FAWS: Fault-Aware Weight Scheduler for DNN Computations in Heterogeneous and Faulty Hardware

Shaswot Shresthamali
Keio University
shaswot@acsl.ics.keio.ac.jp

Yuan He
Keio University
he@acsl.ics.keio.ac.jp

Masaaki Kondo
Keio University
kondo@acsl.ics.keio.ac.jp

***Abstract*—The idea of using inexact computation for overprovisioned DNNs (Deep Neural Networks) to decrease power and latency at the cost of minor accuracy degradation has become very popular. However, there is still no general method to schedule DNN computations on a given hardware platform to effectively implement this idea without loss in computational efficiency. Most contemporary methods require specialized hardware, extensive retraining and hardware-specific scheduling schemes. We present *FAWS: Fault-Aware Weight Scheduler* for scheduling DNN computations in heterogeneous and faulty hardware. Given a trained DNN model and a hardware fault profile, our scheduler is able to recover significant accuracy during inference even at high fault rates. *FAWS* schedules the computations such that the low priority ones are allocated to inexact hardware. This is achieved by shuffling (exchanging) the rows of the matrices. The best shuffling order for a given DNN model and hardware fault profile is determined using Genetic Algorithms (GA). We simulate bitwise errors on different model architectures and datasets with different types of fault profiles and observe that *FAWS* can recover up to 30% of classification accuracy even at high fault rates (which correspond to approximately 50% power savings).**

I. Introduction

Owing to the popularity of Deep Neural Networks (DNNs), specialized accelerator hardware (such as GPUs, TPUs and FPGAs) have become very popular to overcome the resource constraints (e.g., power, computation) in real world applications. DNN computations are embarrassingly parallel so the current trend is to have a large number of parallel Processing Elements (PEs) that operate in SIMD mode. For e.g., NVIDIA GPUs contain CUDA cores/threads grouped in Streaming Multiprocessor (SM) blocks and FPGAs have Deep Learning Processing Units (DPUs) consisting of hybrid PEs.

Contemporary DNN accelerators have densely packed chips/chiplets in various novel architectural organizations (e.g., SambaNova's Datascale Systems, Cerebras's Wafer-Scale Engine (WSE), Tesla's Dojo D1 etc.). However, with current nanometer scale process technologies, yield and reliability are drastically reduced. Disposing entire chips due to the presence of a few faulty PEs is impractical. Adding error correction mechanisms and redundancies to maintain worst-case margins increases power and cost. In addition, chips also degrade with time and external factors causing some PEs to be more faulty than others. Therefore we cannot assume accelerators to be perfectly reliable all the time.

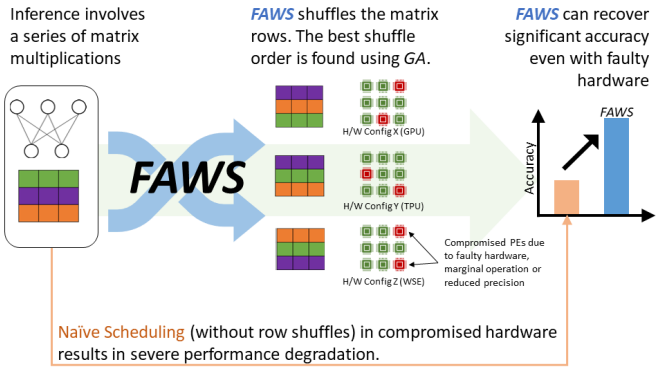

Fig. 1: DNN models deployed in the field are implemented in different (compromised) hardware configurations that tradeoff efficiency for inexact computation. FAWS schedules the DNN computations by shuffling rows to recover performance.

Motivation: Although accelerators maybe unreliable, DNNs have inherent algorithmic resilience to computational errors on account of their distributed parallel nature and over-provisioned parameters. They have been shown to be fault tolerant to Bit Error Rates (BERs) as high as 10^{-4} (in contrast to conventional systems that require BERs in the order of 10^{-15}) [1], [2]. Researchers have cleverly leveraged this resiliency of DNNs to salvage DNN performance even with faulty hardware. By relaxing computational exactness, they have increased energy efficiency and reduced computational memory/latency at the cost of minor performance degradation. There is still room for substantial gains with aggressive voltage underscaling in SRAMs [1] and DPUs [3] as well as using reduced precision [4], [5] if the effects of resulting performance degradation can be minimized.

The general method for recovering performance is to first identify unimportant neurons and then schedule them on compromised hardware. [1] While studying the sensitivity of *neurons* to computational errors is interesting, it does not help

[1]We use the term"compromised hardware" as an umbrella term to indicate hardware that is faulty either due to manufacturing defects, marginal operation or reduced precision.

very much when implementing DNNs on accelerators. This is because optimized code used in many Deep Learning (DL) framework breaks the conceptual one-to-one mapping between a neuron and its computations. For e.g., in the case of GPUs, the computations corresponding to a neuron is actually spread out across multiple CUDA threads to maximize throughput. This is achieved by block-tiling methods used widely in GEMM (General Matrix Multiply) libraries. As a result, many CUDA cores participate in the calculation of one neuron and CUDA cores are reused extensively by different neurons. The scheduling therefore must focus on allocating unimportant *computations* (not neurons) on compromised hardware. Thus, scheduling DNN computation in compromised hardware becomes very difficult (due to the resulting large combinatorial optimization search space). Current solutions are applicable only to very specific DNN models/hardware platforms [5]–[8] and usually involve extensive retraining. Given the variety of DNN models, accelerators and types of fault profiles, it is important to schedule computations on compromised hardware without relying on internal microarchitectural details of the hardware (which is usually not known to the user). Retraining the model is also not usually possible once the hardware is deployed and the dataset may not be easily available.

