(Special Session)

A High Level Synthesis Methodology for Dynamic Monitoring of FPGA ML Accelerators

Ryan F. Forelli*, Rui Shi, Seda Ogrenci, Joshua Agar+
Department of Electrical and Computer Engineering, Northwestern University
*Department of Electrical and Computer Engineering, Lehigh University
+Department of Mechanical Engineering, Drexel University

2024 IEEE 42nd VLSI Test Symposium (VTS) | 979-8-3503-6378-4/24/$31.00 ©2024 IEEE | DOI: 10.1109/VTS60656.2024.10538570

***Abstract*—In this paper, we present concepts towards a HLS-driven dynamic monitoring and debugging framework. Traditionally, in-situ debugging and dynamic monitoring is accessible during the early design stages through costly co-simulation cycles and through invasive tools and interfaces. We propose a methodology where dynamic monitoring is embedded into the high level synthesis description of machine learning (ML) accelerators within the open source hls4ml tool. We discuss the usage of the framework for monitoring FIFO channel utilization, which is a critical structure utilized to implement streaming based ML accelerators on FPGAs.**

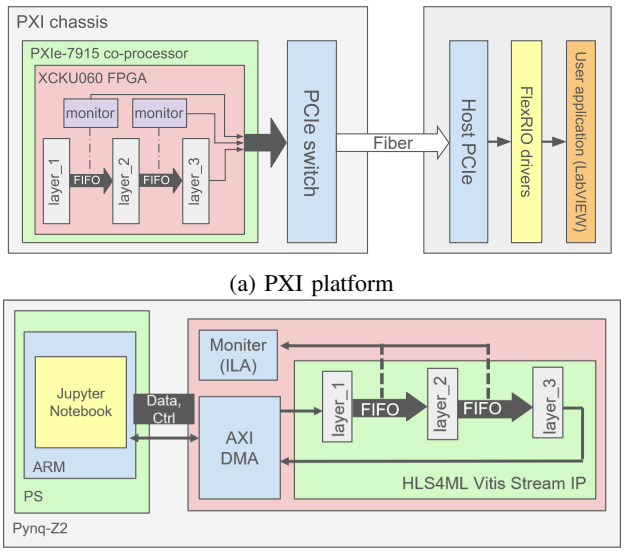
***Index Terms*—dynamic monitoring, HLS, FIFO**

I. Introduction

FPGA HLS tools have been gaining more popularity among designers partly due to the increasing tool flow support from FPGA manufacturers or vendors. These tools allow software developers to express designs in high-level languages, and then translate the code into FPGA board configuration files automatically. Intel introduced oneAPI, which supports a uniform platform for programming across CPU, GPU, and FPGA using Data Parallel C++ [1]. Xilinx provides Vitis tools [2] [3] with similar functionality for OpenCL, C and C++ kernels. These examples and many more emerging HLS tools make FPGAs accessible to a wide range of developers. Utilizing these platforms, open-source tools like hls4ml [4] and FINN [5] provide workflows to create hardware implementations on FPGAs for machine learning algorithms, which further bridge the gap between software models and accelerator inference.

To aid application characterization and architecture design space exploration, advanced profiling and debugging tools have been developed for CPUs (such as, Intel Advisor tool [6], Pin tool [7]), and GPUs (such as, Nvidia Visual Profiler [8] SASSI [9]). Similarly, it is necessary to have a dynamic monitoring tool that is easy to use and provides more detail on the behavior of the accelerator when running on FPGAs. Furthermore, CPUs and GPUs are equipped with hardware performance counters that provide detailed insight into the execution and behavior of the workloads. These monitoring infrastructures support run-time dynamic management mechanisms and they also provide a rich set of features that can be used to build performance models. Performance counters have been used to model performance, power and temperature characteristics for CPUs and GPUs [10]–[13].

This work was partially supported by DoE ASRSP GCFA Grant ID: SP0062070 and NSF POSE Phase II Grant AWD00000665.



(a) PXI platform

(b) Pynq-Z2 SoC platform

Fig. 1: PXI and Pynq-Z2 platforms with hls4ml accelerator.

There are several logic analyzers focusing on in situ FPGA debugging, such as SignalTap [14] and ChipScope [15]. Specifically tailored for HLS generated circuits, Goeders and Wilton [16] proposed a debugging system that allows users to set breakpoints and replay the execution; Pietro et al. [17] examined ways to automate circuit bug detection. One common aspect of these methods is that they require modifications of the HLS compilers and thus, are not applicable for users of commercial HLS tools. Our approach can similarly be used for debugging and more, but it is implemented at the code-level by the designer, hence, it does not require any modification of the HLS tool. We demonstrate its utility through a design flow using hls4ml with the Vitis HLS 2023.2 backend.

