Making Convolutions Resilient via Algorithm-Based Error Detection Techniques

Siva Kumar Sastry Hari, Michael B. Sullivan, Timothy Tsai, and Stephen W. Keckler
NVIDIA Corporation

Abstract—The ability of Convolutional Neural Networks (CNNs) to accurately process real-time telemetry has boosted their use in safety-critical and high-performance computing systems. As such systems require high levels of resilience to errors, CNNs must execute correctly in the presence of hardware faults. Full duplication provides the needed assurance but incurs a prohibitive 100% overhead. Algorithmic techniques are known to offer low-cost solutions, but the practical feasibility and performance of such techniques have never been studied for CNN deployment platforms (e.g., TensorFlow or TensorRT on GPUs). In this paper, we focus on algorithmically verifying Convolutions, which are the most resource-demanding operations in CNNs. We use checksums to verify convolutions, adding a small amount of redundancy, far less than full-duplication. We first identify the challenges that arise in employing Algorithm-Based Error Detection (ABED) for Convolutions in optimized inference platforms that fuse multiple network layers and use reduced-precision operations, and demonstrate how to overcome them. We propose and evaluate variations of ABED techniques that offer implementation complexity, runtime overhead, and coverage trade-offs. Results show that ABED can detect all transient hardware errors that might otherwise corrupt output and does so while incurring low runtime overheads (6-23%), offering at least 1.6× throughput to workloads compared to full duplication.

1 Introduction

Following recent improvement in the ability of Convolutional Neural Networks (CNNs) to perform complex tasks with high efficiency and accuracy, CNNs have made their way into safety-critical and High Performance Computing (HPC) systems. For example, autonomous vehicles (AVs) employ CNNs to perform complex tasks such as vehicle, cyclist, pedestrian, lane, road-sign, and free-space detection [4], [38]. HPC systems also employ CNNs for object classification and detection, image segmentation, and video analytics for application domains such as healthcare, climate analysis, and surveillance, and often in real-time settings [22], [33]. As the compute throughput and power efficiency demands of the CNN-based safety-critical and HPC systems are high, efficient platforms are being designed to meet the throughput demands within limited power budgets [26], [45]. For example, the recently released NVIDIA DRIVE AGX Xavier System-on-Chip and T4 GPU deliver up to 32 and 130 TOPS while consuming just 30 and 70 watts of power, respectively [33], [38].

Safety-critical and HPC systems must be designed to tolerate hardware errors such as those originating from hardware transient, intermittent, and permanent faults. Some market segments require systems to meet strict safety standards, such as the ISO 26262 functional safety standard for AVs [18]. This standard requires the system to be robust to single-point transient, intermittent, and permanent faults either by design or by coverage from safety procedures (such as ECC and parity). The level of robustness a hardware component desires to obtain is determined by the Automotive Safety Integrity Level (ASIL). For ASIL D (the highest safety level), the system is required to be robust to $\geq$99% of faults; the requirements for ASIL C and B are 97% and 90%, respectively [30]. The rate of residual failures, measured in Failure In Time or FITs (where 1 FIT refers to 1 failure per billion hours), must also be $\leq$100 and $\leq$10 FIT for ASIL B/C and D, respectively. While the HPC market also demands high resilience, the requirements are not as rigorous as those of ISO 26262 [44].

With the increasing prevalence of CNNs in safety-critical and HPC systems and the resilience requirements of such systems, correctness of CNNs must be assured in the presence of hardware faults for safety and standards-compliance. Prior studies have analyzed the effects of hardware errors on CNN outputs and observed noticeable corruptions that must be mitigated to ensure safe operation [23], [40]. Processors deployed in such systems employ ECC and/or parity in major SRAM structures. This protection is typically not sufficient to meet the requirements of ASIL B, C, or D for all the hardware error sources. Aggressive employment of ECC/parity on flip-flops and small SRAM buffers comes at an area cost and may still be insufficient for all error sources due to the error rate contribution from non-storage elements. For intermittent and permanent faults, non-storage elements contribute significantly towards the total error rates in GPUs and DNN-accelerators that dedicate significant chip area to logic [9]. Full hardware redundancy can provide the needed safety [2], [19], [46], but it reduces the throughput by 2× or more, which is prohibitive for resource-constrained systems. The goal of this paper is to develop a low-cost CNN-specific resilience solution that allows the full system to meet the target markets' requirements, while incurring far lower overheads compared to full duplication.

Over 90% of the computation during CNN inference and training is in convolutions [24]. Algorithmic methods have been devised to speed up the convolution operation. These methods include using General Matrix Multiplication (GEMM), Fast Fourier Transformation (FFT), and Winograd [45]. Fault tolerance approaches that leverage algorithm knowledge, known as algorithm-based fault tolerance (ABFT), have been shown to provide lower overheads for GEMM and FFT than full duplication algorithms [15], [17]. These techniques leverage the fact that these operations are linear and verify the correctness using a checksum-based approach. These techniques compute checksums for input data, store them with the original data, perform the original and redundant computation, verify

outputs, and possibly correct errors. The number of extra compute operations they introduce is a small fraction of the original computation. While prior ABFT implementations have achieved runtime overheads of about 20% for square matrices [13], our analysis shows that the overheads can be much higher (>50%) for the non-square matrices that are typically used in CNNs. The main sources of overheads include running the larger GEMM, managing storage to store checksums and associated data movement (especially if they are stored with the input data), and computing the checksums online for the input and output matrices.

While ABFT techniques aim to correct errors inline along with detection, the correction capability is limited (e.g., only a single cell error in a row or column can be corrected) and there is no evidence that it is sufficient for real hardware errors. Moreover, detecting an error to prevent silent data corruption (SDC) is more important to safety-critical and HPC systems than the ability to correct it inline. Upon error detection, a low-cost local recovery mechanism can be invoked that either restores the system state [8] or reruns the operation on the same or a spare resource. For rare locally-unrecoverable errors, a heavy-weight fallback mechanism can be invoked (e.g., transition to a degraded mode of operation [39]) to handle the detected error.

Based on this observation, we focus on algorithm-based error detection (ABED) techniques in this paper and avoid additional costs associated with inline error tolerance. Since convolution is also a linear operation, like GEMM, and a similar checksum-based solution can be applied at the convolution level. A recent study explored using such a solution to detect errors in CNN accelerators (via hardware modifications) with the goal of overclocking the system [28] and detecting corruptions online. It proposed using checksums for filters and input feature maps (two inputs to the convolution) to verify the checksum of the output.

CNN inference deployment platforms (e.g., TensorRT, TensorFlow, PyTorch) are commonly used in safety-critical and HPC systems [33], [38]. Employing an algorithmic resilience technique in such platforms for seamless application across architectures (e.g., CPUs, GPUs, or accelerators) is desirable. However, several feasibility-, performance-, and coverage-related challenges remain. (1) The increasing use of reduced-precision data types (e.g., 8 and 4-bit integers) in CNNs introduces new challenges for checksum-based error detection techniques. For example, the checksum calculations can overflow without careful design. (2) Convolution, a linear operation, is often fused with subsequent activation layers, which are non-linear operations, to reduce data movement and improve performance [33]. Checksum-based techniques do not not apply to the non-linear computations and separating the linear and non-linear computations into different operations can incur high overheads, introducing additional design challenge.

We address overflow in checksum calculation and storage such that no information is lost, and full error detection capability is maintained. We also present implementation options that enable ABED techniques to function with the fused (linear and non-linear) layers. We not only addressed the above two challenges for the previously proposed ABED algorithm that employed checksums for both filter and input feature map checksums [28], but also proposed and analyzed two other variations that use checksums just for filters and input feature maps, respectively. We leverage domain knowledge of CNNs to simplify memory management to store checksums and reduce the number of online tasks.

We study the implementation complexity, runtime overhead, and resilience related trade-offs offered by the three ABED techniques. For architecture that protect the memory subsystem well with ECC/parity, a filter-only checksum-based ABED technique offers a lower-overhead solution when used with commonly-used CNN pruning optimization [16], [29]. The filter and input feature map checksum-based technique provides high coverage for architectures that do not sufficiently protect memory subsystem with low overheads even when deployed without CNN pruning optimization. Our results show that ABED can eliminate all transient-error-induced convolution output corruptions with low (6-23%) runtime overhead on state-of-the-art GPUs, offering at least 1.6× throughput improvements to workloads compared to full duplication.

