An Overlay for Rapid FPGA Debug of Machine Learning Applications

Daniel Holanda Noronha[1], Ruizhe Zhao[2], Zhiqiang Que[2], Jeffrey Goeders[3], Wayne Luk[2] and Steve Wilton[1]
[1]University of British Columbia, [2]Imperial College London, [3]Brigham Young University
{danielhn,stevew}@ece.ubc.ca, {ruizhe.zhao15,z.que,w.luk}@imperial.ac.uk, jgoeders@byu.edu

***Abstract*—FPGAs show promise as machine learning accelerators for both training and inference. Designing these circuits on reconfigurable technology is challenging, especially due to bugs that only manifest on-chip when the circuit is running at speed. In this paper, we propose a flexible debug overlay family that provides software-like debug times for machine learning applications. At compile time, the overlay is added to the design and compiled. At debug time, the overlay can be configured to record statistical information about identified weight and activation matrices; this configuration can be changed between debug iterations allowing the user to record a different set of matrices, or record different information about the observed matrices. Importantly, no recompilation is required between debug iterations. Although the flexibility of our overlay suffers some overhead compared to fixed instrumentation, we argue that the ability to change the debugging scenario without requiring a recompilation may be compelling and outweigh the disadvantage of higher overhead for many applications.**

I. Introduction

Field-Programmable Gate Arrays (FPGAs) have emerged as an important implementation option for many machine learning applications. Compared to CPUs and GPUs, FPGAs may lead to training and inference implementations with higher throughput and lower power consumption. These potential advantages have led to numerous proposals for FPGA-based machine learning applications [1]–[4].

Creating a correctly working FPGA-based machine learning application is challenging. Although high-level models (such as those written on top of TensorFlow) can be used to investigate the correctness, accuracy, and convergence of some applications, this often requires long run times, due to the need to train on a large data set or evaluate the response to many input samples. In addition, many applications can only be evaluated by understanding their response to real input traffic. In these cases, accurate characterization can often only be done by running the hardware implementation at speed in a real system.

Frameworks that ease the debug of running FPGA systems have been proposed [5]–[13]. These frameworks provide a mechanism to store the behaviour of key signals in a design as the chip runs for later interrogation, with the goal of providing visibility into the run-time operation of the chip. By understanding the behaviour of key signals, the user may be able to glean information that may help them uncover the root cause of any observed unexpected behaviour.

Although most of this work has targeted general-purpose applications, the framework described in [13] is optimized to debug machine learning applications. Machine learning applications are unique in that they typically contain large matrices (such as weights or activations), and storing the behaviour of all entries in these matrices will quickly exhaust the capacity of on-chip trace buffers. In addition, tracing the raw values of individual entries of large matrices may not be useful during debug. Rather, the "correctness" of machine learning applications depends on the ensemble of a large number of weights and activations acting together. The framework in [13] addresses this by storing the history of statistical measures or approximate weight or activation values in the trace buffers, allowing much better trace buffer utilization than previous techniques.

During the search for an elusive bug, as the user's understanding of the operation of the circuit evolves, he or she may wish to observe a different set of matrices, or record different statistical information about the matrices already being observed. An important limitation of the framework in [13] is that every time a new set of matrices is to be recorded, the user circuit needs to be recompiled. Recompilation is slow, and limiting debug productivity.

In this paper, we address this limitation by presenting a new framework which allows the user to change the matrices recorded or change the information recorded about each matrix *without requiring a recompilation*. Different from overlays that target general-purpose debug, our overlay has capabilities that are specifically tailored for the lossy compression techniques proposed in [13]. The overall approach is shown in Figure 1. At compile time, the user circuit is instrumented

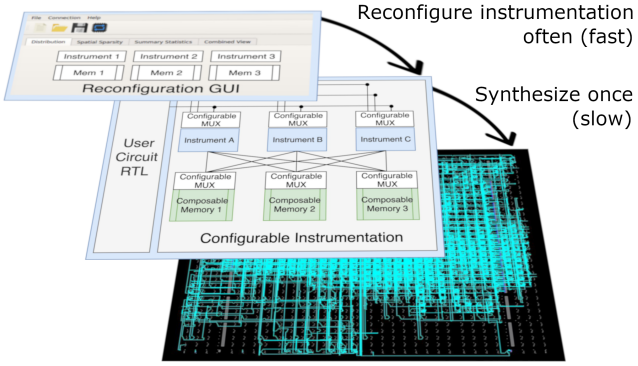


Fig. 1. Debug Framework

978-1-7281-2943-3/19/$31.00 ©2019 IEEE
DOI 10.1109/ICFPT.2019.00024

with a flexible *overlay*. At debug time, the overlay can be configured to store statistical information about identified matrices; the overlay's configuration can be changed between debug iterations allowing the user to record a different set of matrices, or record different information about each matrix. Although the overlay instrumentation is somewhat larger than that in [13], we argue that the ability to change the debugging scenario without requiring a recompilation may be compelling and outweigh the disadvantage of higher overhead for many applications.

This paper is organized as follows. Section II describes the baseline architecture in [13] upon which we build our work. Section III describes our debug framework in detail. The overlay architecture is then described in Section IV, and a graphical user interface is presented in Section V. Our technique is evaluated in Section VI. Section VII describes related work and Section VIII concludes the paper.

II. Baseline

In this section, we describe the flow in [13] upon which we build our work. As described in the introduction, the key to enabling effective debugging is to provide information to the user about the internal operation of the chip. Understanding the behaviour of the running design is essential for a user as he or she tries to uncover the root cause of unexpected behaviour (bugs). Due to bandwidth limitations, this information needs to be stored on-chip while the circuit is running and then read out for later interrogation.

