Flexible Instrumentation for Live On-Chip Debug of Machine Learning Training on FPGAs

Daniel Holanda Noronha[1], Zhiqiang Que[2], Wayne Luk[2] and Steven J.E. Wilton[1]
[1]University of British Columbia, [2]Imperial College London
{danielhn,stevew}@ece.ubc.ca, {z.que,w.luk}@imperial.ac.uk

***Abstract*—FPGAs have recently shown promise for accelerating machine learning training. This has led to research into the co-design of narrow-precision accelerator architectures and the investigation of novel machine learning models. Such research can be extremely expensive, as the steep cost of training a model can increase several-fold due to the need of performing hyper-parameter tuning and adjustments to the model to ensure acceptable convergence speed and accuracy. In this scenario, monitoring key data on-chip is essential to more quickly understand and diagnose problems, significantly reducing training costs.**

Previous work has proposed on-chip debug instrumentation to monitor key signals for both general-purpose circuits and inference algorithms. This instrumentation either performs limited on-chip compression, or is extremely restricted in the amount of run-time customization that may occur. We argue that for training applications, the extremely long and expensive training runs warrant significantly more flexibility in the on-chip instrumentation, even at the expense of some chip area.

In this paper, we propose flexible debug instrumentation that allows for the live debugging of machine learning systems during training. Different from previous debug instrumentation, our instrumentation offers firmware programmability, allowing the researcher to gather data in a large variety of ways that would likely not be anticipated at compile time.

I. Introduction

It has become well-established that FPGA-based hardware accelerators can provide energy-efficient compute horsepower for a variety of complex applications [1]. Major companies such as Microsoft, Amazon, IBM, and Baidu have recently incorporated FPGAs into their data centres [2]–[4]. Today, many of the target applications involve machine learning inference for tasks such as search engine ranking and natural language processing [5], [6]. However, the process of *training* machine learning models still heavily relies on GPUs.

Training machine learning models using FPGAs has been explored by academic work [7]–[10] and has recently spurred the attention of FPGA vendors and FPGA groups from companies with large-scale FPGA-accelerated data centers [11]–[14]. Using FPGAs for training may be compelling since it may offer higher performance-per-watt than a GPU. This is due to optimizations such as custom data paths and arithmetic representations on FPGAs [15], [16]. Cost-effective training is especially desirable for large networks, since training large models like GPT-3 [17] using GPUs may cost several million USD [18], [19].

Implementing training on FPGAs is challenging. In order to take advantage of the customisability of these devices, it may be necessary to significantly redesign and/or refine the underlying machine learning model. As a result, multiple expensive training runs are needed for adjustments to ensure acceptable accuracy and convergence speeds, increasing the cost of training several-fold. Often, the need for these adjustments only become apparent after long run-times; issues such as overfitting, poor generalization performance, sudden drops of accuracy, and long-term numerical instabilities may only become observable after many training iterations. We believe that the ability to diagnose such problems during training by tracking the behaviour of the circuit as it runs is essential to effectively create networks suitable for training on FPGAs.

Frameworks that gather run-time information of a running circuit have been proposed. For machine learning inference applications, frameworks which provide on-chip visibility of large matrices and arrays by recording the behaviour of the design as it runs at speed have been presented [20], [21]. These techniques, however, may not work well for training applications. These techniques rely on identifying and inserting a small subset of *debug instruments* at compile time, limiting the range of behaviours that can be observed. For training, we anticipate that more flexible instrumentation that allows us to observe many different aspects of the training behaviour would be desirable, as would the ability to stream this data off-chip rather than storing it in on-chip buffers. Flexibility will cost chip area, however, since training is often performed on large data-centre FPGAs, it may be more acceptable to insert larger and more flexible instrumentation.

In this paper, we present a flow to accelerate the debug of machine learning training on FPGAs. Our contributions in this paper are the following:

1) We provide motivational examples that highlight the need for increasing the observability of the run-time

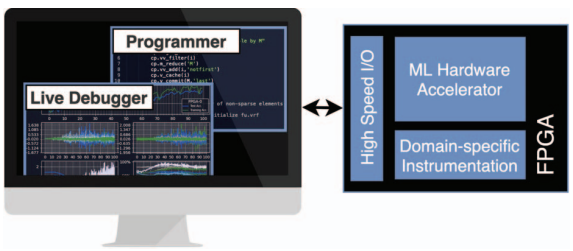

Figure 1. Training Debug Instrumentation

2576-2621/21/$31.00 ©2021 IEEE
DOI 10.1109/FCCM51124.2021.00011

behaviour of training applications,
2) We propose a flexible on-chip debug infrastructure for FPGA machine learning training, describing its architecture and implementation in detail, and
3) We quantify the impact of adding such an infrastructure to hardware accelerators and study how this impact changes according to a set of parameters.

This paper is organized as follows. Section II describes recent domain-agnostic and domain-specific efforts in on-chip debug. Section III then presents a taxonomy on machine learning bugs and presents a motivational example for this work. Section IV introduces our enhanced debug flow and instrumentation architecture. Section V shows examples of data gathering techniques enabled by our instrumentation. Section VI evaluates our proposal in terms of data gathering capabilities, area overhead, and circuit speed.

II. Previous Work and Context

A. Machine Learning Software Debug

High-level software debug of the machine learning model is essential to catch bugs at an early stage. In [22] researchers at Google presented TensorFlow Debugger (tfdbg), a specialized debugger for TensorFlow dataflow-based graphs. Tfdbg focuses on organizing the intermediate and internal graph states and presenting them in a clear and understandable fashion by keeping copies of intermediate values as they flow through the graph. This gives the user more visibility into the model execution graph, which otherwise would be encapsulated as a black-box function call, abstracting away internal graph detail, parallel and potentially distributed execution routines. This command-line interface (CLI) of tfdbg has been extended as a graphical user interface (GUI) in TensorBoard [23]. Tensorboard also includes a graph visualizer that helps users understand complex machine learning architectures by performing a series of transformations to declutter the graph.

Some work has also considered techniques to debug problems based on a reference model. In [24] Uber presented Manifold, a framework that utilizes visual analysis techniques to compare and debug pairs of similar machine learning models (e.g. a full model and its distilled version). This model-agnostic tool does not rely on access to the internal logic of the model. Instead, Manifold visually compares different statistical metrics of the inputs and outputs of the pair models, allowing the user to focus on those discrepancies (symptoms) and make an initial hypothesis of the problem.

Other machine learning software debug work focuses on analysing deep learning models during training. In [25], researchers from Microsoft proposed TensorWatch, which provides a way to perform interactive queries on live processes instead of constantly interrupting the system for queries. TensorWatch also focuses on extensibility, by allowing the user to visualize the logged data in custom ways. and temporarily displaying them to the user without logging.

The main limitation of software debug work is the speed in which data can be gathered at low granularity, making it impractical to diagnose some types of training problems in large models. We believe that if a hardware accelerator is required to make training timely and economically viable, problems that only become apparent after long run-times should be diagnosed on-chip.

B. Machine Learning Hardware Debug

Recent work on on-chip machine learning debug focused on adding instrumentation into the machine learning accelerator to increase visibility into the design by recording selected signals over time.

Unlike traditional on-chip debug [26]–[28], which is domain-agnostic, the authors in [20] explored the creation of an on-chip debug infrastructure specifically tailored to machine learning circuits. The key idea behind this contribution is that it is not necessary to record the raw history of how signals change over time to gain insight on the internal behaviour of a given circuit. Instead, data is compressed in a domain-specific way, allowing the user to decide which kind of information should be recorded on-chip. As a result, the trace buffer memory resources can be utilized to observe the circuit for a substantially longer period, accelerating the diagnosis of more complex problems.

This work was later expanded in [21], which addresses the need for lower debug turns-around times. This need comes from the iterative nature of debug, which requires the user to observe different things as the user refines his or her understanding of the circuit. In this scenario, previous instrumentations were not enough, due to the need to recompile the entire circuit every time the user wanted to observe something different, or observe the same thing in a different way. This problem was addressed by a configurable instrumentation that can select between a few pre-determined data compression circuits at debug time. In addition, the instrumentation in [21] allows for the signals/matrices being traced and the organization of the trace buffer to be adjusted at debug time.

C. Baseline

In this work we adopt the infrastructure proposed in [21] as our baseline. A key limitation of the baseline is the lack of flexibility in how data can be observed. In the instrumentation in

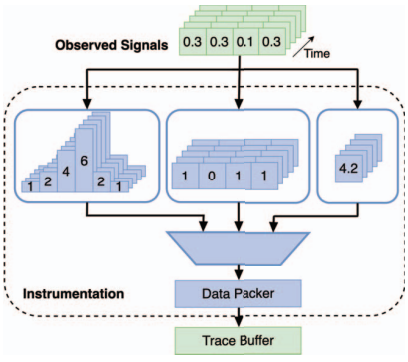

Figure 2. Baseline instrumentation

Figure 2, three data compression schemes have been included in the instrumentation, and the user can switch between these schemes at debug time. However, if a different compression scheme is needed, then new instrumentation must be created and the circuit recompiled. In this paper, we overcome this limitation by proposing a programmable debug infrastructure that can be programmed at the firmware level at run time. Note that although our instrumentation has been optimized to allow for computing statistics that are useful during training, it also allows for the computation of the simpler statistics proposed in [21].

Another important difference from this previous work includes the ability for live monitoring and debugging the circuit, which is essential for debugging training circuits as further discussed in Section IV.

III. Machine Learning Training Bugs

A. Machine Learning Hardware Bug Taxonomy

In this work we adapt the taxonomies presented in [29], [30]. For each description of the accelerator (software baseline, firmware and hardware) we classify bugs into to five types:

Inherited Bugs: This group consists of bugs that predate the beginning of the current development cycle. These include bugs in tools and bugs that were already present in higher-level descriptions of the accelerator.

Data Bugs: These are bugs related to an unexpected behaviour of the input data. This includes problems such as incorrectly labeled samples, values out of range and malformed or missing data samples.

Syntax Bugs: These are bugs related to any failure to comply with the set of rules of the language being used, such as case-sensitivity and the enforced order of operands.

Structural Bugs: These are bugs related to problems in the general logic and semantics of the accelerator, causing the implementation to differ from its original description. Structural bugs range from simple problems, such as a trivial error when converting between units, to complex errors such as a combination of multiple elusive logic mistakes.

Conceptual Bugs: These are bugs related to false assumptions about the suitability of the machine learning model itself and its interactions with the hardware. Conceptual bugs include problems such as assuming that a given data type would be enough to allow for the proper training of a given model, or assuming that a certain model would not overfit given its topology and hyperparameters. Conceptual bugs may result in problems such as lower than expected accuracy, failure to converge or suboptimal convergence speed.

Different from previous work in machine learning hardware debug, which focused on structural bugs during inference, *we focus on conceptual bugs* that only become apparent during training. We consider those bugs especially challenging, since they may be prohibitively long to expose using simulation and may require observing the system in multiple different points of the training process before their overall behaviour can be understood.

B. Motivational Example

To demonstrate an example conceptual bug, we modeled a DNN with multiple dense layers and ReLu activations in all hidden layers to perform a simple classification task. Note that this network is intentionally small in order to allow for rapidly extracting statistics about the training process. Although this bug is illustrated using a small network modeled in software, we anticipate that similar problems may happen with significantly larger networks trained in hardware.

As shown in Figure 3(a), the DNN being trained behaves well during the first few epochs, but the accuracy significantly drops after Epoch 33 and recovers after Epoch 44. This problem would not be visible in an initial RTL-level simulation, since only the first few training steps would be simulated. When running such a network in a hardware accelerator without any instrumentation, the machine learning expert would only be able to observe the drop in accuracy as shown in Figure 3(a), but would have no more information to help diagnose the problem. Moreover, a software-only simulation could also be slow and behave differently from the circuit that has been implemented with limited precision.

In this scenario, there are multiple statistics that could be used to help diagnose this problem as shown in Figure 3(b,c,d,e). Figure 3(b) shows the gradients of different layer over time, which shows that most gradients become zero as soon as the accuracy starts to drop. The cause of this drop is shown in Figure 3(e), which shows that the sparsity of Layer 9 becomes nearly 100% around the same time that the drop in accuracy occurs. This is known as the 'dead ReLu problem', in which layers with most of its ReLus dead will always output approximately the same value for any given input. Once most of the layer ends up in this state, the layer is unlikely to recover, as the function gradient of a ReLu at zero is also zero.

A possible way of addressing this problem is to use Leaky ReLus instead of traditional ReLus, which have a small positive gradient for negative inputs. However, this will cause the sparsity of all layers to be nearly zero, negating some of the benefits of accelerators that profit from sparsity. Moreover, the machine learning expert debugging this system might be interested in understanding the cause of the ReLus dying, since this might be only a symptom of a larger underlying problem.

Note in Figure 3(e) that the sparsity of the network progressively moved towards 100%, which means that no particular batch of inputs was solely the reason for the problem. More interestingly, Figure 3(d) shows that the mean value of activations started to significantly vary between batches after Epoch 60, indicating that the network is possibly overdimensioned or that the training step is too high for the given loss landscape. Note that this behaviour in activations would be averaged out and become less evident if the circuit was only observed every few epochs, showing the need for observations to be performed frequently. Moreover, deciding to observe the training circuit using RTL-level simulation after Epoch 60 would be both slow and not an obvious place to start.

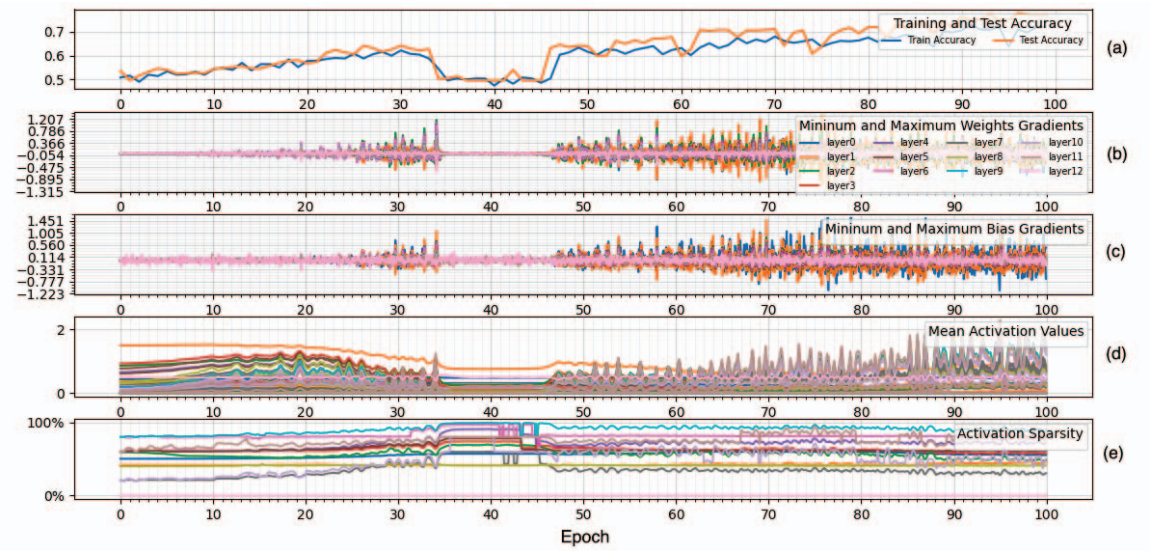

Figure 3. Motivational example showing a debug challenge that can benefit from on-chip instrumentation. Data obtained using a software model.

IV. Enhanced Debug Flow

A. Overall Approach

Similar to previous flows, we insert instrumentation that performs on-chip domain-specific compression to gather aggregated information about the behaviour of the machine learning model, rather than recording raw values over time. Different from previous flows, instead of restricting the user to select among a few predefined ways of observing the circuit during inference, we create an architecture that is optimized for observing training models, while also being flexible enough to allow for a large variety of custom ways to observe the model. Although, as we will show, our architecture requires more chip area than previous techniques; we argue that this may be less critical in training applications which are often performed on large data-centre FPGAs, compared to inference applications which are sometimes implemented in smaller edge-oriented FPGAs.

Our instrumentation also differs from previous work in the way it handles its gathered data. Rather than storing this information using precious on-chip memory resources, we stream this compressed data opportunistically off-chip, allowing for live monitoring and debugging of the machine learning model being trained. As discussed in Section V, this may enable early detection of a variety of problems, significantly saving time and reducing training costs.

We anticipate the typical use of our instrumentation to unfold as follows. First, at compile time, the designer selects which parts of the model could be observed by the instrumentation and adds the instrumentation to the design. The designer then programs the instrumentation by either selecting or creating custom firmware (see Section IV-D) and starts the training process, while live-monitoring different aspects of the learning model. At run time, the designer may reprogram the instrumentation to change how the model is observed. If a certain aspect of the training process is found to be unsuitable, the designer will then perform ad hoc adjustments. At this point, the designer may choose to either quit the possibly failing training process to reduce costs and debug offline, or perform live modifications to the network given that the right amount of controllability is built into the design.

B. Architecture Overview

As illustrated in Figure 4, the user circuit is connected to a single programmable instrumentation unit through a signal selection mechanism. Rather than allowing only a single vector input to reach the programmable instrumentation, the signal selection mechanism allows multiple input vectors to reach the instrumentation via time-multiplexing. As a result, instantiating multiple programmable instrumentation units may be avoided in many scenarios. We believe that time-multiplexing different vectors into the instrumentation is often a good solution, as signals may not be valid at all cycles, and sampling instead of recording all valid signals might also be acceptable when live monitoring the model and searching for the root cause of conceptual bugs.

Note that the programmable instrumentation has no trace buffer to store processed data. Instead, all data is sent off-chip, significantly reducing on-chip memory overhead. This is possible due to the compressing nature of gathering aggregated model information. In scenarios in which the interface to off-chip memory is unable to handle the instrumentation's throughput, back pressure may be applied to the instrumentation to ensure that a somewhat periodic sampling is achieved.

C. Programmable Instrumentation

Figure 5 shows the overall architecture of our programmable instrumentation. We refer to each block that composes our architecture as a *building block*. Building blocks are chained together, allowing data to only follow a predetermined data

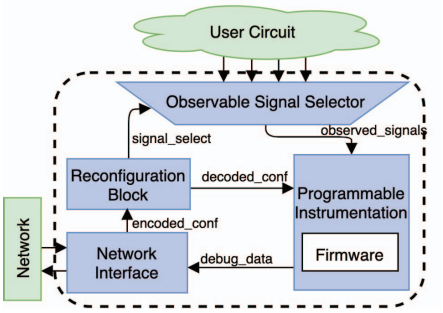

Figure 4. Overall Architecture Image

path. The order in which blocks are connected has been chosen to minimize the number of times data has to traverse the entire instrumentation in order to compute the statistics described in Section V. This order, however, is easily changeable at compile time and might be customized for power users that are not satisfied with the family of programmable infrastructures we provide.

The *Filter Unit* (FU) and the *Matrix-Vector Reduce Unit* (MVRU) are the building blocks responsible for handling data that needs to be within a specific range. The FU handles a vector of N elements and checks whether each of those elements is within M ranges, resulting in a binary $N \times M$ output matrix. A single FU can be used to check multiple different ranges, since those ranges are stored into a programmable memory in the FU. The MVRU then sums this data along the N or M axis and performs zero padding as appropriate. Those blocks may be used, for example, to create a histogram M bins per cycle or to check whether each element of the input array is valid or invalid by simultaneously checking for values out of range, NaNs and subnormals.

The Vector ALU (VALU) performs simple element-wise operations, such as addition, subtraction, and multiplication. This building block also has direct access to a scratchpad that can be used to store intermediate computations when multiple loops though the chained instrumentation are needed. The Vector-Scalar Reduce Unit (VSRU) may be used to reduce all inputs to a single element by either summing or multiplying them.

The Data Packer (DP) is responsible for packing elements into blocks of N elements. Although the DP will always receive N elements as an input, not necessarily all of those inputs are valid. Elements that have been processed by the VSRU, for example, will only have one valid element in its array of N elements. Similarly, the MVRU may also output only M valid elements depending on the operation performed. The DP is crucial to compress the data, minimizing the amount of information that needs to be sent off-chip.

D. Describing the instrument functionality

Different from previous work, which allows the designer to select among a handful ways of observing the model, our architecture enables the designer to create a large variety

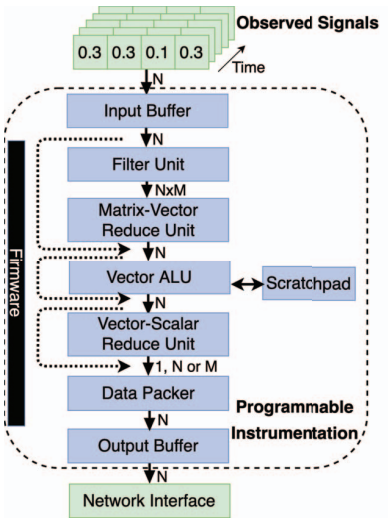

Figure 5. Programmable Instrumentation Architecture

of ways to observe the model at run time. The behaviour of the instrumentation at a given run is described by the *firmware*. Note that changing the firmware does not require resynthesizing the circuit, unless the new firmware requires a specific building block that has not been added to the design at compile time.

A firmware specification is composed of a sequence of instructions that describe the operations that will be performed for each input vector. A firmware specification may contain one or more *chains*, which are sequences of instructions that describe a single pass through the entire architecture. A new chain is initiated each cycle, given the heavily pipelined nature of our instrumentation. The order in which the building blocks are placed in the instrumentation dictates the possible ways in which data may flow and, consequently, the allowed order of operations in a given chain. Complex operations may be achieved by splitting the computations into multiple chains.

```
# Get multiple stats for each set of input vectors
def summaryStats(cp):

    # Sum of all values
    cp.begin_chain()
    cp.vv_add(0,'notfirst')
    cp.v_cache(0)
    cp.v_reduce()
    cp.v_commit(1,'last')
    cp.end_chain()

    # Number of sparse elements
    cp.begin_chain()
    cp.vv_filter(0)
    cp.m_reduce('N')
    cp.vv_add(1,'notfirst')
    cp.v_cache(1)
    cp.v_reduce()
    cp.v_commit(1,'last')
    cp.end_chain()

    return cp.compile()
```

Listing 1. Sample firmware for simple summary statistics

A sample firmware specification that computes simple summary statistics is shown in Listing 1. This firmware specification is composed of two chains, which means that each input vector only needs two clock cycles to be consumed by the instrumentation.

To increase the flexibility of our infrastructure without requiring a costly processor-like control structure, individual instructions may be predicated only by a few key conditions. We found that the existence of these conditions combined with the use of the scratchpad allows us to compute a large variety of statistics that are useful for monitoring and debugging a training model as discussed in Sections V and VI.

V. Training-specific Data Aggregation

In this section, we show how our architecture can be used to increase the observability of a training circuit running on an FPGA. As described in Section IV-D, our instrumentation can be configured at run-time, using firmware, to aggregate data in ways that may be useful to help the designer reason about the model under evaluation. To make our discussion concrete, we focus on six examples that we believe are well positioned to aid in the understanding of training-specific problems. The first three, which we will refer to as our *Baseline Aggregations*, were also used in [21] and were originally intended to help diagnose inference problems. The remaining three, which we will refer to as our *Training Aggregations*, are new to this paper, and were specifically designed with training in mind. In Section VI, we use these six examples as benchmarks to evaluate the effectiveness of our technique.

1. Distribution: This data gathering technique bins the frequency count of the observed values over a user-defined period, resulting in data that can be visualized using histograms.

2. Spatial Sparsity: The Spatial Sparsity gathers whether each specific element is zero or non-zero, allowing the designer to have a low-resolution visualization of activations and weights.

3. Summary Statistics: The summary statistics compresses all data received within a user-defined period of time to a single value, such as the mean or the number of sparse elements.

4. NormCheck: NormCheck is a data aggregation technique inspired by the effect of the Batch Normalization (BatchNorm) layer [31] in a training circuit. Although BatchNorm was originally believed to accelerate training by reducing the *Internal Covariate Shift* (ICS) [31], it was later shown that the actual reason for its success is its impact on the smoothness of the loss landscape [32]. BatchNorm, however, may offer another advantage for hardware accelerators: it increases the initial stability of the distribution of network activations, which may be critical in systems that operate with limited precision.

A designer considering including BatchNorm in a model may wish to understand the distribution of activations in a network as it is being trained. Our instrumentation can be used to gather this information as the circuit is running. Specifically, our NormCheck implementation measures the 15th, 50th, and 85th percentiles of a set of inputs (typically the activations) over a period of time. To demonstrate this, we created a software model and gathered the results in Figure 6. Figure 6 (a) shows the accuracy over time demonstrating the impact of BatchNorm. Figures 6 (b-d) shows the frequency distribution for a typical hidden layer of those networks, which corresponds to the 15th, 50th, and 85th percentiles of the network activations, over a period of time. We propose that we use our instrumentation to compute this later data. The NormCheck shown in Figures 6 (b), shows that the distribution of the activations in the network without batch normalization quickly grow overtime, which, if implemented in hardware, may cause overflows in architectures with lower precision. Conversely, Figure 6 (c) suggests that the use of BatchNorm in all layers is able to make those values stable over time, avoiding numerical problems. Interestingly, Figure 6 (d) show that by the same effect can be achieved, even if only a single layer has BatchNorm.

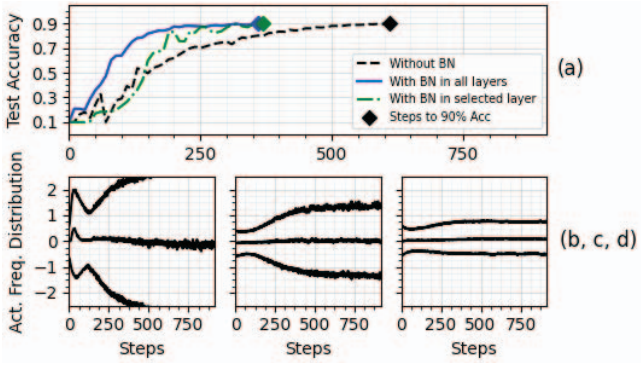

Figure 6. (a) The test accuracy of a network trained with and without Batch Normalization (BN), vs. the number of training steps. (b, c, d) The change in the activation distribution without BN, with BN and with BN in a selected layer, respectively, over time shown as {15, 50, 85}th percentiles. Results obtained using a software model.

The percentiles that compose NormCheck are computed by our instrumentation using an approximation technique. First, a 64-bin distribution of the activations is computed. Those values are then sent off-chip, where the percentiles are approximated and new uneven ranges for the distribution are defined. Those ranges are then opportunistically updated on-chip at run-time, ensuring that a good approximation of the percentiles can be constantly obtained.

5. Activation Predictiveness: We can also use the instrumentation to allow the designer to check whether the activations of a certain layer are somehow correlated with the test accuracy, which would indicate that a numerical problem is manifesting at that specific layer. Figure 7 (a) shows the test accuracy over time of a simple network performing classification, while Figure 7 (b) shows the activation predictiveness of one of its layers (again, computed using a software model). The activation predictiveness is given by:

$$AP_t = \frac{1}{M} \sum_{m=t-M}^{t} max(\bar{a}_{n,m})$$

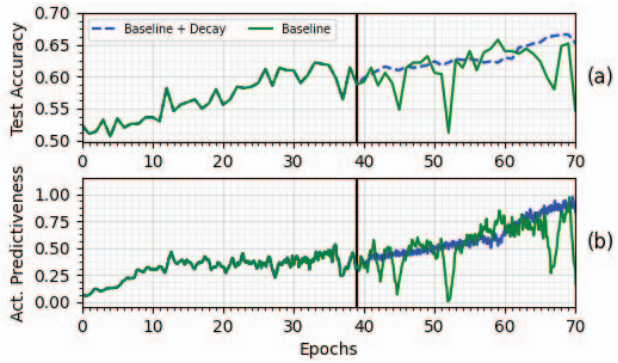

Figure 7. (a) Test accuracy of a sample network over a large number of epochs. (b) Activation Predictiveness of penultimate layer, showing that drops in the predictiveness are correlated with drops in the network's accuracy. Data obtained using a software model.

where $\bar{a}_{n,m}$ is the average node activation of node n at time step m, and M is the number of time steps used for a simple moving average.

Note that the computation of the activation predictiveness does not take any labels into consideration. The similarity between Figures 7(a) and (b) suggests that the sudden drops of accuracy experienced by the network is due to some numerical instability in the network.

The 'Baseline + Decay' plot on Figures 7 (a) and (b) show what would happen if the designer decided halfway through the training process to minimize the effects of this numerical instability by significantly accelerating the network's learning rate decay. As a result, the accuracy slowly increases over time, which is the desired behaviour.

Note that this kind of observation is better performed on-chip, as streaming the activations off-chip to do this analysis would be costly and this is something that must be continuously observed throughout training.

The activation predictiveness is computed by our instrumentation by first calculating the mean of the activations of all nodes, followed by checking the maximum values between those nodes. The moving average is computed offline, as calculating it on-chip would not reduce the amount of information that needs to be sent off-chip.

6. Total Invalidity: During training, weights, activations and gradient values might suffer from different numerical anomalies. Although those anomalies in a small scale may not cause major harm, tracking the total number of invalid elements over time might help better understanding when a numerical problem started to manifest. Total Invalidity simultaneously checks whether each specific element is a subnormal, +inf, -inf or NaN and accumulates this value over time for each training step.

VI. Evaluation

Our programmable debug instrumentation offers run-time programmable data gathering capabilities, and is characterized by a number of parameters that allows the designer to trade-off those capabilities with area overhead. In this section, we will first show how the area, and speed of different variants of our instrumentation compare to the baseline, which only allows data to be gathered in a limited number of ways defined at compile-time. We will then perform an architectural study to investigate the overhead of those variants when different numerical precisions and arithmetical representations are used.

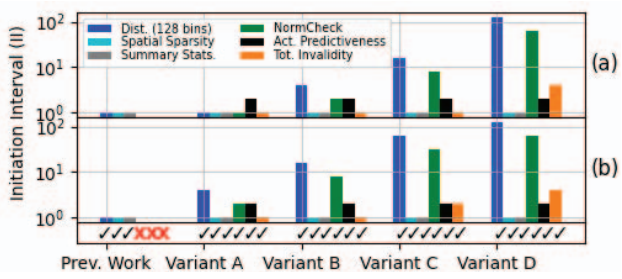

Figure 8. Initiation Interval (II) of different workloads with N=32 (a) and N=128 (b). Some workloads are not possible to compute in previous work.

A. Capabilities and overhead compared to baseline

To compare with previous work, we use four different variants of our debug instrumentation (Variants A-D), each of which is parameterized in four different ways. Those variants differ both in terms of the input vector width (N) connected to the user circuit, as well as in terms of the range parameter M, which dictates many of the instrumentation's capabilities as discussed in Section IV. All of the experiments were performed using Quartus Prime Pro 20.3 and targeting a Stratix 10 1SG280LN2F43E1VG.

In order to allow for a fair comparison, previous work has been adapted to target vectorized circuits. Also note that the different variants of our instrumentation are all able to gather the same kinds of aggregated information. However, different variants may take a different number of clock cycles to perform the same task. Variant A represents the most capable of our instruments, while Variant D corresponds to the least capable of our variants.

Figure 8 shows the Initiation Interval (II) of our instrumentation when compared to previous work under the different workloads presented in Section V. As shown in this figure, the initiation interval of the previous work is always 1, which is ideal for observing short periods of the circuit's execution, but unnecessary for long-term monitoring. In contrast, our instrumentation often needs multiple cycles to perform the computations required in many data gathering techniques, but the more time we allow for the processing of this information, the lower is our area overhead. Importantly, previous work only has the flexibility of computing the baseline aggregations, while ours is capable of being programmed to gather data in a large variety of ways.

Table I shows the area overhead and reported impact on F_{max} of the proposed instrumentation when compared to previous work.

As shown in Table I, both previous work and Variant A may consume an unreasonable amount of area, especially

Table I
OVERHEAD OF DEBUG INSTRUMENTATION COMPARED TO BASELINE

Configuration	Vector Width (N)	FMax (MHz)	Area (ALMs)	Normalized Area
(1) Previous Work† [21]	1	300	0.9k (0.1%)	-
	16	242	14.1k (1.5%)	1x
	32	231	27.7k (2.9%)	1x
	64	216	53.3k (5.7%)	1x
	128	200	106.5k (11.4%)	1x
(2) Variant A (M=N)	16	191	9.8k (1.10%)	0.69x
	32	160	28.7k (3.0%)	1.03x
	64	145	95.3k (10.2%)	1.78x
	128	129	342.6k (36.7%)	3.21x
(3) Variant B (M=N/4)	16	177	6.2k (0.7%)	0.43x
	32	177	14.4k (1.5%)	0.51x
	64	169	38.1k (4.0%)	0.71x
	128	129	114.0k (12.2%)	1.07x
(4) Variant C (M=N/16)	16	178	5.1k (0.5%)	0.35x
	32	173	10.9k (1.1%)	0.39x
	64	185	23.8k (2.5%)	0.44x
	128	165	57.0k (6.1%)	0.53x
(5) Variant D (M=1)	16	178	5.1k (0.5%)	0.35x
	32	184	10.3k (1.1%)	0.37x
	64	177	20.3k (2.1%)	0.38x
	128	168	40.3k (4.3%)	0.37x

† Assuming distribution engine with 128 bins.

when observing wide vectors. This high overhead is caused by the design choice of prioritizing the frequency in which new information can be tapped, instead of allowing for more reuse of the instrumentation. In contrast, Variants B, C, and D show progressively lower area overhead at the cost of additional cycles to gather data. The reported impact on F_{max} of our instrumentation is slightly higher than previous work, and we anticipate that this impact could be further reduced by pipelining the instrumentation.

Note that different from [21], our results don't focus on the overhead in terms of memory, since our instrumentation continuously streams this data off-chip instead of recording data on-chip until the end of the execution. However, if we were to store data on-chip, our compression ratio for the data gathering techniques presented in [21] would be very similar, since we use an analogous data packing mechanism as discussed in Section IV.

B. Study on numerical precision and representation

This architectural study aims to evaluate the overhead of different variants of our architecture when using different numerical precisions and representations. Note that the arithmetical representation being used by the debug instrumentation does not necessarily need to match the arithmetical representation of the user circuit. However, if the user circuit operates in low precision, our instrumentation is in a better position to also use reduced precision, resulting in significantly lower area overhead.

Figure 9 shows the area overhead of multiple variants of our instrumentation when different numerical precisions are used. For this experiment, we fixed the vector width N to 64 elements, as we believe this represents a typical use case scenario. As shown in Figure 9, the area overhead of variant A is significantly larger than the overhead of all other variants,

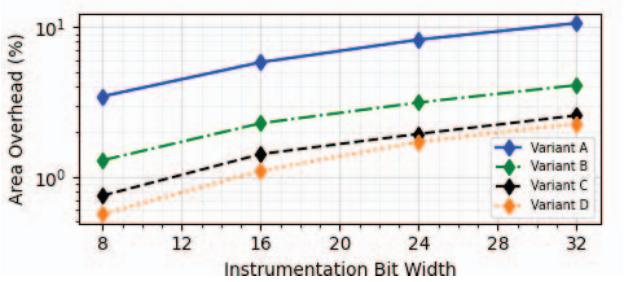

Figure 9. Overhead of instrumentation under different fixed-point bit widths.

even when a lower precision is used. For all widths, the overhead of Variant C is only slightly larger than the overhead of Variant D, although it is significantly more capable.

We also performed initial studies on the possible use of Block Floating Point (BFP) as a way to decrease the area overhead of our debug instrumentation. BFP has been identified as a promising alternative representation for machine learning workloads for both inference and training due to its efficiency when performing multiply-accumulate operations [33]–[35]. However, our experiments have shown that BFP is not a good alternative for debug instrumentations like ours, since a significant part of our area overhead lies in the extensive use of comparators. The need of matching the exponents of the BFP operands before performing those operations requires the use of multiple cycles for an efficient implementation, which causes a significant increase of our initiation interval, outweighing the benefits of the lower overhead.

VII. CONCLUSION

In this paper, we presented a flexible debug instrumentation for live on-chip debug of machine learning training on FPGAs. Different from traditional general-purpose on-chip debug work, our instrumentation generates aggregated data, which compresses information in a domain-specific way, allowing for the live transmission of debug data off-chip. Different from previous work on domain-specific on-chip debug, our infrastructure is firmware programmable, allowing the designer to gather debug information on a large variety of ways, instead of being constrained by a few options defined at compile-time. We showed that this added flexibility allows us to gather information that can be used to more quickly understand efficiency and accuracy problems on training models, significantly reducing training costs of large models. Although the area overhead of such instrumentation can be significant, we show that this overhead can be drastically reduced by trading off area and the time between subsequent circuit observations. Overall, we believe that the inconvenience of the higher overhead is outweighed by the benefits of the flexibility provided by our instrumentation.

DOWNLOAD

The source code for the proposed instrumentation (including examples and documentation) can be downloaded from github.com/danielholanda/LeBug.

REFERENCES

[1] S. Fox, S. Tridgell, C. Jin, and P. H. W. Leong, "Random projections for scaling machine learning on fpgas," in *2016 International Conference on Field-Programmable Technology (FPT)*, 2016, pp. 85–92.

[2] Amazon. Amazon ec2 f1 instances: Enable faster fpga accelerator development and deployment in the cloud. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/

[3] Baidu. Fpga cloud server. [Online]. Available: https://cloud.baidu.com/product/fpga.html

[4] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," *IEEE Micro*, vol. 35, no. 3, pp. 10–22, 2015.

[5] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, and et al., "A configurable cloud-scale dnn processor for real-time ai," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. IEEE Press, 2018, p. 1–14. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00012

[6] E. Nurvitadhi, D. Kwon, A. Jafari, A. Boutros, J. Sim, P. Tomson, H. Sumbul, G. Chen, P. Knag, R. Kumar, R. Krishnamurthy, S. Gribok, B. Pasca, M. Langhammer, D. Marr, and A. Dasu, "Why compete when you can work together: Fpga-asic integration for persistent rnns," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 199–207.

[7] T. Geng, T. Wang, A. Sanaullah, C. Yang, R. Patel, and M. Herbordt, "A framework for acceleration of cnn training on deeply-pipelined fpga clusters with work and weight load balancing," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 394–3944.

[8] H. Zeng and V. Prasanna, "Graphact: Accelerating gcn training on cpu-fpga heterogeneous platforms," in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 255–265. [Online]. Available: https://doi.org/10.1145/3373087.3375312

[9] T. Wang, T. Geng, A. Li, X. Jin, and M. Herbordt, "Fpdeep: Scalable acceleration of cnn training on deeply-pipelined fpga clusters," *IEEE Transactions on Computers*, vol. 69, no. 8, pp. 1143–1158, 2020.

[10] Wenlai Zhao, Haohuan Fu, W. Luk, Teng Yu, Shaojun Wang, Bo Feng, Yuchun Ma, and Guangwen Yang, "F-cnn: An fpga-based framework for training convolutional neural networks," in *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2016, pp. 107–114.

[11] D. Lo, B. D. Darvish, E. S. Chung, Y. Zhao, A. Phanishayee, and R. Zhao, "Adjusting activation compression for neural network training," Microsoft Technology Licensing LLC, U.S. Patent 20200264876A1, Aug. 2020.

[12] D. C. Burger, E. S. Chung, and B. D. Rouhani, "Incremental training of machine learning tools," Microsoft Technology Licensing LLC, U.S. Patent 20200265301A1, Aug. 2020.

[13] K. Denolf, N. Fraser, K. A. Vissers, and G. Gambardella, "Training of neural networks by including implementation cost as an objective," Xilinx Inc, WIPO Patent WO2020068437A1, Apr. 2020.

[14] K. Denolf and K. A. Vissers, "Architecture optimized training of neural networks," Xilinx Inc, U.S. Patent US20190057305A1, Feb. 2019.

[15] S. Kolala Venkataramanaiah, Y. Ma, S. Yin, E. Nurvithadhi, A. Dasu, Y. Cao, and J. Seo, "Automatic compiler based fpga accelerator for cnn training," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 166–172.

[16] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, p. 1–12, Jun. 2017. [Online]. Available: https://doi.org/10.1145/3140659.3080246

[17] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[18] C. Li, "Openai's gpt-3 language model: A technical overview," jun 2020. [Online]. Available: https://lambdalabs.com/blog/demystifying-gpt-3/

[19] K. Wiggers, "Openai launches an api to commercialize its research," jun 2020. [Online]. Available: https://venturebeat.com/2020/06/11/openai-launches-an-api-to-commercialize-its-research/

[20] D. H. Noronha, R. Zhao, J. Goeders, W. Luk, and S. J. Wilton, "On-chip FPGA Debug Instrumentation for Machine Learning Applications," in *Int'l Symposium on Field-Programmable Gate Arrays (FPGA)*, Feb 2019, pp. 110–115.

[21] D. Holanda Noronha, R. Zhao, Z. Que, J. Goeders, W. Luk, and S. Wilton, "An overlay for rapid fpga debug of machine learning applications," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 135–143.

[22] S. Cai, E. Breck, E. Nielsen, M. Salib, and D. Sculley, "Tensorflow debugger: Debugging dataflow graphs for machine learning," 2016.

[23] K. Wongsuphasawat, D. Smilkov, J. Wexler, J. Wilson, D. Mané, D. Fritz, D. Krishnan, F. B. Viégas, and M. Wattenberg, "Visualizing dataflow graphs of deep learning models in tensorflow," *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, no. 1, pp. 1–12, 2018.

[24] J. Zhang, Y. Wang, P. Molino, L. Li, and D. S. Ebert, "Manifold: A model-agnostic framework for interpretation and diagnosis of machine learning models," *CoRR*, vol. abs/1808.00196, 2018. [Online]. Available: http://arxiv.org/abs/1808.00196

[25] S. Shah, R. Fernandez, and S. Drucker, "A system for real-time interactive analysis of deep learning training," *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems - EICS '19*, 2019. [Online]. Available: http://dx.doi.org/10.1145/3319499.3328231

[26] Xilinx, *ChipScope Pro Software and Cores: User Guide*, October 2012.

[27] Intel, *Quartus Prime Pro Edition Handbook*, November 2015, vol. 3, ch. 9: Design Debugging Using the SignalTap II Logic Analyzer.

[28] R. Hale and B. Hutchings, "Enabling low impact, rapid debug for highly utilized fpga designs," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 81–813.

[29] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 510–520. [Online]. Available: https://doi.org/10.1145/3338906.3338955

[30] B. Beizer, *Software System Testing and Quality Assurance*. USA: Van Nostrand Reinhold Co., 1984.

[31] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015. [Online]. Available: http://arxiv.org/abs/1502.03167

[32] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry, "How does batch normalization help optimization?" 2019.

[33] B. Rouhani, D. Lo, R. Zhao, M. Liu, J. Fowers, K. Ovtcharov, A. Vinogradsky, S. Massengill, L. Yang, R. Bittner, A. Forin, H. Zhu, T. Na, P. Patel, S. Che, L. C. Koppaka, X. Song, S. Som, K. Das, S. Tiwary, S. Reinhardt, S. Lanka, E. Chung, and D. Burger, "Pushing the limits of narrow precision inferencing at cloud scale with microsoft floating point," in *NeurIPS 2020*. ACM, November 2020.

[34] M. Drumond, T. Lin, M. Jaggi, and B. Falsafi, "Training dnns with hybrid block floating point," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18. Red Hook, NY, USA: Curran Associates Inc., 2018, p. 451–461.

[35] ——, "End-to-end dnn training with block floating point arithmetic," in *arXiv*, 04 2018.