Proposal: We propose *FAWS: Fault-Aware Weight Scheduler* to schedule DNN computations in compromised hardware to reduce the effect of faults and recover model performance. FAWS achieves this by *shuffling the rows of the matrices* during multiplication Figure 1. This gives it some control over where the computations are allocated in the hardware. FAWS uses Genetic Algorithm (GA) to search the huge optimization space for the best shuffling order such that the majority of the critical operations are assigned to robust computation units. Row-shuffling, while simple, is a general methodology for performance recovery that treats the DNN model as a black box. It does not alter the semantics so code redesign in not necessary. Neither does it interfere with the optimized dataflow graphs specific to different microarchitectures. It also has extremely low computational and time complexity and can be applied to a vast variety of DNNs. This method does not require information about the internal hardware details and no retraining is required.

In this paper, we focus only on inference. Learning the weights of a DNN model is a one-time cost that is performed in a fault-free environment. Once trained, the same model is reused many times over in different hardware platforms with different fault profiles. The GA search to find a suitable shuffle order for a given hardware configuration is also a one-time cost and can be done offline. This requires a fault injection simulator (like in [2]) and the hardware fault profile. The hardware fault profile can be easily obtained before deployment via diagnostic tests like those in [9]–[11] . Once the fault profile is known, the GA search can be rerun offline even if the model is updated.

This paper makes the following contributions:

- We develop ***FAWS*** that recovers lost performance of DNN models on compromised hardware by allocating unimportant computations to compromised PEs (Section III). This is achieved by shuffling the rows of the matrices during multiplication. The row shuffle order is determined by GA search.
- We analyze the fault sensitivity of DNN models and their layers to different fault types and fault rates using bit-level fault injection (Section V-A).
- We determine the most suitable row shuffle orders by using FAWS and recover significant performance (by as much as 30%) from the DNNs in compromised hardware (Section V-B).

II. Related Work

In [12], the authors identify *stuck-at* and *random bit flips* as the most widely and successfully used abstract fault models. So we focus mainly on the errors caused by these faults in this work. Our paper builds up on the work in [1]–[3]. In [2], the authors propose a bitwise fault-injection framework for DNNs to analyze the effect of fault-rate and performance degradation on different DNN models. They show that is possible to leverage implicit fault tolerance properties of DNNs to improve energy efficiency. In [1], this is demonstrated by lowering the SRAM voltages to improve energy performance at the cost of increasing fault rates. They also propose fault mitigation techniques by setting faulty bits/words to zero. In [3] the authors use undervolting in FPGA PEs to increase energy efficiency and compensate for it with frequency underscaling. The approximate computing community has also exploited the fault-tolerance of DNNs by using low-power approximate arithmetic hardware [7], [8]. In [6]–[8], the authors characterize neuron sensitivity and approximate low priority neurons. The drop in performance is recovered by retraining the model. This is not a general methodology and is not always possible due to lack of access to datasets and computational resources. DNNs can also be made more robust by pruning and dropout. These solutions target individual neurons/weights, which as discussed before, is not how hardware faults generally map to the DNN computation. Our paper focuses on a bit-level fault-model at computation level which is more realistic.

Another popular direction for reducing power and latency has been to use reduced precision arithmetic. This is now widely supported by many ML frameworks and libraries. In [4], the authors propose an automated framework to determine the bitwidth required for different computations in a DNN. Their method uses a hardware-aware optimization loop to specialize for different hardware configurations. Other works such as [5] use specialized accelerator architectures with reducible precision for maximizing throughput and minimizing energy. Our work differs from [4], [5] in that it does not rely on hardware details for optimization and is therefore applicable to many different types of accelerators.

III. Fault-Aware Weight Scheduler (*FAWS*)

A. Fault Models

A *fault* is defined as "an anomalous physical condition...which gives rise to an error" [12]. A fault may induce an

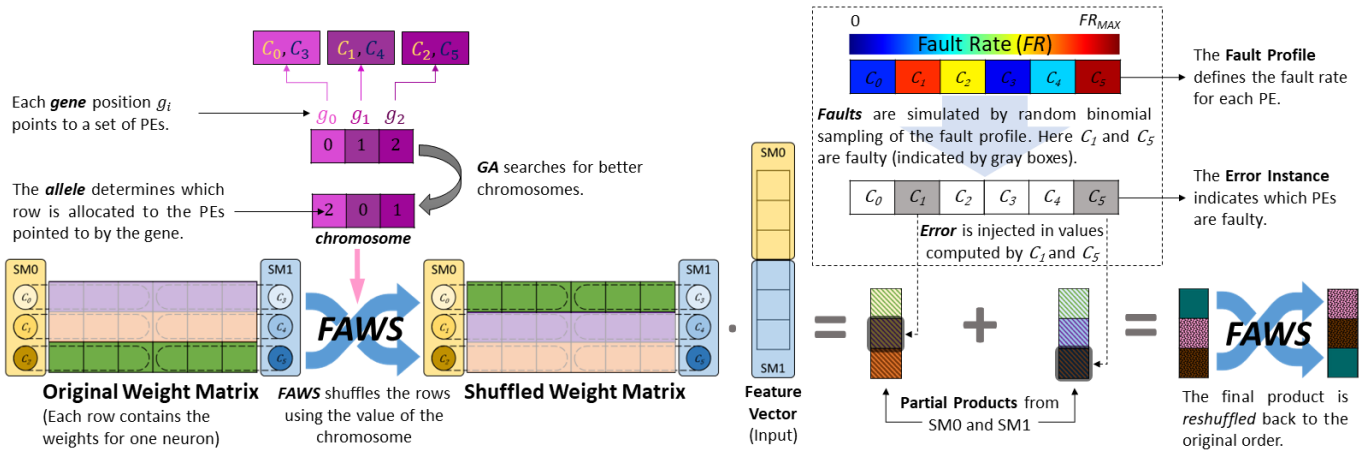

Fig. 2: Matrix-vector tiled multiplication is spread across two SM blocks and six CUDA cores. *FAWS* shuffles the rows of the weight matrix, using the best genes from GA search. The most critical neuron/row (green) is allocated to the most robust cores (C_0 and C_3). As a result, the corresponding element of the final product vector (dark green) is not affected by the faults. The figure also illustrates our error-injection simulation strategy to simulate hardware faults.