2 Background

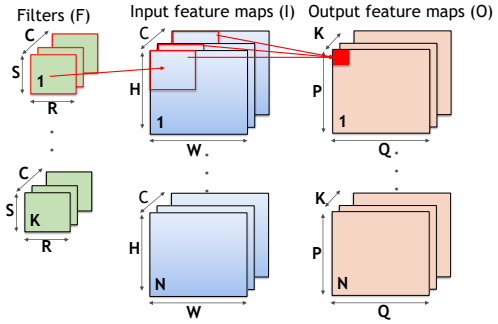
2.1 Related Work

Full Hardware/Software Redundancy: Traditional business-class systems (e.g., IBM Z Series machines [5]) employ expensive hardware-managed dual- or triple-modular redundancy schemes. In safety-critical systems, similar techniques are being employed to meet the highest-levels of safety integrity requirements [2], [19], [46]. Software techniques have been explored that introduce redundancy at different granularities including the process, GPU kernel, thread, and assembly instruction level [12], [27], [47], but they all incur overheads that are high for resource-limited real-time systems.

Algorithm-Based Fault Tolerance (ABFT): ABFT techniques leverage algorithmic knowledge to detect and correct errors with very low overheads and they have been heavily studied in literature for GEMM and FFT [13], [15], [17], [25]. For GEMM, these approaches introduce row checksums for one of the matrices and column checksums for the other. When the matrix multiplication operation is performed with the checksums, an output matrix is produced with an extra row and column. Each of these extra values are expected to match the checksums of the rows and columns of the output matrix. In an event of an error, these techniques can localize the error and use redundant information to correct certain types of errors. Several algorithmic techniques have been proposed to increase the capability of correcting output matrix cell corruptions with little emphasis on studying how low-level hardware faults manifest at that level [7], [48]. Hence it is unclear whether the benefits of the additional correction capability outweigh the costs of higher overhead.

A recent study showed that protecting GEMMs in CNNs via ABFT can provide high protection [10]. While no runtime overhead analysis was included, the paper acknowledged that GEMM kernels are tuned to fully use caches and registers, and adding an extra dimension for checksum storage would not only compromise execution time, but also significantly increase data movement and memory latency. Our experiments confirm this hypothesis and show that the ABFT's overheads for non-square matrices commonly used in CNNs is high (>50%).

Our results show that ABFT's error correction capabilities often introduce higher overheads. A related study suggests that error detection and re-computation can be cheaper than checksum-based ABFT techniques [1]. Moreover, the



Convolution: $O_{n,k,p,q} = \sum_{s=1}^{S}\sum_{r=1}^{R}\sum_{c=1}^{C}\left(F_{k,c,s,r} \times I_{n,c,p+r,q+s}\right) \forall n \in N, k \in K, p \in P, q \in Q$

Fig. 1. A typical convolution operation used by most CNNs.

effectiveness of the ABFT's correction capability is not established for real hardware errors because they may not manifest as correctable (single-cell output) corruptions.

Based on a similar observations, researchers proposed using a checksum-based ABED technique [28] to detect errors during a convolution. Since convolution is a linear operation like GEMM, a checksum-based error detection technique can be extended to it as well. This work used checksums for both the filters and input feature maps to verify the output. The goal of the paper was to overclock a CNN accelerator and detect incorrectly computed convolutions. It extended the hardware accelerator to include the detection technique. Several challenges arise while applying ABED to convolutions in optimized CNN inference platforms (e.g., TensorRT, TensorFlow, PyTorch) commonly used in safety-critical and HPC systems. (1) The use of reduced precision operations is common during inference and without proper care, the checksum arithmetic can overflow. (2) A convolution operation is often fused with a non-linear activation layer, to reduce data movement and improve performance [33]. Checksum-based techniques apply only to the linear operations. Separating the linear and non-linear computations into different operations can incur high overheads due to additional data movement, introducing additional implementation challenge. (3) Without customizing ABED to the CNN inference frameworks, online checksum storage management and generation for the output and both the inputs on every convolution introduces avoidable runtime overheads (explained further in Sections 5.3 and 6.3).

We not only address these challenges in this paper, but also identified and explored two other ABED variants (not previously studied) that use checksums either for filters or input feature maps (explained in Section 3). These variants offer interesting error coverage and performance trade-offs, important in selecting an optimal solution for a target safety-critical system.

2.2 The Convolution Operation

A convolution operation takes two input tensors and produces one output tensor. One of the input tensors is for the input feature maps (or fmaps); these fmaps are either the output of the previous layer or the input to the network. Input fmaps are represented as a 4-D tensor in most CNNs. Each feature map is a 2-D tensor with height (H) and width (W). Typically, many feature maps are stacked to form a 3-D tensor. The number of channels (C) defines the number of feature maps in the stack. Many 3-D input fmaps are batched (batch size = N) together to form the 4-D feature map tensor. The other input tensor is the set of filters, which consists of weights that are computed during the training process. Each filter is a 3-D tensor of weights with dimensions height (S), width (R), and channels (C). Each convolution layer has multiple filters (number of filters = K), adding an extra dimension to produce a 4-D tensor.

Each output feature map value is produced by performing a dot-product between a filter and a same-sized portion of the input fmap's tensor. An example is shown in the highlighted cells in Figure 1, along with the formula to compute each of the output fmap values. As one filter produces one output feature map, the number of channels (feature maps) in the output is the same as the number of filters (K). The number of output fmaps is the same as the batch size (N).

3 Convolution ABED Approach

Verifying every output value of a convolution might require duplicating the entire operation. Instead, the focus of this work is on verifying just the reduced output, i.e., sum of all the output elements. This reduced output can be computed from the inputs directly with far fewer computations. We essentially use a different sequence of sums and products. Since integer sum and product operations are commutative and associative, changing the order of the operations is not a concern. Based on this key insight, we explore the following three schemes to verify a convolution, which are summarized in Figure 2.

3.1 Filter Checksum-Based (FC)

In this scheme, a 3-D filter checksum tensor is computed by performing an element-wise sum (using sum as a checksum function) across all the 3-D filter tensors (❶ in Figure 2(a)). This new checksum filter is convolved with the input fmaps to compute an extra output fmap, which is used to verify the original fmaps' values. The original output fmaps' values are reduced across the channel dimension to generate a reduced fmap, which is compared element-wise for equality with the extra output fmap for verification (❸ in Figure 2(a)). This method protects the computation involved in the convolution, the data storage and transportation of filters (between DRAM/L2 and registers) and output fmaps, but not the storage and transportation of input fmaps. This scheme increases the number of operations in the convolution by a factor of 1/K and introduces PQNK operations for output verification. Here PQNK refers to P×Q×N×K. We omit the multiplication symbols while referring to products of the parameters of the convolution for brevity.

3.2 Input Checksum-Based (IC)

This scheme uses checksums of input fmaps, which can be computed in one of two ways: (1) summing input fmaps' values element-wise across batches to add a new checksum batch, and (2) summing elements of the portions of the input fmaps that are used to perform the dot-product with the filters during the convolution operation to produce a tensor that is the same size as the filter.

The first option effectively increases the batch size by one. The original output fmaps' values are reduced across all the original batches to generate a batch of checksum output fmaps, which is compared for element-wise equality with the

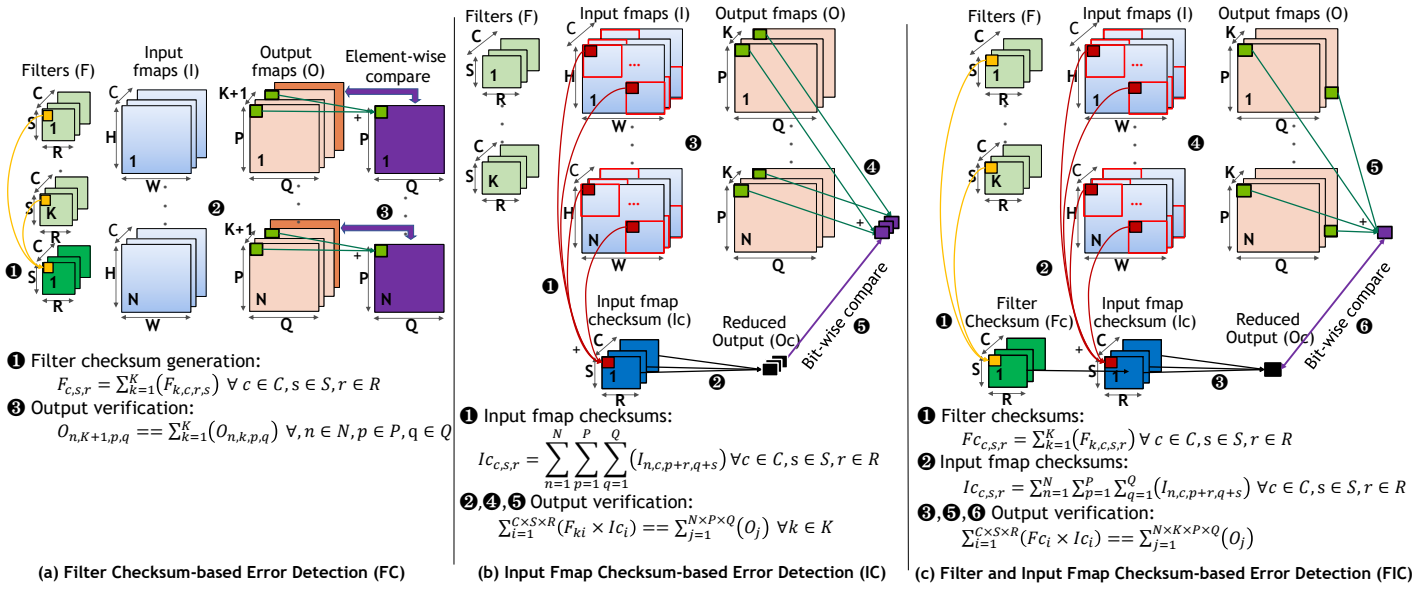


Fig. 2. A depiction of the filter-only (FC), input fmap-only (IC), and filter and input fmap (FIC) checksum-based error detection schemes. Formulae to generate the checksums and verify output fmap values are also shown.