II. Monitoring FIFO Channels for Verification and Optimization of Streaming ML Accelerators

Recent HLS accelerator designs mostly focus on streaming architectures that connect computing elements by FIFOs. This architecture has a simple control logic which can be compiled and optimized by the tools at higher efficiency. Standard

streaming architectures utilize FIFOs with a finite amount of memory allocated to them to facilitate data transfer. When the writer places new data in the FIFO, the receiver ideally consumes the data at some frequency such that the FIFO does not overflow or underflow or initiate a deadlock.

Layer	hls4ml default/ Vitis C-Sim	hls4ml co-sim FIFO opt.	Vitis co-sim DDD
input_1_cpy1	146880	1112	73945
input_1_cpy2	146880	2	3
zp2d_out	148676	146562	146549
conv2d_out	146880	2	2
relu_out	146880	2	2
zp2d_1_out	148676	36622	131374
conv2d_1_out	146880	2	2
relu_1_out	146880	2	2
zp2d_2_out	148676	679	684
conv2d_2_out	146880	2	2
relu_2_out	146880	2	2
zp2d_3_out	148676	679	684
conv2d_3_out	146880	2	2
relu_3_out	146880	2	2
zp2d_4_out	148676	679	684
conv2d_4_out	146880	2	2
relu_4_out	146880	2	2
zp2d_5_out	148676	679	684
conv2d_5_out	146880	2	2
relu_5_out	146880	2	2
zp2d_6_out	148676	679	684
conv2d_6_out	146880	2	2
lambda_out	146880	146880	146880
add_out	146880	66	3
lambda_1_out	1321920	2	3
Time: Our method 3.5h	–	1 day	10 days

TABLE I: Suggested FIFO depths for super resolution NN with skip connection according to hls4ml default algorithm, hls4ml FIFO optimization, and Vitis DDD.

Existing HLS tools for estimating the proper FIFO size in streaming architectures rely primarily on C-simulation and co-simulation. While this technique offers a convenient and relatively accurate software-based method of determining FIFO usage, multiple sources of overhead and uncertainty can impede development. Large or complex neural networks can generate excessive co-simulation runtimes that exceed synthesis and place and route runtimes. Additionally, these simulations can be highly CPU and memory intensive. Finally, the behavior observed in co-simulation does not always match its hardware implementation. In some cases, this can result in either unnecessary over-utilization of the FPGA resources or underallocation of the necessary FIFO resources leading to reader/writer stalls and undetermined behavior due to data loss, deadlocks, or decreases in overall throughput and reliability. Here, we utilize new synthesizable functions of the Vitis HLS stream non-blocking API to enable quick profiling of hardware-implemented FIFOs, thus avoiding lengthy simulation times and yielding accurate FIFO size estimates. Although this feature is available in the tool version, its utilization within extended design automation flows is undefined. In this paper, we present a verification flow involving hls4ml in conjunction with Vitis HLS, where dynamic FIFO monitoring is realized.

A. Existing FIFO Size Estimation Methods in HLS

Current FIFO size optimization methods for Vitis HLS are mainly in C-simulation and co-simulation. Open-source tools, like hls4ml, also provide their own ways of optimizing it. These methods decrease the number of co-simulation cycles needed but they are not up-to-date with Vitis and our methodology is the first automated approach for utilizing the FIFO tracking feature of Vitis within hls4ml.

1) C-simulation: C-simulation is an early step in functional verification during the design process. It is a quick process and can be used to initially determine the appropriate FIFO sizes. However, it often overestimates the size needed to avoid deadlocks, as pragmas, which apply dataflow/function-level pipelining are disregarded during c-simulation. As model architectures grow larger, the inherent inaccuracies of C-simulation can lead to excessive memory usage on the host CPU for simulation and on the FPGA for the final design.

2) Manual Co-simulation: Co-simulation provides a more detailed view of FIFO fullness during the development process. It allows designers to manually tune the FIFO size through step-by-step tuning. However, this requires intuition and analysis from the designer. As the model becomes larger, converging on the optimal FIFO sizes becomes more difficult and the time commitment becomes too costly.

3) Dynamic Estimation using Co-simulation: Similar to the manual simulation, newer Vitis tools incorporate Dynamic Deadlock Detection (DDD) algorithms which incrementally increase FIFO depths throughout the design until performance no longer improves and no FIFO writers are blocked. Moreover, this process involves the execution of multiple successive co-simulations and as such, larger designs can generate excessive runtimes which impede development.