error, which is a deviation of the logical state from the correct one. Permanent faults are continuous and always present, arising mostly due to irreversible physical damage. A transient fault occurs for a short period of time mostly due to process and environmental variations as well as degradation. *Intermittent* faults, usually caused by marginal device operation, are recurring transient faults and these are the most common. We abstract the effects of these various faults using the following bitwise fault model similar to [1], [2], [12]:

- *Flip-to-0/1*: a random register bit is held at either 0/1.
- *Bitflip*: the value of a random register bit is flipped.

The Flip-to-0/1 fault model represents many of the permanent faults in hardware. The Bitflip fault models the transient faults in PE SRAM registers/memory elements due to marginal operation or external effects.

In addition to bitwise fault models, we also define a fault model for mixed-precision computations. Strictly speaking, reduced precision is not a result of an anomalous physical condition, we can model it as a type of marginal operation. In this work, we model reduced precision as a type of fault where a block of the least significant bits of the mantissa are set to zero. We use this fault model for no other reason than for simplifying the description and analysis of our experiments.

B. Fault Injection Methodology

A bitwise fault model is much more realistic w.r.t. hardware implementation than a neuron-wise model. We further only consider faults that affect the intermediate computations of DNNs and not the main memory elements that contain the input data stream, weights and biases. The effects of errors in the weights and biases in the main memory and their mitigation have been addressed in [1], [2]. We assume that the input, weights and biases are not corrupted. This way if there is any degradation in model performance, it is solely due to incorrect computation and not corrupt data.

1) Tiled Matrix Multiplication: We focus on the effects of hardware faults on matrix multiplications because they constitute the majority of the computations for DNNs. Matrix multiplications consist of many multiply and accumulate (MAC) operations that can be performed in parallel. For general matrices, tiled matrix multiplication is preferred because it maximizes parallelization and minimizes memory conflicts by exploiting data locality. For instance, in NVIDIA GPUs, large multiplying matrices are broken up into small tiles and each tile is computed by a SM which consists of a *warp* of 32 CUDA threads/cores. The partial products from each of the blocks are summed together to give the final product. This is why we cannot assume one-to-one mapping between neurons and PEs for DNN accelerators. The calculation for one neuron is performed by multiple PEs and each PE is reused many times for different neurons (Figure 2). More details on optimizing GEMM for GPUs can be found in [13].

We are interested in the effect of faulty behavior of the CUDA cores in the SM blocks. Hence we simulate faults by injecting errors separately into the *partial products* from each SM block. This fault-injection methodology is illustrated for the case of matrix-vector multiplication in Figure 2. Note that injecting errors *after* the final matrix product has been computed does not capture the faulty behavior of the CUDA cores in the SMs.

In this work, we use GPU only for the sake of example. The basic principle behind using tiled multiplication to distribute the MAC operations among a vast array of PEs (similar to CUDA cores) to maximize parallelization is applicable to most accelerators (e.g., TPUs, FPGAs etc.).

2) Fault Profiles and Error Instances: In this work, we assume the fault rate (*FR*) is the same as the error rate i.e., faults always result in errors. This is simply to make the analysis straightforward and has no loss of generality during analysis and experiments. We simulate faults in the GPU hardware using a bitwise fault simulator similar to [2]. The

simulator injects errors into the computation executed by a PE with a probability that is determined by its fault rate (Figure 2). The *fault profile* of the accelerator (GPU) describes the different fault rates of its PEs (CUDA cores). At each time step, an *error instance* is sampled from the fault profile. The error instance determines whether or not if a particular PE is faulty. We assume the fault profile has been obtained via diagnostic tests.

3) im2col Optimization: Naive implementation of the convolution operation is inefficient. It is also easily one of the most compute intensive components of DNNs especially when there are multiple kernels and channels. A popular approach for implementing convolution operations is to flatten the kernel matrices, extract patches of the images into columns (*im2col*) and perform Multiple Channel Multiple Kernel (MCMK) convolution using existing GEMM libraries [14]. This decreases the latency of the convolutional operation at the expense of larger memory use. This method of convolution is the most widely used and is present in many of the popular deep learning frameworks. We assume convolution operations are optimized using *im2col* method and implemented as GEMM. Thus fault injection in convolution operation is similar to that for matrix-matrix/vector multiplications.

C. GA Problem Statement for FAWS

The main objective of *FAWS* is to achieve better scheduling of computation in faulty hardware to recover lost performance. *FAWS* achieves this by shuffling the matrix rows before multiplication. After computation, it "reshuffles" the rows of the product back to the original order after so that the mathematical semantics are preserved. By permuting (shuffling) the order of the rows, we get some (limited) control over where these computation take place within the accelerator. The number of possible ways in which one can shuffle the rows of a matrix with n rows is $n!$. This is a huge optimization space and exhaustive brute force search is not possible. *FAWS* uses GA search to find a suitable shuffle order. Exchanging rows does not have high computational/temporal complexity so the overhead is minimal for *FAWS*. The row-exchange usually involves changes in the metadata of the tensor i.e., only the tensor "view" changes and no actual data is copied to/from the memory. When implemented this way, the temporal complexity is $O(1)$ (for each row swap). Furthermore, since all the rows for a PE block are usually loaded into a shared memory pool, row-exchanges do not increase the cache miss rates significantly.