extra batch of output fmaps generated using the checksum input fmaps for verification. This option is attractive if the batch size is large because the effective overhead of running the larger convolution would be small. However, for small batch sizes, which is common in safety-critical systems [33], [42], this option can result in high overheads.

The second option reduces the input fmaps into a separate checksum tensor, which is the same size as a filter for the layer (❶ in Figure 2(b)). This checksum tensor is then convolved with K filters to produce exactly K values. The output fmaps are reduced across height, width, and batch dimensions and then compared with these K values element-wise for equality for verification (❹ and ❺ in Figure 2(b)). This method protects the computation involved in the convolution, the data storage and transportation of input and output fmaps, but not the storage and transportation of the filters. The number of additional computations needed for the convolution is CRSK, input fmap checksum generation is PQNCRS, and output fmap checksum generation for verification is PQNK.

3.3 Filter and Input Fmap Checksum-Based (FIC)

The scheme, similar to the one proposed by prior work [28], creates checksums for both the filter and input fmaps (❶ and ❷ in Figure 2(c)). Using the two checksums, we perform an extra convolution (❸ in Figure 2(c)). This operation can also be implemented as a vector-vector dot-product because the filter checksum size is same as the input fmap checksum size. This operation produces a single value, which is used to verify the original computation. The original convolution is run with the original parameters and the output is reduced to a single value, which is verified using the value generated by the dot-product. (❹, ❺, and ❻ in Figure 2(c)). This method protects the computation involved in the convolution and the data storage as well as the transportation of filters and the input and output fmaps. The number of additional computations needed for dot-product is CRS, input fmap checksum generation is PQNCRS, and output fmap checksum generation for verification is PQNK.

3.4 Trade-offs

Table 1 summarizes the trade-offs offered by the FC, IC, and FIC techniques in terms of the number of tasks that must be performed online and the protection they provide. The table shows that FIC offers better protection than FC and IC by protecting the storage and data movement of both the filters and inputs. The filter checksums can be generated offline because the weights are known before a CNN is deployed. However, the input and output checksum generation and verification tasks must be performed online. Since the online tasks needed for the FIC and IC techniques are similar, the runtime overheads also expected to be similar. Given that FIC offers superior protection compared to IC but the runtimes are expected to be similar, we do not investigate IC further.

Since FC must run a larger convolution, the overhead can be higher than FIC. However, FC can be faster if the larger convolution adds minimal overheads, which is possible with the use of network pruning techniques [16], [29]. Network pruning improves network performance by identifying and removing filters that contribute minimally to the accuracy of the network. With the use of pruned networks, the number of filters per layer may diminish so that adding the checksum filter introduces minimal overheads (explored further in Section 6).

4 Implementation

This section addresses challenges that arise while implementing ABED on a GPU-based system. Specifically, we explain (1) how to maintain high coverage by avoiding overflow while using reduced-precision operations and storage, the use of which is prevalent in inference platforms, (2) task-fusion based optimizations/modifications we propose to the highly-optimized inference platforms to minimize the overheads introduced by ABED, and (3) modifications needed to the inference deployment frameworks for seamless integration with ABED.

TABLE 1
Trade-offs between the FC, IC, and the FIC techniques. Entries marked Yes/Offline and No/Online are favorable and unfavorable, respectively.

	Criteria		FC	IC	FIC
Additional Work	Filter checksum generation		Offline	-	Offline
	Input fmap checksum generation		-	Online	Online
	Avoid running a larger convolution		No	Yes	Yes
Protects	Computation		Yes	Yes	Yes
	Storage and transportation	Filters	Yes	No	Yes
		Input fmaps	No	Yes	Yes

TABLE 2
The bit requirements to accurately represent the results while verifying intb (e.g., int8 for b=8) convolutions.

	FIC	FC
Input fmaps	b	b
Input fmap checksum	b+log_2(K)	-
Filters	b	b
Filter checksum	b+log_2(PQN)	b
Output fmap	2×b+log_2(CRS)	2×b+log_2(CRS)
Reduced output fmap	2×b+log_2(PQNKCRS)	2×b+log_2(CRSK)
Dot-product output	2×b+log_2(PQNKCRS)	-

4.1 Handling Reduced-Precision Operations

The use of reduced-precision fixed-point data types has been explored both in research as well as many commercial products. For example, 8-bit integer arithmetic is supported in Google's Tensor Processing Unit, NVIDIA's Volta and Turing GPUs, Intel Xeon Scalable Processors, and ARM CPUs [3], [21], [32], [41]. Fixed-point arithmetic suffers from overflow if the result does not have sufficient bits to represent the full value. Here we describe a method to ensure full error coverage while using reduced-precision fixed-point data types.

Convolutions that use 8-bit integers (int8) store the filters and input fmaps using int8 data types. Each output fmap value is obtained by performing a dot-product using one filter (of size CRS) with a same sized portion of the input fmap, illustrated by the highlighted portion in Figure 1. In this operation, CRS 16-bit values, each of which is a product of two int8 values, are summed together. Making no assumption about the values, the result can be accurately represented using $[16+log_2(CRS)]$-bit integers. For most practical values of C, R, and S ($CRS \leq 64K$), int32 is sufficient to avoid overflow during convolutions.

We use two's complement integer summation as the checksum function. To avoid overflow during checksum generation, we use int32 operations. For the FC technique, we store the int32 checksums as a tuple consisting of up to four int8 values, creating up to four checksum filters. No information is lost with this scheme. The extra output fmap values are shifted left by 1, 8, 16, and 24, respectively, and added together across the channel dimension. These values are then compared with the output fmap checksums, which are obtained by summing the original output fmap values across the channel dimension (K additions). The reduced result can be accurately represented using $[16+log_2(CRSK)]$-bit integers. For most practical values of C, R, S, and K, 64-bits are sufficient.

For the FIC technique, the filter checksum is obtained by summing K int8 filters and can be accurately represented by $[8+log_2(K)]$-bit integers. Each input fmap checksum value is computed by summing PQN int8 input fmap values, requiring up to $[8+log_2(PQN)]$ bits. The result of the convolution of the checksums would require up to $[16+log_2(PQNKCRS)]$ bits. For most practical purposes, int32 and int64 are sufficient to store the checksums and convolution results, respectively.

Table 2 summarizes the maximum number of bits needed to accurately represent the values at different points during the convolution operation for the FC and FIC techniques based on worst-case overflow analysis. We assume unsigned integers for this analysis. The requirements can be slightly less for signed integers. For example, the result of multiplying two signed 8-bit integers (with 1 sign bit) can be accurately represented using 15-bit signed integers (with 1 sign bit). Since all parameters are known prior to a neural network deployment, the ABED precision requirements can also be determined. For the networks used in this paper, int64 checksums are sufficient to avoid overflow. If the precision requirements increase beyond int64, modular arithmetic can be used to limit the highest precision operation used by the verification kernel (to reduce runtime overhead) with some loss in coverage, which we do not explore in this paper.

Floating-point arithmetic suffers from both overflow and rounding. While we explored ways to address these issues, the discussion to maintain high error coverage using floating-point data types is deferred to Section 7, as commercial implementations increasingly use fixed-point arithmetic due to its performance and energy advantages.

4.2 Framework Modifications

Once a neural network is trained, it is optimized and prepared for deployment using a platform for high-performance inference (e.g., TensorRT). The optimizations involve pruning, weight and activation precision calibration (also known as quantization), layer and tensor fusion, and kernel auto-tuning. These optimizations are typically performed once to create an inference engine, which is then serialized to avoid preparation overheads. We perform the following during the optimization step. (1) We create checksum filters and store them along with the filter tensor and in separate storage for the FC and FIC techniques, respectively. (2) We introduce all the additional online tasks that should be executed during inference as part of the ABED scheme (e.g., input and output checksum generation). Figure 3 summarizes the proposed changes for seamless integration of ABED in a tool-chain used to deploy trained models for inference. ABED is independent from all optimization described above, except the layer and kernel fusion. We next describe how ABED can be applied to highly-optimized convolutions that are fused with subsequent layers in CNNs.

4.3 Kernel Modifications

As described in Section 2, common convolution operations take two 4-D tensors as inputs, one each for input fmaps and filters (I and F) and produce a 4-D tensor of output fmaps (O). Convolution, bias, and activation operations are typically fused together for performance. Such fused operations perform $O = activation(conv(x) + bias)$. For int8 convolutions, I and F use int8, and O uses either int8 or fp32. Figure 4 explains the logical flow of computation within such fused kernels. For int8 convolutions, the output of the convolution operation

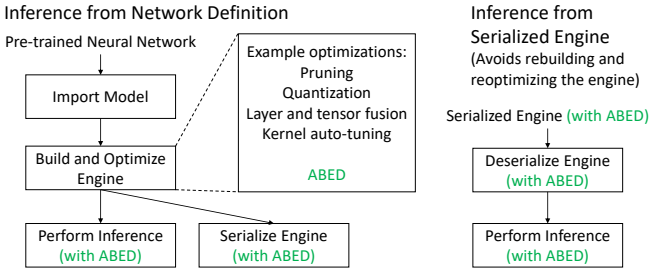


Fig. 3. Inference deployment steps and where ABED can be included for seamless integration.

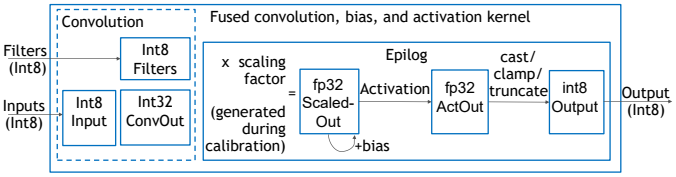


Fig. 4. The logical computation flow in a fused convolution, bias, and activation kernel.

is an int32 result (ConvOut in the figure). This intermediate result is then scaled using a scaling factor that is generated during the calibration step, which produces an fp32 result (ScaledOut in the figure). This step assumes that the scaled int32 result can be accurately represented using an fp32 data type. Bias is added to ScaledOut. The activation function is then applied to it to produce ActOut (another fp32 value). If this value (ActOut) is too large to be accurately represented using int8 data type, it will be clamped and truncated to produce an int8 output value. We refer to all the operations after the convolution operation as *epilog*.

The ABED techniques verify just the result of the convolution. Hence, the intermediate output (ConvOut in Figure 4) must be verified before the epilog is applied, which can be performed by either using un-fused kernels or fusing some part of the output fmap checksum generation task with the fused convolution + epilog kernel. Figure 5 lists some of the options for the FIC technique. For each option (one row in the table), we list the tasks that must be performed (in columns) and show the unfused/fused kernels that the option will execute. For each of the kernels, we show the data types and sizes of its inputs and outputs.

FIC Technique: The following seven tasks (one more than what is listed in Figure 2(c)) must be performed for the FIC technique to detect errors in the fused convolution + epilog kernel: (1) filter checksum generation, (2) input fmap checksum generation (ICG), (3) dot-product of the filter and input fmap checksums, (4) convolution operation, (5) output fmap checksum generation (OCG), (6) epilog, and (7) verifying the output fmap checksum with the output of step (3). The first task is performed offline, as described above. All other tasks are performed online. Since task (7) involves comparing just two values for bit-wise equality, it can be performed on the host CPU. There are several ways to perform the remaining online tasks on a GPU. A simple option is to use different GPU kernels to perform each of the tasks, shown as option Unfused in Figure 5. The input checksum generation kernel reads the input fmaps in int8 data type of size NHCW and generates an int32 checksum vector of size CRS. The convolution kernel reads in the int8 filters and input fmaps of sizes KCRS and NCHW, respectively, and writes out an int32 output of size NKPQ (ConvOut in Figure 4). The next kernel reads ConvOut and applies epilog to produce an int8 output of the same size as the input. The output fmap checksum generation kernel reads the convolution output again to generate a single int64 checksum value. Lastly, the dot-product kernel reads in filter and input fmap checksums, and two int32 vectors of sizes CRS. It produces a single int64 value, which is compared with the output fmap checksum for equality. This implementation does not protect the epilog output and introduces several additional data transfers including the convolution output stored in int32 data type (4× the size of an equivalent int8 structure).

Task fusion can significantly reduce data movement. While we explored multiple options to (partially/fully) fuse tasks, we describe only two options here for brevity. (1) The convolution, epilog, and output fmap checksum generation kernels can be fused into a single kernel to limit data movement introduced by ABED (FusedOCG in Figure 5). It generates a int64 value for output fmap checksum along with the output of the fused convolution + epilog tasks, i.e., int8 output fmap tensor of size NKPQ. (2) To further reduce data movement and provide coverage for epilog's output, the output checksum generation task can be modified to produce the input fmap checksum for the subsequent layer, if the next layer is a convolution layer (FusedIOCG in Figure 5). It essentially fuses input checksum generation task with the fused convolution + epilog + output checksum generation kernel. This optimization duplicates the epilog, but we assume that data movement saving will improve runtime more than the overhead introduced by duplicating compute-bound epilog. The FusedOCG and FusedIOCG options require changes to the existing fused convolution + epilog kernel offered by frameworks such as cuDNN and TensorRT.

FC Technique: The following are the tasks needed for the FC technique (one more than what is listed in Figure 2(a)): (1) filter checksum generation, (2) convolution operation, (3) epilog, and (4) output fmap reduction and verification. The first task is done offline. The verification task involves computing checksums from the original and extra output feature maps across the channel dimension separately and comparing them for bit-wise equality. This task must be performed before the epilog and can be implemented in multiple ways. The first option is to not fuse the operations and launch separate kernels for the convolution, epilog, and output fmap checksum generation and verification. This option is similar to Unfused for FIC. Fusing the output fmap checksum generation and verification task with the already fused convolution + epilog kernel will reduce data movement. This option is similar to FusedOCG for FIC.

Since this technique adds checksum filters, the runtime of the larger convolution can be higher. The increase can be super-linear in the number of filters for certain architectures and convolution parameters. The convolution implementation is often tiled. The runtime may not increase if a tile boundary is not crossed and can increase significantly if it is crossed. Coordination with the pruning techniques may help in reducing this overhead. When the network is being prepared for deployment, filter-pruning is commonly employed to optimize the network's performance and storage [16], [29]. Reducing the number of filters at this step such that the checksum filters can be included without introducing a new

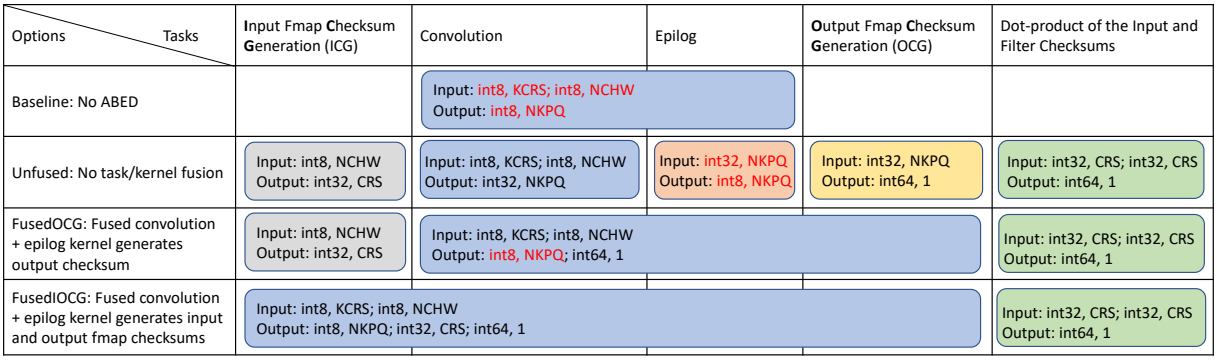


Fig. 5. Implementation options for the FIC technique are shown. Each colored box represents a GPU kernel and shows the data type and size of the inputs and outputs. Inputs/outputs for which the data transportation is not protected are shown in red.

tile of work can reduce the overhead significantly.

The output of the fused operation must be trimmed such that the extra feature maps are ignored by the subsequent layer. Trimming can be fused with the output verification task. We do not explicitly study the effects of trimming because implementing it simply requires skipping some writes.

5 Evaluation Methodology

We evaluate the overheads introduced by the ABED techniques to the convolutional layers from different CNNs. We use VGG16, ResNet18, and ResNet50 for analysis [14], [43]. We evaluate using two different image sizes—224x224, the size of the images in the ImageNet dataset [11], and 1080x1920, the resolution of the images in a full-HD or 1080p video.

5.1 Compute/Data Movement Overheads Estimation

We first analytically evaluate the increase in the compute and data movement operations when we apply the FC and FIC ABED techniques to the networks. In this analysis, we abstract away implementation details and only consider the arithmetic operations such as multiplication, addition, fused multiply-addition (FMA), activation, and type-casting. Similarly, we also count the bytes of data that form the inputs and outputs of different implementations for the FC and FIC techniques, as listed in Section 4.3.

5.2 Runtime Overhead Evaluation

We experimentally evaluate the runtime of convolutions by creating a cuDNN-based workload that sets up, initializes, and runs convolutions in a loop. We compile this workload using CUDA 10 and use cuDNN 7.3 [37] on both a Jetson AGX Xavier system and an x86-based desktop with a V100-based GPU (Titan V) [32], [34]. For performance analysis, we lock the CPU, GPU, and memory clocks on the Jetson board and lock the application clocks on the V100 GPU. We run the convolution (and other operations needed by the ABED techniques) 200 times, recording the average. Since real-time applications (including safety-critical systems) use small batch sizes, we use batch size of two on Jetson and eight on V100-based system [33], [42]. We ignore the first layer in each network because it is not well optimized by cuDNN. We use NHWC memory layout to tensor storage, as int8 cuDNN convolutions are optimized for this layout.

For a baseline, we invoke the fused convolution and epilog kernel provided by cuDNN (called cudnnConvolutionBiasActivationForward). We implement versions that are similar to Unfused for the FIC and FC techniques. As cuDNN does not offer a convolution kernel that takes in two int8 tensors and produces an int32 tensor as output, we employ a version that produces fp32 output. The epilog, when performed separately, invokes two GPU kernels (one each for adding bias and applying activation) and generates a fp32 tensor as output (instead of an int8 tensor, as mentioned in Section 4.3, due to cuDNN API limitations). For analyzing the Unfused implementation options for ABED, we also collect results with just the unfused convoltuion and epilog kernels (without ABED). This version launches one kernel each to perform the convolution operation, add bias, and apply activation. We refer to it as the unfused baseline.

The checksum generation kernels are written in CUDA. Since the checksum generation has similarities to the reduction operation, we use previously-established optimizations. Specifically, we use the functions and primitives (such as DeviceReduce and WarpReduce) provided by the *CUB* library optimized implementations [35]. We try to minimize the use of atomics and leverage faster memory (e.g., registers and shared memory) as much as possible. We ensure that global loads are coalesced across warps and use wide loads per thread (e.g., LD.128). We avoid control flow and use the dot-product instruction (DP4A) whenever possible to avoid a compute bottleneck [36]. We specialize kernels for filter sizes 1 and 3, strides 1 and 2, and for data types int8 and int32. For the FC technique, we add 8 filters (4 for checksums and 4 with zeros for int8) because the cuDNN version we use on our target device chooses efficient kernels when the number of filters is a multiple of 8. We obtain the aggregate runtimes per network by adding all of the GPU kernel runtimes and show the runtime relative to one of the baselines mentioned above.

Effect of Task Fusion-based Optimization: Since we could not modify the closed-source cuDNN kernels to fuse checksum generation and output verification tasks with the convolution for the FusedOCG versions, we model the runtimes. We write separate CUDA kernels to capture the overheads associated with performing the additional work that will be fused with the convolution + epilog kernel. For FIC-FusedOCG, the new kernel fully reduces the output and writes out just one int64 value. Since using atomic operations for reduction can be a performance bottleneck, we hierarchically reduce the output similar

to the prior optimized reduction task implementations [35]. For FC-FusedOCG, the new kernel reduces the values across the channel dimension for verification and sets a flag on an error detection. We measure the runtimes of these kernels by running them on silicon and add the runtimes to Baseline-Fused as an estimate of Kernel 3's runtime (from Figure 5).

5.3 Overhead Analysis of a Traditional ABFT Technique

As convolutions can be implemented using GEMM, ABFT can be employed at the GEMM-level to provide protection. While prior work explained the resilience benefits, no performance assessment was included [10]. We quantify the overheads associated with the ABFT technique for GEMM and highlight the benefits of our ABED solutions to protect convolutions. An ABFT approach performs the following tasks: (1) allocate larger input and output matrices with space for checksums, (2) copy data from the original matrices to the new locations, (3) generate checksums for both the input matrices, (4) run the larger GEMM, instead of the original GEMM, (5) generate both row and column checksums for the output (by reading the output matrix twice) and compare them with the extra row and column values generated by the GEMM, and (6) copy the output matrix to the original location. We measure overheads for tasks (2)-(6) by developing CUDA kernels. We compare the runtime of our implementation of the checksum generation task with the reduction operation provided by CUB [35] and use that whenever it is faster (assuming checksum generation can be as fast as reduction). ABFT can be implemented to store the input checksums separately to avoid copying large matrices. Such an implementation launches separate kernels to perform original GEMM and generate extra output row and column via vector-matrix products, which can be as slow as the original GEMM for some matrix sizes (used in CNNs). Due to such overheads, we do not explore this option in this paper.

5.4 Resilience Evaluation

We evaluate the resilience improvements offered by the ABED techniques using three methods—analytical modeling, input-output error injections, and accelerated-particle beam experiments. The first method uses the same model used above to analyze the increase in compute and data movement operations. Here we analyze the fraction of compute and data movement operations that are protected by the ABED techniques. For the second method, we run the second convolution layer from ResNet18 with input (fmap and filter) values initialized to 1. We perform three error injection campaigns to study the effect of injecting single bit-flips into input fmaps, filters, and output fmaps. For each experiment in a campaign, we randomly flip a bit in a randomly chosen location in the tensor and study whether the ABED approach detects the error.

We conducted two accelerated-particle beam experiments at ChipIr at Rutherford Appleton Laboratory and ICE-II at LANSCE [6], [31] using our implementations of the Unfused option for the FIC and FC techniques and baseline. We excluded epilog in these experiments. The input tensors were initialized to 1. After each convolution we verified the output with the expected golden values that were collected during fault-free runs. Any mismatch was recorded as an SDC. Our investigation suggests that the output tensor was corrupted for some runs after the ABED checks were completed, likely when the output was being reduced to compare against the golden checksum to determine the SDC. These output corruptions are out of the coverage scope for ABED and hence we do not consider them as SDCs in this study. We also recorded whether our ABED scheme was able to detect the error. We tested with NVIDIA Quadro GV100 GPUs with HBM2 ECC always On, and on-chip ECC On and Off.

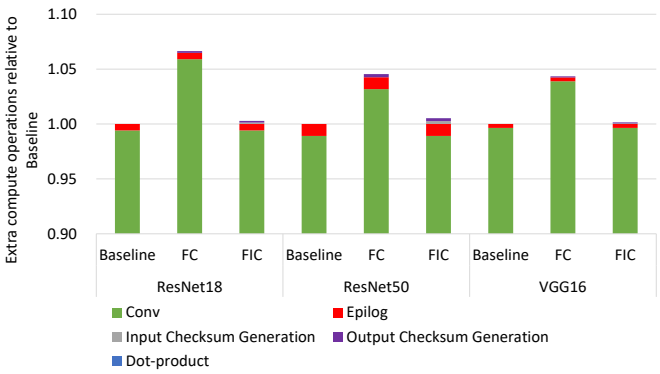


Fig. 6. The increase in the number of logical compute operations for the two ABED techniques relative to the baselines.

6 Results

6.1 Compute/Data Movement Overheads Estimation

We first study the increase in the logical compute and data movement operations based on the model described in Section 5.1. Figure 6 shows a breakdown of the number of arithmetic operations in convolution, epilog, checksum generation, and dot-product of the checksums for the baseline and the FC and FIC techniques. The average increase in the number of operations is small, <7% for FC and <1% for FIC for the studied networks compared to the respective baselines. The increase is relatively higher for the FC technique because it increases the size of the convolution, unlike the FIC technique. Results show that the extra computations added for checksum generation and performing the dot-product of the checksums are significantly less than 1%.

Different implementations listed in Section 4.3 perform the same logical compute tasks, but differ in terms of data movement. Figure 7 shows the bytes of data that form the inputs and outputs of the different implementation options for ResNet18 using two input image sizes. Results for other networks and input sizes show similar trends (not shown for brevity). The figure shows that the fused versions transport significantly less data compared to the versions that do not fuse tasks. Introducing separate kernels for checksum generation and verification introduces more data movement, as is the case for FIC Unfused and FC Unfused. Results also show that the FC FusedOCG requires less data movement than FIC FusedOCG, but FIC FusedOCG offers better data movement protection by using input and weight checksums, while FC FusedOCG protects just the weight storage and movement.

6.2 Runtime Overheads

Figure 8 shows the average runtimes of the Unfused options of the baseline and FC and FIC techniques. Results for the three networks using 1080p inputs are shown here. The results are

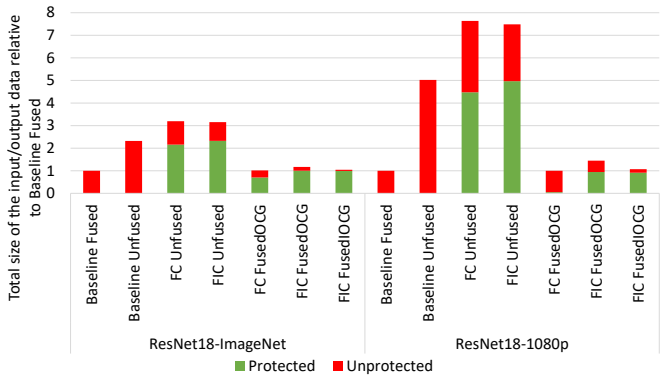


Fig. 7. The relative increase in the data that forms input and output of all the kernels for different implementations of the ABED techniques is shown here.

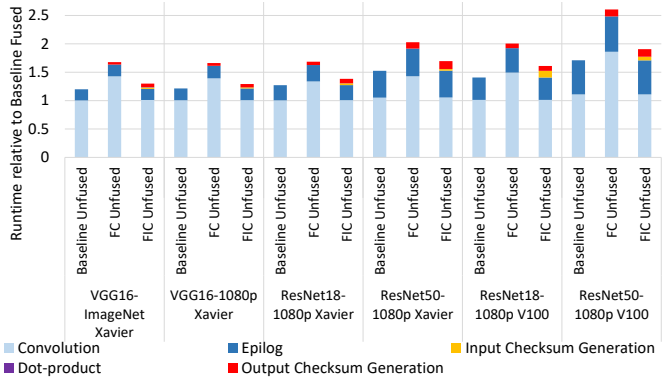


Fig. 8. The runtimes of the Unfused versions of the baseline and FC and FIC techniques for different neural networks are shown here.

normalized to the Fused baseline. The runtime overheads for the FC technique stem from running a larger convolution with additional checksum filters and output checksum generation. For the FIC technique, the overheads stem from running the input and output fmap checksum generation tasks and the dot-product of the filter and input fmap checksums.

FC vs. FIC: Results show that the runtime overhead introduced by the output checksum generation is similar between the FC and FIC techniques. The dot-product kernel used during the FIC technique introduces negligible overhead across all the studied networks and architectures. The difference in overheads between the FC and FIC techniques is mainly due to running the larger convolution versus generating input fmap checksums online. The former introduces higher overheads for all the networks and architectures we studied (Figure 8). The results show that the overheads introduced by the checksum generation and verification tasks are small (4-20%). The overhead introduced by the separate data-movement-heavy epilog is high which can be avoided by the task-fusion-based implementations discussed below.

Model-specific Sensitivities: The overhead of output checksum generation for ResNet50 is higher compared to VGG16 and ResNet18. A primary reason for this difference is that the overhead for verifying 1x1 convolutions is much higher compared to verifying 3x3 convolutions, and ResNet50 uses many 1x1 convolutions while ResNet18 and VGG16 do not use any. The fraction of the work (and data movement) performed for checksum generation to that of the baseline convolution is much higher for 1x1 convolutions compared to 3x3 convolution. Fusing the output checksum generation task with the convolution operation can help reduce the overheads.

Architecture-specific Sensitivities: We show the results obtained from Jetson AGX Xavier (with batch size of two) and a V100-based GPU (with batch size of eight) in Figure 8. The V100-based GPU offers approximately 10× compute throughput and 5× memory bandwidth compared to the Xavier GPU. As expected the baseline work is signficantly faster on the V100-based GPU. Since the throughput to bandwidth ratio is higher for the V100-based GPU, overheads of the memory-bound tasks such as checksum generation and epilog are higher. In fact, the tasks that are not memory-bound in Xavier become memory bandwidth/latency limited in V100.

The runtime overhead of generating input checksums is significantly lower than generating output checksums. The

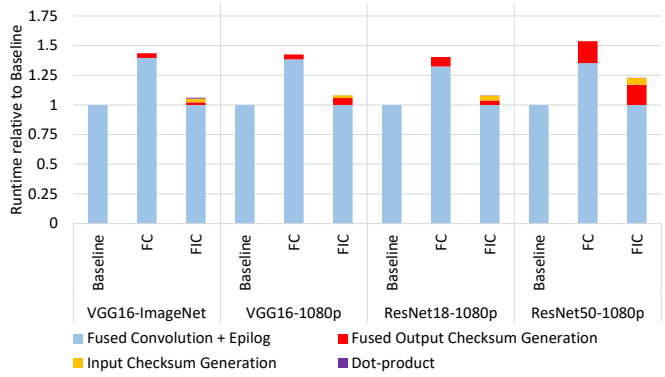


Fig. 9. The runtime overheads of FusedOCG options for the FC and FIC techniques relative to the fused convolution + e05ilog kernel (Baseline) are shown here.

main contributing factor is that the input fmaps are 4× smaller compared to the non-scaled 32-bit integer output fmap values. One exception to this finding is ResNet-18 with image size of 1080x1280 on the V100-based GPU. Input checksum generation for ResNet-18 incurs high overhead because it becomes memory-bandwidth limited on this GPU.

Input-specific Sensitivities: Figure 8 shows the relative runtime for VGG16 with different input sizes (224x224 vs. 1088x1920). Since no significant difference is observed, we do not analyze results with smaller image size for other networks.

Effect of Task Fusion-based Optimization: The runtime overhead results for the FusedOCG optimization for FC and FIC techniques obtained using the methodology described in Section 5.2 are shown in Figure 9. These experiments were run on Jetson Xavier. These results suggest that task fusion is highly effective in reducing the overheads by reducing memory traffic associated with epilog and additional ABED tasks. It shows that the inference-level overheads for the FIC technique (6-23%) are far lower than full duplication. The overheads for the FC technique are higher mainly due to running the larger convolution (with additional checksum filters).

Reducing overheads for the FC technique: In our FC implementations, we increase the filter counts by 8 (as described in Section 5.2). The runtime of the convolution, however, increases disproportionately for some layers. We illustrate this behavior by executing a convolutional layer with a varying number of filters. Specifically, we ran an int8 convolution with 112x112 input fmap, 3x3 filters, 64 input channels, and stride and

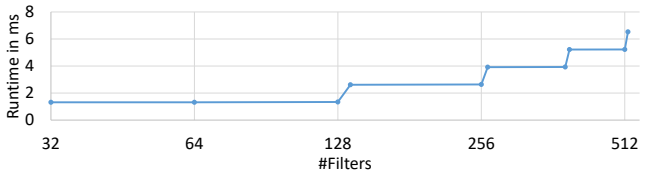


Fig. 10. Convolution runtimes with varying filter counts.

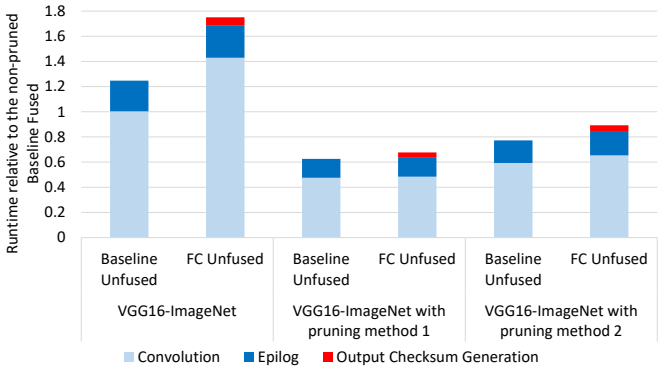


Fig. 11. The runtimes for VGG16 and two pruned versions for the Unfused baseline and FC technique are shown here.

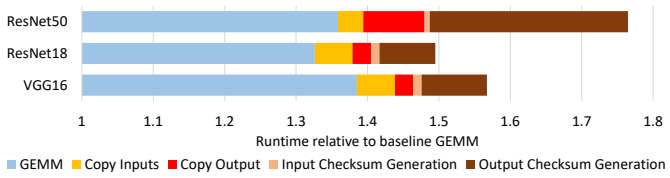


Fig. 12. Runtime breakdown for an ABFT implementation.

padding of 1. We vary the filter counts and show that adding just eight filters can introduce up to 2× overhead. Figure 10 shows these results. Since cuDNN uses GEMM as a method to perform the int8 convolutions and GEMMs use tiling, the sharp increase in the runtime is likely due to the use of an additional tile. While such tiling effects are not a concern for GEMMs with large dimensions, they can be problematic for the commonly used convolutional layer dimensions (where K is small).

Instead of increasing the filter counts of the baseline network, creating space for the filter checksums can eliminate this overhead. As mentioned in Section 3.4, network pruning techniques, which are being adapted as a way to improve network performance, may create space for filter checksums. These techniques identify and remove filters that contribute minimally to the accuracy of the network. With the use of pruned networks, the number of filters per layer may reduce even after adding the checksum filter, the sharp increase in the convolution runtime can be avoided. To test this hypothesis, we conducted an experiment for the FC technique using VGG16 and two pruned versions of the network. We obtain the number of pruned filters per layer from a previously published result. Huang et al., studied two methods to prune the network [16]. The first approach ranks filters on per layer basis, while the second ranks them across all the network. Our results in Figure 11 demonstrate that the overheads from running the larger convolution become small, 2% or 10% for the two pruned versions, respectively, compared to the 42% overhead for the non-pruned version.

6.3 Overhead Analysis of a Traditional ABFT Technique

As convolutions can be implemented using GEMM, ABFT for GEMM (a well known technique) can provide high protection to CNNs. A recent study explained the resilience benefits of such an approach [10]. In this section, we quantify the overheads associated with the ABFT technique for GEMM, as described in Section 5.3, and highlight the benefits of our ABED solutions to protect convolutions in CNNs. Results for the three networks for 1080p input are shown in Figure 12. These results show that running a larger GEMM incurs high overhead (similar to the FC technique). This overhead can be reduced using pruned networks (not explored here). As our ABFT implementation embeds the row and column checksums along with the input matrices to perform a larger GEMM, online data management (i.e., copying input to larger matrices) introduces significant overheads. This overhead can be avoided by allocating larger matrices in the first place for the inputs and output with broader application knowledge and framework support, which is what we propose for the FC technique. The FIC technique avoids running the larger convolution altogether, simplifying data management. Lastly, processing output matrix twice to generate both the row and column checksums for error correction capability can also introduce high overheads. Optimized implementations that process the output just once will reduce this overhead. By focusing on error detection alone, ABED significantly speeds up this step by generating a single checksum.

6.4 Resilience Evaluation

Scope of protection: Figures 6 and 7 show the scope of protection offered by different the FC and FIC techniques. The ABED techniques protect computation in the convolution, input and output checksum generation, and dot-product kernel. Other than convolutions, CNNs include activation, pooling, and fully-connected layers. Activation layers are typically merged with the convolution layer and they constitute a small fraction of the total compute (0.6% for ResNet18 and 1.1% for ResNet50 as shown in Figure 6). Only a few pooling layers are typically used in CNNs (just two in ResNet18 and ResNet50, for example) and their compute requirement is also small (<0.3%). Fully-connected layers can be converted to GEMMs and checksum-based ABED techniques can be applied for their protection. For full network protection, ABED can be applied to convolutions and fully-connected layers, and duplication can be used to protect the rest, which constitutes a small fraction of the total compute, <1.4% for ResNet18 and ResNet50. Compared to full inference-level duplication, the ABED approach offers overheads that are lower by >4×.

The amount of protection offered by ABED for data storage and movement is important for architectures that do not protect (with ECC/parity) storage and transportation structures sufficiently against transient, intermittent, and permanent errors. We show the levels of protection ABED techniques offer for data movement in Figure 7. Since FIC technique protects the input data to the original convolution kernel, it provides better data storage and movement protection than the FC technique. As the input fmap size increases, the coverage offered by the FC technique reduces (compare FC FusedOCG results between ResNet18-ImageNet and ResNet18-1080p). While the coverage also reduces for the FIC technique, the reduction is much less.

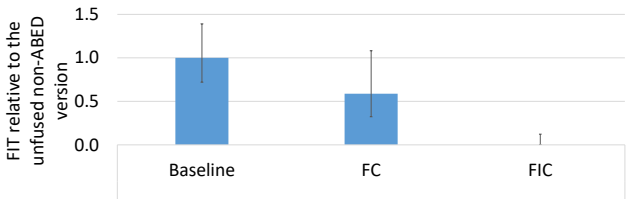


Fig. 13. The SDC FIT rate improvement with the FC and FIC techniques without on-chip ECC.

Results show that FIC FusedIOCG offers highest data storage and movement coverage among all the FIC options.

ECC/parity deployed in architectures used in HPC and safety-critical systems provide no protection to computational units, one of the major sources of intermittent and permanent errors. ABED techniques provide very high protection for computational units along with storage and data transportation protection.

Error injections: We perform error injections into the input and output tensors as described in Section 5.4. Our results for the FC technique show that all single-bit injections into non-zero filters and output fmaps are detected by the ABED technique and no single-bit injections into input fmaps are detected, as expected. A similar experiment for the FIC technique shows that errors in the input fmaps, filters, and output fmaps are detected.

Beam testing: To accurately quantify the vulnerability improvement, we conducted accelerated-particle beam experiments as outlined in Section 5.4. For the on-chip ECC Off experiments, the results show a clear SDC FIT rate reduction trend for the FC and FIC techniques. We observed some SDCs when the FC technique was employed. Up on inspecting the SDCs that were not detected by the FC technique, we found the output to be corrupted such that the values in original fmaps when reduced to be compared to the checksums result in no error detection (both the original output and checksum values were corrupted). Such manifestations indicate corruption to the input fmap, which is not covered by this technique. With on-chip SRAM (register file, L1/L2 cache) ECC/parity protection enabled, the likelihood of such errors will be low. We observed no SDCs while running convolutions with the FIC technique. The ABED techniques detected a few errors when no SDC was observed. These extra errors could be the result of a fault in the verification kernel. Figure 13 shows the FIT rate improvement results with on-chip ECC Off.

With on-chip ECC On, we expect both the FC and FIC techniques to offer high and comparable protection. Due to the cost of experimentation and limited availability of the beam time, we verify this only for FIC. In this experiment, the FIC technique detected all observed SDCs. The error bars for this experiment were relatively large, however, due to limited beam time and relatively lower SDC rate of the GPU (compared to on-chip ECC Off).

7 Discussion: Managing Rounding Error

While commercial systems increasingly use fixed-point data types during inference, the use of half-precision floating-point data type (fp16) is also common. All the techniques described in the paper are applicable to fp16 convolutions. Due to the non-associative nature of floating-point operations, the final comparison cannot be exact. We can test whether the two reduced values computed through different ways are close enough using a threshold. Corruptions can be detected if the error changes the values such that the difference is greater than the threshold. The lower the threshold, the higher the coverage.

The threshold depends on the rounding error introduced by the ABED tasks (e.g., checksum generation) and baseline convolution. We explored methods to significantly reduce the error introduced by the ABED tasks. Since the filter checksums are generated offline, very high precision operations can be used to reduce rounding error. Most architectures support accumulators that use higher precision compared to inputs (e.g., fp32 accumulation for fp16 input is common). Leveraging such hardware features, the error in input fmap online checksum generation can be reduced. For the FC technique, the resulting checksums are stored with the filters in fp16 format. The checksums can be stored as multiple fp16 values (or filters) such that the error introduced by rounding to fp16 is eliminated. For the FIC technique, the checksum can be stored in a fp32 data type. The output verification step can also use a higher-precision accumulator to reduce the rounding error. We estimate the error introduced by the checksum generation and verification steps (not shown here) and found it to be much smaller than the average error introduced by the convolution.

Since the error introduced by the baseline convolution is challenging to bound (due to the varying implementations, algorithms, and input value distributions), we rely on heuristics to estimate average rounding error. For some uncommon input values, the observed error can be higher than the threshold used for error detection, causing check failures in fault-free runs (we call them false positives). The system must recover from false positives to guarantee forward progress. False positives can be handled using a combination of techniques, if the rate is low: (1) rerun the layer on a different component (CPU/GPU/DLA) by incurring higher latency, or (2) notify a higher layer in the system to determine whether the it can tolerate skipping the inference. A recent study found that many low-level errors are tolerable at the system-level [20]. If the false positive rate is high, a diagnostic module can be invoked to tune the threshold or switch to the low-throughput duplication mode.

