Enabling Overclocking through Algorithm-Level Error Detection

Thibaut Marty
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
thibaut.marty@irisa.fr

Tomofumi Yuki
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
tomofumi.yuki@inria.fr

Steven Derrien
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
steven.derrien@irisa.fr

***Abstract*—In this paper, we propose a technique for improving the efficiency of hardware accelerators based on timing speculation (overclocking) and fault tolerance. We augment the accelerator with a lightweight error detection mechanism to protect against timing errors, enabling aggressive timing speculation. We demonstrate the validity of our approach for the convolution layers in convolutional neural networks. We present an implementation of a fault-tolerant convolution layer accelerator combined with the lightweight error detection. The error detection mechanism we have developed works at the algorithm-level, utilizing algebraic properties of the computation, allowing the full implementation to be realized using High-Level Synthesis tools. Our prototype on ZC706 demonstrated 68 %–77 % higher throughput with negligible overhead.**

I. Introduction

Timing speculation, also known as overclocking, is a well known approach to increase the computational throughput of programmable processors and hardware accelerators. When used aggressively, timing speculation can lead to incorrect/corrupted results due to timing anomalies that typically occur within long combinational paths. As reported in the literature [1]–[3], timing errors can cause large numerical errors in the computation. Although many applications are robust to low amplitude errors (e.g., errors due to quantization), occasional large errors can have devastating effect even for such applications.

The frequency of such errors depends on a number of factors, including the intensity of overclocking, operating temperature, voltage drops, variability within and across boards, input data, and so on. This makes it extremely difficult to determine a "safe" overclocking speed analytically or empirically as acknowledged in prior work. This has led to the use of online monitoring to avoid having excessive slacks based on static timing analysis [4], [5].

In this work, we propose to combine timing speculation with algorithm-level error detection to make overclocking a viable option. Online error detection is necessary to prevent errors from affecting the final output, and it must be lightweight so that the gains by overclocking are not nullified. Although many error detection techniques exist, they are too low-level and are tied to the exact design at hand [4], [5]. This limits the ease of design space exploration for a given algorithm, especially in the context of High-Level Synthesis where these techniques cannot be directly expressed.

We therefore propose a higher level error detection scheme by extending earlier results on algorithm-based fault tolerance [6]. ABFT is widely used in High Performance Computing as a lightweight protection from both soft and hard errors [7], [8].

The original ABFT for matrix operations has been used in the context of FPGAs as well [9], [10]. However, we are not aware of other work that uses algorithm-level error detection in combination with frequency scaling. We provide an alternative way to safely overclock FPGAs that can be expressed at the algorithm-level, giving additional flexibility to designers especially in the HLS context.

We use convolutional neural networks as a case study to demonstrate our approach. CNN is a variant of multi-layered neural networks that constructs features from local information through convolutions. CNN models used in state-of-the-art applications are computation intensive and often need to be accelerated on GPUs or FPGAs to achieve high performance and/or obtain better energy efficiency [11], [12]. The core computations of CNNs have abundant parallelism, both task-level and fine-grained, that can be efficiently mapped to these accelerators. Furthermore, CNNs are known to be tolerant to noise. The reasons above make CNN an interesting class of computation to target.

Specifically, our contributions are the following:

- A lightweight error detection for convolution layers building on the classical ABFT. The developed method gives two-degree savings in complexity with respect to the full convolution layer, whereas the original ABFT is limited to one-degree savings in matrix operations.
- A method for dynamic frequency scaling reacting to monitored errors.
- A prototype implementation of the above, fully implemented with HLS tools.

The remainder of this paper is organized as follows. Section II provides background on timing speculation techniques and CNNs. We describe our proposed approach with lightweight error detection in Section III and IV. We demonstrate the approach with a prototype implementation in Section V, and then discuss our results and related work in Section VI. We conclude and give directions for future work in Section VII.

II. Background

In this section, we introduce timing speculation and convolutional neural networks. We then discuss existing FPGA designs for CNN accelerators.

A. Timing Speculation through Overclocking in FPGAs

The minimum clock period at which a given FPGA design is expected to work is obtained from a static timing analysis (STA). This analysis assumes the worst case scenario, and hence the design may be operated on higher operating frequencies without observing incorrect behaviors. However, this technique, widely known as *overclocking* or *timing speculation*, also has many pitfalls:

- Variability among chips and operating conditions makes it difficult to determine how much overclocking can be tolerated. The fact that errors do not manifest often (or the inability to observe errors in a given setup) does not mean that errors do not happen.
- The impact of timing errors on the circuit output is difficult to evaluate a priori. Unlike errors arising from truncation/quantization, the impact is not limited to least significant bits. Thus, it may result in large numerical errors, compromising the design functionality.

These challenges have led to circuit-level techniques for dynamically checking for incorrect behaviors [4], [5], [13]. Our work also aims at enabling aggressive overclocking through algorithm-level error detection.

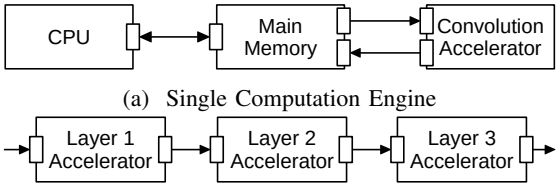
B. Convolutional Neural Networks

In this paper, we are interested in the forward pass of CNNs, i.e., when a trained network is applied to new inputs. In a typical configuration, a forward pass of CNNs processes three-dimensional matrices in a pipelined manner through multiple *layers*. The input is usually an image (with color depth), and the final output is a vector of length M indicating the likelihood of each category, which can be viewed as a $1\times1\times M$ matrix.

We are interested in the convolution layer of CNNs, which is known to be the main bottleneck. Convolution layers act as local feature extractors. Given a $P \times Q \times N$ input matrix x, it computes a $R \times C \times M$ output y. Each of the M two-dimensional outputs are computed as a three-dimensional convolution over x with a kernel (weights) of size $K \times K \times N$. The convolution may be strided by some factor S. The R and C dimensions of the output matrix may become smaller than the input for a non-unit stride, or depending on the padding of the boundaries. Different layers take different values of the parameters described - the output $R \times C \times M$, called feature maps, can be viewed as an "image" where the depth is the extracted features.

Given a $P \times Q \times N$ input matrix x and $K \times K \times N \times M$ matrix holding the weights w, a convolution layer outputs a $R \times C \times M$ matrix y through the following equation:

$$y_{r,c,m} = \sum_{n=0}^{N-1} \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} x_{Sr+i,Sc+j,n} \cdot w_{i,j,n,m} \qquad (1)$$



(a) Single Computation Engine

(b) Streaming Accelerator

Fig. 1: System-level architectures.

In this paper, we describe the approach for convolution layers with stride factor $S = 1$. How our approach can be extended to more general cases and how others layers may be handled are discussed in Section VI-C.

C. Accelerating CNNs with FPGA

There has been numerous work on accelerating CNNs on FPGAs and other platforms. We describe the system-level architectures that are typically used, and then describe the accelerator for the convolution layer.

The system-level architectures are largely distinguished into two types as illustrated in Figure 1 [14]. Single computation engine (SCE) architecture offloads the main kernel to the accelerator, processing the layers in a sequential manner. Architecture of this type typically employ a form of decoupled access/execute model through macro-pipelining to process blocks of computation that fit on-chip memory. Streaming accelerator (SA) uses multiple accelerator instances, typically one for each layer, that are pipelined to provide high throughput.

Our approach can fit both types of architectures described above. Our only requirement is that the inputs/outputs of the convolution accelerator is streamed through asynchronous FIFOs. This enables the accelerators to operate in a separate clock domain. For our prototype implementation, we used the SCE architecture.

Our accelerator kernel follows an implementation strategy proposed by Zhang et al. [11], which is described in Figure 2. One important notion in their work that we also use is the *tiles*. The CNN computations are largely data parallel, and can be decomposed into smaller chunks to reduce on-chip memory cost. The granularity of these chunks is usually selected such that the accelerator fits on the target board. We use the word *tile* to refer to the unit of computation performed by the accelerator.

III. Proposed Approach

In this section, we first motivate the need for online error detection as an overclocking enabler by empirical observation of the impact of overclocking on application-level accuracy. We then describe our modified accelerator that allows the error rate to be monitored online to avoid excessive degradation in the final output.

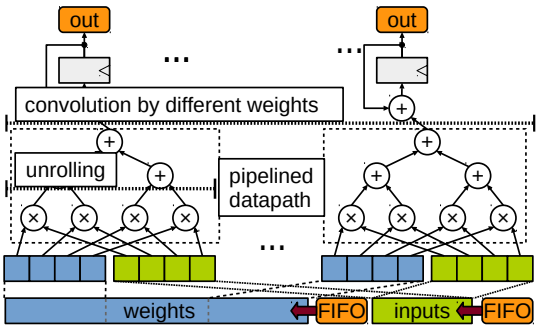


Fig. 2: The convolution kernel based on the design by Zhang et al. [11]. The convolution kernel has three sources of parallelism. The main convolution has ample parallelism, which is used as the fine-grained parallelism (unrolling in HLS). This datapath is also aggressively pipelined to process different inputs. Furthermore, this datapath itself is replicated for convolutions by different weights to the same input. The factor of unrolling and/or replication controls the throughput of the accelerator.

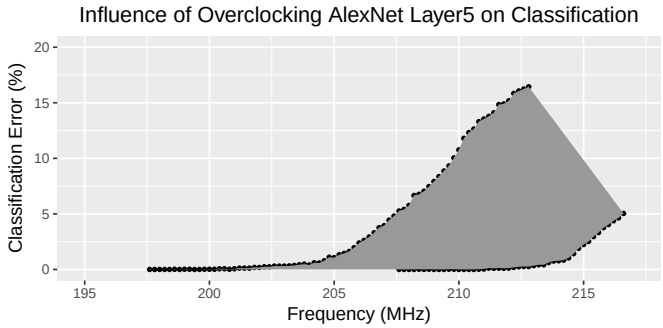


Fig. 3: Impact of overclocking on classification; classification error is when the output class differs from that of error-free execution, regardless of the ground-truth class. The data points are the highest/lowest error rate observed for the given frequency, and the shaded regions show the range of possible error rate influenced by various sources of variability.

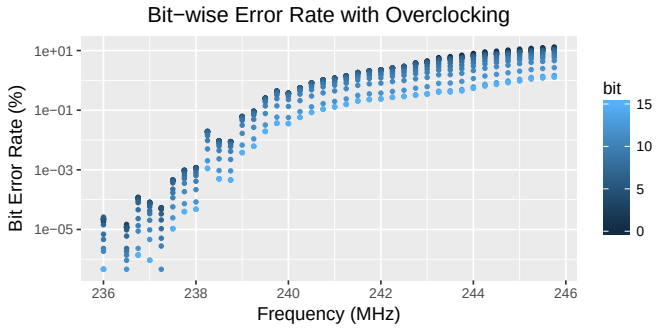


Fig. 4: Observed bit-wise error rate in the outputs.

A. *Impact of Overclocking on Classification Accuracy*

We performed a series of experiments to asses the impact of overclocking on final output. In this section, we report some of the results obtained from classifying 5000 images (the first 10 % of ILSVRC 2012 validation dataset) using AlexNet. We used our accelerator prototype to execute the last convolutional layer. The last convolutional layer is selected for these experiments because the last layers are known to be more sensitive to errors than the first couple of layers. We used 16-bit fixed-point data. We obtained the input data for the accelerator by quantizing the previous layer's error-free floating-point output in order to isolate effect on accuracy. Using low precision fixed-point requires special attention to training and quantization to avoid accuracy drop, which is beyond the scope of this work.

We used a set of Zybo boards to empirically observe variability. We used Zybo for this experiment because we had access to a number of them, which was important to capture inter-board variability. Figure 3 depicts the impact of overclocking on classification accuracy showing that there are significant variation when the accuracy degradation starts to happen. We have also performed the same experiment on a larger board (ZC706), and observed similar impact on accuracy.

Figure 4 reports the bit-wise error rate observed on our ZC706. There are several interesting observations:

- Although it is less likely than the lower bits, the most significant bits are also affected.
- We observed that MSB flips are dominant when we force the use of LUT-based multipliers, but that LSB flips dominate when using DSP48 based multipliers.
- The relative probabilities between bit flips seem constant across frequency — overclocking increases the total number of errors without influencing the ratios.
- A 10 MHz frequency range covers 6 orders of magnitude error rate difference. We believe that this is due to the paths being balanced by the synthesis and P&R tools, resulting in many paths with similar length. This explains the large number of errors manifesting within a narrow range of frequencies.

It is acknowledged by prior work that a "safe" frequency is impossible to determine statically due to multiple sources of variability [5]. For example, when considering only a subset of variability (inter-board variability), there is as wide as 10 MHz gap in the frequency where accuracy starts to drop. In other words, it is difficult to ensure that the overclocked accelerators do not significantly alter the application output.

B. *Online Error Detection*

Our approach builds on a *lightweight* mechanism for detecting errors based on algorithmic properties, similar to algorithm-based fault tolerance known for matrix operations [6]. However, we emphasize that the error detection we use is not the same as the one for matrix operations; see Section VI-A for further discussion.

This error detection mechanism uses two checksums, one computed from the output, and another computed directly

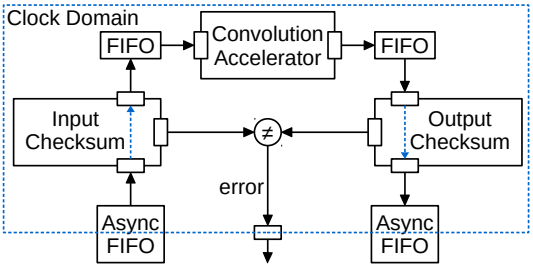


Fig. 5: Accelerator with Error Detection.

from inputs, which we refer to as output-checksum and input-checksum, respectively. We flag the computation as erroneous whenever the checksums do not match. We detail how the checksum calculation can be made lightweight in Section IV. The key intuition is that the computation of input-checksum can be simplified taking advantage of algebraic properties. Instead of redundantly computing the output with another instance of the accelerator (i.e., double modular redundancy), the checksum is directly computed from the inputs, with lower complexity, using algebraic properties.

Our modified architecture is depicted in Figure 5. Two additional modules are added between the convolution kernel and the FIFOs that computes the checksums. The important property is that the checksums are computed in a streaming fashion such that the latency of the accelerator is unaffected. The entire accelerator is in its own clock domain managed by software through dynamic reconfiguration of the mixed-mode clock manager (MMCM) module.

C. Frequency Scaling

Given an accelerator with the capability to detect errors online, there are many possible ways to react to the errors. We use a simple algorithm to dynamically scale the frequency at runtime. The algorithm has two parameters:

- G: granularity of frequency adjusted at a time;
- I: interval; number of successful tiles before attempting to increasing frequency.

The algorithm initially increases the frequency by G after each tile until an error is observed. Then the algorithm repeatedly takes one of the following actions:

- decrease by G if an error is observed;
- increase by G after I successful tiles.

In our empirical study, we found that a few errors do not significantly influence the classification accuracy. Hence, it is possible to allow for a limited number of errors. If occasional miss-classification is not a problem, then short intervals can be used to aggressively attempt to overclock. If no error is tolerated, the interval can be made long to minimize error rate and recompute the erroneous tiles.

D. Failure Recovery Cost

When a small number of errors are tolerated, the errors detected do not have to be corrected. Hence, there is no additional latency overhead with our approach. If error-free output is desired, the erroneous tiles must be recomputed, adding some overhead. This overhead depends on the system-level architecture discussed in Section II-C.

For a single compute engine, the erroneous tile is simply fed back to the accelerator at a later time. Since the tiles are parallel, they can be executed in any order, the macro-pipeline does not have to be stalled.

For a streaming accelerator, the overhead depends on the details of the architecture. If there are buffers between layers, e.g., for crossing FPGA boundaries, then full pipeline stall can be avoided. In the worst case, the full pipeline must be restarted from the first layer, discarding all intermediate results.

As a simplistic model, consider the following:

- each tile takes 1 unit of time with baseline frequency;
- failed tiles are executed at the baseline frequency;
- N: total number of tiles;
- O: mean overclocked frequency, normalized to the baseline;
- E: error Rate, probability of a tile to fail;
- S: number of stages in SA.

Note that the time it takes to switch the frequency is negligible — less than $1\,\%$ of a tile computation in our prototype.

Then the total time T can be modeled as: $T = \frac{N}{O} + S.N.E$ where the SCE is a special case when $S = 1$.

The gain by overclocking is $N - \frac{N}{O}$, which should be lower than the overhead $S.N.E$ for overclocking to be profitable. Clearly, the error rate plays an important role, and we may derive the break-even point (overhead = gain) as: $E = \frac{O-1}{S}$

In other words, even for a small gain in frequency, completely negating its gain takes a high error rate. For instance, with $25\,\%$ overclocking on average with 5-stage SA architecture, it requires $5\,\%$ error rate to negate the gain. With a strict dynamic scaling policy that only attempts to increase frequency in long intervals once an error is detected, the error rate is extremely low ($< 0.1\,\%$), rendering the recovery cost negligible. In addition, we expect that the error-free execution is not necessary in most use cases.

IV. Algorithm-Level Error Detection

This section describes our checksum computation technique. We start with a simpler case for 2D convolution kernel, and then generalize to convolution layers in CNNs.

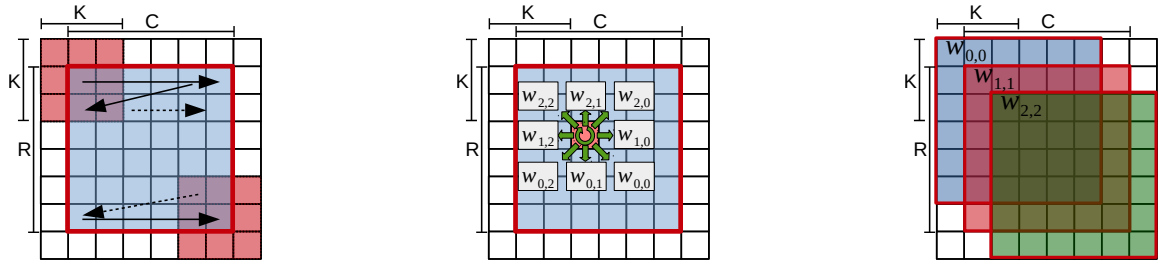
A. Intuition with 2D Convolution

The 2D case is missing the depth dimension in the processed matrices, but the main ideas carry over to the 3D case. Given the 2D output y, the output-checksum is:

$$\sigma = \sum_{r=0}^{R} \sum_{c=0}^{C} y_{r,c}$$

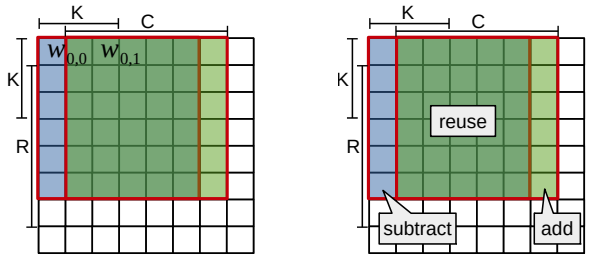
Substituting the definition of y (Eq. 1 without m, n and $S = 1$) gives the direct computation from the inputs:

$$\rho = \sum_{r=0}^{R} \sum_{c=0}^{C} \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} (x_{r+i,c+j} \cdot w_{i,j})$$



(a) 2D convolution without zero-padding (b) Contribution of an input by weights (c) Input data groups by weights

Fig. 6: Illustration of the factorization for 2D case. As depicted in Figure 6a, 2D convolution can be viewed as a dot product between the weights and the neighboring inputs with a sliding window. An alternative view shown in Figure 6b is that an input value is used to compute K^2 output values, contributing to each of them through multiplication by different weights. We can thus group the input data into (overlapping) subsets based on the weights, which is what is shown for three weight values in Figure 6c. Since sum of convolution outputs is completely linear, the multiplication can be factorized to save work.



(a) Two neighboring groups (b) Reuse pattern

Fig. 7: The reuse between two input groups corresponding to weights $w_{0,0}$ and $w_{0,1}$. The sum of all elements in group $w_{0,1}$ can be computed by addition/subtraction of columns from that of $w_{0,0}$, instead of repeating $R \times C$ additions.

The goal is to simplify the computation of ρ so that the checksum comparison can be performed with reduced cost.

The additional two-dimensional summation provides two sources for simplification. The combination of the two simplifications described below reduces the cost of computing the checksum for 2D case from $O(RCK^2)$ to K^2 multiplications and $O(RC)$ additions.

1) Factorization: Multiplications can be factored out to eliminate RC multiplications. This may be viewed as a reordering of the summations followed by factorization:

$$\rho = \sum_{i=0}^{K-1}\sum_{j=0}^{K-1} w_{i,j}\left(\sum_{r=0}^{R}\sum_{c=0}^{C} x_{r+i,c+j}\right)$$

A graphical illustration is given in Figure 6. This reduces the number of multiplications from K^2RC to K^2.

2) Reuse in Summations: The groups of summations after factorization have significant overlaps, which can be used to reduce the number of additions. This simplification concerns the computation of the inner two summations:

$$X_{i,j} = \sum_{r=0}^{R}\sum_{c=0}^{C} x_{r+i,c+j}$$

Note that each value of X is a summation over slightly different regions of x due to the offsets by i and j. These values of X takes K^2RC additions when computed naïvely. However, once a value of X for a specific instance of i, j is computed, the remaining instances can be computed by only $O(C)$ or $O(R)$ additions as explained in Figure 7.

There are multiple ways to rewrite the definition of X to take advantage of this reuse. One example is as follows:

$$X_{i,j} = \begin{cases} \sum_{r=0}^{R}\sum_{c=0}^{C} x_{r,c} & : \begin{matrix} i=0 \\ j=0 \end{matrix} \\ X_{i-1,j} + \sum_{c=0}^{C} x_{R-1+i,c+j} - \sum_{c=0}^{C} x_{i-1,c+j} & : i > 0 \\ X_{i,j-1} + \sum_{r=0}^{R} x_{r+i,C-1+j} - \sum_{r=0}^{R} x_{r+i,j-1} & : \begin{matrix} i=0 \\ j>0 \end{matrix} \end{cases}$$

In the above, the $R \times C$ summation for $X_{0,0}$ is performed first. Then the remaining instances of $X_{i,j}$ is computed by addition and subtraction of one row/column. The $R \times C$ summation is not repeated for all each $X_{i,j}$ ($K \times K$ instances) reducing the complexity by $O(K^2)$ in exchange for $2R$ or $2C$ additions.

B. Lightweight Checksum for 3D Convolution Layers

For the 3D case, the output y has the third dimension corresponding to the different kernels applied to the input. The checksum is over all three dimensions of the output:

$$\sigma = \sum_{m=0}^{M-1}\sum_{r=0}^{R-1}\sum_{c=0}^{C-1} y_{m,r,c} \tag{2}$$

Substituting Eq. 1 (again, assuming unit stride) gives the direct checksum computation from the inputs:

$$\rho = \sum_{m=0}^{M-1}\sum_{r=0}^{R-1}\sum_{c=0}^{C-1}\left(\sum_{n=0}^{N-1}\sum_{i=0}^{K-1}\sum_{j=0}^{K-1} x_{n,r+i,c+j} \cdot w_{m,n,i,j}\right)$$

Reordering of the summations permits the factorization of the multiplication by weights:

$$\rho = \sum_{n=0}^{N-1}\sum_{i=0}^{K-1}\sum_{j=0}^{K-1}\left(\left(\sum_{r=0}^{R-1}\sum_{c=0}^{C-1} x_{n,r+i,c+j}\right) \cdot \sum_{m=0}^{M-1} w_{m,n,i,j}\right)$$

Note that in the 3D case, the different convolution kernels applied to the same input (the m dimension) can be first added together, since m is invariant to the expression involving x.

$$\rho = \sum_{n=0}^{N-1} \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} \left(X_{n,i,j} \cdot \sum_{m=0}^{M-1} w_{m,n,i,j} \right) \tag{3}$$

where X takes the same form as the 2D case, except for the additional n dimension, which does not have any reuse.

C. Implementation of Checksum Calculations

The output-checksum is implemented as a simple accumulator over convolution outputs, with a small area overhead compared to the rest of the datapath. The hardware component responsible for computing the input-checksum (Eq. 3) is more costly, as it involves a multiplier, storage for partial sums, and non-trivial control logic.

Both of these components are significantly less complex compared to the main kernel. The main performance constraint is to ensure that the data processing rate matches that of the convolution kernel. The checksum calculations need sufficient parallelism to keep up with the rate of input consumption as well as output production. The parallelism in these calculations are realized as aggressive pipelining (II=1) to minimize area cost.

The output-checksum has a trivial parallelism to process the outputs. Since the output-checksum is an accumulation, the computation is independent except for the aggregation at the end.

The input-checksum computes the input groups and the summation of weights over M as the data is streamed through. Then, the rest of the computation is overlapped with the convolution kernel: the computation of X using the input groups, multiplication by the summed weights, and the final accumulation. Note that the inputs memory transfer (and summations) for next tile is overlapped with the kernel, the rest of the input-checksum calculation, and the outputs memory transfer for the previous tile. This ensures that the input-checksum calculation does not impact the overall latency of the accelerator.

D. Correctness of Error Detection

The algebraic property guarantees that there is no possibility to have false negatives provided that there is only one error. If multiple errors happen, there is a chance that the errors cancel each other such that the checksum remains the same. If the wordlength is b, this chance is $\frac{1}{2^b}$ assuming all bits have the same probability to flip. In practice, the chances may be higher due to the probability of bit flips not being uniform, and having higher concentration on a subset of the bits.

It is possible to have false positives if an error happens at the checksum calculations that are also overclocked. However, these modules are much simpler than the convolution kernel, and we have empirically observed that the errors start to occur at much higher frequencies compared to the convolution kernel. We have never seen false positives in our experiments.

We have observed that the false negative rate was around $0.33\,\%$ for our design with 8 bits wordlength. This is $0.33\,\%$ of the erroneous tiles that contains more than one error, which is an extremely small fraction of all the tiles executed. Combined with the fact that a small number of errors are not expected to affect the final output, we believe that the false negatives are not an issue.

V. Proof of Concept

We present our empirical study in this section. The study consist of two parts: (i) the area overhead, and (ii) frequency scaling on AlexNet image classification.

A. Experimental Platform

The whole accelerator was designed using SDSoC (2018.2), demonstrating the suitability of our approach to modern FPGA tools that operate at higher level of abstractions. We use the ZC706 board (XC7Z045) from the Xilinx Zynq-7000 family as our target platform.

The hardware designs used in this section targets the fifth convolutional layer in the AlexNet CNN architecture [12] with the following standard configuration: $N = 192$, $M = 128$, $R = C = 13$, $K = 3$, and $S = 1$. Preliminary experiences showed that this layer was the one that most impact classification rate when impacted by timing errors. Note that the third and fourth layers of AlexNet also have similar configurations with larger N and/or M.

We report the results for various wordlengths used in the convolution kernel. Other design parameters (tile sizes and unroll factors) were selected to achieve highest throughput on the target board with 16 bits, and the same parameters were used for lower wordlength designs. All designs were synthesized with several target frequencies in order to get the maximum STA frequency. All reported numbers are after P&R. We encountered an issue in SDSoC that made the synthesis to fail because of the use of the dynamic frequency reconfiguration (clocking wizard) when using a target frequency different than $100\,\mathrm{MHz}$. Thus, the experiments were run with the $100\,\mathrm{MHz}$ synthesised designs, while area numbers come from designs without dynamic reconfiguration ability. As our best effort to make a fair baseline, we use the maximum STA frequency as the baseline frequency.

B. Area Overhead

The following can be observed from results in Table I:

- DSP and BRAM cost, which are the limiting resource on this board, does not increase by adding online error detection.
- Only slice usage changes showing effective overhead after P&R.

This clearly shows that the overhead is negligible. Enabling overclocking with online error detection gives more than $50\,\%$ boost in throughput for "free" as shown in the next section.

TABLE I: ZC706 area results, maximum STA frequency (in MHz) and GOPS for different wordlengths. Designs with algorithm-level error detection (ALED) include the cost for all components: the convolution kernel, input-checksum, output-checksum. The frequency and GOPS are not relevant for ALED equipped ones, as the frequency will be scaled at runtime.

WL	Type	BRAM	DSP	SLICE	F_{max}	GOPS
16	base	13.2 %	57.0 %	36.7 %	134.8	43.2
	ALED	13.2 %	57.0 %	37.4 %	-	-
8	base	10.3 %	58.7 %	24.4 %	134.4	43.0
	ALED	10.3 %	58.7 %	25.8 %	-	-
4	base	4.4 %	58.8 %	20.8 %	141.5	45.3
	ALED	4.4 %	58.8 %	20.6 %	-	-
2	base	4.4 %	1.0 %	14.3 %	229.9	73.6
	ALED	4.4 %	1.0 %	14.7 %	-	-
1	base	2.9 %	0.1 %	14.3 %	231.7	74.2
	ALED	2.9 %	0.1 %	14.5 %	-	-
Available		1090	900	54 650		

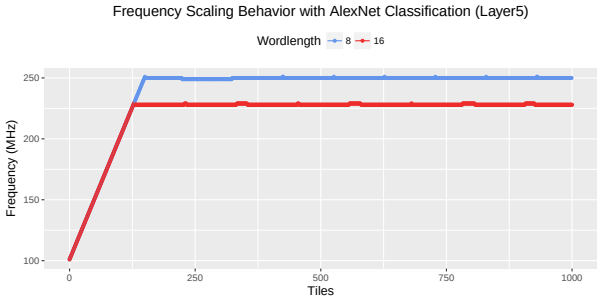


Fig. 8: Frequency scaling applied to AlexNet.

C. Dynamic Frequency Scaling

We show the result of applying frequency scaling on a classification task of 1000 images using AlexNet with 8 and 16 bits designs in Figure 8. Note that 16 bits data come from Alexnet's fifth layer output data while 8 bits data was uniformly generated data. The algorithm in Section III-C was configured to $G = 1\,\text{MHz}$ and $I = 100$. The frequency scales up to around 230 and then stabilizes with occasional bumps. The mean frequency over the 1000 images was 226.5 MHz for 16 bits and 238.7 MHz for 8 bits. Compared to the baseline frequency, this is about 68 %–77 % increase in throughput. Note that this result is from a single board, single setup. The gains are expected to vary under different operating conditions.

VI. Discussion and Related Work

In this section, we place our work in context with related work. We then discuss extensions to more general CNN layers.

A. Comparison with Existing ABFT Techniques

Our approach builds on the ABFT technique for matrix operations [6]. The convolution layer and the fully-connected layer can be viewed as a collection of matrix products, making the classical ABFT directly applicable. Some ABFT extensions employ a variant of the original method by identifying pieces of computations that can be viewed as matrix operations [15], [16]. However, ABFT is a more general concept that can be applied to other computations, e.g., FFT [17].

The lightweight checksum calculation proposed in this paper is not a direct application of the ABFT for matrix operations. We use algorithmic invariants in collections of convolutions to further reduce the cost of checksums. This is evident in the fact that we reduce the algorithmic complexity by twofold, exploiting reuse in two dimensions, whereas the original ABFT brings one-degree savings.

Some of the ABFT techniques employ two or more checksums to enable error correction. It is possible to use a similar method for the convolution layer by giving up one-degree of complexity savings. However, this is not attractive because:

- Error correction based on checksums would not work if there are two or more errors (or at the cost of less complexity reduction), which is frequently the case with overclocking.
- Recovery by recomputing erroneous outputs does not add significant cost as discussed in Section III-D.
- Adding the error recovery logic in hardware can be costly in terms of area.

B. Other Techniques for Timing Error Detection

Some error detection techniques have been designed specifically to detect timing errors. Both Razor [4] and online slack measurement [5] use shadow registers to detect timing violation or to measure timing slacks. These approaches adjust the frequency/voltage using the measured errors or timing slack, which is similar to our work.

The main difference between our approach and those based on shadow registers is that we provide error detection at the algorithm-level in contrast to register/circuit level. This makes our approach more flexible, as it can be reused without additional design effort over a large design space (tile size, unrolling, data wordlength, etc.) of CNN accelerators. In contrast, circuit-level protections must be redesigned for every new design. Although some proof of concept toolchains have been proposed [18] to automate such approach, they are not publicly available, and it is difficult to assess how robust they are in practice.

Existing circuit-level techniques trade coverage with area overhead. One of the main difficulty is to properly select the smallest subset of the registers to protect/measure to keep the overhead low. This process is complicated as the addition of the shadow registers may impact the timing, and is also limited by the static timing analysis to determine the (near-)critical paths (not even mentioning LUT-level variability as reported by Gojman et al. [19]).

Lastly, many key FPGA components such as BRAM and DSP blocks include internal (e.g., pipelining) registers that cannot be protected due to the lack of adequate routing resources. This severely limits the error coverage that can actually be obtained. Our error detection scheme does not suffer from such limitations, while providing excellent error coverage for low hardware overhead.

The main limitation of our work is that we require the computation to have algorithmic properties to enable lightweight error detection. Although convolutions is not the only kernel

where ABFT is available, our technique is not applicable to arbitrary computations unlike those based on shadow registers. An interesting direction of future work is to automatically identify the necessary properties to improve the applicability of our technique.

C. More General CNN Layers

We have assumed unit stride to simplify the presentation. We do not give the full detail due to space reasons, but the same principles apply to non-unit strides as well. The main difference is in the simplification of the summations. With non-unit strides, the input groups (Figure 6c) also becomes strided. This reduces the reuse across the input groups, but this is natural since non-unit stride essentially corresponds to subsampling, i.e., the number of uses of each input for computing the output is reduced. For a stride factor S, there are S^2 independent input groups. This reduces the complexity savings to $O\left(\frac{RC}{S^2}\right)$ for additions and $O\left(\frac{K^2}{S^2}\right)$ for multiplications.

However, we note that layers with non-unit strides typically have larger values of R, C, and K increasing the reuse. For AlexNet, the first layer uses $S = 4, R = C = 55$, and $K = 11$, which corresponds to a factor of 189 savings on additions and a factor of 8 savings on multiplications. These savings are similar to those for the last layers with $S = 1, R = C = 13$, and $K = 11$, which corresponds to savings on additions and multiplications with a factor of 169 and 9, respectively. Thus, we expect similar levels of overhead even for strided layers.

Other layers such as pooling or rectified linear unit are not compute-intensive. These layers may simply be executed without overclocking to ensure that no errors are introduced.

VII. Conclusions and Future Work

In this paper, we propose timing speculation coupled with lightweight error detection as an approach to further improve the performance of hardware accelerators for CNNs. We have demonstrated the efficacy of our approach with a prototype implementation and an extensive empirical study. In addition, our approach is well suited for implementation in high-level design tools such as Vivado HLS/SDSoC, which is becoming more and more attractive for productivity reasons.

We believe that similar techniques can be applied to many other application domains (bioinformatics, iterative solvers, etc.) by taking advantage of existing ABFT techniques or by devising new algorithms tailored for this task. This is part of our ongoing work.

Acknowledgements

This work receives financial support from the Brittany Region. The authors would like to thank the anonymous reviewers for their insightful comments.

References

[1] S. Chaudhuri, J. S. J. Wong, and P. Y. K. Cheung, "Timing speculation in FPGAs: Probabilistic inference of data dependent failure rates," in *Proceedings of the 2011 International Conference on Field-Programmable Technology*, ser. FPT '11, 2011, pp. 1–8.

[2] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17, 2017, pp. 8:1–8:12.

[3] K. Shi, D. Boland, and G. A. Constantinides, "Accuracy-Performance Tradeoffs on an FPGA through Overclocking," in *Proceedings of the IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '13, 2013, pp. 29–36.

[4] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation," in *Proceedings of the 36th IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36, 2003, pp. 7–.

[5] J. M. Levine, E. Stott, and P. Y. Cheung, "Dynamic Voltage & Frequency Scaling with Online Slack Measurement," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '14, 2014, pp. 65–74.

[6] K.-H. Huang and J. A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 518–528, 1984.

[7] Z. Chen, "Online-ABFT: An Online Algorithm Based Fault Tolerance Scheme for Soft Error Detection in Iterative Methods," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13, 2013, pp. 167–176.

[8] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, "Algorithm-based Fault Tolerance for Dense Matrix Factorizations," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12, 2012, pp. 225–234.

[9] A. Jacobs, G. Cieslewski, and A. D. George, "Overhead and reliability analysis of algorithm-based fault tolerance in FPGA systems," in *Proceedings of the 22nd International Conference on Field Programmable Logic and Applications*, ser. FPL '12, 2012, pp. 300–306.

[10] J. J. Davis and P. Y. K. Cheung, "Achieving low-overhead fault tolerance for parallel accelerators with dynamic partial reconfiguration," in *Proceedings of the 24th International Conference on Field Programmable Logic and Applications*, ser. FPL '14, 2014, pp. 1–6.

[11] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15, 2015, pp. 161–170.

[12] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

[13] M. Fojtik, D. Fick, Y. Kim, N. Pinckney, D. M. Harris, D. Blaauw, and D. Sylvester, "Bubble Razor: Eliminating Timing Margins in an ARM Cortex-M3 Processor in 45 nm CMOS Using Architecturally Independent Error Detection and Correction," *IEEE Journal of Solid-State Circuits*, vol. 48, no. 1, pp. 66–81, 2013.

[14] S. I. Venieris, A. Kouris, and C.-S. Bouganis, "Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 56:1–56:39, 2018.

[15] A. Roy-Chowdhury, N. Bellas, and P. Banerjee, "Algorithm-based error-detection schemes for iterative solution of partial differential equations," *IEEE Transactions on Computers*, vol. 45, no. 4, pp. 394–407, 1996.

[16] G. Bosilca, A. Bouteiller, T. Herault, Y. Robert, and J. Dongarra, "Composing resilience techniques: ABFT, periodic and incremental checkpointing," *International Journal of Networking and Computing*, vol. 5, no. 1, pp. 2–25, 2015.

[17] S.-J. Wang and N. K. Jha, "Algorithm-based fault tolerance for fft networks," *IEEE Transactions on Computers*, vol. 43, no. 7, pp. 849–854, 1994.

[18] J. M. Levine, E. Stott, G. A. Constantinides, and P. Y. K. Cheung, "SMI: Slack Measurement Insertion for online timing monitoring in FPGAs," in *Proceedings of the 23rd International Conference on Field Programmable Logic and Applications*, ser. FPL '13, 2013, pp. 1–4.

[19] B. Gojman, S. Nalmela, N. Mehta, N. Howarth, and A. Dehon, "GROK-LAB: Generating real on-chip knowledge for intra-cluster delays using timing extraction," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 4, pp. 32:1–32:23, 2014.