1) Genetic Representation: GA is especially suitable for this search problem because it is highly parallelizable and defining the fitness functions and chromosomes is quite straightforward. *FAWS* uses GA to find the row shuffle order for a given matrix multiplication for a known hardware fault profile.

We define the chromosome to be an array of integers that represents the row-exchange order for the *multiplier* matrix. The length of the chromosome is equal to the number of rows of the matrix. The *genes* (i.e., the element positions of the chromosome) represents the PEs that compute the row indicated by the *allele* (i.e., the value of the gene). This means each gene position g_i maps to a set of PEs (CUDA cores) $S_i \in \{C_i, C_j, ...\}$. If g_i contains the allele (value) a_i, then row a_i of the multiplier matrix is scheduled to be computed by the PEs in S_i (Figure 2).

The mapping of genes to actual hardware is not in our control and is usually not known. However, this is not a problem as long as the internal scheduling of the accelerator is consistent. Consequently, our method is independent of the inner workings of the accelerator. Of course, it is possible to get more control over the scheduling by accessing the accelerator firmware. However, this results in a highly specialized solution for a very specific hardware-DNN model pair for a very specific fault profile. This is not a general solution and usually not possible in proprietary hardware. Our method is agnostic to the microarchitecture of the accelerator hardware. If the fault profile changes, it is always possible to re-run the GA search find a suitable scheduling scheme (shuffle order) for *FAWS*.

2) Fitness Function: A DNN model inference involves a number of matrix multiplications. We associate one chromosome for each matrix multiplication that is optimized by *FAWS*. For a given fault profile and a set of chromosomes, the fitness function is simply the top-1 classification accuracy of the model when the computation is implemented on that hardware with the row shuffle orders as determined by the corresponding chromosomes. Every generation, the best N individuals from a population are selected to be parents. These parents generate new offspring through mutation and crossover. The fitness of the offspring is evaluated and the best N individuals become parents for the next generation.

IV. Evaluation Setup

A. DNN models

We use two types of DNN models for evaluation: *mnist32-cnn* and *fashion-cnn*. *mnist32-cnn* consists of one convolutional layer (*c0*) with 32 (4×4) kernels followed by three dense layers (*h0*,*h1*,*h2*) and a final output layer (*op*). The model is trained using the MNIST dataset [15] with the images resized to 32×32 (for faster fault-injection simulation). *fashion-cnn* consists of two convolutional layers, *c0* and *c1*, each with 32 (4×4) kernels. This is followed by two dense layers (*h0* and *op*). It is trained with dropout on the Fashion-MNIST dataset [16] with 28×28 images. The fault-free accuracy of *mnist32-cnn* is about 99% whereas *fashion-cnn* is about 92%. We train three instances of *mnist32-cnn* and *fashion-cnn* each using different seeds.

We analyze the fault sensitivity of each of the model instances by injecting different types of errors into each of their layers one-by-one. If faults in a particular layer significantly degrade the performance, we use GA search to find the best chromosome (row-exchange order) for *FAWS*.

TABLE I: Different types of errors

Error	Description	Max. Fault Rate
Flip-to-0	Set random exponent bit to 0	1E-1, 2E-1, 5E-1
Flip-to-1	Set random exponent bit to 1	1E-3, 2E-3, 5E-3
BitFlip	Flip random exponent bit	1E-3, 2E-3, 5E-3
TF32	Set 13 mantissa LSBs to 0	1E-1, 2E-1, 5E-1
BF16	Set 16 mantissa LSBs to 0	1E-1, 2E-1, 5E-1

B. Fault Profiles

We consider an NVIDIA GPU-based DNN accelerator for our experiments. This is simply for the sake of example and there is no loss in generality during fault-sensitivity analysis and recovery with *FAWS*. We assume the GPU has 20 streaming multiprocessors (SMs) and each multiprocessor has 32 PEs (i.e., CUDA cores). We consider only 20 SMs per GPU to keep our simulations tractable. Each CUDA core has a fault rate i.e., probability of a fault manifesting and resulting in an error.

The probability distribution of the *FR* of each CUDA core is determined by the *fault profile* of the GPU. We use six different types of fault profiles in our experiment and distinguish between them by the maximum fault probability (FR_{max}) . Thus a GPU with a fault profile characterized by FR_{max} means that each of its CUDA cores have different fault probabilities (including zero for non-faulty PEs) but none exceed FR_{max}. At each timestep, each CUDA core is either faulty or non-faulty with a probability distribution that depends on its particular fault rate. We assume that faulty operation of the CUDA cores are independent events for the sake of generality; although in practice, there may be some strong correlation between the *FR*s of spatially neighbouring CUDA cores due to shared power/data bus, thermal hotspots etc.

For our evaluations, we generate artificial fault profiles of the GPU for a given FR_{max} by randomly assigning a *FR* $\in (0, FR_{max})$ for each CUDA core. For a given FR_{max}, we instantiate two different fault profiles using different seeds to represent different hardware profiles. These hardware profiles represent different types of hardware with manufacturing/aging defects, or the marginal behavior due to power optimization schemes, or heterogeneous PEs with reduced precision.

C. Error Types

At each time step, *error instances* are derived from the fault profile using random binomial sampling. Based on the fault model described in Section III-A, we simulate five different types of errors. These errors and their corresponding FR_{max} are listed in Table I. We are primarily interested in bitflips that happen only in the exponent field of the FP32 data because they affect the performance most strongly. Our preliminary evaluations show that bitflips in mantissa are not very serious because they do not cause large enough deviations from the correct value.