4) hls4ml FIFO Optimization Flow: hls4ml translates neural network models from high-level languages such as PyTorch, Keras, and ONNX to an HLS representation. hls4ml utilizes a FIFO-based streaming approach to construct a neural network architecture where FIFOs transport layer activations from one layer to the next. By default, these FIFOs are sized to match the output size of the corresponding layer. However, typical hls4ml implementations utilize only a small fraction of the full allocated FIFO capacities, thus inflating BRAM resource usage beyond what is necessary.

Currently, hls4ml only offers co-simulation based tools for FIFO depth optimization with Vivado HLS. In contrast to the Vitis DDD scheme, this tool executes only a single neural network inference in co-simulation, and extracts the necessary signals to identify the maximum occupancy of each transport FIFO in the network. These maximum values are then incremented by 1 and back-annotated to the HLS architecture. While quicker than Vitis DDD, this scheme fails to account for successive inputs which can result in undersized FIFOs, impede throughput, and yield a higher initiation interval. Excessive runtimes are also a staple of this optimization with larger neural networks.

5) Size and Capacity Function of HLS Stream API: Newer versions of the Xilinx design suite include new synthesizable

methods in the HLS stream non-blocking API for stream control and live diagnostics. Two of these methods include ".capacity()", and ".size()", which return the hardware implemented runtime capacity and the current occupancy of the FIFO-based stream. We leverage this new feature to develop a Xilinx HLS-based framework for active FIFO monitoring in hardware for ML accelerators generated with hls4ml.

B. Comparison of Co-simulation-based FIFO Sizing Methods

We present a comparison between the default hls4ml FIFO depths (equal to layer output sizes), the hls4ml FIFO optimization results, and the Vitis DDD reported maximum depths. We utilize a relatively large super resolution model which features a skip connection to emphasize the advantages, shortcomings, and differences between the current software methods of FIFO depth calculation and optimization. Deadlocks are also commonly characteristic of neural network architectures with skip connections. The architecture of this neural network is shown in Table I. The model input is copied to input_1_cpy1 and input_1_cpy2 creating two branches which merge at add_out. Furthermore, Table I outlines the by-layer FIFO depths recommended by each method. hls4ml sets the size of each transport FIFO of the network equal to the size of the previous layer's output. For example, since the model input is a 680x216 image with three color channels, hls4ml defines the input stream depth of 146,880. Vitis HLS C-simulation reports verify that this depth is reached. However, as previously mentioned, pipelining is not applied during C-simulation, and thus hls4ml utilizes a more conservative configuration strategy.

hls4ml's FIFO optimization suggested depths are also shown in Table I. They reflect the maximum achieved occupancy of each network FIFO during a single co-simulation inference execution plus one. We observe a ˜90.4% decrease in resource consumption across LUTs, BRAMs, and FFs. Finally, Vitis DDD yields similar results to the hls4ml FIFO optimization, yielding an ˜88.5% reduction in resources across LUTs, BRAMs, and FFs. The latter two methods are often successful in identifying the optimal FIFO sizes and reducing overall resource consumption. However, lengthy runtimes paired with exorbitant CPU usage render these solutions less than ideal. In this case, hls4ml's optimization runtime exceeded **one day** and Vitis DDD lasted **10 days**. Given that the total duration from C-synthesis to bitstream generation is three and a half hours for this model, an in-hardware monitoring solution would decrease development time. Additionally, the disagreement between these two methods highlights the potential inaccuracy of software-based simulation. Thus, a more robust, time-efficient, and less compute-intense profiling scheme will greatly decrease development time.

C. FIFO Monitoring in Hardware Neural Networks

One way to overcome the limitations of software simulation-based methods discussed above is to set these hyperparamaters by monitoring the hardware directly instead. Thus, we implement a profiling scheme for *in situ* monitoring of FIFO occupancy utilizing the new Vitis HLS Stream API features previously mentioned. For this work, we implemented and tested our framework using two hls4ml models. Both consist of two convolutional layers, one dense layer, and ReLU activations. However, one model is completely "linear" from input to output, and the second features a skip connection in similar fashion to the super resolution model.

1) HLS Framework Implementation: We modify the existing hls4ml template convolutional and ReLU activation layers to incorporate a ".size()" API call at the layer input to fetch the current depth of the stream. This depth value, $size\_moc$, is exposed at the top-level IP output so that it can be viewed in real time by IP core monitoring tools. It is implemented with an "ap_vld" port-level protocol which asserts a separate valid signal when a new depth value is valid for reading. Thus, we also incorporate conditional logic to latch only the maximum achieved stream occupancy to the IP output. These maximum occupancy values can then be back-annotated to the HLS model to achieve the optimal resource consumption.

```
template <class data_T, ...>
void f(hls::stream<data_T> &data, unsigned &
    size_moc, ...) {
  STREAM_DATA_READ_IN:
  for (int i = 0; i < N; i++) {
    ...
    data_T in_data;
    io_section: {
        #pragma HLS protocol fixed
        size_moc = data.size();
        in_data = data.read();
    }
    ...
}
```

Listing 1: Probing functions for FIFO fullness measurement

D. Experimental Hardware Setup

We target two platforms to demonstrate the operation of our FIFO monitoring framework: Python productivity for Zynq for PYNQ-Z2, and the National Instruments PCI eXtensions for Instrumentation (PXI). The results presented here will primarily reflect our implementation on a Pynq-Z2 SoC and an NI PXIe-7915 co-processor which is equipped with a XCKU060 FPGA. Furthermore, the Pynq stack is applicable to many FPGA boards and hls4ml users can benefit from the accelerator system template for Vitis HLS we develop here.

1) HLS4ML Vitis HLS Accelerator Development for Pynq: Currently hls4ml does not have official support for the Vitis HLS accelerator. We first extend it to support it. We create a system template to insert the streaming IP into the Pynq system, using a wrapper to bridge the gap between the streaming IP and the DMA. The DMA controls the data transfer between the hls4ml IP and the PS system through a Jupyter Notebook. The Pynq-Z2 platform is illustrated in Fig. 1.

2) HLS4ML Vitis HLS Accelerator Development for National Instruments PXI: Similar to Pynq, we begin developing a PXI-based accelerator by starting with the hls4ml Vivado HLS accelerator template. The NI LabVIEW FPGA package greatly simplifies the process of integrating custom IPs

and seamlessly manages DMA communication with the host system. LabVIEW FPGA provides vhdl-implemented versions of the standard library functions in their graphical language, G. Through LabVIEW, we integrate the hls4ml model IP, implement the surrounding AXI-stream control logic, and design front panel user interface indicators to display the model output, latency, and maximum depth values obtained from the hardware. The template design we develop here can be utilized to implement any hls4ml model with a streaming interface. The PXI platform is illustrated in Fig. 1.

E. Results

1) CNN without skip connection: First, we target a "linear" CNN architecture with no skip connections or abnormal architectural elements. Table II shows the actual hardware implemented maximum occupancy for the Conv2D and ReLU transport FIFOs. Also shown are the default hls4ml FIFO configuration for this neural network and the Vitis co-simulation DDD maximum depths for comparison. The hardware results indicate the true FIFO occupancy for this model on this FPGA. In testing, we back-annotated these maximum depth values (plus one) to the hls4ml model configuration to validate the optimized version of the model. The model exhibited the correct behavior with the FIFO depths from Vitis DDD and from the hardware monitoring framework.

Layer	hls4ml Defaults	Vitis Co-sim DDD	PXI Hardware	Pynq-Z2 Hardware
conv2d	64	55/56	57	57
relu	36	1	1	1
conv2d_1	36	19	19	19
relu_1	16	1	1	1
dense	16	1	1	1

TABLE II: Comparison of hls4ml default depths, Vitis co-simulation suggested depths, and hardware-implemented FIFO monitoring results on hls4ml CNN with no skip connection.

2) CNN with skip connection: Next, we target a similar CNN with a skip connection incorporated into its architecture. In this test, we aim to demonstrate the ability of this framework to combat deadlocks in non-linear streaming neural network and other firmware architectures. As such, we purposefully induce a deadlock in co-simulation and hardware by setting all FIFO depths to two or less. In contrast, hardware inference using the previously presented linear architecture was successful even with FIFOs of this truncated size. This highlights the importance of carefully considering FIFO sizes when utilizing different streaming architecture configurations. Upon sizing the model's FIFOs according to the depths (plus one) reported by our hardware monitors, the hardware model with a skip connection also completed execution and produced the correct output. The hardware reported depths are in Table III.

3) Resource Impact: The addition of these FIFO monitors within the microarchitecture has a negligible impact on resource usage on PXI. The increased usage on Pynq-Z2 is because of the ILA initiation, which can be greatly decreased by buffering results to the PS side in the future development. The resource comparison between before and after the addition of these monitors for each model is shown in Table IV.

Layer	hls4ml Defaults	Deadlock	Vitis Co-sim DDD	PXI Hardware	Pynq-Z2 Hardware
conv2d	100	2	82/84	86	86
relu	64	2	1	1	1
conv2d_1	100	2	70	70	70
relu_1	64	2	1	1	1
dense	64	1	1	1	1

TABLE III: Comparison of hls4ml default depths, Vitis co-simulation FIFO suggested depths, and hardware-implemented FIFO monitoring results on hls4ml CNN with skip connection.

Board	CNN w/ skip?	FIFO Monitors Enabled?	BRAM	DSP48	Register	LUT
PXI	N	N	4.3%	0.0%	7.3%	12.9%
PXI	N	Y	4.3%	0.0%	7.4%	13.2%
PXI	Y	N	4.4%	0.0%	7.6%	13.2%
PXI	Y	Y	4.4%	0.0%	7.7%	13.4%
Pynq-Z2	N	N	8.6%	0.0%	6.3%	8.5%
Pynq-Z2	N	Y	13.6%	0.0%	9.8%	12.9%
Pynq-Z2	Y	N	17.5%	0.0%	8.7%	9.7%
Pynq-Z2	Y	Y	22.5%	0.0%	12.2%	14.1%

TABLE IV

III. Conclusion

In this paper, we present a systematic framework to dynamically monitor FIFO structures within HLS-based FPGA ML accelerators. This is accomplished by providing the necessary architectural interfaces so that users can seamlessly insert a monitoring utility into their design at the HLS programming stage without involving intrusive in-situ debugging and time and CPU-intensive co-simulation tools. The framework imposes little impact on the accelerator's performance and requires insignificant area overhead. With this monitoring ability, users can better understand the run-time behavior of the FPGA designs.

The framework is general and easily extendable. In future work, we will investigate other useful performance metrics and add them to the framework. We will also add support for monitoring a greater variety of hls4ml layers and expand testing to large assortment of architectures. We will also investigate how to compress the dynamic information saved at execution time within optimized trace buffer structures so that a longer history can be maintained and accessed by users post deployment.

References

[1] I. Corporation. (2020) "Intel oneAPI Specification.". [Online]. Available: https://spec.oneapi.io/versions/latest/oneAPI-spec.pdf

[2] V. U. S. Platform. (2019) "Vitis Unified Software Platform". [Online]. Available: https://www.xilinx.com/products/design-tools/vitis.html
[3] L. Wirbel, "Xilinx sdaccel: a unified development environment for tomorrow's data center," *The Linley Group Inc*, 2014.
[4] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, and et al., "Fast inference of deep neural networks in fpgas for particle physics," *Journal of Instrumentation*, vol. 13, no. 07, p. P07027–P07027, Jul 2018. [Online]. Available: http://dx.doi.org/10.1088/1748-0221/13/07/P07027
[5] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu, M. Leeser, and K. Vissers, "Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, pp. 1–23, 2018.
[6] I. Corporation. (2020) "Intel Advisor User Guide". [Online]. Available: https://software.intel.com/en-us/advisor-user-guide
[7] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.
[8] Nvidia. (2019) "Profiler Users Guide".
[9] M. Stephenson, S. K. Sastry Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Connor, and S. W. Keckler, "Flexible software profiling of gpu architectures," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 185–197.
[10] K. Zhang, S. Ogrenci-Memik, G. Memik, K. Yoshii, R. Sankaran, and P. Beckman, "Minimizing thermal variation across system components," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 1139–1148.
[11] K. Zhang, A. Guliani, S. Ogrenci-Memik, G. Memik, K. Yoshii, R. Sankaran, and P. Beckman, "Machine learning-based temperature prediction for runtime thermal management across system components," *IEEE Transactions on Parallel and Distributed Systems*, 2017.
[12] N. Ardalani, C. Lestourgeon, K. Sankaralingam, and X. Zhu, "Cross-architecture performance prediction (xapp) using cpu code to predict gpu performance," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*. New York, NY, USA: Association for Computing Machinery, 2015, p. 725–737. [Online]. Available: https://doi.org/10.1145/2830772.2830780
[13] X. Ma, M. Dong, L. Zhong, and Z. Deng, "Statistical power consumption analysis and modeling for gpu-based computing," in *Proceeding of ACM SOSP Workshop on Power Aware Computing and Systems (HotPower)*, vol. 1, 2009.
[14] I. Corporation. (2019) "Intel Quartus Prime Pro Edition User Guide: Debug Tools".
[15] Xilinx. (2019) "Vivado Design Suite User Guide: Programming and Debugging".
[16] J. Goeders and S. J. 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, 2016.
[17] P. Fezzardi, M. Castellana, and F. Ferrandi, "Trace-based automated logical debugging for high-level synthesis generated circuits," in *2015 33rd IEEE International Conference on Computer Design (ICCD)*. IEEE, 2015, pp. 251–258.