8 Conclusions

CNNs have made their way into safety-critical and HPC systems. GPU and accelerator-based systems are preferred platforms for CNNs, with the convolution consuming most of their execution time. Since safety is paramount for such systems, it is important to ensure the correctness of convolutions in the presense of hardware errors. This paper proposes an algorithm-based error detection (ABED) solution for convolutions, providing a much lower overhead approach compared to full duplication. We demonstrate how this solution can be employed during highly optimized CNN inference executions that fuse multiple layers and use reduced-precision operations. Results show that ABED eliminates convolution output corruptions for all studied hardware errors with low (6-23%) runtime overhead, at least 4× lower than full duplication.

References

[1] Al-Yamani et al. Performance Evaluation of Checksum-Based ABFT. In *Proc. of the Int. Symp. on Defect and Fault Tolerance in VLSI Systems (DFT)*, 2001.

[2] S. Alcaide, L. Kosmidis, C. Hernandez, and J. Abella. High-Integrity GPU Designs for Critical Real-Time Automotive Systems. In *Proc. of the Design, Automation & Test in Europe Conference (DATE)*, 2019.

[3] ARM. Arm A64 Instruction Set Architecture: Armv8, for Armv8-A architecture profile. https://developer.arm.com/docs/ddi0596/latest/a64-sve-instructions-alphabetic-order, 2018.

[4] Baidu. Apollo Open Platform. http://apollo.auto, 2019.

[5] W. Bartlett et al. Commercial Fault Tolerance: A Tale of Two Systems. *Trans. on Dependable and Secure Computing*, pages 87–96, 2004.

[6] C. Cazzaniga et al. First Tests of a New Facility for Device-Level, Board-Level and System-Level Neutron Irradiation of Microelectronics. *IEEE Trans. on Emerging Topics in Computing*, pages 1–1, 2018.

[7] J. Chen et al. GPU-ABFT: Optimizing Algorithm-Based Fault Tolerance for Heterogeneous Systems with GPUs. In *Proceedings of the IEEE International Conference on Networking, Architecture and Storage (NAS)*, 2016.

[8] J. Chung et al. Containment Domains: A Scalable, Efficient, and Flexible Resilience Scheme for Exascale Systems. In *Proc. of the Int. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[9] C. Constantinescu. Intermittent Faults and Effects on Reliability of Integrated Circuits. In *Proc. of the Annual Reliability and Maintainability Symp.*, 2008.

[10] F. F. d. Santos et al. Analyzing and Increasing the Reliability of Convolutional Neural Networks on GPUs. *IEEE Trans. on Reliability*, 68(2):663–677, 2019.

[11] J. Deng et al. ImageNet: A Large-Scale Hierarchical Image Database. In *Proc. of the Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2009.

[12] M. Dimitrov et al. Understanding Software Approaches for GPGPU Reliability. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, 2009.

[13] C. Ding et al. Matrix Multiplication on GPUs with On-Line Fault Tolerance. In *Proc. of the Int. Symp. on Parallel and Distributed Processing with Applications (ISPA)*, 2011.

[14] K. He et al. Deep Residual Learning for Image Recognition. *CoRR*, abs/1512.03385, 2015.

[15] K.-H. Huang et al. Algorithm-Based Fault Tolerance for Matrix Operations. *IEEE Trans. on Computers*, C-33(6):518–528, 1984.

[16] Q. Huang et al. Learning to Prune Filters in Convolutional Neural Networks. In *Proc. of the IEEE Winter Conference on Application of Computer Vision*, 2018.

[17] Z. Hui and other. Optimized Software-Based Hardening Strategies for Matrix Multiplication and Fast Fourier Transform. In *Proc. of the Int. Conf. on Algorithms, Computing and Systems*, 2018.

[18] ISO. ISO 26262-9:2011 Preview Road Vehicles Functional Safety. https://www.iso.org/standard/51365.html, 2011.

[19] X. Iturbe et al. A Triple Core Lock-Step (TCLS) ARM® Cortex®-R5 Processor for Safety-Critical and Ultra-Reliable Applications. In *Proc. of the Int. Conference on Dependable Systems and Networks Workshop (DSN-W)*, 2016.

[20] S. Jha et al. ML-based Fault Injection for Autonomous Vehicles: A Case for Bayesian Fault Injection. *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN)*, 2019.

[21] N. P. Jouppi et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proc. of the Int. Symp. on Computer Architecture (ISCA)*, 2017.

[22] T. Kurth et al. Exascale Deep Learning for Climate Analytics. In *Proc. of the Int. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, 2018.

[23] G. Li et al. Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications. In *Proc. of the Int. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, 2017.

[24] S. Li et al. Enabling Sparse Winograd Convolution by Native Pruning. *CoRR*, abs/1702.08597, 2017.

[25] X. Liang et al. Correcting Soft Errors Online in Fast Fourier Transform. In *Proc. of the Int. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, 2017.

[26] S.-C. Lin et al. The Architectural Implications of Autonomous Driving: Constraints and Acceleration. In *Proc. of the Int. Conf. on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2018.

[27] A. Mahmoud et al. Optimizing Software-Directed Instruction Replication for GPU Error Detection. In *Proc. of the Int. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, 2018.

[28] T. Marty et al. Enabling Overclocking through Algorithm-Level Error Detection. In *Proc. of the Int. Conf. on Field-Programmable Technology (FPT)*, 2018.

[29] P. Molchanov et al. Pruning Convolutional Neural Networks for Resource Efficient Transfer Learning. *CoRR*, abs/1611.06440, 2016.

[30] A. Nardi et al. Functional Safety Methodologies for Automotive Applications. In *Proc. of the Int. Conf. on Computer-Aided Design (ICCAD)*, 2017.

[31] S. F. Nowicki et al. The Los Alamos Neutron Science Center Spallation Neutron Sources. *Physics Procedia*, 90:374–380, 2017.

[32] NVIDIA. NVIDIA Tesla V100 GPU Architecture, The World's Most Advanced Datacenter GPU. http://www.nvidia.com/object/volta-architecture-whitepaper.html, 2017.

[33] NVIDIA. NVIDIA AI INFERENCE PLATFORM Giant Leaps in Performance and Efficiency for AI Services, from the Data Center to the Network's Edge. "https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/t4-inference-print-update-inference-tech-overview-final.pdf", 2018.

[34] NVIDIA. NVIDIA Jetson AGX Xavier Developer Kit — NVIDIA Developer. https://developer.nvidia.com/embedded/buy/jetson-xavier-devkit, 2018.

[35] NVIDIA. CUB: Main Page - NVlabs. https://nvlabs.github.io/cub/, 2019.

[36] NVIDIA. CUDA Binary Utilities :: CUDA Toolkit Documentation. http://docs.nvidia.com/cuda/cuda-binary-utilities/index.html, 2019.

[37] NVIDIA. cuDNN Developer Guide :: Deep Learning SDK Documentation. https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html, 2019.

[38] NVIDIA. NVIDIA DRIVE — NVIDIA Developer. https://developer.nvidia.com/driveworks, 2019.

[39] NVIDIA. Self-Driving Safety Report. https://www.nvidia.com/content/dam/en-zz/Solutions/self-driving-cars/safety-report/auto-print-safety-report-final-web.pdf, 2019.

[40] B. Reagen et al. Ares: A Framework for Quantifying the Resilience of Deep Neural Networks. In *Proc. of the Design Automation Conference (DAC)*, 2018.

[41] A. Rodriguez et al. Lower Numerical Precision Deep Learning Inference and Training. https://www.intel.ai/nervana/wp-content/uploads/sites/53/2018/05/Lower-Numerical-Precision-Deep-Learning-Inference-Training.pdf, 2018.

[42] S.Han et al. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *CoRR*, abs/1510.00149, 2015.

[43] K. Simonyan et al. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[44] M. Snir et al. Addressing Failures in Exascale Computing. *The Int. Journal of High Performance Computing Applications*, 28(2):129–173, 2014.

[45] V. Sze et al. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proc. of the IEEE*, 105(12):2295–2329, 2017.

[46] Tesla. Tesla Autonomy Investor Day — Tesla, Inc. https://ir.tesla.com/events/event-details/tesla-autonomy-investor-day, 2019.

[47] J. Wadden et al. Real-World Design and Evaluation of Compiler-Managed GPU Redundant Multithreading. In *Proc. of the Int. Symp. on Computer Architecture (ISCA)*, 2014.

[48] P. Wu et al. Towards Practical Algorithm Based Fault Tolerance in Dense Linear Algebra. In *Proc. of the Int. Symp. on High-Performance Parallel and Distributed Computing (HPDC)*, 2016.