We also define mantissa truncation as an "error" for sake of consistency in description. While individual bitwise errors in mantissa are not very serious, truncating out a block of mantissa bits affects the performance of the model significantly [4]. TF32 and BF16 are popular truncation schemes used in DNN and ML frameworks. We emulate this reduced precision by setting some number of mantissa least significant bits to zero. Fault sensitivity analysis using TF32 and BF16 errors gives us an indication of how sensitive the DNN is to the precision offered by the mantissa bits and how much recovery can be expected. This scenario may arise in cases when the same model has to be implemented on different hardware platforms that contain a mixture of PEs that compute using different precision [5], and where hardware-in-the-loop optimization may not be practical. In such a case, we would like to know whether shuffling the rows using *FAWS* can recover some of performance lost due to mixed precision arithmetic. For the sake of generality, we consider an extreme case (i.e., a harder optimization problem) of a GPU with heterogeneous CUDA cores that randomly switch between different levels of precision (FP32, TF32 or BF16) with a fixed probability. Of course, this is unrealistic and in practice, the CUDA cores would be regularly arranged and their precision would be controlled deterministically.

D. Metrics and Evaluation Parameters

The model performance is measured by evaluating its classification accuracy on the 5120 test images (i.e., images not seen during training). Since errors manifest stochastically for a given fault profile, we evaluate the model three times with that profile and report its mean and standard deviation. The fitness function for the GA algorithm also uses the mean over three evaluations of the model for a given fault profile and shuffle order.

Each GA optimization run is performed three times and we use the best chromosomes out of each run (one run lasts for 100 generations). We limit the population size to 20 with a Crossover Rate (CR) at 0.6 and Mutation Rate (MR) of 0.2. During each generation, the fittest 20 individuals are selected for breeding and generating the next batch of individuals (*truncation selection*). This is faster than probabilistic selection like the Roulette wheel. Moreover, we observe that most individuals have very close fitness values so Roulette wheel type selection mechanisms are not worth the additional time and computation. The hyperparameters for GA were determined using a grid search. We use the same GA hyperparameters across all our experiments for consistency. However, there is much room for improvement if one were to use different hyperparameters depending on the model and the DNN layer. We leave this finer hyperparameter search problem for future work.

V. Experimental Results

A. Fault Sensitivity Analysis

1) mnist32-cnn: Figures 4 and 5 show the fault sensitivity for different layers of *mnist32-cnn*. The figures report the mean and the standard deviation of the performance degradation across all model seeds and error profile seeds. In Figure 4, we observe that the classification accuracy degrades gracefully

TABLE II: Sizes of matrices in different layers

Model	Layer	Multiplier Matrix	Input Matrix/Vector
mnist32-cnn	*c0*	32×16	$16 \times (29 \times 29)$
	h0	1024×6272	6272×1
	h1	256×1024	1024×1
	h2	64×256	256×1
	op	10×64	64×1
fashion-cnn	*c0*	32×16	$16 \times (28 \times 28)$
	c1	$32 \times (16 \times 32)$	$(16 \times 32) \times (32 \times 32)$
	h0	1024×1568	1568×1
	op	10×1024	1024×1

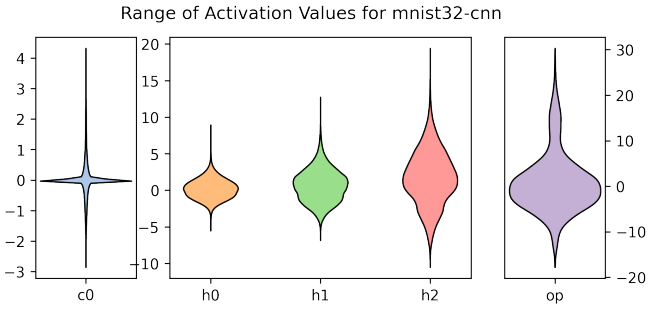

Fig. 3: The range and distribution of the matrix multiplication products in different layers for *mnist32-cnn* (before ReLU or softmax).

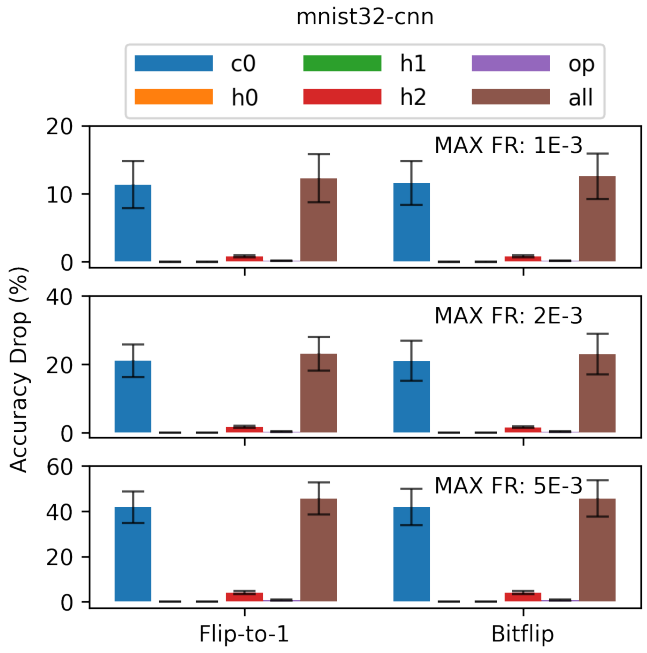

Fig. 4: Sensitivity of different layers to bitwise errors in the exponent field. Layer *c0* is most sensitive due to high kernel reuse.

with increasing fault rates. This is encouraging because it shows that errors in computation do not automatically imply catastrophic failures in DNNs. For *mnist32-cnn*, layer *c0* is the most sensitive to Flip-to-1 and Bitflip errors and contributes most to performance degradation. These observations concur with the results in [2]. The degradation becomes significant when fault rates exceed $5e{-}3$.