In this framework, the user first describes a machine learning circuit using a high-level framework (such as TensorFlow) to evaluate convergence and accuracy, and select meta-parameters such as the size, type, and number of layers and their interconnection. Once the user settles on an architecture and a particular set of meta-parameters, the design is translated to RTL, either manually, or using an automated tool (such as [14]).

To increase visibility into the design, instrumentation is added that monitors selected signals in the design, and records them on-chip. Unlike earlier work [12], [15] or commercial tools [16], [17], in which individual source code variables or signals are monitored, the instrumentation described in [13] is optimized to monitor large matrices, such as those that might be found in a machine-learning application to store weights and activations. Since it is not possible to monitor *all* matrices in a large design, the user must be judicious in selecting which matrices would provide the maximum value during debug, and adjust the instrumentation to monitor only those matrices.

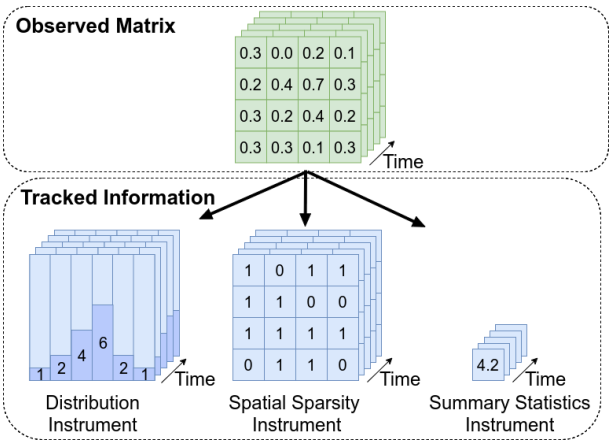


Fig. 2. Instruments

The user design, along with the instrumentation is then compiled using the normal back-end FPGA CAD flow (the work in [13] uses Quartus Pro). The circuit then runs on an FPGA, and the instrumentation records information about the run-time behaviour of the selected matrix(ces). After the run is complete, or when the circuit reaches a pre-determined breakpoint, the trace buffer information can be read out, and used, along with a graphical user interface to help the user understand the behaviour of their design.

Since a full history of these large matrices will quickly overwhelm the size of the on-chip trace buffers, the instrumentation in [13] contains compression circuitry which records information about the values in the selected matrices, rather than the raw values of the elements. In [13], three types of compression circuitry are described, shown graphically in Figure 2 and are described briefly below:

Distribution instrument: The distribution instrument bins the frequency count of values within the selected matrix in a histogram-like fashion. This is shown on the left side of Figure 2. The size of each bin and the number of bits for each histogram value can be optimized based on the user circuit and available memory. A separate histogram is created for each "frame" of the input stream; a frame is a user-defined period and may correspond, for example, to a single input image in a CNN. Such an instrument may be useful if the user wishes to determine whether a given matrix contains values that span an expected range, or whether values are biased high or low.

Spatial Sparsity Instrument: Often, a user may wish to determine the distribution of zeros within a matrix. Such information might be useful in a neural network for example, where the distribution of non-zero values may be related to the attention region of a particular layer. The spatial sparsity instrument shown in the centre of Figure 2 provides the ability to record which elements in the selected matrix are zero (within some tolerance). Again a separate map is created for each frame in the input stream.

Summary Statistics Instrument: Sometimes, it may be sufficient for the user to understand overall statistics regarding the run-time behaviour of the design. The Summary Statistics instrument stores the sparsity of the observed matrix(ces) without storing the location of the individual zeros.

The data is stored in the available trace buffers in a round-robin fashion as the circuit runs (with newer data replacing the oldest data in the buffer). Since the trace buffer is typically larger than the size of a single frame, at the end of the run, data regarding multiple frames would be stored in the trace

buffer; the more frames for which we have data, the more useful it will be to help the user deduce the cause of a bug.

In [13], a GUI is also described. The user can use the GUI to step through the operation of the design, using recorded data. After each run, as the user refines his or her view of the operation of the design, the user can change the matrix(ces) or instrumentation type, recompile the design, and repeat until the root cause of the bug is found. Each of these iterations is often termed a "debug turn".

After debugging is complete, the user may choose to remove the overlay for production or leave the overlay in place but disable it.

III. Enhanced Debug Flow

A limitation of the baseline flow is that every time a new set of matrices are to be recorded, or every time a new compression instrument is to be used, the user circuit (and instrumentation) needs to be run through the entire FPGA vendor CAD flow, including synthesis, place and route. This might take hours, and significantly limits debug productivity. In this section, we describe our flow which allows the user to change the matrix(ces) to be observed, the compression scheme used for each observed matrix, and the allocation of trace buffer memory to each instrument, without requiring a recompilation.

As in the baseline flow, the user inserts debug instrumentation into the user circuit, compiles using a back-end tool flow, and runs the circuit on an FPGA. Unlike the baseline, however, this instrumentation circuitry is flexible at run-time, without requiring a recompilation. After the circuit has been compiled, but before it is run, the instrumentation can be configured in the following three ways:

Selective Matrix Tracing: Selective Matrix Tracing refers to the capability to configure the overlay, at debug time, to specify which matrices in the user circuit should be traced. In our implementation, we focus on weight and activation matrices, since these are common in many machine learning applications, however, any large array could be monitored using our infrastructure.

Selective Compression: Storing the run-time behaviour of the raw values of all entries in large matrices will quickly overwhelm any reasonably-sized trace buffer. As in [13], our instrument contains three types of compression circuitry: spatial sparsity, histogram, and summary statistics. Unlike [13], the compression circuit used can be configured at debug time.

Flexible Trace Buffer: In the baseline flow, each instrument feeds one trace buffer. The size of each trace buffer must be statically determined at compile time. In our flow, the allocation of trace buffer space to instruments can be configured dynamically at run-time. Different instruments may use trace buffer space at different rates, meaning if the trace buffer space is not allocated properly, some trace buffers may fill before others, leading to uneven recording lengths for different matrices in the design. This flexibility allows the user to customize the amount of space devoted to each monitored matrix, and change this allocation as debugging proceeds. It also allows degenerate configurations, such as allocating all trace buffer space to a single instrument, maximizing the history for a selected matrix.

After the instrumentation has been configured (by writing the configuration through the JTAG port), the circuit is then run, and the instrumentation records information regarding each identified matrix into the trace buffers (as in the baseline flow, the trace buffers are configured as circular buffers). Each buffer continues to record data until either a breakpoint is reached or a predetermined condition indicates that recording should stop. At that point, the user can start a debug GUI (to be described in Section V) to analyze the recorded data.

As in the baseline flow, since the on-chip trace buffer is of a limited size, we can not store the entire run-time history of all matrices in the trace buffer. This means that, when debugging, the user can only view the behaviour of the identified matrices. As users refine their view of the operation of the circuit, they may wish to modify the debug scenario, and repeat the process until the root cause of bug is determined. Importantly, the design does not need to be recompiled when the debug scenario is changed.

IV. Architecture

Key to our technique is instrumentation that is flexible, yet adds as little overhead as possible to the user design. In this section, we describe our instrumentation architecture.

A. Overall Architecture

Figure 3 shows the overall architecture. The instrumentation consists of a number of Processing Elements (PEs) (three PEs are shown in the figure, however, this can be selected as the instrumentation is inserted, depending on the amount of chip area available). The instrumentation also contains a number of composable memory blocks (CM), each of which will be mapped to some number of embedded memory arrays in the target FPGA. The number of CMs can be selected depending on the size of the FPGA. The inputs of the instrumentation are hardwired to the address and data lines of the circuitry used to write data to selected matrices in the user circuit. A flexible

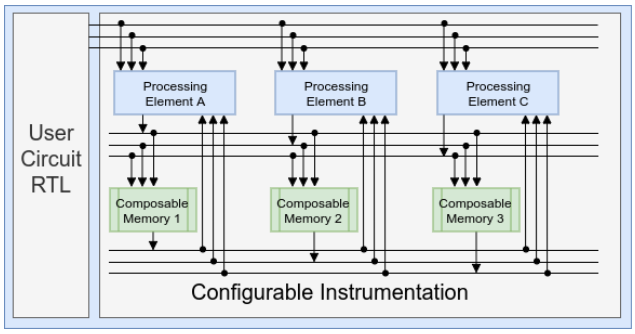
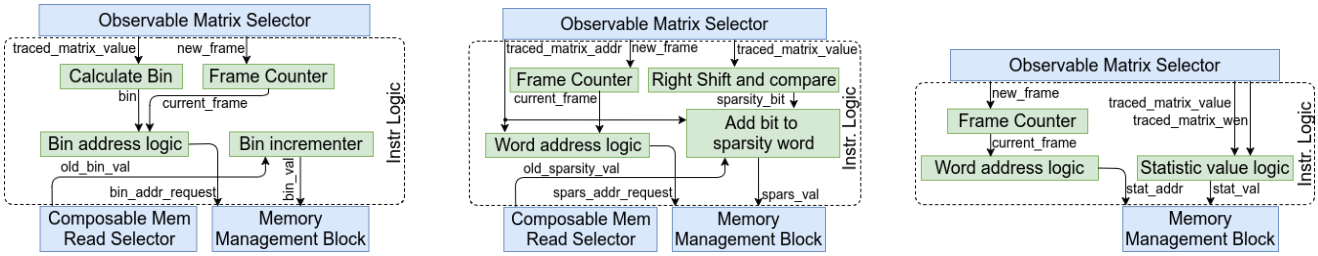


Fig. 3. Overall Instrumentation Architecture



(a) Distribution (b) Spatial Sparsity (c) Summary Statistics

Fig. 4. Instrumentation Logic Block Diagrams

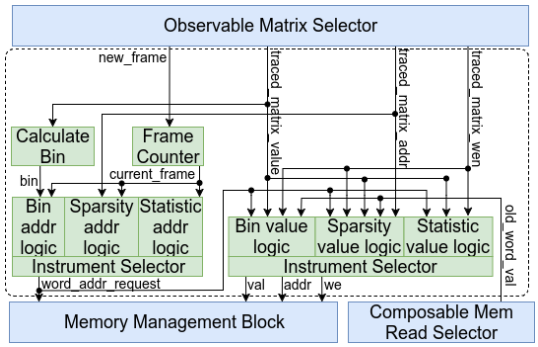


Fig. 5. Instrumentation Logic Block of Super Instrument

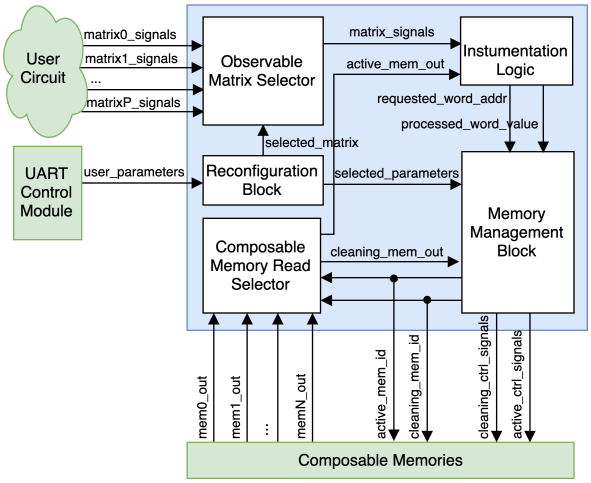


Fig. 6. PE Architecture

crossbar connects the inputs to the PEs; two additional flexible crossbars connect the PEs to the CMs. Each interconnect point within each crossbar is controlled by a register which can be written through the JTAG port at run-time. The crossbars are flexible enough that any input can be connected to any PE, and any PE can be connected to one or more CMs. Multiple CMs can be connected to a single instrument, allowing larger storage regions to be associated with a single matrix in the user circuit. The "return path" between the CMs and PEs is included for the histogram and spatial sparsity instruments since they require both reading and writing to the trace buffer.

We consider two variants of the architecture. In the first variant, each PE is statically fixed as either a histogram, spatial sparsity, or summary statistics instrument. In this variant, although the user can change which instrument is connected to which user matrix, the mix of instruments is static. In the second variant, each PE can be configured to be any of the three (we call such an instrument a *super instrument*), meaning the mix of instruments is flexible at run-time. The second variant is more flexible, but as we will show in the next section, has a higher overhead.

B. Processing Element

A block diagram of each PE is shown in Figure 6. The heart of each PE is the Instrumentation Logic sub-block which contains the compression circuitry described earlier. The contents of this block depends on the instrumentation being used for this PE; Figure 4 shows the structure of the sub-block for the distribution instrument, the spatial sparsity instrument and the summary statistics instrument, and Figure 5 shows combined architecture used to create the super instrument. Among all the sub-blocks that make up the PE, this is the only sub-block that differs for different compression methods.

Each PE also contains matrix selectors, which implement the crossbars described earlier, a Reconfiguration Block which allows the user to change the configuration of the crossbars and internal instrument operation at run-time using the JTAG port, and a memory management block which will be described in the next subsection.

C. Flexible Trace Buffer Memory Organization

In [13] each trace buffer memory can have a width tailored according to the instrument being used and the size of the matrix being traced. This allowed for a very high memory utilization. However, it is not adequate for a scenario in which the instrumentation can change at debug time. When used to store data from the distribution instrument, each data element must be wide enough to store the number of elements in each bin; this is a function of the matrix size, and changes as the matrix being recorded change. For the spatial sparsity instrument, each element is a single bit and the number of bits depends on the size of the matrix. The summary statistics instrument only stores one value per frame. Therefore, an adaptive packing mechanism is required to pack data elements into a fixed-size memory width. This is shown graphically in Figure 8.

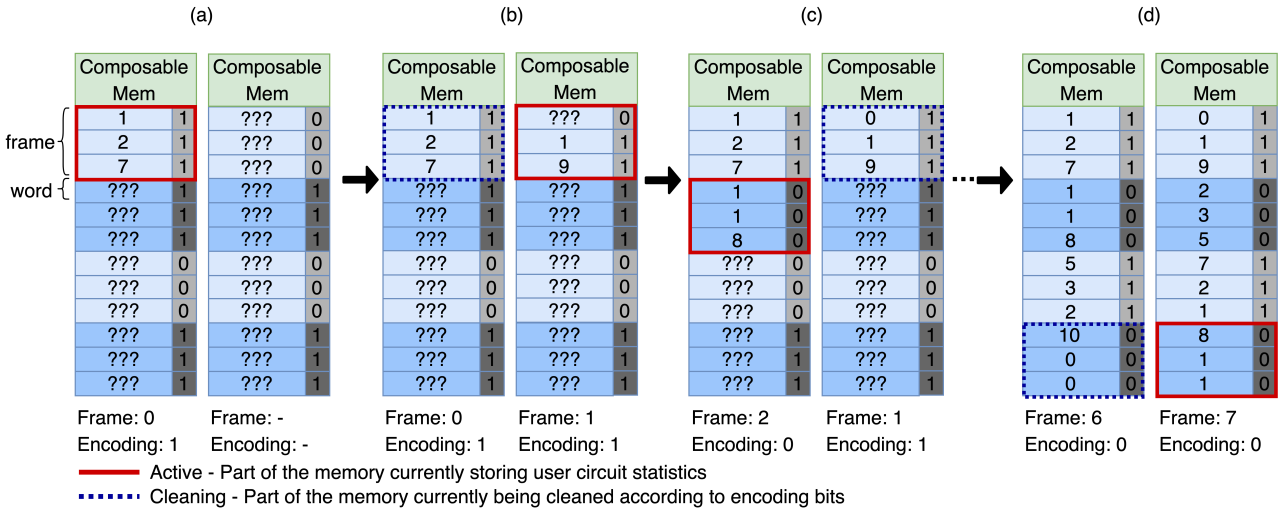


Fig. 7. Example showing Progression of Trace Buffer Memory

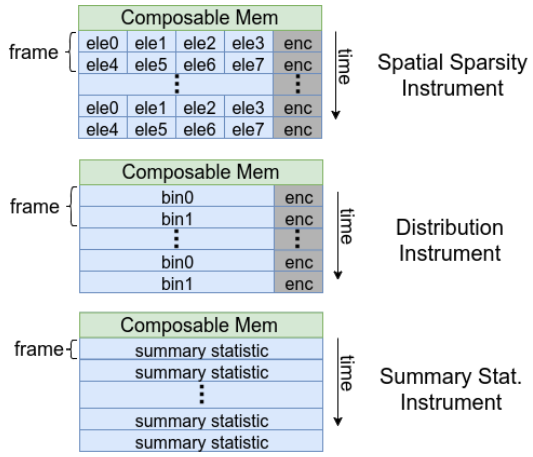


Fig. 8. Memory content organization according to instrument

The memory organization is further complicated by the fact that the instrumentation must be fast enough to write new information to the trace buffer every cycle. This has two implications. First, the spatial sparsity and distribution instruments construct their results in an incremental way; each sample requires updating either a bin or an entry in the sparsity map. Therefore, both reading and writing the trace buffer is required in each cycle. This can be satisfied by using dual-port embedded memories to implement trace buffers; dual-port SRAM blocks are common on FPGAs.

Second, it is necessary to "zero out" a region of memory between frames to avoid incrementally computing new statistics on top of old values (recall the trace buffers are implemented as circular buffers, so each frame replaces an older frame). Critically, since we are tracing data in real-time, we do not wish to stop the circuit between frames.

The instruments presented in [13] solved this issue by keeping track of dirty bits in a second memory. The width of this dirty bit memory was given by the number of elements that need to be stored for each frame. This solution is not preferred in our case, since the dirty bit memory rarely has the same width as the memory being used to store the values traced. In our case, all memories should ideally have the same width to be efficiently used by different instruments.

We solve this by associating a minimum of two composable memory blocks with each distribution or spatial sparsity instrument. One of the memories is actively being used to store the new information from the user circuit (*active state*), while the other memory is used to "clean" the words that have not been written to during the previous frame (*cleaning state*). Each word is marked as clean or dirty using a *frame encoding bit* which changes every time that a new frame is written into the same memory. This encoding bit is used to guarantee that we can differentiate between values that we can use during the current frame (e.g. a bin of a histogram that we have to increment) and values from an unrelated frame that has been previously stored in this memory location.

An example of our technique is shown in Figure 7. In this example, we use two memories to store the distribution histogram with three bins. Since there are three bins, each frame requires three words in the trace buffer plus one encoding bit. Bin values in the diagram are labeled "???" if they are uninitialized, or contain data from a previous frame.

In Figure 7(a), the memory on the left is in the active state. At the end of the first frame (Frame 0), the three words contain the count of data elements for each bin (1, 2, and 7). The encoding bit (final bit in each word) is set to 1 for all words corresponding to this frame.

At the end of the next frame (Frame 1), the memory on the right contains the count of data elements for Frame 1 as shown in Figure 7(b). Again, a 1 is used in each encoding bit. In this case, there were no elements mapped to the first bin, so this entry contains old data, and the encoding bit for this entry was not properly set to 1 (this will be "cleaned" later).

In Figure 7(c), at the end of the third frame (Frame 2), the memory on the left contains information about Frame 2. In this case, the encoding bit for each entry is set to 0 to differentiate these words from those corresponding to the previous frame. The memory on the right has been "cleaned" in that the missing value in the histogram (the first word) is set to 0 and the encoding bit is updated properly.

Figure 7(d) shows both memories a few frames later. In this figure, the memory on the left is cleaning Frame 6, while the memory on the right is active and storing information from Frame 7.

Since this is a circular trace buffer, the next frame (Frame 8) will overwrite the information written into the three first words of the matrix on the left. The encoding bits allow us to differentiate previous information from information that is being processed at the current frame.

An alternative architecture would "zero out" frames before they are written rather than fixing missing entries after the frame is written. Such an architecture would not require encoding bits, but would be able to store one fewer frame in the steady state. Since matrices can be large, we have elected to implement the first architecture.

V. Graphical User Interface

We provide a graphical user interface (Figure 9) that serves three purposes. First, at *compile time*, the designer can select the number and type of PEs and composable memory blocks, and can identify all the matrices in the user circuit that may be potentially traced (perhaps all of them). Second, at *run-time*, the user can select which of the matrices should be traced in a given run, which PEs should be connected to each matrix, the configuration of any super-instrument, and how the composable memories should be allocated to the PEs (the latter can optionally be done automatically based on the data production rate of each instrument). Third, at *debug-time*, after the circuit has been run and the trace history is read out, the user can visualize the compressed data that has been stored in the trace-buffer of those instruments in a way that makes sense for a machine learning domain expert.

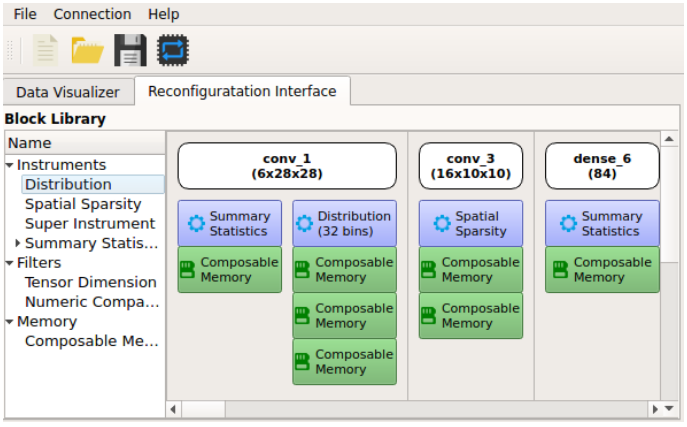


Fig. 9. Graphical user interface

VI. Evaluation

The overlay architecture is characterized by several parameters that allow the user to trade-off the amount of information that can be traced and area overhead. In this section, we will evaluate different architectures to show how those different parameters impact memory and area utilization.

A. Overhead of overlay when compared to baseline

To compare with previous work, we create circuits made to observe kernels that are part of Convolutional Neural Networks (CNNs). This is very similar to the approach taken in [13]. The generated circuits contain kernels that consist of 28x28 convolutions with 1, 8 and 16 channels.

We compare twenty different debugging configurations by varying the instrumentation being used and the matrices being observed. Similar to [12], [13], [15], 100Kb of total trace memory is assumed for all scenarios. For all experiments, each data point corresponds to an average of 20 circuits placed and routed using different seeds.

Table I shows the overhead in terms of area and speed of our architecture when compared to the baseline. In this experiment, all matrices were instrumented by our overlay. Which of those instrumented matrices is actually observed by each overlay instrument can be changed at debug time, while the observed matrix in the baseline is only configurable at compile-time. For all scenarios in this experiment, our instrumentation uses two composable memory blocks, while the baseline has a single trace buffer with the same total size.

As shown in Table I, when only a single instrument is used, the area increase of our instrumentation when compared to the baseline is very low for almost all instrument types. The only exception is the spatial sparsity instrument, which uses packing logic to store the data in multi-bit words instead of using single-bit words as is done in the baseline. The critical path delay of the distribution instrument remains approximately the same, while the delay of the summary statistics instrument slightly decreases due to the trace-buffer being split into two smaller memories. The delay of the spatial sparsity increases due to the packing logic used to compress the data. We anticipate that we could reduce this impact by further pipelining the instrumentation.

The impact in terms of how many frames we are able to store information about the matrix we are observing (trace size) is also shown in Table I. The trace sizes of the distribution instrument and spatial sparsity in our overlay for all matrices are the same as in the baseline for the largest matrix. This happens because each word of the trace buffer needs to be wide enough to store information in the worst-case scenario (which is given by the largest instrumented matrix). Nevertheless, the trace size of the spatial sparsity varies according to the size of the matrix being traced. This is possible due to the flexible bit compression scheme developed for this instrument, making its trace size very similar to the baseline.

Table II shows results from the same experiment when the super instrument is used instead of other instruments with fixed compression types. In this scenario, the super instrument

TABLE I
OVERHEAD OF ADDING CAPABILITY TO OBSERVE DIFFERENT MATRICES COMPARED TO BASELINE

Configuration	Baseline [13] (Compile-time Configurable)									Ours (Debug-time Configurable)		
	Observing Matrix A			Observing Matrix B			Observing Matrix C			Observing Matrices A, B or C		
Instrument	**FMax (MHz)**	**LEs**	**Trace Size**[§]	**FMax (MHz)**	**LEs**	**Trace Size**[§]	**FMax (MHz)**	**LEs**	**Trace Size**[§]	**FMax (MHz)**	**LEs**	**Trace Size**[§]
Distribution Instrument (32 Bins)	196.5	830	284	197.4	823	223	199.7	828	208	201.6	976	A: 208 B: 208 C: 208
Distribution Instrument (128 Bins)	185.1	1557	71	187.1	1552	55	187.5	1558	52	181.1	1508	A: 52 B: 52 C: 52
Spatial Sparsity Instrument	188.0	471⋆	127	184.6	466⋆	15	191.0	463⋆	7	162.2	867	A: 120 B: 14 C: 6
Summary Statistics Instrument	203.2	473	10000	208.9	476	7692	204.5	478	7142	212.7	584	A: 7142 B: 7142 C: 7142

§ Number of times information about the entire 32-bit frame could be tracked.
† Matrix A is 1x28x28, matrix B is 8x28x28 and matrix C is 16x28x28 (circuit being instrumented has all 3 matrices).
⋆ Baseline implementation uses single-bit words, while ours uses a packing logic to compress the data into multi-bit words.

TABLE II
OVERHEAD OF ADDING CAPABILITY TO OBSERVE DIFFERENT MATRICES COMPARED TO BASELINE

Configuration	Ours (Debug-time Configurable)											
	Observing Matrix A			Observing Matrix B			Observing Matrix C			Observing Matrices A, B or C		
Instrument	**FMax (MHz)**	**LEs**	**Trace Size**[§]	**FMax (MHz)**	**LEs**	**Trace Size**[§]	**FMax (MHz)**	**LEs**	**Trace Size**[§]	**FMax (MHz)**	**LEs**	**Trace Size**[§]
Super Instrument⋆	165.18	1098	Dist.:182 Spat.:120 Summ.:5882	164.4	1107	Dist.:182 Spat.:14 Summ.:5882	164.8	1115	Dist.:182 Spat.:6 Summ.:5882	164.3	1143	Dist.:182 Spat.:6-120 Summ.:5882

§ Number of times information about the entire 32-bit frame could be tracked.
† Matrix A is 1x28x28, matrix B is 8x28x28 and matrix C is 16x28x28 (circuit being instrumented has all 3 matrices).
⋆ Either distribution (32 bins), spatial sparsity or summary statistics instrument.

can be configured at debug-time to work either as the spatial sparsity, summary statistics or distribution instrument with 32 bins. Since there is significant component reuse, the total area utilization of this instrument is similar to the area of the largest instrument that it is composed of. The critical path delay of this instrument was given by the spatial sparsity component.

As shown in Table II, the configurability of the super instrument can also cause the reduction of the trace size in some cases. Note that all instruments of the super instrument need to be able to share the same trace buffer. This means that the trace buffer memory width must be given by the worst-case scenario for all of those instruments, reducing the trace size of instruments that could otherwise utilize a narrower memory.

B. Impact of number of instruments and memories

Figure 10 shows a different experiment in which a circuit with eight instrumented kernel matrices can be observed by either 1, 2, 4 or 8 instruments, while the number of composable memories in the design is also varied. Note that the number of memories and instruments do not need to be a power of two in order for the instrumentation to be efficient. Each data point corresponds to an average of twenty placed and routed circuits and the total memory for the trace buffer of those circuits is kept at 100Kb.

As shown in Figure 10, the area grows according to the number of instruments and memories added to the circuit. However, this number is never larger than 5,000 LEs when using up to 4 instruments and 8 memories. This corresponds to approximately 0.5% of a Stratix V FPGA or less than 0.2% of a modern Stratix 10 device.

As expected, the area overhead of the distribution and spatial sparsity instruments, which need to both read and write to the composable memories, are larger than the area for the summary statistics instrument. The area of the super instrument is comparable with the area of the distribution and spatial sparsity instruments, which indicates that using this flexible instrument might very often be a better alternative than using instruments with a fixed compression scheme.

We anticipate that both area and delay could be further decreased by limiting the number of simultaneous connections supported between instruments and memories by using a concentrator and time-multiplexing the memory accesses.

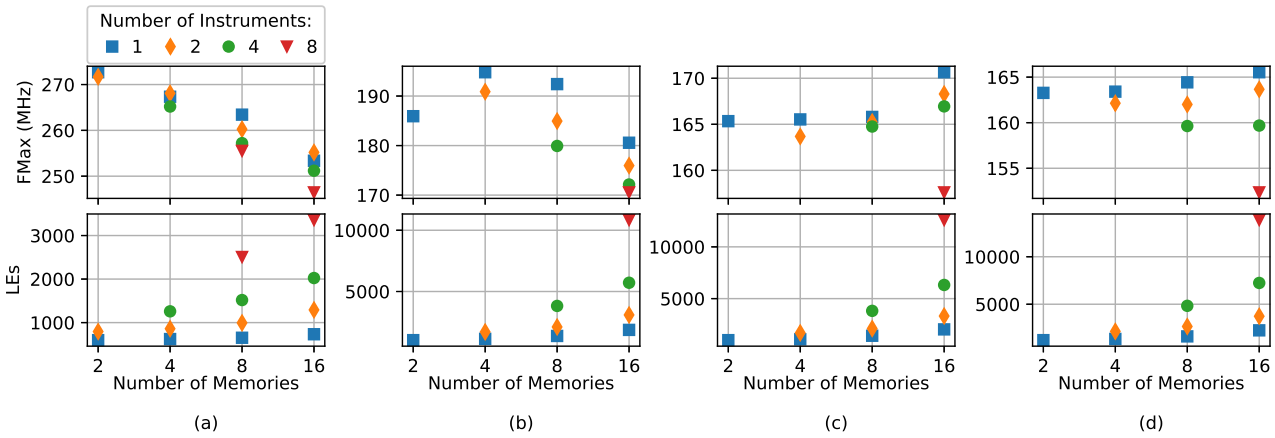


Fig. 10. Impact of number of memories and instruments: (a) Summary Statistics Instrument; (b) Distribution Instrument (32 bins); (c) Spatial Sparsity Instrument; (d) Super instrument.

VII. Related Work

Early work on enhancing the visibility of FPGA circuits focused on using scan-chains and similar techniques to capture a snapshot of the circuit at a certain point in time [18], [19]. This allows a very high visibility at a low area cost, but does not provide a history of how values change over time if it is not possible to resume execution after the circuit stopped.

There has been much effort developing techniques for storing data at run-time into circular buffers including building instruments that contain run-time compression [20]–[23]. This has led to commercial products that insert trace buffers and associated circuitry into FPGA designs [5], [6]. Compared to the work in this paper, the previous techniques focus on general-purpose designs and are not optimized for the large matrices that are common in machine learning designs.

Improving the turn-around time between debug turns, including limiting recompiling the part of instrumentation that changes, partial reconfiguration, and post-place and route debug insertion have also been proposed by various authors [7], [24]–[27]. Overlays have also emerged as an interesting option to achieve even lower debug turn-around times at a higher area cost [15], [28], [29]. Again, unlike the work in this paper, these previous studies were not optimized for machine learning.

Work in optimizing the instrumentation for circuits developed using high-level synthesis (HLS) techniques have been presented [9], [10], [12], [30]. In these frameworks, since a schedule of the HLS-generated circuit is available, the debug instrumentation can be optimized on a circuit-by-circuit basis. This approach also attempts to imitate software debugging as closely as possible. Since many machine learning implementations are created using high-level synthesis tools, these techniques may be useful for debugging machine learning circuits, however, they do not contain the custom compression for large matrices that are used in this work. It is possible to create instrumentation that is optimized for both HLS and large matrices and use both at the same time; this is an interesting area for future work.

In terms of debug for machine learning circuits on FPGAs, the only other work we are aware of is [13]. Compared to that work, our approach allows for debug-time overlay customization leading to faster turn-around times. As described earlier, we have built our framework using some ideas from [13].

The most similar work to that presented here is that of [15], which described a debug overlay that allows the user to rapidly reconfigure the general-purpose HLS-oriented debug infrastructure proposed in [12]. Our overlay is quite different from that in [15]. First, our technique is optimized for tracing large matrices rather than individual variables. Second, the lossy compression techniques used in our instrumentation are not present in [15]. Again, it may be possible to combine features from the overlay in [15] with ideas from this paper, and this is an interesting area for future work.

VIII. Conclusions

In this paper, we presented a flexible overlay for on-chip debug instrumentation of machine learning applications, providing software-like debug turn around time. Similar to previous work, domain-specific debug instrumentation is added to the RTL at compile-time. This instrumentation records information about the design as it runs at speed for later interrogation. Different from previous work, the instrumentation being used can be reconfigured at debug-time, instead of requiring a full new synthesis. We explored overlay variants with architectural support for selective matrix tracing, selective compression, and flexible trace buffer allocation. Instrumentation with different sizes and capabilities were compared to previous work, showing that only a small area overhead is added to allow this flexibility. Together, the overlay capabilities proposed in this work have the potential to significantly accelerate the process of debugging machine learning circuits on FPGAs.

Acknowledgments

The support of NSERC COHESA, Intel ISRA, Vanier CGS, Corerain Technologies, EPSRC Grants EP/L016796/1, EP/N031768/1 and EP/P010040/1 is gratefully acknowledged.

REFERENCES

[1] T. Chen *et al.*, "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," in *OSDI*, 2018.

[2] Y. Fu, "Xilinx ML Suite Overview," Xilinx, Tech. Rep., 2018.

[3] E. Chung *et al.*, "Serving DNNs in real time at datacenter scale with Project Brainwave," *IEEE Micro*, vol. 38, pp. 8–20, 2018.

[4] M. S. Abdelfattah *et al.*, "DLA: Compiler and FPGA Overlay for Neural Network Inference Acceleration," in *FPL*, 2018.

[5] Xilinx, "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.

[6] Intel, *Quartus Prime Pro Edition Handbook*, May 2017, vol. 3, ch. 8: Design Debugging with the Signal Tap Logic Analyzer.

[7] A. Kourfali and D. Stroobandt, "Efficient hardware debugging using parameterized FPGA reconfiguration," in *Int'l Parallel and Distributed Processing Symposium Workshop*, May 2016, pp. 277–282.

[8] F. Eslami and S. J. E. Wilton, "An adaptive virtual overlay for fast trigger insertion for FPGA debug," in *Int'l Conf. on Field Programmable Technology (FPT)*, Dec 2015.

[9] K. Hemmert, J. Tripp, B. Hutchings, and P. Jackson, "Source level debugger for the Sea Cucumber synthesizing compiler," in *Symposium on Field-Programmable Custom Computing Machines.*, April 2003, pp. 228–237.

[10] N. Calagar, S. Brown, and J. Anderson, "Source-level Debugging for FPGA High-Level Synthesis," in *Int'l Conf. on Field Programmable Logic and Applications*, Sept 2014.

[11] J. Goeders and S. Wilton, "Effective FPGA debug for high-level synthesis generated circuits," in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, Sept 2014.

[12] J. Goeders and S. J. E. Wilton, "Signal-tracing techniques for in-system FPGA debugging of high-level synthesis circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 1, pp. 83–96, Jan 2017.

[13] 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.

[14] D. Holanda Noronha, B. Salehpour, and S. J. E. Wilton, "Leflow: Enabling flexible FPGA high-level synthesis of tensorflow deep neural networks," in *International Workshop on FPGAs for Software Programmers*, Aug 2018.

[15] A. Jamal, J. Goeders, and S. Wilton, "Architecture exploration for hls-oriented FPGA debug overlays," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18, 2018, pp. 209–218.

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

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

[18] T. Wheeler, P. Graham, B. Nelson, and B. Hutchings, "Using design-level scan to improve FPGA design observabilty and controllability for functional verification," in *Int'l Conf. on Field-Programmable Logic and Applications*, 2001, pp. 483–492.

[19] H. u. H. Khan, A. Kamal, and D. Goehringer, "An intrusive dynamic reconfigurable cycle-accurate debugging system for embedded processors," in *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, N. Voros, M. Huebner, G. Keramidas, D. Goehringer, C. Antonopoulos, and P. C. Diniz, Eds. Cham: Springer International Publishing, 2018, pp. 433–445.

[20] E. A. Daoud and N. Nicolici, "On using lossy compression for repeatable experiments during silicon debug," *IEEE Transactions on Computers*, vol. 60, no. 7, pp. 937–950, Jun. 2010.

[21] ——, "Real-time lossless compression for silicon debug," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 28, no. 9, pp. 1387–1400, Aug. 2009.

[22] H. F. Ko, A. Kinsman, and N. Nicolici, "Design-for-debug architecture for distributed embedded logic analysis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 8, pp. 1380–1393, Jun. 2010.

[23] B. Quinton, A. Hughes, and S. Wilton, "Post-silicon debug of complex multi clock and power domain ics," in *IEEE Int'l Workshop on Silicon Debug and Diagnosis*, 2010.

[24] A. Kourfali and D. Stroobandt, "Superimposed in-circuit debugging for self-healing FPGA overlays," in *Latin America Test Symposium*, March 2018.

[25] P. Kumar, J. Goeders, and S. Wilton, "Accelerating in-system FPGA debug of high-level synthesis circuits using incremental compilation techniques," in *Int'l Conf. on Field-Programmable Logic and Applications*, Sept 2017.

[26] E. Hung, J. B. Goeders, and S. J. E. Wilton, "Faster FPGA debug: Efficiently coupling trace instruments with user circuitry," in *Reconfigurable Computing: Architectures, Tools, and Applications*, D. Goehringer, M. D. Santambrogio, J. M. P. Cardoso, and K. Bertels, Eds. Cham: Springer International Publishing, 2014, pp. 73–84.

[27] 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)*, Aug 2018, pp. 81–813.

[28] F. Eslami, E. Hung, and S. J. E. Wilton, "Enabling effective FPGA debug using overlays: Opportunities and challenges," *CoRR*, vol. abs/1606.06457, 2016. [Online]. Available: http://arxiv.org/abs/1606.06457

[29] Z. Poulos, Y.-S. Yang, J. Anderson, A. Veneris, and B. Le, "Leveraging reconfigurability to raise productivity in FPGA functional debug," in *IEEE/ACM Design and Test in Europe (DATE)*, 2012, pp. 292–295.

[30] J. S. Monson and B. L. Hutchings, "Using Source-Level Transformations to Improve High-Level Synthesis Debug and Validation on FPGAs," in *Int'l Symp. on Field-Programmable Gate Arrays*, 2015, pp. 5–8.