On-chip FPGA Debug Instrumentation for Machine Learning Applications

Daniel Holanda Noronha[1], Ruizhe Zhao[2], Jeff Goeders[3], Wayne Luk[2] and Steven J.E. Wilton[1]
[1]University of British Columbia, [2]Imperial College London, [3]Brigham Young University
danielhn@ece.ubc.ca,ruizhe.zhao15@imperial.ac.uk,jgoeders@byu.edu,w.luk@imperial.ac.uk,stevew@ece.ubc.ca

ABSTRACT

FPGAs provide a promising implementation option for many machine learning applications. Although simulations or software models can be used to explore the design space of these applications, often the final behaviour can not be evaluated until the design is mapped to the FPGA and integrated into the target system. This may be because long run-times are required, or because the environment can not be adequately described using a software model. Once unexpected behaviour is observed, on-chip debug is notoriously difficult; typically a design is instrumented with on-chip trace buffers that record the run-time behaviour for later interrogation.

In this paper, we describe instrumentation that can accelerate the process of debugging machine learning applications implemented on an FPGA. Unlike previous work, our instrumentation is optimized to take advantage of characteristics of this application domain. Our instruments gather useful domain-specific information about the observed variables instead of recording the raw values of those elements. Results show that the proposed instruments provide at least 17.8x longer visibility in the most conservative of our experiments at a low area and latency cost.

CCS CONCEPTS

• **Hardware → VLSI**; **EDA**; **Design for debug**.

ACM Reference Format:
Daniel Holanda Noronha, Ruizhe Zhao, Jeff Goeders, Wayne Luk and Steven J.E. Wilton. 2019. On-chip FPGA Debug Instrumentation for Machine Learning Applications. In *The 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19), February 24–26, 2019, Seaside, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3289602.3293922

1 INTRODUCTION

Recent years have seen tremendous interest in machine learning algorithms and techniques. Although CPUs and GPUs are often sufficient for training and inference tasks, Field-Programmable Gate Arrays (FPGAs) may lead to implementations that are faster and/or lower power. These advantages have led many researchers to propose novel FPGA-oriented architectures and techniques, and have spurred significant commercial activity.

FPGA '19, February 24–26, 2019, Seaside, CA, USA

ACM ISBN 978-1-4503-6137-8/19/02…$15.00
https://doi.org/10.1145/3289602.3293922

Designing an FPGA-based machine learning application can be done using RTL-based design, high-level synthesis (HLS), or domain-specific translation flows (eg. [7, 12]). Regardless of how these applications are implemented, debugging these designs can be extremely difficult, for at least three reasons. First, unlike many application domains, inference and training tasks normally require very long run-times (many training or inference samples) before their overall behaviour can be understood. As an example, consider an error in a training application that causes many weights to be incorrectly clamped to zero; this could not be observed or understood without running many training samples. This means that hardware-oriented simulation techniques may not be sufficient to understand the operation of the design. Second, the "correctness" of machine learning applications can often not be determined by looking at individual signals/variables in isolation. As an example, during training, there is no "correct" value for a weight; the correctness depends on the ensemble of weights acting together. This means existing debug tools, which are optimized for examining the behaviour or specific signals are variables, are less useful. Third, machine learning applications are often designed at a high level (eg. C in HLS-based flows or Python in flows such as [7]) and automated tools are used to translate these high level designs into a circuit. This means that hardware-oriented debug flows, which provide visibility in the context of the hardware design, may not provide information in a form that is useful for the designer (this is similar to the observation in [5]).

In this paper, we present a flow to accelerate the debug of machine learning applications on FPGAs. Like existing hardware debug flows, we allow the user to add instrumentation that records the behaviour of the design as it runs at speed. Unlike existing flows, our instrumentation is optimized specifically for machine learning applications. Specifically, this work provides novel histogram, spatial sparsity, and summary generating instrumentation. These instruments are designed to summarize data on-chip in a way that maximizes the utilization of trace buffer space, while providing information that is meaningful to an engineer trying to understand the behaviour of the chip.