We hypothesize that the reason behind *c0*'s sensitivity to these errors is due to i) the limited range of the output of the convolution layer and ii) the high reuse of the kernel matrix. From Figure 3, we observe that the outputs of the convolution matrix multiplication (before ReLU activation) are tightly concentrated around 0. When an exponent bit is accidentally flipped to 1, the resulting error is quite large. If the change is in the positive direction, this error is propagated through the ReLU and maxpool layers and thus affecting the rest of the DNN computation pipeline. Secondly, from Table II, we see that *c0* convolution has a small kernel matrix which spans over a few PEs. These PEs are reused many times over the *im2col* patches extracted from the image. Thus, recurring errors in the PEs containing the kernel matrix are expressed many times during the convolution thus amplifying the effect of the error. The work in [2] also follows a similar reasoning.

In contrast, layers *h0-h2* span over a large number of PEs due to their large sizes but are used only once for multiplying a feature vector (because inference is usually one image at a time). Hence, the errors in *h0-h2* layers do not cause significant degradation. Among the fully connected layers, *h2* is the most sensitive to Flip-to-1 and Bitflip errors. This is probably due to its proximity to the output layer. Any bitwise errors in its exponent causes a large deviation resulting in wrong neurons being activated in the output layer.

In Figure 5, we observe that Flip-to-0 errors cause very little performance degradation (less than 5%) even at very high fault rates. This seems to hold true irrespective of the model and the layer in which this error manifests (see also Figure 6). In fact, the authors in [1] purposefully set bits to zero when error is detected, as a *fault-correction* technique. As stated in [1], only a few neurons fire at a time while the rest are inhibited. Since it is more probable that a neuron does not fire, a Flip-to-zero (which inhibits firing) is more benign that Flip-to-1 (which may activate accidental firing).

Truncating the LSBs of mantissa (TF32, BF16) does not cause dramatic degradation even at very high fault rates (Figure 5). This observation is in agreement with the idea behind aggressive precision reduction techniques used in [4]. Mantissa truncation errors affect output layer the most. It is interesting to note that the output layer is relatively more immune to Flip-to-1/Bitflip error than mantissa truncation when compared with the other layers.

We reason that this is due to the precision sensitivity of the final softmax activation as a result of the "squashing" effect of the exponential function in softmax. The product of the output layer matrix multiplication is a vector of ten elements (one for each image class). This vector is fed into the softmax activation which squashes them using an exponential

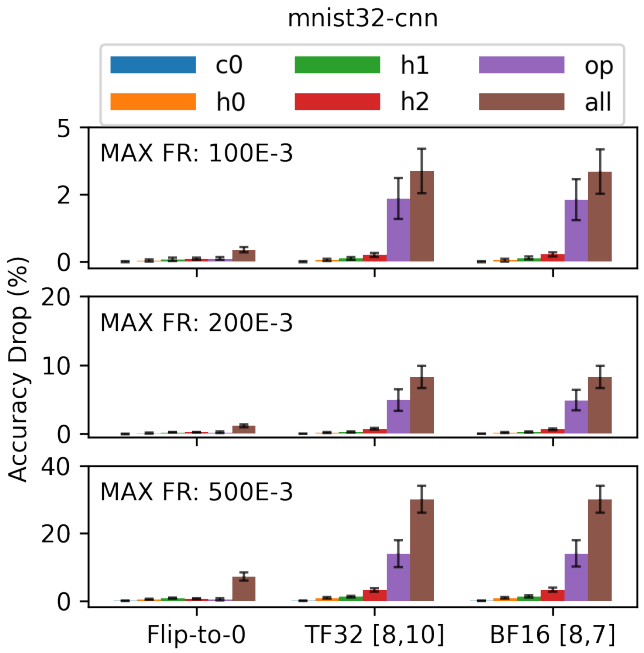

Fig. 5: Flip-to-0 errors are benign even at very high fault rates. Mantissa truncation affects output layer (*op*) the most severely.

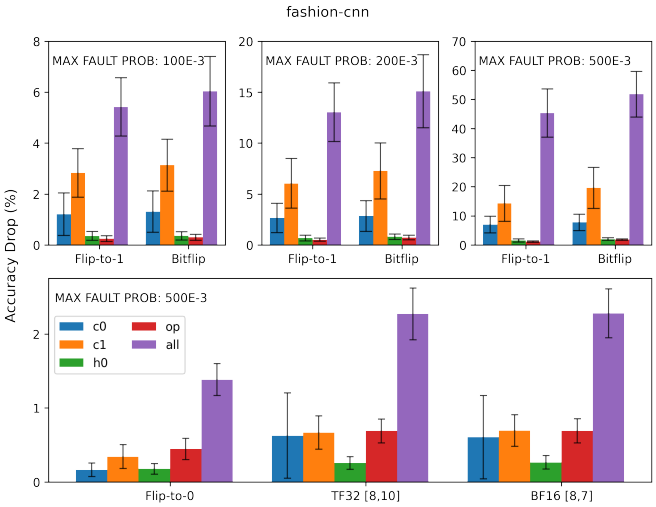

Fig. 6: *fashion-cnn* is more robust to errors. At FR_{max}=500e−3, *c0* and *c1* are most affected causing significant degradation.

followed by normalization. When the elements of the vector have similar values, the softmax operation is sensitive to the precision. Mantissa truncation reduces this precision and therefore introduces significant error into the final class scores. On the other hand, exponent bit errors result in very large changes that overwhelm the softmax output. However, since the *op* layer multiplication for *mnist32-cnn* requires only a few PEs, the overall effect of exponent bitwise errors is not very pronounced.

