now being adopted in form factor-constrained environments, requiring a combination of high throughput, low latency, and energy efficiency. Therefore, relying solely on conventional processors for DNN execution is inadequate, highlighting the need for dedicated hardware accelerators, which are specialized to enhance the performance of the computational patterns inherent to the DNN workloads.

Despite the recent successes in the consumer domain of machine learning, there is still more work to enhance the reliability of DNN-enabled systems, particularly when deployed in safety-critical scenarios, such as health and autonomous driving. An often overlooked aspect is that even robustly validated models may fail to perform correctly on defective hardware. Consequently, understanding the relationship between a new hardware accelerator and the DNN models it is expected to support in the presence of hardware faults is increasingly important alongside other assessment criteria for hardware accelerators, such as performance and energy efficiency. Importantly, developing this understanding during the design space exploration phase can have a significant impact by allowing early design revisions before a full RTL implementation is carried out.

Reliability is particularly important for safety-critical applications and industrial domains where hardware accelerators are designed to run for extended periods, such as
aerospace, for example. Recent studies have demonstrated a high sensitivity of DNN inference accuracy to applications running on faulty hardware. The study by [5] has revealed that the fault resiliency of the hardware is influenced by both the hardware architecture and the data reuse patterns that impact its fault resilience. Transient bit flips can occur in flops (FFs) and other memory structures resulting in temporary but detrimental effects [5–11]. Concurrently, the growing use of machine learning technologies in fields that require hardware with long lifespans has motivated an interest in exploring how permanent hardware faults affect system-level behavior [11–14]. Although these studies have emphasized the importance of assessing the reliability of DNN hardware, their scope is limited to errors in memory and/or storage elements within the datapath. Due to the inherent trade-off between the types and number of faults that can be evaluated and the length of fault injection experiments, there is an underexplored area in understanding how permanent hardware faults in large-scale arrays of processing engines can impact the accuracy of DNN inference. This challenge is particularly pronounced in designs that feature large-scale arrays of arithmetic units, such as those described in [2], where even register-transfer level (RTL) simulations are expensive [15]. Additionally, it is important to conduct this type of fault assessment experiments at an early stage of design space exploration to ensure that design revisions are made in a timely manner.

Arithmetic units can require 27–40% of the DNN accelerator hardware area [2,3,12,16]; however, fewer works are presented in the literature to evaluate the resiliency toward faults that are internal to arithmetic units at the DNN application level. The work from [17] has established how, under simple DNN workloads, faults in integer multipliers will cause deviations in the output of the arithmetic units. However, further investigations are needed to understand how these deviations will impact the accuracy of DNN inference at the application level. As shown in Figure 1, there are different stages of the chip development process during which fault resiliency assessment can be performed (a more detailed description of this figure is provided in Section 4). It is important to clarify that the term Early used in the rest of this work refers to the system-level exploration design stages before the hardware architecture is fully developed and implemented.

In addition to the motivation provided above, recent works have reported increases in Silent Data Corruption (SDC) [18,19]. The assumption that hardware reliability is safeguarded with post-fabrication testing to detect and isolate fabrication defects from causing impactful data corruption is now more challenging than before. SDC can manifest long after the initial installation of cloud infrastructure; for example, defects in mercurial cores [18] can impact the computed result long after the error has first occurred, making it even harder to localize the error in a fleet of compute nodes or long workloads. Several reasons are contributing to this: the widespread demands from the DNN community to push more processing engines on the same chip, staying on track with Moore’s law with a lower transistor feature size, the increasing interest in cloud decentralized computing, and the movement toward adopting machine learning technologies in safety-critical applications. For these reasons, there is a higher chance that latent defects reach consumers [20]. Mitigation of the impact of faults in functional blocks, such as arithmetic units, is much less developed when compared to dealing with memory faults because of a lack of error correction mechanisms; this is reinforced by the study from [19], which found that permanent faults in functional blocks occur at a higher rate than soft errors. To this end, the cloud providers are in need to better understand the underlying problems and find solutions to deal with SDC [18,19]. A major direction in research to address these issues is to use more accurate models during the premanufacturing verification stages. While several works address transient faults from this perspective, in terms of scale, negligible initiatives focus on permanent faults in processing engines. One possible reason for limited contributions in this niche is due to the challenges faced when scaling frameworks for fault resiliency assessment and mitigation to deal with larger hardware systems and workloads. Therefore, in this work, we are proposing a solution to overcome these challenges by carrying out
fault resiliency assessment early in the design process when it is less expensive to carry out design revisions.

One key point worth articulating is that accurate models of the arithmetic units are only available during the later stages of the development process. For example, circuit models of arithmetic units are not available at RTL [21]. While gate-level models can replace selected RTL modules [22], the RTL simulation speeds [15] can still pose challenges for carrying out fault injection experiments for very large accelerators. One way to address the above concerns is to use accurate models of the hardware components at very high-levels of abstraction, as detailed in this paper, before the RTL description has been developed. This strategy not only accelerates fault injection experiments in arithmetic units but also provides early feedback to system designers about the fault resiliency of their architecture before RTL implementation.

![Figure 1](image-url)

**Figure 1.** The main motivation for the proposed framework is to enable early fault assessment during the design phase of new DNN accelerators. The top of the figure highlights four different hardware development stages during which fault assessment can be performed. The rest of the figure compares them according to usage, accuracy, hardware redesign cost, and simulation speed.

In this work, accurate circuit models of targeted hardware components, i.e., arithmetic units, are utilized to facilitate fault assessment integration within High-Performance Computing (HPC) DNN frameworks, e.g., [23]. This setup provides an abstracted interface for running DNN inference tasks while enabling fault injection configurations across a diverse range of fault models and hardware architectures, as elaborated in Section 5. By increasing
the abstraction level for fault injection experiments, there is a limited spectrum of the
types of faulty behaviors that can be assessed. However, given that embedded memories
and arithmetic units are predominant in DNN accelerators, this focus is justified. While
system-level fault injection experiments in memory have been previously investigated,
the effects of faults within large arrays of processing engines on DNN inference at the
application level remain largely underexplored.

By using precise circuit models for arithmetic units, hardware designers are able to
make decisions and revisions before the full implementation is complete. A key challenge
that needs to be addressed is how to seamlessly incorporate fault injection and modeling
custom hardware accelerator architectures within HPC frameworks, such as PyTorch [23],
which have been optimized for peak application performance. The framework must
also provide a broad spectrum of fault injection capabilities and remain adaptable to
various hardware configurations. To summarize, the main contributions of this study are
outlined below:

• We propose a DNN simulation framework designed to assess the fault resilience
  of hardware accelerators. This framework utilizes data streaming patterns known
  prior to RTL implementation and integrates gate-level accurate arithmetic circuit
  models into our high-level model. The methodology enables designers to evaluate the
  resilience of new DNN accelerators very early during the design space exploration
  phase, providing insights that can guide adjustments before more detailed and time-
  consuming implementations are finalized.

• To validate the generality of our framework, we evaluate the fault resilience of different
  hardware accelerators from the literature for different DNN models. We provide a
  comparative analysis in terms of fault resilience that, to our knowledge, has not been
  reported in the literature for large-scale designs and models.

The rest of the paper is organized as follows. Section 2 outlines the key differences
between our work and recent studies from the literature. Section 3 presents the terminology
and core concepts from the literature needed to elaborate our methodology. Section 5
describes the proposed fault assessment framework’s design stages and structure. The
experimental results, including using the framework with different state-of-the-art DNN
models and hardware accelerators from the literature, are discussed in Section 6. Section 7
concludes this paper.

2. Related Works

DNN fault simulation frameworks need to support various features, such as modeling
the target hardware, mapping DNN operations to hardware, and fault models. Develop-
ing a comprehensive simulation framework with such capabilities poses a challenge due
to the processing demands of state-of-the-art DNN models and the implementation of
accurate hardware models. Such complexities depend on the modeled hardware architec-
ture and the fault scenarios that can be supported by the framework ([10–12,24–31]). For
example, enabling transient faults in memory during simulation may require the usage
of accurate models of the hardware only during the simulation periods when faults are activated ([10,25,31]).

Another type of faults to be considered for hardware resiliency assessment is per-
manent faults in the datapath ([11,12,24]). Unlike transient faults, modeling faults in the
datapath requires more detailed information about the hardware architecture that needs to
be captured in the hardware models. For instance, the mapping of operations to the pro-
cessing engines, data types used, memory footprints, and data-streaming patterns should
be considered when modeling the datapath for fault resiliency assessment. A permanent
fault in one of the inputs to an arithmetic unit will affect different operations and variables
when running the DNN application. This impact is unique to the particular hardware
architecture under study and how it is being used to process the different calculations
during DNN inference.
The main challenge in modeling faults at this level in a comprehensive simulation framework is providing a flexible and configurable hardware modeling tool to allow hardware designers to assess different architectures while, at the same time, maintaining adequate simulation speeds. One option is using a compiled programming language, such as C++, to build a complete and efficient DNN fault resiliency assessment framework ([24]). However, supporting and extending such frameworks with emerging DNN architectures makes these frameworks less favorable due to the extra effort needed to rebuild them. Another approach is to utilize one of the existing HPC DNN frameworks and incorporate the hardware-specific and fault injection features into it ([24,31]).

In order to support faults in arithmetic units, a more detailed level of hardware modeling with various added complexities is required. Faults in the operands of operations of the datapath can be modeled by modifying the data at the output of tensor operations ([24,29]). In contrast, permanent stuck-at faults in arithmetic circuits cannot be generally represented at the boundaries of tensor operations as they alter the output based on both operand values and the fault site within the arithmetic circuit. Therefore, to support a broader range of fault models and hardware accelerator architectures, the hardware designers should have the ability to introduce faults at the level of logic gates and capture the mapping of operations from the DNN model to the arithmetic units within the accelerator.

To model fault injection in arithmetic units using HPC DNN frameworks, it is essential to consider that the hardware models for arithmetic units need to be integrated within the tensor or kernel routines. However, it should be noted, as further elaborated in Section 5, that modifying highly optimized tensor operations in an HPC DNN framework can lead to a substantial decrease in computational performance. In [27], a gate-level, stuck-at fault-injectable operator for a single neuron increases the simulation time by more than 25%. Replacing all the neurons in the DNN with fault-injectable neuron modules, based on the criteria in [27], is expected to significantly increase the simulation time by several orders of magnitude. In [26], to enable acceleration using parallel-computing Graphics Processing Units (GPUs), DNN models of the gates’ logic are used instead of logic operations to model processing engines’ circuitry. Scaling this type of method to larger hardware clusters, DNN models or a higher number of classes may necessitate several iterations of feature-map extraction, training, and updating of the DNN models.

Some previous studies have aimed to balance simulation time and fault injection accuracy by integrating RTL-accurate models in performance DNN frameworks. Verilator [32] can convert RTL descriptions to C++ modules [30]. However, RTL models lack accurate gate-level representations of arithmetic units [25]. Additionally, creating an RTL model of the target architecture is necessary before conducting fault injection experiments, leading to increased development cycle time when design revisions are required. In contrast, this work attempts to bridge the gap between system-level design and gate-level simulation accuracy by using gate-level accurate arithmetic hardware models alongside configurable accelerator architecture models while maintaining manageable simulation speeds.

Figure 2 outlines the key differences between our work and recent studies from the literature, which are categorized by fault injection support, redesign cost, and simulation speed [11,12,25–27]. The work in [25] provides fault assessment for transient faults using RTL models of the hardware with more affordable simulation demands compared to studies supporting permanent faults in the datapath [11,12,24]. Although supporting faults within arithmetic units significantly increases the simulation time [26,27], studying this type of fault is important because they occupy up to 40% of the silicon area [2,3,16] and they can lead to Silent Data Corruption [18,19] due to the lack of error correction mechanisms that are common for protecting systems from memory faults.

Taking the above context into account, we introduce our framework, referred to as Architecture-Level Pre-Register-Transfer-Level Implementation Fault Injection (ALPRI-FI), which operates at a new level of abstraction for carrying out fault injection experiments. This framework allows for the integration of data streaming patterns, known before the RTL implementation, into existing DNN frameworks, such as PyTorch [23], while also
providing the capability to inject faults in gate-level accurate models of the arithmetic circuitry. Although the scope of the fault injection experiments is restricted to arithmetic units for different hardware accelerator architectures, the scale of the experiments, as shown in Section 6, is more comprehensive than what has been reported in the literature. This framework also offers additional flexibility, enabling early-stage assessment of how certain architectural choices will impact reliability and allowing for revisions at an early design stage when they are less costly.

Figure 2. Positioning of our framework, called ALPRI-FI, relative to recent studies [11,12,25–27]. ALPRI-FI offers gate-level accurate fault assessment in arithmetic circuitry during early design stages using an extendable framework that operates at the application level.

3. Background

This section introduces the terminology and core concepts from the literature needed to introduce and elaborate our methodology.

3.1. Notation

The purpose of this section is to introduce the terminology used in the presented work. While the findings and approaches designed in this work are devoted to the fundamental operations for convolution Deep Neural Network (CONV DNN) models, the same methods can be applied to other types of DNN models such as transformers.

\[
O[im][d_h][d_w][c_o] = ACT\left( B[c_o] + \sum_{i_h=0}^{K_h-1} \sum_{i_w=0}^{K_w-1} \sum_{d_c=0}^{D_c-1} I[im][S_T \times d_h + i_h][S_T \times d_w + i_w][d_c] \times W[c_o][i_h][i_w][d_c] \right), \quad H_o = \frac{H_i - K_h + S_T}{S_T}, \quad W_o = \frac{W_i - K_w + S_T}{S_T}, \quad 0 \leq im < N_{im}, \quad 0 \leq d_c < D_{in}, \quad 0 \leq c_o < D_{out}, \quad 0 \leq d_h < H_o, \quad 0 \leq d_w < W_o
\] (1)

Equation (1) evaluates the Output Feature Map (OFmap) \(O\) using bias \(B\), Input Feature Map (IFmap) \(I\), and weights \(W\). OFmaps \(O\) is a four-dimensional (4D) structure, where the outer two dimensions of \(O\) scan different images in the patch pool and different input channels for each image, respectively. The inner two dimensions scan the OFmap channel height and width, respectively. Zero padding is assumed for simplicity. Table 1 describes the structural parameters, and Figure 3 illustrates the operation. It is important to note that although the discussion in this work focuses on convolutional layers (CONV), the examples
capture the intuition behind the operations performed in other DNN model types, such as transformers and MLP models.

Table 1. Convolution parameters of Equation (1).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_{out}$</td>
<td>Number of OFmap channels</td>
</tr>
<tr>
<td>$D_{in}$</td>
<td>Number of IFmap channels</td>
</tr>
<tr>
<td>$K_h/K_w$</td>
<td>Convolution kernel height and weight</td>
</tr>
<tr>
<td>$H_i/W_i$</td>
<td>IFmap height and weight</td>
</tr>
<tr>
<td>$H_o/W_o$</td>
<td>OFmap height and weight</td>
</tr>
<tr>
<td>$S_T$</td>
<td>Scanning stride (default = 1)</td>
</tr>
<tr>
<td>$N_{im}$</td>
<td>Number of images/data samples</td>
</tr>
</tbody>
</table>

Figure 3. Paper notation: convolution operation for IFmap $I[H_i][W_i][D_{in}]$ using $D_{out}$ kernels $W[D_{out}][K_h][K_w][D_{in}]$.

3.2. DNN Acceleration

Various types of hardware accelerators have been introduced for DNN acceleration [2–4,16,33]. In the inference mode, data streaming through DNN layers occurs sequentially with the output of one layer calculated before activating the next layer. However, the calculations within each layer can often be performed in parallel using multiple data streams. Furthermore, the majority of DNN processing relies on Multiplication-and-Accumulation (MAC) operations. These two characteristics of DNN processing serve as the primary motivation for designing hardware accelerators. Diverse hardware architectures are precisely optimized for specific tasks like convolution acceleration to enhance the computational efficiency for targeted applications. Conversely, some hardware designers advocate for versatile, general-purpose DNN accelerators, which are engineered to support a wide array of DNN models.

3.3. Accelerating DNN Convolution

Most of the computational time in convolutional DNNs is devoted to processing CONV layers [34]. Consequently, custom accelerators have undergone extensive optimization for CONV layers, particularly in terms of data reuse, which can be classified into three main types:

- **Weight Stationary**: Weight-stationary accelerators exploit weight reuse by storing the kernel weights in faster and smaller memories allocated for each PE. Examples of weight-stationary accelerators include refs. [2,35]. Equation (1) shows that processing each kernel necessitates scanning the IFmap along two or more dimensions (in this example, two dimensions). By reusing kernel weights across different scanning positions, the number of weight reloads from the main memory is reduced, resulting in savings in both energy and time.

- **Output Stationary**: In accelerators of this type, PEs are allocated to perform operations related to a specific number of outputs. The intermediate partial sums are stored in the PE memory and are updated with additional partial sums as more IFmaps
and weights are streamed through the hardware. Output partial sum stationary accelerators leverage input and partial sum data reuse [3,36,37].

- **Weight and Input Reuse:** In addition to leveraging weight reuse, input reuse is further employed to minimize memory reads, enhancing the performance of these accelerators. Convolution Primitives, as implemented by [4], is enabled by processing IFmap and kernel weights utilizing distinct sets of PE units, resulting in the aggregation of partial sums to generate the desired OFmap with less inter-core memory transactions.

As elaborated in the next section, a key novelty of our methodology is the integration of the data streaming pattern specification into our high-level fault assessment framework. This allows for the evaluation of the fault resiliency of the accelerator at an early stage in the design space exploration phase before the completion of the full implementation of the accelerator.

3.4. General DNN Frameworks

General DNN frameworks are designed to efficiently support various types of DNN layers. One of the reasons is that the processing of each layer can be simplified to a series of tensor operations. For instance, the processing of linear layers can be transformed into a single matrix multiplication task, where the first dimension of the activation matrix corresponds to the batch size and the number of output nodes equals the number of columns in the weights matrix. Similarly, the processing of a convolutional layer can be converted to matrix multiplication by applying convolution unfolding techniques.

An example illustrating the unfolding of CONV layer processing to matrix multiplication is presented in Figure 4. Matrix $A$ represents the input activation for a batch of images with a size of $N_{im}$ images. Each group of $H_0 \times W_0$ rows in matrix $A$ corresponds to one unfolded input image, where each row contains the IFmap pixels corresponding to one OFmap sample. It is assumed that the deepest dimension is the number of input channels, which is denoted as $D_{in}$. The weight matrices are represented by matrix $B$, where each column corresponds to one kernel. The output partial product before activation (assuming activation occurs separately in the subsequent stage) is represented by matrix $C$.

\[
\begin{align*}
& A \rightarrow B \\
& \text{Layer Output} = C \\
& D_{out} = \begin{bmatrix}
H_0 \times W_0 & D_{in}
\end{bmatrix}
\end{align*}
\]

**Figure 4.** Converting CONV layer processing to matrix multiplication, known as convolution unfolding. One kernel scan position ($d_w, d_h$) corresponds to one row of $A$. The lower part of the figure shows rows from matrix $A$ related to one input image, which, when multiplied to weights matrix $B$, the result corresponds to the $O[im][d_h][c_o]$, before activation, such that $im$ is the image index as per Equation (1).
When presenting our main contribution in Section 5, we will utilize the same terminology and design abstraction outlined in Figure 4. This choice is motivated by the widespread use of this level of abstraction in software frameworks for DNN implementation, such as PyTorch [23]. More importantly, this level of abstraction is not dependent on any specific type of hardware accelerator employed for deploying DNN models. Therefore, by developing a fault injection framework at this level of abstraction, it is possible to evaluate the potential impact of hardware faults on new hardware acceleration proposals early during design space exploration, i.e., during the pre-implementation phase when the design revisions are more easily manageable.

4. Motivation

In Figure 1, we show four different fault resiliency assessment options in four different environments during the design and implementation process.

Using silicon prototypes to emulate hardware faults (silicon environment) can help assess the hardware fault resiliency in a fast operating environment. However, this requires the insertion of hardware structures that facilitate the fault injection, and the results from such experiments can be used for reporting fault resiliency rather than making important decisions for design revisions. A gate-level environment offers an accurate circuit model of the hardware for fault injection experiments. Nonetheless, this model is only available very late in the design and implementation stages, and the simulation speed is known to be significantly slow, thus making the assessment of DNN application performance at this stage practically infeasible.

While RTL gives more accurate description of the hardware, the scope of the model is limited to data transfers, and therefore, it does not offer natively accurate circuit models to be used for fault injection. For example, the impact of faults that are internal to the arithmetic units cannot be evaluated unless more accurate circuit models are introduced.

Driven by the computational speed constraints of current fault injection methodologies in large arrays of arithmetic units within DNN accelerators, we propose adapting existing DNN frameworks, such as PyTorch [23], to facilitate fault assessment experiments at the application level.

To emphasize the reasons behind our choice, in Figure 5, we show an example of a MAC operation (hardware unit) in three different simulation environments: system level (System-Env), RTL level (RTL-Env) and gate level (Gate-Env). The figure shows an example of multiplying activation $X$ by weight $Y$. The result partial product $Z$ is added to the DNN node accumulator $acc$. It is assumed that the hardware is faulty with fault $Z[0]/0$ (Stuck-at-0). In the Gate-Env, a netlist serves as the design model, enabling direct fault injections. However, the simulation times in Gate-Env tend to be excessively long because it involves processing the entire netlist. This level of detail is unnecessary when the goal is specifically to carry out fault injections in the arithmetic units, suggesting a need for more targeted fault injection strategies that balance detail with efficiency.

The System-Env and RTL-Env environments provide faster simulation speeds compared to Gate-Env with System-Env achieving superior performance. This advantage in System-Env is largely due to its reliance on higher levels of abstraction, which are facilitated by environments like SystemC [38]. Nevertheless, both environments lack accurate circuit models needed for performing fault injection in arithmetic units, necessitating certain design updates to accommodate this requirement. In particular, RTL-Env can incur higher costs, as modifications to the arithmetic units often necessitate updates across other components within the RTL model. For instance, changing the number of bits of a hardware multiplier in RTL-Env requires updates to all associated memory and connections. Additionally, fault assessment using RTL-Env is only viable once the RTL description of the accelerator is fully developed, which delays the possibility of early design feedback.

In contrast to RTL-Env, System-Env leverages High-Performance Computing frameworks designed to accelerate algorithm design and model development. Utilizing widely adopted DNN frameworks such as PyTorch [23] and TensorFlow [39], running inference
jobs in System-Env offers several advantages over RTL-Env, which can be summarized as follows:

- The system environment is readily accessible during the initial phases of development, allowing for early development and validation.
- It can be used to provide early feedback on fault resiliency, which can reduce the cost and time associated with redesigns.
- Offers greater reconfigurability, utilizing accurate models only when necessary for the types of components that are assessed, e.g., arithmetic units.
- Enables experimentation at a wider scale, supporting the assessment of larger DNN models and hardware configurations.
- Facilitates keeping pace with emerging DNN models by integrating with widely used, open-source DNN frameworks like PyTorch [23] and TensorFlow [39].

Figure 5. Comparing fault injection mechanisms using Multiply–Accumulate (MAC) hardware models in System-Env, RTL-Env and Gates-Env simulation environments. RTL functional description of the target hardware does not offer detailed circuit models, limiting the support of fault resiliency assessment. The colours of the accelerator model (square) and the multiplier model (circle) indicate the simulation speed, with red presenting the longest simulation time.

5. Fault Assessment Framework

This section describes the proposed fault assessment framework’s design stages and structure, including a novel method used to model the operation-PE mapping, which is consistent with the data streaming patterns from the evaluated HW accelerator that leverages the organization of data structures in state-of-the-art HPC-based frameworks for DNNs, e.g., PyTorch [23]. Note, it is worth recalling from Figure 1 that a key benefit of developing a high-level fault assessment framework for DNN accelerators is to employ it not only after an accelerator has been implemented but also during the early design space exploration phase when the cost of design revisions is low.
5.1. Framework Requirements

The main objective is to build a framework capable of performing logic fault assessment of DNN hardware accelerators. This can be translated into a series of three key requirements, R1, R2, and R3, as shown in Table 2.

The design of the framework is aimed at meeting the above requirements. To support requirements R1 and R3, a high-performance DNN framework (PyTorch) is chosen as the base framework. Note, however, that there is conceptually no reason the methods presented in this work cannot be applied to other base DNN frameworks. The main contributions of this work are providing key support to conduct fault injection in a hardware multiplier model (R2-a) and modeling operation-PE mapping (R2-b), as per the data streaming patterns defined at the architectural level, within an HPC DNN framework. These two modeling features are anticipated to be key characteristics of the hardware utilized for DNN acceleration, and both introduce significant challenges upon their integration within high-performance DNN frameworks, especially when scaled up.

Table 2. Framework requirements.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>Flexible to simulate different DNN networks, able to perform both post and pre-implementation fault assessment, and upgradable to support future DNN network types and HW features.</td>
</tr>
<tr>
<td>R2</td>
<td>Support different hardware modeling features: a Modeling the arithmetic circuitry of the HW design b Modeling the mapping of operations to processing engines (PEs) based on data streaming patterns.</td>
</tr>
<tr>
<td>R3</td>
<td>Consume manageable computational resources.</td>
</tr>
</tbody>
</table>

Before discussing the ALPRI-FI structure, it is essential to highlight the challenges of modeling fault injection for a large DNN model on an HW accelerator and quantify the performance overhead the work from this paper addresses. To achieve this, Table 3 summarizes how adding new modeling features gradually introduces a performance penalty, while at the same time, it shows how, by optimizing the implementation of these added features, some of the shortcomings can be mitigated. These optimizations, which are detailed in the following sections, are the main contributing factors that facilitate fault assessment for low-level netlist models of arithmetic units for HW accelerators, executing large DNN models on a scale not reported previously in the literature.

Table 3. Framework performance at three different development stages. At each stage, a framework’s requirement is designed and verified (a), and then the implementation is further optimized (b). The results are calculated using FBGEMM ([40]) library benchmarking tests of matrix multiplication.

<table>
<thead>
<tr>
<th>Stage</th>
<th>Feature</th>
<th>Fault-Injectable Multiplier Model (R2-a)</th>
<th>Operation-PE Mapping (R2-b)</th>
<th>Run Time Compared to Baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>(a) Native Matrix Multiplication (MM) routine</td>
<td>No</td>
<td>No</td>
<td>56x</td>
</tr>
<tr>
<td></td>
<td>(b) FBGEMM Machine-Specific MM</td>
<td>No</td>
<td>No</td>
<td>1 (Baseline)</td>
</tr>
<tr>
<td>S2</td>
<td>(a) Add Initial Fault-Injectable Multiplier Model</td>
<td>Yes</td>
<td>No</td>
<td>2700x</td>
</tr>
<tr>
<td></td>
<td>(b) Optimized Fault-Injectable Multiplier Model</td>
<td>Yes</td>
<td>No</td>
<td>900x</td>
</tr>
<tr>
<td>S3</td>
<td>(a) Add Basic Operation-PE mapping</td>
<td>Yes</td>
<td>Yes</td>
<td>1400x</td>
</tr>
<tr>
<td></td>
<td>(b) Operation-PE mapping with the new method</td>
<td>Yes</td>
<td>Yes</td>
<td>1100x</td>
</tr>
</tbody>
</table>

The framework development can be divided into stages S1, S2 and S3 as follows.
5.1.1. (S1) Selection of Base DNN Framework for Injecting Faults to Arithmetic Logic

Simulating logic faults in arithmetic circuits requires developing circuit models of the arithmetic units and the injection of faults to the circuit nodes of the model. For example, to model logic faults in HW multipliers, it is required to replace multiplication operations in DNN inference jobs with the circuit model of the multiplier. One option is to build a custom DNN framework using the HW multiplier model. However, this approach does not meet the requirements R1 and R3. Another option is integrating the hardware models into a high-performance DNN framework like PyTorch. For example, benchmarking PyTorch on large matrix multiplication operations, it is found to be 56x faster compared to a native implementation using loops (stage S1 in Table 3), which justifies the integration of HW models into PyTorch’s high-performance DNN framework.

5.1.2. (S2) Using Fault Injectable Arithmetic Unit Model

In order to carry out fault resiliency assessment experiments, it is required to replace the multiplication operations in DNN inference with a netlist multiplier model that supports fault injection, e.g., forcing the target node to 1 or 0. Exploratory experiments have established that the initial implementation of requirement R2-a—specifically, substituting the multiplication operator with a fault-injectable hardware multiplier netlist model—resulted in a substantial increase in simulation time, exceeding 2700x-equivalent compared to the base framework implementation. For example, this increases the inference job simulation time of the ImageNet [41] validation test set using ResNet50 [42] from approximately 220 min (on a reference 6-core desktop machine) to an estimate of 412 days, which is very large. To mitigate the excessive slowdown in runtime, and acknowledging that integrating feature R2-b would further exacerbate this issue, we decided to investigate basic optimization techniques (specific technical details will be provided in the following subsections) targeting general simulation platforms. The simulation speed is further boosted by using parallel computing clusters. One reason for this decision is to keep the framework portable, i.e., no machine-specific code is introduced; another reason is to offer a plug-and-play experience for the accelerator design team to try different arithmetic structures. The basic optimizations reduced the simulation time from 2700x-equivalent to 900x-equivalent compared to baseline, as shown in stage S2 of Table 3.

5.1.3. (S3) Operation-PE Mapping Feature

Moving to requirement (R2-b), the first exploration used a list that maps the target operation to the assigned PE, as clarified later in this section. This implementation increased the simulation time by 50% (1400x equivalent in stage S3-a compared to 900x equivalent in stage S2-b of Table 3) for small activation and weight matrices. More importantly, the memory requirements for the initial operation-PE design are substantial such that assessing large DNN network layers was not feasible on one machine. By introducing a novel method, discussed in detail in this section, the operation-PE mapping information is stored and processed in a more compact representation. This approach to supporting different data streaming patterns by modeling different operation-PE mappings offered lower memory requirements proportional to the data size rather than the number of operations and a simulation time overhead of only 20% over not supporting this critical feature needed for high-level fault assessment (1100x equivalent compared to the baseline as in stage S3-b of Table 3).

Having motivated the need for and summarized the requirements for an HPC-based framework for fault assessment during the early design phase of DNN accelerators, we introduce the general framework structure in the following subsection.

5.2. General Framework Structure

Figure 6 depicts the general diagram of the framework. It is divided into three main parts: (1) top-level application, (2) DNN framework, and (3) General Basic Linear Algebra Subroutine (BLAS) back-end. The processing of one DL layer for a batch of inputs
is converted to one matrix multiplication job and assigned to a BLAS General Matrix Multiplication (GEMM) routine. Without loss of generality, in this framework, we use PyTorch [23] as the base framework. Currently, PyTorch uses Facebook General Matrix Multiplication (FBGEMM) [40] as the main GEMM for processing inference jobs on 8-bit quantized DNN models.

**Figure 6.** ALPRI-FI framework structure: The left side describes the breakdown of processing activation matrix $A$ and weights matrix $B$ to produce pre-activation layer output $C$ through software layers consisting of DNN application, DNN framework, GEMM subroutine, and low-level MM kernel. Operation-PE mapping description in $IA$ and $IB$ matrices is efficiently propagated from the top-level application to the low-level kernel (Conf Block(im, ik, in)) as shown on the right side. The MM kernel uses the mapping information to model the operation-PE mapping of the target hardware accelerator. The thickened black arrows indicate the transformation of data structures from an upper to a lower software layer.

In Figure 6, $A$, $B$, and $C$ are the activation, weight, and partial output matrices, respectively. The conversion process includes convolution unfolding of the convolution layers and converting the processing of convolution (and linear) layers to matrix multiplication. It also features tiling the input matrices to several smaller matrices with fixed dimensions. The GEMM job is broken down into calls to several machine-coded matrix multiplication.
kernel routines. The data are re-ordered in the memory to increase data cache hits, and the efficiency of data streaming to the simulation hardware CPU cores.

The base framework can be upgraded to support modeling of the various blocks within the DNN accelerator being assessed. One key feature is modeling the behavior of arithmetic units. While the results from this study focus on logic faults in integer multipliers, the framework is expandable to investigate adders, bit shifters, etc., that may exist in the processing units of a DNN accelerator. As detailed in the next subsection, the multiplier model is based on Baugh and Wooley [43] signed multiplier with added fault injection support. While one signed multiplier circuit model is used in this study, the generic framework can be configured to other multiplier circuit architectures. Another key feature in assessing the impact of logic faults on the DNN accelerator is modeling the operation-PE mapping. In this work, as discussed in detail in the following subsections, we introduce an effective way to manage fault assessments by using two new matrix-like data structures, which are referred to as $IA$ and $IB$. As shown in Figure 6, $IA/IB$ are defined at the top level (application) and are used at the low level (kernel). Supporting these two key fault injection features, the built-in kernel is replaced with a custom kernel that uses the operation-PE mapping information taken from $IA/IB$ and the fault-injectable multiplier model.

After introducing the multiplier model (for the sake of completeness when presenting results) in the following subsections, we will discuss how the introduction of the $IA/IB$ mapping works efficiently with different data streaming patterns and how the implementation integrates smoothly with the DNN/GEMM performance layers.

5.3. Multiplier Model

While the main contribution of our work is concerned with how to assess the impact of faults in arithmetic units based on an injection mechanism at the level of vector operations (as detailed in the following subsections), for the sake of technical completeness, we first elaborate on the details needed to be accounted for to model the arithmetic circuit’s faulty behavior at a higher level of abstraction. A multiplier model is illustrated in Figure 7, where a 9-bit signed multiplier is based on Ripple Carry Adders (RCAs) [43].

![Multiplier Model Diagram]

Figure 7. A 9-bit signed multiplier model based on Baugh–Wooly model [43].
As highlighted in Table 3, the initial benchmarking of the multiplier model showed a drastic increase in runtime. The performance of the model was enhanced by applying several techniques summarized below. One method to accelerate the evaluation of the faulty output for a pair of inputs and a specific fault is by using look-up tables (LUTs) for the top adder rows (in Figure 7) preceding the row where the fault is being injected. Another technique is implementing multi-bit fault masks to remove the conditional statements for each fault injected into the model’s nodes. For each logic operation in the model, one pair of pre-evaluated masks (AND and OR masks) is used to inject SA-0 or SA-1 faults. Finally, a header-only implementation is also used (in contrast to source file implementation, header-only implementation implement software logic in header files to allow better compiler optimization at later application build stages), which overall leads to a runtime that is three times faster compared to the non-optimized faulty model (900x equivalent vs. 2700x equivalent compared to baseline, as shown in stage S2 of Table 3).

5.4. (Baseline) Operation-PE Mapping Using Mapping Lists

The data streaming patterns vary from one DNN accelerator to another; these patterns define how operations map to PE units, which is called operation-PE mapping. To motivate a new technique for managing operation-PE mapping, we will refer to the example of multiplying matrix $A$ of sizes $(2, 4)$ by matrix $B$ of size $(4, 3)$ shown in Figure 8.

Figure 8. Operation-PE mapping example for matrix multiplication of $A$ of size $(2, 4)$ and $B$ of size $(4, 3)$ resulting in matrix $C$ of size $(2, 3)$. Referring to Figure 4, these dimensions can be the result of convolution parameters $K_w = K_h = 2$, $D_{in} = 1$, and $D_{out} = 3$. As described in Section 3, the PE grid must be pre-configured with the weights in weights-stationary systolic arrays. It is also required that the streaming of activation into the grid is synchronized with the accumulation and the streaming of partial products from the grid. Figure 9 shows the weights-stationary configuration using a $4 \times 3$ systolic-arrays grid. Assuming a fully pipelined implementation with a PE latency of 1 clock cycle per PE, $C_{00}$ and $C_{12}$ are ready at the output of $M[3][0]$ and $M[3][2]$ PE units at the 4th and 7th cycle, respectively. The mapping list can be accessed using the iterators $k$, $n$, and $m$ used to access $A/B$. The multiplication operation corresponding to $A[m][k] \times B[k][n]$ is mapped to PE $(PE\_ID[m][k][n])$ as described in Equation (2), where $c_t$ is the number of grid columns ($c_t = 3$ in this example).

$$PE\_ID[m][k][n] = MP\_listIB[k][n + m \times c_t]$$

It is important to note that the mapping list size is larger than the input matrices added together. This imposes a challenge for larger DNN layers requiring more memory storage and bandwidth. In addition, the memory footprint of the list is different from either $A$ or $B$, which increases the likelihood of cache misses. Consequently, a more effective operation-PE
mapping method is needed to assess the impact of logic faults when a data streaming pattern is reassessed during the early stages of designing a new DNN accelerator.

![Diagram of matrix multiplication]

Figure 9. Streaming pattern for matrix multiplication from Figure 8 in a $4 \times 3$ systolic array and the operation-PE mapping description using mapping list $\text{MP}_{\text{list}}[M][M^*N]$, where $M$, $K$, and $N$ are 2, 4 and 3, respectively.

5.5. A New Representation for Operation-PE Mapping

ALPRI-FI stores the operation-PE mapping information in two matrix-like data structures. Introducing $IA$ and $IB$ matrices, the designer can define the number of PEs in the cluster and the operation-PE mapping with minimal impact on the system performance. The contents of $IA$ and $IB$ define the mapping of operations to the cluster PE units. The sizes of $IA$ and $IB$ match the sizes of the activation matrix $A$ and the weights matrix $B$, respectively. $IA$ and $IB$ are packed and tiled, resulting in a memory footprint similar to $A$ and $B$, respectively. Each pair of elements taken from $IA$ and $IB$ is processed by an operation-PE mapper function to calculate the index of the PE unit. Figure 6 highlights how a pair of $(IA_{kj}, IB_{kj})$, taken from $IA$ and $IB$, is translated to the ID of the cluster PE $PE_{id}$. By defining the multiplier model and the $IA/IB$ pair, the designers of the hardware accelerator can define the number of PE units and the operation-PE mapping of the cluster under study. In the following subsections, we discuss how the proposed method can be used to describe in detail the operation mapping for a variety of DNN accelerators.

5.6. Supporting Data Streams for Generalized Matrix Multiplication Acceleration

In this section, we introduce a more effective and efficient method to describe the operation-PE mapping for a general matrix multiplication accelerator to be able to assess the impact of faults on logic circuits, such as multipliers. As discussed in Section 3, the processing of different DNN layers can be converted to matrix multiplications.

Using the proposed method to describe the operation mapping of weights-stationary data streaming in systolic arrays, Figure 10 shows general operation-PE mapping configuration for a TPU-like accelerator [2] of size $rt \times ct$. Each PE is associated with one weight. $IA$ is redundant, and $IB$ only is needed to describe the operation-PE mapping. As shown in the figure, $IB$ can be defined as tiles of two-dimensional (2-D) blocks of size $rt \times ct$. The contents of $IB$ are simply the ID of the PE units to be used with the operations associated with the corresponding weights.
Figure 11 shows how IA and IB can be configured to describe the operation-PE mapping for the example from Figure 8, which is processed by weights-stationary systolic-arrays. Since only IB is needed to define the operations mapping, the multiplication operation corresponding to iterators \( m, k, \) and \( n \) is processed by a PE identified as outlined below:

\[
PE\_ID[m][k][n] = IB[k][n]
\]  

Figure 10. Operation-PE mapping for a TPU cluster (weights-stationary systolic arrays) of size \( rt \times ct \) using IB configuration matrix. Each block of \( rt \times ct \) provides mapping information of one cluster configuration.

Figure 11. Operation-PE mapping description for weights-stationary streaming patterns through a grid of \( 4 \times 3 \) systolic arrays using IA and IB matrices for the example in Figure 8.

A natural question is whether the IA/IB-type matrices can be used for different data streaming patterns in the same cluster. Figure 12 depicts partial results-stationary data streaming into the arrays and how IA/IB can describe the mapping. In partial results-
stationary arrays, rows of activation matrix $A$ perform a dot-product with columns of weights matrix $B$. The same PE performs the accumulation of partial results for the same output pixel. After the final results are ready, the output is streamed out of the processing grid. One way to describe the mapping is to store the row index and column index of the target PE in $IA$ and $IB$, respectively, as shown in the figure. In this case, the operation-PE mapping depends on both the activation and the weights. Consequently, both $IA$ and $IB$ are needed to describe this mapping such that the multiplication operation corresponding to $m$, $k$, and $n$ is mapped to $\text{PE}_{\text{ID}}[m][k][n]$, which is defined as

$$\text{PE}_{\text{ID}}[m][k][n] = IA[m][k] \times ct + IB[k][n]$$

(4)

Figure 12. Operation-PE mapping description for partial results-stationary streaming patterns through a grid of $4 \times 3$ systolic arrays using $IA$ and $IB$ operation-PE mapping matrices for the example in Figure 8. $IA$ and $IB$ hold the row and column ID of the corresponding PE, respectively.

It is worth mentioning that changing the mapping from weights-stationary to partial product-stationary reduces the utilization of PE units. For the latter, each PE presents a higher contribution toward the result by calculating four multiplications per PE compared to only two for the former. This utilization will affect the susceptibility to logic faults for different DNN workloads, leading to different PE utilizations for the respective accelerator. Investigating this relationship between PE utilization and hardware fault resilience within a DNN environment constitutes a key feature of our framework, as elaborated in the results section.

5.7. Supporting Data Streams for Specialized DNN Accelerators

This subsection shows how our method can be adapted to capture custom mappings for specialized DNN accelerators, such as convolution accelerators whose mapping depends on convolution parameters. As an illustrative example, we will discuss the mapping of convolution operations to the Eyeriss [4] convolution accelerator.

Eyeriss [4] consists of a grid of $r_e \times c_e$ PE units, where $r_e$ and $c_e$ are the numbers of grid rows and columns, respectively ($r_e = 12$ and $c_e = 14$ in [4]). Figure 13 shows the kernel, IFmap, and OFmap data streaming patterns. The rows of the kernels, IFmap, and OFmap are streamed horizontally, diagonally, and vertically, respectively. While the grid size and data streaming can be customized, we will show how $IA/IB$ can be configured to describe the operation-PE mapping according to the specifications from [4].
Before drawing a generalized format for $IA/IB$ for Eyeriss, it is important to illustrate how structuring the operation-PE mapping in 2-D matrix format can be used to describe a multi-dimensional tensor operation, e.g., convolution between activation in $A$ and kernels in $B$. In a similar manner as convolution unfolding described in Section 3, the operation-PE mapping of convolution jobs can be described by 2-D matrices $IA/IB$. To help with the illustration, we show $IA/IB$ for a 2-D convolution job with an IFmap with $(W, H, D_{in}) = (3, 2, 1)$, 3 kernels with $(k_w, k_h, S_T, N_{im}, D_{out}) = (2, 2, 1, 1, 3)$ and OFmap with $(H_o, W_o) = (1, 2)$. If the same convolution unfolding rules described in Section 3 are followed, the convolution job unfolds to a matrix multiplication job with a format similar to the example in Figure 8. Hence, the same annotation is used.

Figure 13 shows a configuration of $IA$ and $IB$ describing the operation-PE mapping of a general convolution job streamed to Eyeriss. In this format, it is assumed that the Eyeriss grid is evaluating the OFmap for only one input channel and one output kernel at a time. As noted from the figure, this mapping results in the columns of each of $IA$ and $IB$ being identical. This means further memory savings if one column is used to define each matrix.

![Diagram](image)

**Figure 13.** Eyeriss ([4]) specialized convolution accelerator is a PE grid of size $r_c \times c_c$. In Eyeriss, kernel rows are streamed horizontally while IFmap rows are fed diagonally. The output partial-sums are captured vertically.

In a similar manner as convolution unfolding described in Section 3, the operation-PE mapping of convolution jobs can be described by 2-D matrices $IA/IB$. To help with the illustration, we show $IA/IB$ for a 2-D convolution job with an IFmap with $(W, H, D_{in}) = (3, 2, 1)$, 3 kernels with $(k_w, k_h, S_T, N_{im}, D_{out}) = (2, 2, 1, 1, 3)$ and OFmap with $(H_o, W_o) = (1, 2)$. If the same convolution unfolding rules described in Section 3 are followed, the convolution job unfolds to a matrix multiplication job with a format similar to the example in Figure 8. Hence, the same annotation is used.

Figure 14 is sectioned into two rows, illustrating the segments of the data used to produce one value of an OFmap on different views, as shown from the left: (i) the convolution kernel scanning diagram, (ii) activation matrix $A$ and kernel matrix $B$, (iii) Eyeriss data streaming pattern to the cluster, and (iv) the corresponding operation-PE mapping in the $IA/IB$ matrix format. The processing of the first kernel is shown. There are two horizontal kernel scanning points ($W_o = 2$) and one vertical kernel scanning point ($H_o = 1$). Due to the nature of the convolution unfolding, some entries on $A$ map to the same IFmap value (e.g., $a_{10} = a_{01}$).

On the first row from Figure 14, selecting the first row of each of $A$ and $IA$ corresponds to the kernel’s first horizontal scanning position. The next row of $A$ and $IA$ corresponds to the second kernel horizontal kernel scanning position. Only two PE units are used from the cluster since there are two kernel rows and one vertical scanning point ($H_o = 1$). Configuring $IA/IB$ such that $IA$ holds the cluster column index and $IB$ holds the cluster row index is sufficient to describe operations mapping to the cluster for each pair of values from $A$ and $B$. Equation (5) can be used to calculate the PE index $PE_{ID}$ for each pair taken from $IA$ and $IB$.

$$PE_{ID}[m][k][n] = IA[m][k] + c_c \times IB[k][n]$$

(5)

Figure 15 shows a configuration of $IA$ and $IB$ describing the operation-PE mapping of a general convolution job streamed to Eyeriss. In this format, it is assumed that the Eyeriss grid is evaluating the OFmap for only one input channel and one output kernel at a time. As noted from the figure, this mapping results in the columns of each of $IA$ and $IB$ being identical. This means further memory savings if one column is used to define each matrix.
5.8. Contrasting the New Methodology to the Baseline

To assess the impact of logic faults on DNN accelerators early during the implementation phase, one has to capture the relation between arithmetic operations in the DNN model executed on the accelerator and accelerator’s processing engines (PEs). Describing the operation-PE mapping for the example in Figure 8 highlights a few key insights of the proposed methodology compared to mapping lists.
One key feature is providing a compact representation of operation-PE mapping. The size of the mapping information in the proposed method matches the size of the inputs, unlike mapping lists (described in Section 5.4), in which the mapping information matches the number of operations. In the example from Figure 8, it is required to perform 24 operations using 24 different operands from $A$ of size $(2, 4)$ and $B$ of size $(4, 3)$ to evaluate $C$. For larger $A$ and $B$ values, the number of operations is generally higher than the combined number of samples and weights. The reduction in memory facilitated by IA and IB is defined by $M_R$, where $(M, K), (K, N)$ are the dimensions of $A$ and $B$, respectively, as follows:

$$M_R = 1 - \frac{MK + KN}{MKN}$$  \hspace{0.5cm} (6)

For example, for Resnet50 [42], the size of the CONV layer 20 corresponds to a kernel of size $K_h = K_w = 3$, output channels of size $D_{out} = 128$, output channel dimensions of size $H_o = W_o = 28$ and input channels $D_{in} = 128$. For a patch of $N_{im} = 100$ images, the memory consumption needed for the reduction in memory consumption using the proposed method is $M_R = 99.2\%$ compared to mapping lists (describing operation-PE mapping using IA and IB requires less than 1.76 GB of memory, whereas utilizing a mapping list for operation-PE mapping will need 225 GB of memory for this layer only). This reduction is achieved by breaking down the operation mapping information into two parts: activation-dependent, captured in IA, and weights-dependent, captured in IB. These savings in memory storage and bandwidth made it feasible to fulfill the requirements for completing an inference job supporting configurable operation-PE mapping on one simulation machine.

Another key feature of using this methodology is related to runtime. Pre-processing IA and IB is fused with the pre-processing of $A$ and $B$, respectively. This results in leveraging the same platform-specific optimization routines. It has been observed that it is faster to pre-process $A$ and IA together in the same pre-processing routine rather than processing each separately. One reason for this is using the same memory access iterators and similar memory access patterns. This efficient design and implementation of the IA/IB PE-operation mapping technique consumes as little as an extra 20% more processing compared to the fault-injectable kernel without this feature, which is critical when exploring the impact of hardware faults on new accelerators.

To summarize, in addition to operation-PE mapping support, the framework features different fault injection capabilities, allowing to run versatile types of experiments. Table 4 summarizes the fault injection support by the framework that will be assessed in Section 6.

**Table 4.** Fault injection features supported by the framework.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNN Models</td>
<td>Support configurable DNN model</td>
</tr>
<tr>
<td>HW Accelerator</td>
<td>Support modeling of custom HW accelerator</td>
</tr>
<tr>
<td>Multiplier Arch</td>
<td>Support customized multiplier circuit design</td>
</tr>
<tr>
<td>Fault Type</td>
<td>Stuck at 0/1 (SA 0/1), fixed or randomized.</td>
</tr>
<tr>
<td>Fault Site</td>
<td>Node selection in multiplier netlist</td>
</tr>
<tr>
<td>Injection Rate</td>
<td>Number of faulty PEs with respect to the total number of PE units</td>
</tr>
<tr>
<td>DNN Model Fault Region</td>
<td>Selection of specific DNN model layer</td>
</tr>
<tr>
<td>Cluster Fault Location</td>
<td>Selection of randomized or specific PE set</td>
</tr>
</tbody>
</table>
6. Experiments

This section shows the benefits of the framework described in this paper to assess the resiliency of a hardware accelerator running DNN models against logic faults in arithmetic units. As highlighted in Figure 1, the main motivation for the framework is to enable early fault assessment during the design phase of large-scale DNN accelerators.

In each of the following experiments, the changes in inference accuracy are analyzed for a different set of parameters. Each experiment studies the impact of a subset of parameters on the application performance. This is performed by randomizing all the other remaining parameters and plotting the average. By calculating the average, the effect of the unwanted parameters is removed (additional averaging jobs were performed to enable clearer data visualization). Different pre-trained DNN networks based on refs. [44,45] are used in the analysis. The framework is implemented in C++ and is integrated into a custom-build of PyTorch C++ open-source library in a Linux environment. The verification was completed in three stages: (i) multiplier model verification in Hardware Description Language (HDL) and C++ environment; (ii) matrix multiplication kernel verification and benchmarking using FBGEMM verification and benchmarking library tests on a local machine (6-core Intel Xeon CPU E5-2620 v3 with 48GB RAM); (iii) DNN inference performance verification using custom-built PyTorch 1.11 on Niagara compute cluster (operated by SciNet [46]). Niagara enables large parallel jobs on 1040 cores or more clustered in nodes of 40 Intel “Skylake” cores and 202 GB of RAM. While collecting the results in the third stage, each simulation point (e.g., for a specific fault injection rate) is repeated multiple times (30–500), and the average is plotted.

The results from Sections 6.1–6.3 assess the impact of different types of faults, e.g., stuck-at-0 (SA-0) or stuck-at-1 (SA-1) or critical nets in multipliers on the prediction accuracy of the DNN models whose arithmetic runs on such faulty multipliers. The first set of experiments does not account for operation-PE mapping. The results from Section 6.4 demonstrate how data streaming patterns used by a specific HW accelerator (as captured by operation-PE mapping) impact the prediction accuracy of the DNN models executed on them. In Section 6.5, the PE utilization distribution for each accelerator under study is presented. This feature helps system designers optimize for the resource utilization and hardware resiliency at the system level. This section concludes with a qualitative comparison against related studies in Section 6.6.

6.1. Fault Resilience of Different DNN Models

This experiment injects random logic faults before running inference jobs for an accelerator with a specific number of pre-defined PE units (e.g., 10,000). To assess the impact of random faults, the mapping of operations to the PE units is ignored for this experiment by randomization of operation to PE mapping and calculation of the average. A case study sweeping different fault injection rates for nine different convolution DNN models is shown in Figure 16. The inference accuracy for CIFAR10 and ImageNet DNN nets are evaluated based on top-k = 1 and top-k = 5, respectively. For high injection rates, the inference accuracy for CIFAR10 DNN nets settled at 10%, whereas for ImageNet, DNN nets settled at 0.1%, reflecting the number of classes in each. The simulation points are manually selected to reduce the simulation time/resources; more simulated points are chosen within the region of high decline in accuracy.

The simulation results show a significant decline in the inference accuracy for fault injection rates as low as 250–2000 faults per million PE. This implies that a DNN model running on an accelerator with the same dimensions as a 256 × 256 TPU [2] may mispredict an object in the input image if 16–130 PE units are faulty. Projecting this range to Eyeriss with 168 PE units [4], only one fault can cause a similar accuracy decline. The differences between the nine DNN nets shown in the figure do not reflect a significant change in the trend of how prediction accuracy degrades.
6.2. Susceptibility to Different Fault Types

In this experiment, the sensitivity of a DNN model to different fault types is investigated. The objective is to understand the effect of SA-0 and SA-1 logic faults in arithmetic units on different DNN models, e.g., VGG16 and Resnet18. For the results from Figure 17, the simulation experiments were performed with the exact selection of a PE and fault site. The results show more decline in inference accuracy when injecting SA-1 faults than SA-0. This is because the ReLU activation function passes positive numbers. SA-1 pushes the kernel output (or node in linear layers) toward a positive increase. This argument is valid for positive and negative convolution layer outputs before activation. On the other hand, SA-0 works in the opposite direction. Flipping a bit from 1 to 0 reduces the output kernel value. The ReLU activation function shunts changes in negative kernel outputs, whereas faults causing discrepancies in the positive range will likely affect the output. A similar observation was reported when injecting fault in the datapath [11].

Figure 16. The performance of DNN inference for nine different DNN models and two different datasets, CIFAR10 and ImageNet. The performance of the models declines for fault rates as low as 100–1000 faulty PE units per million.

Figure 17. Comparing injection of SA-0 and SA-1 for CIFAR10 [47] on VGG16 and ImageNet [41] on ResNet18.
6.3. Critical Faults

This test evaluates the DNN network sensitivity toward faults at different fault sites of the multiplier circuit. The findings of the experiment can be used to guide the design of fault-tolerant techniques to protect against the most critical faults, since mitigating against all faults is not feasible. For the multiplier model in Figure 7, there are 876 different faults for 438 nodes. A study of the general trends in inference accuracy as a result of injecting random faults (SA-0, SA-1) in two node groups at different areas of the multiplier circuit is performed as a guiding example.

Figure 18 shows the inference accuracy at different fault injection rates for M and S node groups, at the input to full adder blocks FA(i, j), such that i = 2 and 3 <= j <= 8 (M23:M28 and S23:S28). The same notation as in Figure 7 is used. Faults injected into nodes closer to the most significant bits (MSBs) are more likely to propagate discrepancies at the multiplier output than those injected into nodes closer to the least significant bits (LSBs). Injecting faults at a rate higher than 10^{-3} faults/PE to node M28 is sufficient to reduce the inference accuracy significantly. On the other hand, a fault injected to M23 can be mitigated up to a rate of 10^{-1} faults/PE. A similar trend can be observed for the node group S. Comparing the M and S node groups in the same row, both node groups show a comparable decline in inference accuracy for the nodes closer to LSB. However, as we move closer to the MSB, injecting faults into M nodes shows a more significant decline in prediction accuracy. One reason is that M MSB nodes have a higher chance to flip the sign bit than S MSB nodes.

![Figure 18. Inference accuracy for M23:M28 and S23:S28 node groups for the example multiplier from Figure 7. VGG16 DNN model for CIFAR10 dataset is used in this experiment.](image)

6.4. Different Hardware Accelerator Architectures

In this experiment, a study of the impact of logic faults for different hardware accelerator architectures, TPU ([2]), Eyeriss ([4]) and Origami([3]), is presented. Its primary purpose is to assess the impact of data streaming patterns represented by different operation-PE mappings in our model. As detailed in Section 5, these mappings are captured by the IA/IB configuration matrices. Figure 19 shows the trend in inference accuracy decline when random faults are injected into a set of PE units randomly selected from all available PE units in each accelerator regardless of the utilization of each chosen PE. It is worth noting that the PE units’ utilization and workload distribution are not necessarily equivalent for all PE units in an accelerator due to the unique architecture and data streaming patterns. For example, in Eyeriss, for kernels with a smaller number of rows (K), some cluster rows may not be utilized. To highlight the impact of each accelerator’s unique operation-PE mapping
scheme, the experiment is repeated such that faults are injected only to the utilized PE units, and the chances of selecting a PE for injection are based on how much this PE is used compared to other PE units. The results are depicted in Figure 20.

**Figure 19.** Inference accuracy for random, grid-wise, logic fault injection with simplified operation-PE mapping for TPU [2], Eyeriss [4] and Origami [3]. VGG16 DNN model for CIFAR10 dataset is used in this experiment.

In Figure 19, the accuracy is shown for different injection rates. The graph indicates that the accuracy decreases as the number of injected faults increases. The graph shows the performance of TPU, Eyeriss, and Origami under fault injection. The accuracy is measured as a percentage, and the x-axis represents the injection rate in terms of faults per million PE units.

**Figure 20.** Inference accuracy for random logic fault injection with simplified operation-PE mapping. Selecting a PE unit to inject a fault is according to the utilization of that specific PE unit compared to the other used PE units. Showing the results for TPU [2], Eyeriss [4] and Origami [3]. VGG16 DNN model for CIFAR10 dataset is used in this experiment.

Injecting faults to Eyeriss below $1/N_{PE\text{Eyeriss}}$ Faults/Million PE does not impact the output, where $N_{PE\text{Eyeriss}}$ is the number of PEs of Eyeriss (e.g., 168 in [4]). The same observation can be made for Origami. Comparing Figures 19 and 20 indicates that injecting...
faults to the higher utilized PE units leads to higher fault resilience. The reason is not that the more utilized PE units contribute less to the output, but rather that another factor has a greater influence. This is because the inference workloads are not evenly distributed between the accelerator PE units for DNN layers closer to a model’s inputs, and faults in these layers have a higher chance of being mitigated. This uneven distribution of workload is due to the specific structure of the DNN model and the particular operation-PE mapping scheme for an accelerator. For example, the first two layers of VGG16 have 64 kernels vs. 512 kernels in layers 8–13. This results in using only 64 columns of the TPU PE grid out of 256 in the first two layers. The more utilized PE units are more likely to be in the first 64 columns of the grid, and injecting faults to the respective PE units will mainly impact layers 1–2. For Origami and Eyeriss, the uneven distribution of workloads between layers depends more on the kernel’s dimensions (for example, VGG16 uses only kernels of size $3 \times 3$). For this reason, there is a smaller difference in the PE workload distribution between the DNN layers closer to the model’s input and the layers closer to the output when compared to the TPU. Overall, comparing the resilience of the three accelerators in Figure 20, it is evident that for higher injection rates, when the PE utilization is taken into account, TPU performs better than Eyeriss and Origami because of the better workload distribution between the PE units. It should be highlighted, from this experiment, that understanding the dominating factors (from the DNN model and/or hardware) in system performance is possible by the assessment of hardware fault resilience at this level of abstraction offered by our framework.

6.5. Pre-RTL Accelerator PE Utilization

A key property of our framework ALPRI-FI is that it operates at the pre-RTL level. By functioning before RTL development, ALPRI-FI can provide feedback early in the design process, allowing for revisions before a more refined model is developed. Through abstraction layers from the ALPRI-FI framework, one can identify (for various types of models) the distribution of the workload for PE units. In Figure 21, the PE workload distributions for TPU [2] (a), Eyeriss [4] (b), and Origami [3] (c) are compared. This analysis helps in identifying which PEs are under heavier loads and can help system designers identify where faults are most likely to have significant impacts at the application level. This type of insight can be used in decisions on how to allocate fault resources at locations where they are most needed.

![Figure 21](image)

Figure 21. PE utilization distribution for TPU [2] (a), Eyeriss [4] (b) and Origami [3] (c). The normalized distribution statistics are collected from running the same inference jobs on VGG16. Evaluating the impact of operation-PE mapping on workload distribution and hardware fault resilience can be performed efficiently using our method, thus gaining early feedback for design revisions before developing an RTL model.

It is important to highlight that the PE workload distributions plotted on Figure 21 are based on the basic operation-PE mapping features that are unique to each accelerator as
described in Section 5. For instance, the mapping for Eyeriss shown in the figure initially assumes processing only one kernel at a time. However, a modification in the operation-PE mapping would alter the workload distribution across the processing grid. Therefore, to enhance resource utilization, whenever deemed necessary and assessed to be feasible, multiple kernels can be processed simultaneously. The value of the ALPRI-FI framework is that it allows for updates to the operation-PE mapping and provides estimates of the impact of different workload distributions on hardware fault resilience; thus, designers can make early decisions on the development of fault-tolerant strategies.

6.6. Comparison to Related Studies

Table 5 summarizes the main differences between ALPRI-FI and studies from [12,24–27]. In ref. [12], faults in control and management units are emulated by inserting specific instructions into the software kernels of accelerators. Although this method is very fast, as it is applicable to assess the faults using silicon prototypes, it is complementary to the focus of our study, which is focused on the effects of permanent faults in arithmetic units on the accuracy of DNN inference at the application level. The impact of such faults depends on the values of the input operands, making it challenging to adapt the approach from [12] for large-scale arrays of arithmetic units common in DNN accelerators. Additionally, our research aims to enable designers to identify potential reliability concerns due to permanent faults in arithmetic units early in the design process even before the RTL code has been developed.

Table 5. Comparing support and performance of ALPRI-FI against literature studies [12,24–27].

<table>
<thead>
<tr>
<th>Study</th>
<th>Simulation Environment</th>
<th>Fault Assessment (Support Arithmetic Permanent Faults?)</th>
<th>HW Architecture Support</th>
<th>Evaluated DNNs/Datasets</th>
<th>Inference Speed (Minutes/Image/Core)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[12]</td>
<td>RTL-Env</td>
<td>Permanent faults to control/management units. Faults emulated on silicon (NO)</td>
<td>• GPU with microarchitecture ready • Tested on NVIDIA G80</td>
<td>General GPU load/applications</td>
<td>No DNN Inference Data</td>
</tr>
<tr>
<td>[25]</td>
<td>RTL-Env</td>
<td>Transient faults to activation, weights and control (NO)</td>
<td>• Configurable: Convert Complete HW RTL to C++ • Tested with Systolic Arrays by Gemmini [48]</td>
<td>ResNet, GoogLeNet/ImageNet</td>
<td>0.178</td>
</tr>
<tr>
<td>[27]</td>
<td>Gates-Env</td>
<td>After-synthesis fault simulation (YES)</td>
<td>HW = DNN Structure</td>
<td>LeNet-5/MNIST</td>
<td>No Data (slow)</td>
</tr>
<tr>
<td>ALPRI-FI</td>
<td>System-Env</td>
<td>Wide support summarized in Table 4 (YES)</td>
<td>• Configurable: describe operation mapping using IA/IB • Tested with TPU, Eyeriss, Origami</td>
<td>Configurable: Tested with 9 models (Figure 16)/CIFAR10, ImageNet</td>
<td>29</td>
</tr>
</tbody>
</table>

The study from [25] utilizes RTL models to assess resiliency against transient faults. This approach achieves fast simulation times (0.178 min per image) by converting RTL models to C/C++ and activating accurate models only when transient faults occur during
simulation. However, extending this method to assess permanent faults in arithmetic units presents several challenges. Unlike transient faults, permanent faults require that accurate circuit models of arithmetic units be active throughout the entire simulation. Adding accurate circuit models for arithmetic units, which are not natively available in RTL models, and the activation of these models during all simulation times are expected to increase the simulation time significantly. Furthermore, unlike our approach that can operate at the pre-RTL stage, the RTL models become available during the later design stages.

Using accurate gate-level models for all the accelerators is another option to evaluate permanent faults. In [27], a complete gates-level hardware model is produced by synthesizing DNN model operations. Evaluating fault resiliency using such a complete gate-level model of the accelerator is expected to result in an accurate fault assessment. However, the simulation time is very long, and it is unfeasible to conduct fault injection campaigns efficiently at the DNN application level. According to [27], replacing each multiplication operation with a fault-injectable hardware model increases the simulation time by 7 s. Scaling this to the number of multiplications for a state-of-the-art DNN model is expected to require an excessive amount of runtime. Therefore, this approach is limited to a final assessment before the physical implementation rather than during the early stages of design space exploration, as it is one of the critical objectives of our work, as highlighted in Figure 1.

One attempt to increase simulation performance using parallel processing is the study in [26], which models the logic gates of the arithmetic circuits using neural network-based models. Using neural network-based models facilitates the usage of GPU engines. However, aside from the impact of such models on the assessment accuracy, this type of modeling of the circuit logic increases the simulation demands because of the large number of models that need to be executed to evaluate MAC operations. For example, a relatively small LeNet-5 network consumes 1670 processor minutes for each of the 28 \times 28 MNIST images.

In addition to the simulation time increases from using accurate circuit models of arithmetic units, which can be in the order of 1000x from Section 5, it is important to highlight another key challenge when considering a DNN framework to test the resiliency of different hardware architectures. This challenge is the significant amounts of memory needed during simulation to support configurable operations mapping to arithmetic units. In [24], the purpose is to assess memory faults, focusing only on faults affecting the operands of arithmetic operations. Hence, the simulation CPU requirements are sufficient to simulate larger DNN models such as ResNet50 performing inference for the ImageNet dataset (34 min per image in Table 5). However, it should be noted that describing the mapping of operations to the processing elements takes more considerable amounts of memory than running inference without operation mapping support. According to [24], it is required to free the memory of the simulation machine belonging to one DNN layer before processing the next layer. This can limit the performance when more fault assessment features are added, such as using an accurate model of the arithmetic units. Through the proposed method, memory requirements to describe the mapping of operations are reduced to be proportionate to the model size and not to the number of operations. This helped to keep manageable simulation performance (29 min per image) even with the inclusion of gate-level models of the arithmetic units, as shown in the last row of Table 5.

To summarize, the primary goal of our framework is to assist system and hardware designers in evaluating the resilience of new hardware architectures against permanent faults in arithmetic units, which can take up to 40% of the silicon area. This evaluation can be made at the DNN application level and is intended to be carried at the very early stages of the development process even before the RTL model is developed. For these reasons, we are considering ALPRI-FI as a suitable candidate for early hardware fault resiliency assessment in DNN accelerators.
7. Conclusions

Hardware accelerators are essential for machine learning workloads due to their performance and energy efficiency in handling the large computational demands of DNN models. Assessing the impact of hardware faults on these accelerators is critical to maintaining reliable operation in safety-critical applications and cloud infrastructure. However, fault assessment is time consuming because it requires extensive analysis of various fault scenarios. Additionally, accurately simulating and evaluating the impact of these faults on DNN models with large datasets is extremely computationally demanding. To address this concern, we introduced ALPRI-FI in this paper, which is a comprehensive framework for evaluating the fault resilience of DNN accelerators during the early design and implementation stages.

While prior work has investigated the impact of transient faults or permanent faults in memory, there is a lack of understanding of how faults in arithmetic units impact the functionality of DNN models, particularly when fault injection experiments need to be carried out at scale. To address this, ALPRI-FI relies on information from data streaming patterns in hardware accelerators, determined prior to the RTL implementation, and gate-level accurate arithmetic circuit models, which are used to perform precise fault injection experiments. It was demonstrated how accurate fault injection experiments in arithmetic units could be incorporated into hardware accelerator models within state-of-the-art frameworks, such as PyTorch, thus reducing both development time and computational runtimes. More importantly, these experiments are conducted during the early design space exploration stage, providing insights that can guide adjustments before more detailed implementations are finalized. To validate the generality of our framework, we evaluated the fault resilience of various hardware accelerators from the literature for different DNN models. We provided a comparative analysis in terms of fault resilience that, to our knowledge, had not been reported in the public domain for large-scale designs and models.

Author Contributions: Conceptualization, K.M. and N.N.; methodology, K.M.; software, K.M.; validation, K.M.; formal analysis, K.M.; investigation, K.M.; resources, K.M.; data curation, K.M.; writing—original draft preparation, K.M.; writing—review and editing, K.M. and N.N.; visualization, K.M.; supervision, N.N.; project administration, K.M.; funding acquisition, K.M. and N.N. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Natural Sciences and Engineering Research Council (NSERC) of Canada grant number RGPIN-2020-06884. Computations were performed on the Niagara supercomputer at the SciNet HPC Consortium and were enabled in part by support provided by Compute Ontario (computeontario.ca accessed on 1 January 2021) and the Digital Research Alliance of Canada (alliancecan.ca accessed on 1 April 2022). SciNet is funded by Innovation, Science and Economic Development Canada; the Digital Research Alliance of Canada; the Ontario Research Fund: Research Excellence; and the University of Toronto.

Data Availability Statement: The data that support the findings of this study are available from the corresponding author, K.M., upon reasonable request.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

- ALPRI-FI: Architecture-Level Pre RTL Implementation Fault Injection
- SDC: Silent Data Corruption
- DNN: Deep Neural Network
- IFmap/OFmap: Input/Output Feature Map
- MAC: Multiply-Accumulate (operation)
- FF: Flip Flops
**References**


20. Hollister, S. *There Is No Fix for Intel’s Crashing 13th and 14th Gen CPUs—Any Damage Is Permanent*; The Verge: New York, NY, USA, 2024.
38. systemC. Available online: https://systemc.org/ (accessed on 7 July 2024).

**Disclaimer/Publisher’s Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.