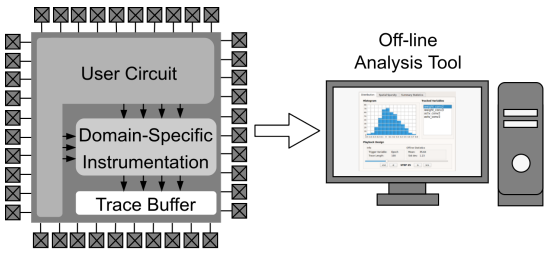

Figure 1: Domain-Specific Debug Instrumentation

Although the flow in this paper could be applied to any hardware implementation of machine learning algorithms, it is especially relevant for FPGAs for two reasons. First, FPGAs are often the first hardware implementation method for a new product (either as a prototype or before cost reduction), so this type of debugging is most likely to be performed in an FPGA. Second, this technique enables rapid debug and design, which is essential as FPGAs start to appear in cloud-based systems. We anticipate that FPGA companies are especially well-positioned to take advantage of our techniques.

This paper is organized as follows. Section 2 provides context for our work by describing recent efforts in on-chip debug instrumentation. Section 3 then describes our overall approach and instrumentation architecture. Section 4 shows how the data obtained from the instruments can be displayed to the user in a meaningful way. Section 5 evaluates our proposal in terms of trace length, area overhead, and circuit speed.

2 PREVIOUS WORK AND CONTEXT

Although software simulators are an important part of any debug ecosystem, simulation alone many not be sufficient to find the root cause of many types of bugs. Bugs that require long run-times to manifest, or those that occur due to specific input patterns may be missed by simulation. The only method to find the cause of these types of bugs is to execute the hardware in a real system, running at speed, driven by real input traffic.

Understanding the behaviour of a design running at speed is difficult due to limited I/O pins; FPGA vendors provide tools such as Intel's SignalTap II and Vivado's ILA [1, 14] to instrument a user design, at compile time, such that the behaviour of important signals can be stored on-chip during execution for later interrogation. Examining the traces of these signals can allow an engineer to understand the behavior of the design and try to track down the root cause of unexpected behaviour. Academic work has also considered adding such instrumentation at run-time [3, 11], decreasing the time between debug iterations.

Recent academic work has extended this instrumentation, optimizing it specifically for designs that are generated by high-level synthesis compilers [2, 4–6]. These works show that significant compression is possible by understanding the schedule of the design (which is available from the HLS tool) and using this information to only record signal values when they are scheduled to change. This compression allows a longer trace history to be stored on-chip, meaning fewer debug iterations are typically required to identify the root cause of a bug.

Debugging for machine learning applications is often done using software tools such as Tensorboard. Our approach is different in that Tensorboard is aimed at debugging a software model, while our approach is for debugging an application running on an FPGA.

3 DOMAIN-SPECIFIC INSTRUMENTATION

3.1 Overall Approach

Similar to previous flows, we instrument the user design at compile time to enable runtime recording of circuit behaviour (Figure 1). However, while previous work stores the raw values of variables

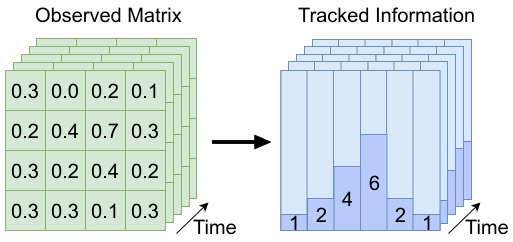

Figure 2: Distribution Instrument Overview

or signals into the on-chip trace buffers, we use on-chip domain-specific compression to store aggregated information about important variables or signals, making much more efficient use of memory. After the chip has been run, this information can be retrieved and used with our off-line analysis tool to allow the user to relate specific values to quantities in the original design, allowing them to better understand the behaviour of the circuit.

3.2 Debug Instruments

The domain-specific compression is done using one or more *instruments*, each of which monitors the behaviour of signals and summarizes the behaviour in the associated trace buffer(s). Each instrument summarizes the behaviour in a different way. The instruments we have selected are inspired by Tensorboard, which is used for debugging software implementations of machine learning applications. Below, we describe three of these instruments.

3.2.1 Distribution Instrument. Many machine learning applications consist of large arrays (eg. activations or weights). Often, during debugging, it is useful to understand the overall distribution of values in an array. This may help, for example, to determine if

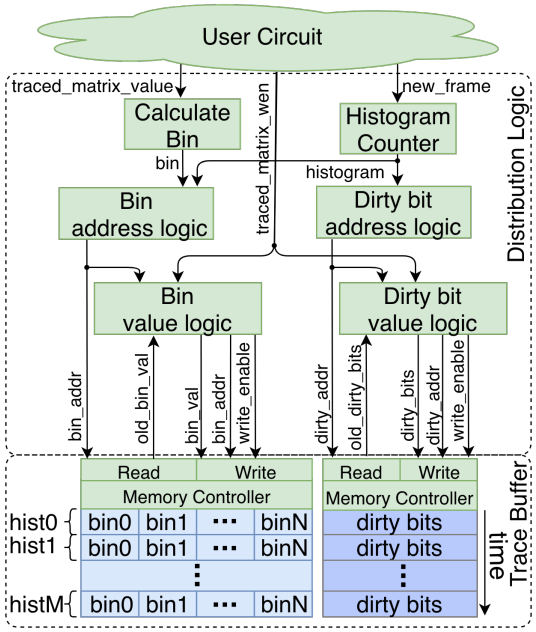

Figure 3: Distribution Instrument Architecture

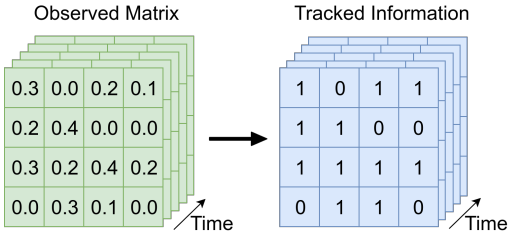

Figure 4: Spatial Sparsity Instrument Overview

an error is causing activations to "clamp" at a minimum or maximum value, or whether, during training, weights are spanning the entire space provided by their representation's bit-width. Using techniques such as [5], that capture raw variable values, it would be possible to record all values in an array, and then perform the distribution analysis off-line. However, for large arrays, this may result in very inefficient use of trace buffer memory; every change to every element in the array would consume an entry in the trace buffer.

Instead, we provide an instrument that monitors all words in a specified array (memory) and aggregates the values into a *histogram*. This is shown conceptually in Figure 2, and an architecture is shown in Figure 3. The trace buffer contains one word per bin in each histogram. Each time an element of an identified array is updated, comparators are used to select the appropriate bin, and increment the associated word in the trace buffer. After the run is complete, the histogram can be read, and the user can use this information to understand the nature of the run-time values of the activations or weights stored in the array.

Rather than storing a single histogram, our trace buffer is divided into *frames*, each large enough to store one histogram. In a CNN, a frame may represent all calculations corresponding to a single input image; in other types of deep learning networks, a frame may represent a single training or inference sample. At the start of each frame (sample or image), a new histogram is initiated. Since we implement our trace buffers using embedded memories with limited ports, we cannot reset all entries in a histogram to zero in one cycle. Instead, we use a second memory containing dirty bits to track which bins have valid or invalid values. Each word in the dirty bits memory contains the dirty bits for all the bins of one entire histogram. Every time a new histogram is initiated (start of a new frame), all the bits in the word corresponding to this new histogram are marked as dirty. If a bin needs to be incremented and its value is marked as dirty, the value stored to this bin will be one, and the associated dirty bit will be deasserted.

The user can choose multiple variables (arrays) to observe using our proposed distribution instruments. For each of the arrays selected, a copy of the architecture shown in Figure 3 is inserted.

This association of a histogram to a frame is unique in this work. In [5], events to signals and variables are stored in order, however, there is no explicit association between those events and the context (in our case a frame) for which they occur. This association provides debug information in a manner which may be more natural for the user, possibly leading to more insight and faster debug cycles. An envisaged user interface will be described in Section 4.

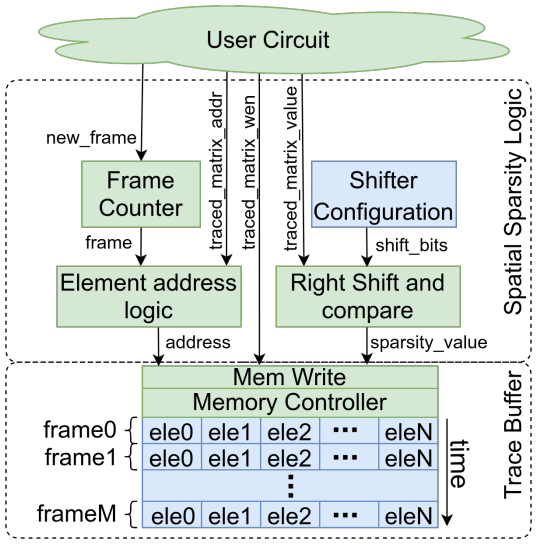

Figure 5: Spatial Sparsity Instrument Architecture

3.2.2 Spatial Sparsity Instrument. Many errors in machine learning applications can manifest as arrays of weights and/or activations that are zero (or close to zero). The Spatial Sparsity instrument monitors specified activation or weight arrays, and, rather than storing all updates to these arrays, stores an indication whether the array is zero (within a predetermined threshold) or non-zero. This provides information about the sparsity of a given array. This is shown conceptually in Figure 4 and an architecture for this instrument is shown in Figure 5. The same logic could also be used to track elements close to 1, another upper bound or not a number (NaN).

Again, the trace buffer is treated as a set of frames, where each frame corresponds to a single input sample or image. Since only one bit is required to represent each element of the observed array a very long trace length can be achieved when compared to simply tracking all the changes as in [5].

3.2.3 Summary Statistics Instrument. Multiple summary statistics can also be used to represent the data we are trying to observe. Some examples of summary statistics include measure of tendency, such as arithmetic mean; measure of statistical dispersion, such as standard deviation; and measure of shape of the distribution, such as skewness. Although many of those statistics can be explored for most of applications, there are some summary statistics that can be especially useful for debugging machine learning applications.

In our instrumentation we focused on calculating the sparsity of the observed matrix to illustrate the example of gathering summary statistics to assist debugging machine learning circuits. Differently from the spatial sparsity instrument, the sparsity summary statistic instrument will not retain information regarding the location of elements that are equal to zero. Instead, a simple counter is used to track how many of the elements written into the observed matrix are zero valued.

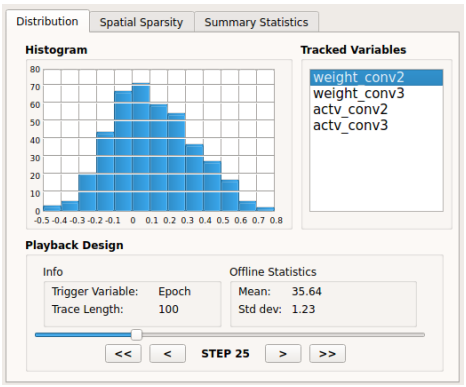

Figure 6: User interface for distribution instrument

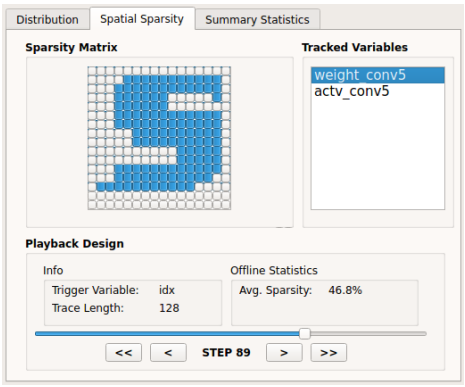

Figure 7: User interface for spatial sparsity instrument

4 USER INTERFACE

After the chip has been run, the data is obtained from the trace buffers and displayed to the user. Differently than hardware-oriented debug flows [1, 13] which display trace data using waveforms, or HLS-based debug flows [2, 5, 6] which display trace data in terms of user-visible C variables, we display data in the context of higher-level machine learning constructs. In the instrumentation architectures described in Section 3, data is associated with frames, each frame typically representing an input image in a CNN or input data item in other machine learning applications. In our user interface, we display the data in the context of frames in a form that we believe is useful for the user.

Our user interface consists of a set of tabs, as shown in Figures 6 and 7. Each tab corresponds to one type of debug instrument. Each debug instrument can be used multiple times to track different variables (arrays). For the distribution instrument, the associated tab shows a histogram. For the spatial sparsity instrument, the tab shows a graphical representation of sparsity. In both cases, the user can step through frames (rather than single-stepping lines of code) providing insight into how the traced matrix corresponds to the input image or dataset. Although it would be possible to obtain the same insight using a code-oriented debugger [5], the relationship between frame and variable value may not be clear, especially in the presence of hardware optimizations, possibly complicating the debugging process.

5 RESULTS AND DISCUSSION

In this section, we first compare our technique to the general-purpose debug instrumentation from [5]. We then present two architecture studies.

5.1 Comparison to General-Purpose Debug Instrumentation

To compare to previous work, we use kernels that are part of Convolutional Neural Networks (CNNs) generated using a tool that constructs CNNs of varying sizes and capabilities [15]. We use three kernels, one consisting of a single 28x28 convolution, one consisting of eight 28x28 convolutions and one consisting of 32 28x28 convolutions.

We compare seven different debugging configurations for each kernel. The first two configurations are from the general-purpose debug techniques in [5]. Unlike this work, these are not tailored to machine learning applications, and the debug logic is designed to simply capture raw variable values. The first configuration (1) tracks all user-visible variables, while the second (2) only tracks the elements in the array corresponding to the output of the convolutions. The remaining five configurations correspond to using the domain-specific debug instruments presented in this work to also observe the output of the convolutions. This includes configurations for the (3) distribution instrument with 32 bins, (4) the distribution instrument with 128 bins, (5) the spatial sparsity instrument, (6) the summary statistics instrument, and (7) a final configuration combining the distribution (32 bins) and spatial sparsity instruments (summary statistics sparsity is calculated off-line). As in [5, 9], 100Kb of trace memory was assumed in all configurations; in the configuration in which all instruments are included, the trace memory is partitioned among the those instruments.

For each kernel and configuration, the instrumentation is added, and the design synthesized, placed, and routed using Quartus II 16.1 (an average of 10 runs with different seeds is used). Table 1 shows the results. The fifth column shows the number of times that information about the frames can be stored in the trace memory. In the general-purpose approaches, where raw signal values are recorded, not even a single frame of convolution results can be stored in the trace buffer. However, using the data aggregation approaches presented in this paper, data for many frames can be stored in the same trace buffer space. The eighth column in the table shows the normalized trace size compared to configuration (2), which corresponds to the best general-purpose approach. When all proposed instruments are included, information about the frames can be stored for 21.8–24.1x longer.

Table 1 also shows area and F_{max} results (post place-and-route). As can be seen, the instrumentation is small, even when all three instruments are included. The impact of the instrumentation on F_{max} is similar to the previous work for large kernels; we anticipate we could reduce this impact by pipelining the instrumentation.

Although the proposed instruments significantly increase the trace size of large signals, it is not appropriate for tracing small signals. We believe that integrating those instruments with general purpose debug is an interesting venue for research.

Table 1: Resources and trace length when compared to general purpose debug

Configuration	Kernel	FMax (MHz)	LEs	Trace Size[§]	Normalized FMax	Normalized LEs	Normalized Trace size
Uninstrumented User Circuit	**32x28x28**	214.06	2483	-	1.00x	0.73x	-
	8x28x28	264.38	2439	-	1.01x	0.73x	-
	1x28x28	281.94	2308	-	0.97x	0.72x	-
(1) General Purpose [5] - Trace all signals	**32x28x28**	212.19	3575	0.003	0.99x	1.05x	0.024x
	8x28x28	255.02	3534	0.015	0.98x	1.06x	0.030x
	1x28x28	266.79	3397	0.132	0.92x	1.07x	0.033x
(2) General Purpose [5] - Single Matrix Trace	**32x28x28**	213.79	3391	0.124[†]	1x	1x	1x
	8x28x28	260.05	3324	0.498[†]	1x	1x	1x
	1x28x28	287.89	3167	3.985[†]	1x	1x	1x
(3) Distribution Instrument - 32 bins	**32x28x28**	200.48	2867	195	0.93x	0.84x	1,572.5x
	8x28x28	227.65	2834	223	0.87x	0.85x	447.7x
	1x28x28	229.87	2676	284	0.79x	0.84x	71.2x
(4) Distribution Instrument - 128 bins	**32x28x28**	189.62	3670	48	0.88x	1.08x	387.0x
	8x28x28	225.17	3600	55	0.86x	1.08x	110.4x
	1x28x28	228.98	3488	71	0.79x	1.10x	17.8x
(5) Spatial Sparsity Instrument	**32x28x28**	200.46	2547	3	0.93x	0.75x	24.1x
	8x28x28	211.13	2531	15	0.81x	0.76x	30.1x
	1x28x28	214.70	2393	127	0.74x	0.75x	31.8x
(6) Summary Statistics Instrument - Sparsity	**32x28x28**	213.17	2557	6666	0.99x	0.75x	53,758.0x
	8x28x28	258.75	2531	7692	0.99x	0.76x	15,445.7x
	1x28x28	285.30	2390	10000	0.99x	0.75x	2,509.4x
(7) Proposed instruments combined[♯]	**32x28x28**	189.23	2930	3	0.88x	0.86x	24.1x
	8x28x28	206.69	2927	14	0.79x	0.88x	28.1x
	1x28x28	220.51	2786	87	0.76x	0.87x	21.8x

[§] Number of times information about the entire 32-bit frame could be tracked.
[†] All memory bits are used for dataflow trace buffer and only the values of the observed matrix are traced.
[♯] Distribution (32bins) and spatial sparsity instruments; Sparsity summary statistic is calculated off-line.

5.2 Architecture Study 1: Distribution Instrument

In this experiment, we vary the number of bins used in the distribution instrument while keeping the number of histograms in the trace buffer constant, and measure the impact on the total memory bits, area, and speed. In all cases, the output of a 32-bit 32x28x28 convolution is tracked.

The results are shown in Figure 8. The horizontal axis corresponds to the number of bins used in each histogram, while different lines represent the size of the trace buffer in terms of the number of histograms that this trace buffer is able to store. The left-most point in each graph corresponds to an uninstrumented circuit (0 bins).

As shown, the frequency drops as the number of bins increases, however, the impact is less than 5% when using 64 bins and 64 frames. In terms of area, the relationship between the number of LEs and the number of bins is linear; this is because most of the area is due to the comparators required to build the histogram circuit. The number of memory bits needed by the distribution instrument also increases with the number of bins; this quantity can be calculated using:

$$mBits = nHist * nBin * (binWidth + 1), \quad (1)$$

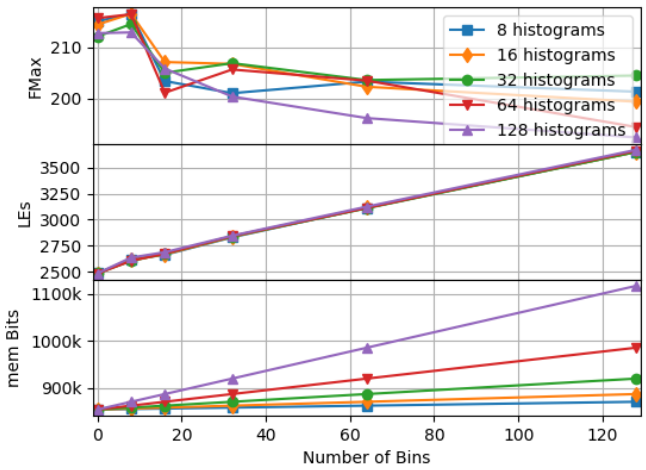

Figure 8: Maximum frequency, logic elements and memory bits used by distribution instrument when tracking 32x28x28 matrix

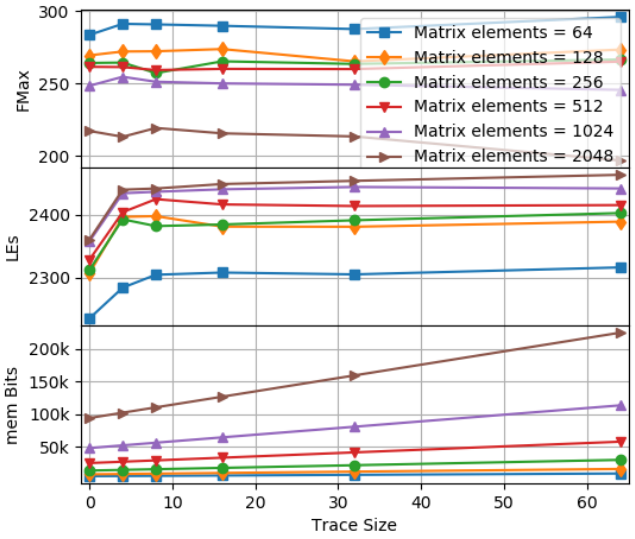

Figure 9: Maximum frequency, logic elements and memory bits used by spatial sparsity instrument

where *mBits* is the number of memory bits, *nHist* is the number of histograms tracked in the trace buffer, *nBins* is the number of bins of each histogram and *binWidth* the bit-width of each bin in the histogram. The *binWidth* is given by

$$binWidth = \lceil log2(sigLen + 1) \rceil \quad (2)$$

in which *sigLen* corresponds to the number of elements in the matrix being traced.

5.3 Architecture Study 2: Spatial Sparsity Instrument

In this experiment, we focus on the spatial sparsity instrument. We vary the number of frames traced while keeping the size of each frame constant, for several kernels. The results are shown in Figure 9. Each line in the graph corresponds to a different kernel. The left-most point is the uninstrumented circuit (trace size = 0).

As shown in the central plot of Figure 9, the spatial sparsity instrument has an initial area overhead that does not increase with the trace size and is approximately the same for all circuits. This is different from the distribution instrument, which presents almost no initial area cost, but a growing number of logic elements when the number of bins increases.

The latency of the circuit has not shown to be sensitive to the spatial sparsity instrument for most cases. The only exception is a slight decrease in the maximum frequency when tracking 2048 elements.

Figure 9 also shows the number of memory bits required to implement this instrument. This quantity can be calculated using

$$mBits = tSize * mEle, \quad (3)$$

where *tSize* corresponds to the trace size and *mEle* corresponds to the number of elements of the matrix being tracked.

6 CONCLUSIONS

In this paper, we have presented a debug infrastructure that can be used either as a stand-alone tool or in concert with existing debug infrastructure to enhance visibility of machine learning circuits. The proposed infrastructure allows the user to run the design at speed and record information for later interrogation. Unlike previously published debug tools, our instrumentation leverages the specific characteristics inherent in machine learning algorithms to provide an insight of the behaviour of the observed variables without tracking all changes to a variable. We show that our system is able to trace signals for a fraction of the area cost, enabling the designer to achieve a longer trace-lengths when compare to state-of-the-art debug infrastructures.

Current and future work includes evaluating and adapting the proposed approach to various deep learning networks, combining it with high-level debugging aids such as assertions [8] for edge computing and for cloud applications, exploring the possibility of extensions to cover the debugging of application-specific devices such as the TPU [10] and integrating this work with Tensorboard for a unified software/hardware debug framework.

ACKNOWLEDGEMENTS

We thank both NSERC COHESA and Intel for their FPGA Programming Optimizations ISRA (Intel Strategic Research Alliance).

REFERENCES

[1] Altera. 2015. *Quartus Prime Pro Edition Handbook*. Vol. 3. Chapter 9: Design Debugging Using the SignalTap II Logic Analyzer.
[2] N. Calagar, S.D. Brown, and J.H. Anderson. 2014. Source-level Debugging for FPGA High-Level Synthesis. In *Int'l Conf. on Field Programmable Logic and Applications*.
[3] F. Eslami and S. J. E. Wilton. 2015. An adaptive virtual overlay for fast trigger insertion for FPGA debug. In *Int'l Conf. on Field Programmable Technology (FPT)*.
[4] J. Goeders and S.J.E. Wilton. 2014. Effective FPGA debug for high-level synthesis generated circuits. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*. https://doi.org/10.1109/FPL.2014.6927498
[5] J. Goeders and S.J.E. Wilton. 2017. Signal-Tracing Techniques for In-System FPGA Debugging of High-Level Synthesis Circuits. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 36, 1 (Jan 2017), 83–96.
[6] K.S. Hemmert, J.L. Tripp, B.L. Hutchings, and P.A. Jackson. 2003. Source level debugger for the Sea Cucumber synthesizing compiler. In *Symposium on Field-Programmable Custom Computing Machines*. 228–237.
[7] D. Holanda Noronha, B. Salehpour, and S. J. E. Wilton. 2018. LeFlow: Enabling Flexible FPGA High-Level Synthesis of Tensorflow Deep Neural Networks. In *International Workshop on FPGAs for Software Programmers*.
[8] E. Hung, T. Todman, and W. Luk. 2017. Transparent in-circuit assertions for FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36, 7 (July 2017), 1193–1202.
[9] A. Jamal, J. Goeders, and S.J.E. Wilton. 2018. Architecture Exploration for HLS-Oriented FPGA Debug Overlays. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '18)*. ACM, New York, NY, USA, 209–218.
[10] N. Jouppi, C. Young, N. Patil, and D. Patterson. 2018. A domain-specific architecture for deep neural networks. *Commun. ACM* 61, 9 (Sept 2018), 50–59.
[11] A. Kourfali and D. Stroobandt. 2016. Efficient Hardware Debugging using Parameterized FPGA Reconfiguration. In *Int'l Parallel and Distributed Processing Symposium Workshop*. 277–282.
[12] S.I. Venieris and C. Bouganis. 2016. fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs. In *2016 Int'l Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 40–47.
[13] Xilinx. 2012. *ChipScope Pro Software and Cores: User Guide*.
[14] Xilinx. 2016. Integrated Logic Analyzer v6.1: LogiCORE IP Product Guide. http://www.xilinx.com/support/documentation/ip_documentation/ila/v6_1/pg172-ila.pdf. (April 2016).
[15] R. Zhao, H. Ng, W. Luk, and X. Niu. 2018. Towards Efficient Convolutional Neural Network for Domain-Specific Applications on FPGA. *arXiv preprint arXiv:1809.03318* (Sept 2018).