2) fashion-cnn: The *fashion-cnn* model has two convolutional layers. It is trained with multiple dropout layers and therefore it is more robust and can tolerate error rates almost two orders of magnitude higher than *mnist32-cnn* (Figure 6). Similar to *mnist32-cnn*, the convolutional layers *c0* and *c1* are most sensitive to Flip-to-1 and Bitflip errors. We reason that *c1* is more sensitive than *c0* due to larger kernel size and the more kernel reuse. Although the degradation due to only one of the convolutional layers is not very much, when both *c0* and *c1* have errors, the degradation can be very high.

The *fashion-cnn* model is very robust to mantissa truncation. Even at error rates as high as 0.5, the degradation is only a few percentage points. This indicates that the precision of PEs can be aggressively reduced without significant performance degradation.

B. Performance Recovery with FAWS

Now that we have identified the which layers are most sensitive to which errors, we use our GA-based *FAWS* and observe what performance can be salvaged. We observe from Figures 4 to 6 that some layer-error combination scenarios have very little degradation. Our evaluations show that performance recovery using FAWS on these layers has very little effect. Thus, for the sake of brevity we only discuss the performance of *FAWS* in the following layers. Of course, we cannot expect the complete performance restoration on faulty hardware but as we shall see, *FAWS* comes pretty close.

- *c0* layer in *mnist32-cnn* for Flip-to-1 and Bitflip errors (FR_{max} = 1e−3, 2e−3, 5e−3).
- *h2* and *op* layers in *mnist32-cnn* for mantissa truncation errors - TF32 and BF16 (FR_{max} = 100e−3, 200e−3, 500e−3).
- *c0* and *c1* layers in *fashion-cnn* for Flip-to-1 and Bitflip errors (FR_{max} = 500e−3).

1) mnist32-cnn: The stacked bar plots in Figure 7 show the performance degradation due to Flip-to-1 and Bitflip errors in *c0* and the subsequent recovery by *FAWS* using the shuffling order obtained from GA search.

We observe that *FAWS* is able to recover almost 30% of the lost accuracy (from ∼55% to ∼85%) when FR_{max} = 5e−3. This is a huge gain in performance by simply shuffling the rows. For lower fault rates, *FAWS* is able to recover almost all lost performance. If we refer to [1], fault rates of 1e−3 correspond to about 50% power savings. Thus, by using *FAWS*, we can aggressively lower SRAM voltages and *gain 50% power savings with almost no loss in model accuracy or latency.* This is quite significant, especially when all it costs is to shuffle the rows of the tensors before and after multiplication.

The performance degradation due to mantissa truncation and recovery using *FAWS* for *h2* and *op* layers is shown in Figure 8. *FAWS* recovers 2-4% accuracy points for *h2* and *op* layers individually. When both layers undergo mantissa truncation, it is possible to recover almost 5% accuracy points (*h2-op*). While this is not as dramatic as the 30% recovery for *c0* layer, it still shows that it is possible to recover from some performance degradation when executing models in heterogeneous hardware with mixed precision. It is worth noting we have used extremely unrealistic and severe truncation errors

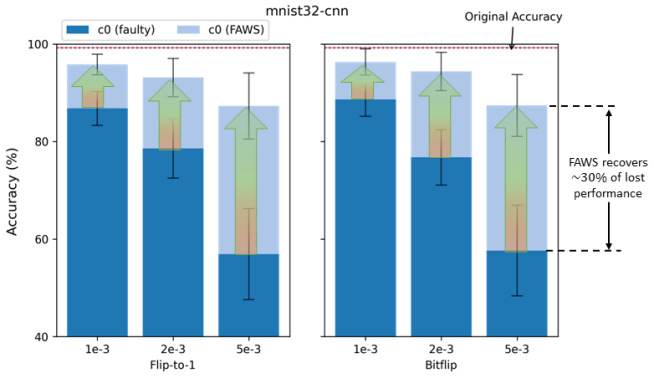

Fig. 7: *FAWS* recovers model performance by as much as 30% in *mnist32-cnn* models for Flip-to-1 and Bitflip errors.

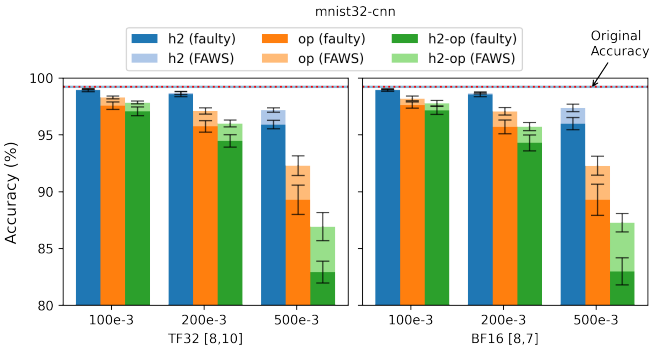

Fig. 8: *FAWS* can recover model accuracy when using reduced precision without any retraining or sophisticated hardware-in-loop optimization

for the sake of generality. During actual implementation, PEs do not sporadically change their precision and they are not randomly scattered throughout the hardware. Rather, they are arranged in a structured manner with controllable deterministic precision scheduling. In such a case, we can expect GA to find a much more optimized solution for much higher performance recovery.

2) fashion-cnn: The *fashion-cnn* model is much more robust than *mnist32-cnn*. Significant degradation is observed only when error rates reach as high as 500e−3 for Flip-to-1 and Bitflip errors. This translates to power savings of approximately 60%. We use *FAWS* to recover some of the lost performance due to errors in *c0* and *c1*, which is shown in Figure 9. We observe that *FAWS* is able to recover about 5% accuracy for each layer individually and around 10% when errors are present in both layers (*c0-c1*). When errors are present only in *c0*, we can expect almost full performance recovery. Even at such high error rates, we can expect *FAWS* to recover the performance of the model by simply shuffling the rows. At "lower" fault rates (100e−3, 200e−3) there is little degradation, so using *FAWS* doesn't have any significant improvement.

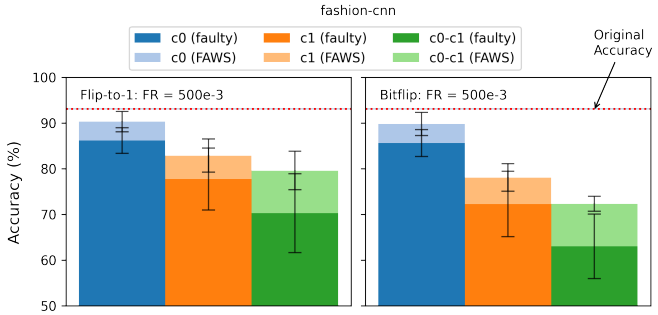

Fig. 9: Even with fault rates as high as 0.5, *FAWS* can still recover upto 10% of performance for *fashion-cnn* model.

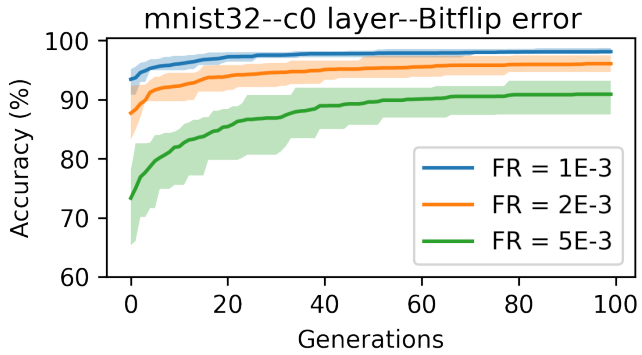

Fig. 10: GA convergence rates for different fault profiles. The rate differs with types of layers, models and fault profiles.

C. GA Convergence

Figure 10 shows how the accuracy of the model improves non-decreasingly as we increase the number of generations for GA optimization. This means that it is possible to get better chromosomes if we let the GA optimization run for longer period of time. This is a major advantage over random search where it is not guaranteed that the solutions will get better as the search progresses. The designer can also decide when to stop the GA optimization. Furthermore, we see that the starting points and the rate of improvement is different for different fault rates, layers and models. Thus, the designer is free to choose different GA parameters when optimizing for different layers/error types and fault profiles.

VI. Conclusion

It is possible to extract significant energy gains and reduced latency for DNN computations by using faulty/marginal hardware. Our proposed *FAWS: Fault-Aware Weight Scheduler* allocates non-critical computations of DNNs to compromised PEs to minimize the performance degradation. It achieves this by shuffling the rows of the matrices during matrix multiplication. The row-shuffling order is determined using GA-based search. With *FAWS*, we are able to recover up to 30% of classification accuracy for fault rates which correspond to power savings of approximately 50%.

ACKNOWLEDGMENT

This work was supported in part by the JST MIRAI Program Grant Number JPMJMI18E1.

REFERENCES

[1] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 267–278.

[2] 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 *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.

[3] B. Salami, E. B. Onural, I. E. Yuksel, F. Koc, O. Ergin, A. C. Kestelman, O. Unsal, H. Sarbazi-Azad, and O. Mutlu, "An experimental study of reduced-voltage operation in modern fpgas for neural network acceleration," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2020, pp. 138–149.

[4] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "Hardware-centric automl for mixed-precision quantization," *International Journal of Computer Vision*, vol. 128, no. 8, pp. 2035–2048, 2020.

[5] S. Venkataramani, V. Srinivasan, W. Wang, S. Sen, J. Zhang, A. Agrawal, M. Kar, S. Jain, A. Mannari, H. Tran *et al.*, "Rapid: Ai accelerator for ultra-low precision training and inference," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 153–166.

[6] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan, "Axnn: Energy-efficient neuromorphic systems using approximate computing," in *2014 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 2014, pp. 27–32.

[7] Q. Zhang, T. Wang, Y. Tian, F. Yuan, and Q. Xu, "Approxann: An approximate computing framework for artificial neural network," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2015, pp. 701–706.

[8] X. He, L. Ke, W. Lu, G. Yan, and X. Zhang, "Axtrain: Hardware-oriented neural network training for approximate inference," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2018, pp. 1–6.

[9] M. B. Sullivan, N. Saxena, M. O'Connor, D. Lee, P. Racunas, S. Hukerikar, T. Tsai, S. K. S. Hari, and S. W. Keckler, "Characterizing and mitigating soft errors in gpu dram," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 641–653.

[10] B. Nie, D. Tiwari, S. Gupta, E. Smirni, and J. H. Rogers, "A large-scale study of soft-errors on gpus in the field," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 519–530.

[11] G. Shi, J. Enos, M. Showerman, and V. Kindratenko, "On testing gpu memory for hard and soft errors," in *Proc. Symposium on Application Accelerators in High-Performance Computing*, vol. 107, 2009.

[12] C. Torres-Huitzil and B. Girau, "Fault and error tolerance in neural networks: A review," *IEEE Access*, vol. 5, pp. 17 322–17 341, 2017.

[13] NVIDIA, "Cuda c++ programming guide." [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[14] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 675–678.

[15] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, vol. 2, 2010.

[16] H. Xiao, K. Rasul, and R. Vollgraf. (2017) Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms.