Evaluating high-level synthesis for neural network inference on FPGA with hls4ml

Karlis Susters
ks980@cam.ac.uk
Christ's College
University of Cambridge

Abstract

Implementing field-programmable gate array (FPGA) designs with high-level synthesis (HLS) has been effective in reducing development time for a variety of acceleration applications. However, the higher abstraction layer usually comes at a performance cost. In this report, I investigate the performance overheads of using HLS for inference of neural networks on FPGA, by creating equivalent layer implementations using Verilog. For designs with equal latency, I find an average HLS FPGA resource overhead of 47%, which matches previous research evaluating HLS for other acceleration domains. I also perform a qualitative evaluation of optimising the HLS implementations to match Verilog performance.

All code is available at `https://github.com/karlis-susters/hls4ml-vs-verilog/`

1 Introduction

Field-programmable gate arrays (FPGAs) have been used for accelerating various algorithms, including neural network inference, outperforming GPUs by typically showing lower latencies and higher energy efficiency [4]. However, programming FPGAs directly with low-level hardware description languages significantly differs from programming CPUs or GPUs and is can be error-prone, thus alternative approaches have been proposed, such as high-level synthesis (HLS), which can compile a C++ program with hardware-specific annotations into a lower-level hardware description language.

Several frameworks facilitate implementing neural network inference algorithms on FPGA, for example, Vitis AI [14], FP-DNN [2], but in this experiment I will specifically focus on hls4ml [9], which uses Vivado high-level synthesis to translate a model description in PyTorch or Keras to an FPGA implementation for inference of that neural network.

There is research comparing the quality of results when using HLS versus using a hardware description language (finding HLS performance overheads of around 50%) [5], however, as far as I am aware, there is no published research evaluating HLS specifically for neural network inference algorithms. As such algorithms are suited towards GPUs, they usually show significant parallelism and simple control flow, and might thus allow HLS to easily create an efficient implementation, as opposed to complex control flow algorithms, where significant transformations are required to extract the available parallelism and tailor the algorithm to FPGA.

Therefore, in this report, I will attempt to answer two questions concerning neural network inference on FPGA: how large are the performance overheads from using HLS versus using a low-level hardware description language, and what prevents the creation of optimised HLS applications? To investigate this, I will replicate the HLS implementations of several neural network layers in hls4ml using a hardware description language (Verilog), and compare the latency, maximum clock frequency, and required amount of FPGA resources. Then, I will explore both implementations, analyse sources of overhead, and attempt to create an optimised HLS implementation.

2 Background

2.1 Field-programmable gate arrays (FPGA)

An FPGA is a reprogrammable integrated circuit, which is used for a variety of acceleration tasks, mostly excelling at low-latency and parallel processing. It consists of a grid of look-up tables (LUTs), which can implement combinational logic, and flip-flops (FFs), which can store state. These two logic elements allow an FPGA to emulate any digital logic electronic circuit, albeit at an approximately 10 times speed penalty when compared to manufacturing a custom silicon chip implementing the same circuit. To lessen this penalty, FPGAs also contain fixed silicon elements implementing higher-level operations, such as integer multiplication, called digital signal processing blocks (DSPs).

As programming an FPGA amounts to designing an electronic circuit, performance metrics like latency, area usage and throughput are entirely deterministic and can be calculated after the design is run through FPGA synthesis, using a program like Vivado. This means that programming the design onto an FPGA is not required for an accurate evaluation, which is a step I have also omitted in this report.

This fundamental difference in design also implies that the FPGA programming semantics are entirely different to those of CPUs and GPUs. Thus, FPGAs are still mostly programmed using specialized hardware description languages, such as Verilog, however, in the last 2 decades, many alternative programming approaches have been proposed, often drawing from software engineering practices [1].

2.2 High-level synthesis (HLS)

High-level synthesis (HLS) is a method of implementing hardware designs for FPGA, which converts a program in a high-level programming language (like C++) to a hardware design in a hardware description language (like Verilog). As there is some ambiguity in how software code should be executed as an electronic circuit, special code annotations (pragmas) should be used. Of special focus in this report are pragmas that specify how loops should be executed. There are 2 main possibilities: unrolling a loop, which will execute all its iterations in parallel, thus requiring many hardware copies of the loop logic, or pipelining the loop, which will execute iterations sequentially, however, pipelining the computations of sequential iterations wherever possible. Pipelining will require fewer hardware resources but will take more cycles for the whole loop to execute.

However, the convenience of the HLS programming model tends to add a performance cost. Lahti et al. (2018) [5] compiled results from 46 papers that implemented an application using both HLS and Verilog, finding on average a 41% LUT usage overhead and 70% higher execution time for HLS implementations. On the other hand, the latency overhead was only 5%, and the reported development time of HLS was only a third of that when using Verilog, showing there are significant achievable productivity gains by using HLS if the performance overheads are tolerable, or can be decreased.

2.3 The hls4ml library

hls4ml [9] is a library for implementing neural network inference on FPGA, providing a multitude of layer implementations, pruning, and arbitrary precision quantization with quantization-aware training. It has been used in several publications, showcasing low-latency CNN and RNN physics applications like jet identification at the CERN Large Hadron Collider [7, 12], as well as real-time semantic segmentation for autonomous vehicles [11]. At its core, the library contains HLS C++ implementations of various layer types and activation functions. Then, given a Keras or PyTorch model description, it can generate HLS C++ code that parametrizes and chains together the pre-written HLS layer implementations, implementing the whole neural network on FPGA (top pipeline in Figure 1). In this report, I will be using hls4ml 0.8.0, which works with Vivado 2019.1.

There has been sizable effort devoted to optimising the HLS layer implementations in hls4ml, exploring and eliminating various issues to obtain better latency, throughput and area results [11, 13]. Carini (2022) [10] compared the efficiency of CNN implementations in hls4ml against Vitis AI, a closed source tool developed by AMD/Xilinx, showing that hls4ml gives 2-20 times lower inference latencies than Vitis AI. Carini also notes that hls4ml supports more layer types, and has more tunable parameters to produce efficient inference implementations for FPGA. This indicates that the HLS

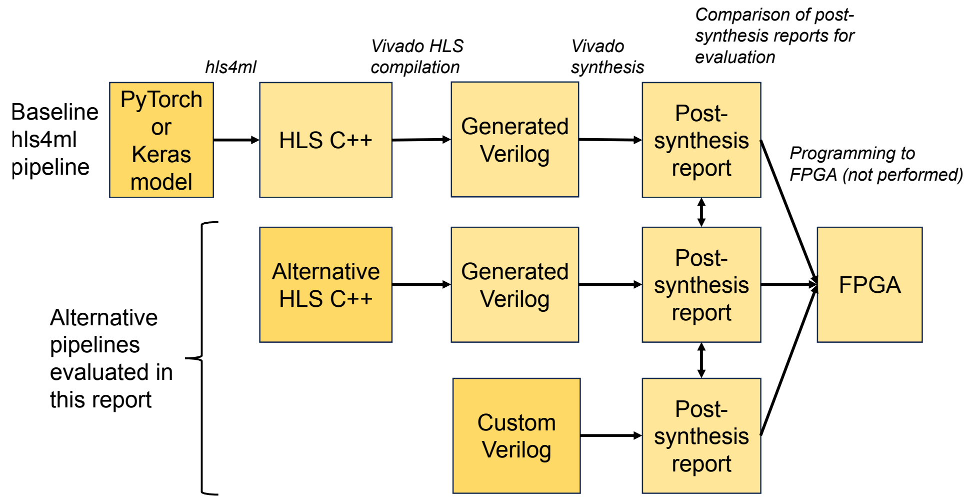

Figure 1: Outline of the hls4ml compilation pipeline, and the alternative programming pipelines evaluated in this report.

implementations generated by hls4ml are reasonably well optimised, and can thus serve as a fair benchmark to represent the high-level synthesis technique in my report.

3 Implementation

I will produce custom Verilog implementations to replicate and replace the HLS implementations in hls4ml of 3 commonly used layer types for inference on FPGA – dense, 1D convolution and 2D convolution. These layers are central in convolutional neural networks, which are frequently accelerated on FPGA [3]. Following hls4ml, all weights and biases in the custom Verilog implementations are also encoded using LUTs on the FPGA, which limits both hls4ml and my custom Verilog implementations to small layer sizes.

Although my Verilog layer implementations have been designed with good hardware design in mind (as demonstrated in the following section), the Verilog implementations I produced were intentionally not optimised, and synthesis results in the evaluation were taken from the first implementation that passed the tests. This is because my report specifically focuses on the gap between an HLS implementation and a Verilog implementation with a similar design time – there is no doubt that much more complex and performant Verilog layer implementations can be created, at the cost of significant design and verification effort.

I also considered using existing open-source Verilog layer implementations instead of writing my own, but from my investigation, I discovered several issues: namely, they might implement the layer in an entirely different fashion to hls4ml, could be integrated with other features I don't require, might have a different weight storage mechanism to the hls4ml implementations, could have a different IO interface, or not be generic enough and restricted to specific kernel sizes. Therefore, I had to partially re-invent the wheel, but I believe this should give a more representative comparison between an HLS implementation and a similarly designed and identically functioning Verilog replacement.

3.1 Dense layer implementation

An overview of my approach to implementing the dense layer is shown in Figure 2. Given a fixed number of DSP blocks (which can each perform one multiplication per cycle), the key decision in the dense layer implementation is which multiplications should be performed at the same time. Following the design in hls4ml, in each cycle, I decided to multiply all elements in a matrix column (with one vector element) and cover as many columns in parallel as the number of DSP blocks allowed. Then, in each cycle, all multiplied elements in one row need to be added together, which is done using a pipelined adder tree. Parallelising more rows and fewer columns also ensures this adder tree has low depth, which decreases latency and should decrease hardware resource usage.

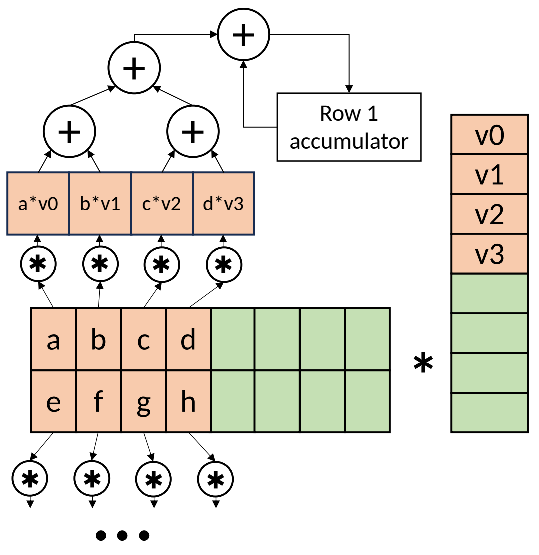

Figure 2: The custom Verilog dense layer implementation. In this example, orange and green denote separate cycles of performing the computation, and 8 DSP blocks are used.

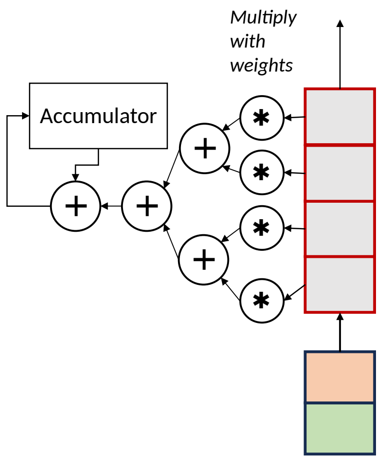

Figure 3: The custom Verilog 1D convolution layer implementation. The shift register (grey) is shifted by one element each cycle.

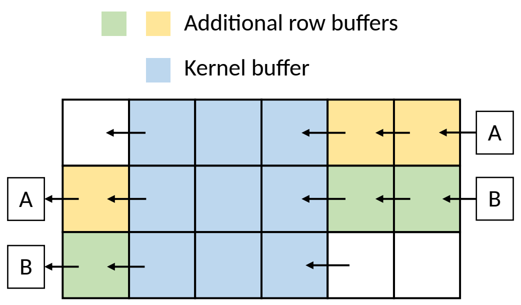

Figure 4: The kernel and row buffers in the custom Verilog 2D convolution layer. In every cycle, all elements move according to the arrows. Every element in the kernel buffer is also multiplied by weights and accumulated akin to the 1D convolution implementation.

3.2 1D convolution implementation

The Verilog 1D convolution implementation (Figure 3) consists of a shift register, where every cycle it is shifted by one element, taking one new input element. All elements are multiplied by the weights, and subsequently accumulated using a pipelined adder tree.

3.3 2D convolution implementation

The 2D convolution implementation (Figure 4) is directly inspired by the HLS implementation in hls4ml. A kernel buffer is maintained, which contains all pixels currently in the kernel, and several row buffers are kept, which contain pixels that are currently outside the kernel, but are in the same rows as the kernel (and will thus later slide into the kernel). In every cycle, all elements in the kernel buffer are multiplied with weights, and accumulated as in the 1D convolution implementation.

3.4 Integration with hls4ml

To evaluate my Verilog implementations in an identical setting as the generated Verilog from hls4ml (with the same IO interface and weight storage mechanism), I would first run hls4ml and HLS

16-bit fixed point	Dense (16 to 64 neurons)				Dense (64 to 16 neurons)			
	hls4ml (latency)	hls4ml (resource)	Optimised HLS	Custom Verilog	hls4ml (latency)	hls4ml (resource)	Optimised HLS	Custom Verilog
LUTs	19678	6891	5558	4361	20545	5874	4722	2714
FFs	17683	8988	7447	1064	17578	11692	10016	789
Latency (cycles)	10	20	21	10	11	22	24	11
Est. maximum frequency	153 MHz	129 MHz	120 MHz	153 MHz	153 MHz	131 MHz	131 MHz	153 MHz

Table 1: Area, latency, and timing results for the Dense layer implementations (using 16-bit fixed point). DSP usage was fixed and observed at 128 blocks for all implementations.

compilation to generate a Verilog implementation of the neural network. Then, I would overwrite the generated Verilog implementation of a neural network layer with my custom Verilog implementation. To do this, I also had to reverse-engineer some control signals in the generated Verilog so that my custom Verilog implementations function correctly. This solution also enabled me to use the built-in tests in hls4ml, that simulate the neural network in Verilog and check its outputs.

Hls4ml also supports two types of HLS implementations for most layers – 'latency' and 'resource', each prioritizing their respective metrics. Results from both will be reported. In addition, for the dense layer, the 'io_parallel' IO type was used (which means each input and output neuron is a separate wire into the layer module), but for 1D and 2D convolution, 'io_stream' was used (which meant that the input and output pixels were sequentially sent over a single wire).

3.5 Synthesis

To compare the custom Verilog implementations and the hls4ml generated ones, I run both implementations through logic synthesis using Vivado 2019.1 (the first step in FPGA compilation) targeting a ZYNQ XC7Z020 SoC (used in the Pynq-Z2 boards). For some configurations, I ran the full FPGA compilation (including place & route) but observed that the used resource estimates from logic synthesis were inaccurate by at most 5%. Thus, following the methodology in other publications [6, 7, 12], I documented my results using only the post-synthesis reports for both custom and generated Verilog implementations. The reported latencies (in cycles) are fixed in place by the Verilog design and are thus perfectly accurate, however, the estimated maximum clock frequency values were quite unstable (as they are determined by the single longest logic path), and could vary by up to 50%. Therefore, the estimated maximum clock frequency in my evaluation should be examined as a general indicator that the hardware implementations are reasonable, however, the obtained area and latency measurements should be very accurate.

4 Evaluation

For my evaluation, I selected several configurations of the Dense, Conv1D, and Conv2D layer types, and ran logic synthesis to obtain the area, latency, and estimated maximum frequency metrics.

As low-precision quantization is a common theme of exploration on FPGA [3, 4, 8], I experimented using both 16-bit fixed point precision (6 integer bits) for all weights, biases, accumulators and activations, and 8-bit fixed point precision (3 integer bits). The key difference between these two variants (as will be observed later), is that FPGA synthesis tools prefer to use a dedicated DSP block for 16-bit multiplication while implementing 8-bit multiplication using regular LUTs. All layers also apply the ReLU activation function to outputs, but due to its simplicity, it should not consume much additional logic.

8-bit fixed point	Dense (16 to 64 neurons)				Dense (64 to 16 neurons)			
	hls4ml (latency)	hls4ml (resource)	Optimised HLS	Custom Verilog	hls4ml (latency)	hls4ml (resource)	Optimised HLS	Custom Verilog
LUTs	8436	9222	9417	10859	7843	8039	8136	8717
FFs	7770	7939	7795	2567	7824	6339	6439	2513
Latency (cycles)	6	5	5	6	7	6	6	7
Est. maximum frequency	180 MHz	120 MHz	120 MHz	139 MHz	180 MHz	142 MHz	142 MHz	150 MHz

Table 2: Area, latency, and timing results for the Dense layer implementations (using 8-bit fixed point). 0 DSP blocks were used by all implementations.

4.1 Dense

For Dense, I experimented with both a 16x64 matrix and a 64x16 matrix, as those should result in fairly different implementations due to the different size adder trees employed. Hls4ml also exposes a parameter that can configure the maximum amount of DSP blocks used by the design ("reuse factor"), where using more blocks would result in lower latency but higher resource usage. For the 16-bit implementation, I fixed the DSP block usage to 128. As both dense layers require performing 1024 multiplications, and a DSP block can perform 1 multiplication per cycle, all implementations will require at least 8 cycles of latency.

The results when using 16-bit quantization are shown in Table 1. Firstly, comparing the two hls4ml implementations in both 64x16 and 16x64 dense, there is a stark 3-4x LUT usage difference between them. The custom Verilog implementation does better, using 37% or 54% fewer LUTs than the "resource" hls4ml implementation.

The latency for the Verilog and hls4ml "latency" is slightly above the absolute lower bound, at 10 cycles. The "resource" implementation latency was higher at around 20 cycles, however, investigating this issue I discovered that 10 of those cycles are a constant overhead, meaning that if the layer sizes were scaled up (or fewer DSP blocks were allocated), the latencies of all implementations should converge.

All 16-bit HLS implementations, however, use around 8x more flip flops than custom Verilog. However, this is not a critical issue, because in FPGAs flip-flops come bundled with LUTs at a ratio of 2 FFs for 1 LUT, therefore in none of the implementations would FF usage present a bottleneck. Thus, in a way, HLS is being more efficient by utilizing resources more proportionately, while the custom Verilog implementations are under-utilizing FFs.

Evaluating the Dense layer with 8-bit quantization (Table 2), a rather different picture emerges, where the custom Verilog implementation consumes around 10% more LUTs than hls4ml. This is because for the hls4ml implementations, the HLS compiler decided to decrease the latency of the module to 5-6 cycles (and didn't react to the hls4ml latency controls), which meant that I also had to scale up the custom Verilog implementation to equalize latency for a fair comparison. This process required increasing the size of the adder tree in my implementation, which doesn't seem to scale too well, thus drastically increasing the required area. This example shows that firstly, the HLS compiler can sometimes be slightly hard to control to get the exact area-latency tradeoff that is desired, but also that it can be more flexible in changing underlying structures when a different latency is targeted.

4.2 Optimising HLS for the dense layer

After observing the efficiency of the custom Verilog implementation, I attempted to replicate my results using HLS C++. However, my results here were mostly negative, and I failed to produce a more efficient HLS implementation than the hls4ml "resource" version. While my initial idea was to write very low-level HLS that would specify what operation should happen in each cycle, I was slightly frustrated at the lack of such low-level control over the timing of operations.

	Conv1D (16 bit)			Conv1D (8 bit)		
	hls4ml (latency)	hls4ml (resource)	Custom Verilog	hls4ml (latency)	hls4ml (resource)	Custom Verilog
LUTs	636	802	517	630	661	432
FFs	864	1202	268	687	751	539
DSPs	32	32	32	0	0	0
Latency (cycles)	141	5433	133	137	5178	133
Est. maximum frequency	167 MHz	173 MHz	167 MHz	173 MHz	173 MHz	173 MHz

Table 3: Area, latency, and timing results for the 1D convolution layer implementations (using 16 or 8-bit fixed point with a size 32 kernel and 128 input items).

Then, I turned to optimise the "resource" implementation and found that removing a "rewind" loop annotation would slightly increase the latency, but give a 20% LUT reduction in the 16-bit fixed point dense layer (Table 1). This annotation should decrease the latency of successive invocations of the loop, however, it is not obvious to me why it was consuming significant additional hardware resources. It should also be noted that my optimisation had no effect in the 8-bit fixed point version.

4.3 1D convolution

For the 1D and 2D convolution layers, I wanted to increase the kernel sizes to obtain larger and more representative logic usage results, however, the hls4ml implementations would fail to compile with large kernels, so I had to use a size 32 kernel for 1D convolution and a 5x5 kernel for 2D convolution (using 128 inputs or 16x16 inputs respectively)

The synthesis results for 1D convolution are shown in Table 3. The custom Verilog implementation consumes 20-40% fewer LUTs than hls4ml, with similar FF usage patterns as for the dense layer. Also, the previously seen trend of implementations being equal in 8-bit precision is not visible here, and custom Verilog still performs better.

An extremely high latency (5433 and 5178 cycles) can be observed for the hls4ml resource implementations. I investigated the cause of this and discovered it was because pragmas in some crucial loops were severely limiting parallelism by executing the iterations sequentially instead of unrolling. This meant that, for example, each element in the kernel was being shifted in a separate cycle (instead of all being shifted in the same cycle simultaneously). While generally executing loop iterations sequentially requires fewer hardware resources, in this case, it is rather unlikely that shifting elements one by one should give any area savings. Indeed, I also verified this, as modifying the hls4ml "resource" implementation to unroll the kernel shifting loop gives a 5x reduction in latency and a 10% reduction in LUTs. This example should show that correctly using pragmas in HLS can be tricky and does require some low-level hardware knowledge to produce an efficient implementation.

4.4 2D convolution

For 2D convolution, the observed trends are similar to 1D convolution (Table 4), where Verilog uses 20%-50% fewer LUTs for both 16-bit and 8-bit precision. The "resource" implementation also has a large latency because of the same reasons as for 1D convolution, that is, very aggressive pragmas on loops attempting to decrease area usage.

5 Conclusion

I have successfully compared the HLS implementations in hls4ml to similarly designed implementations in Verilog across 8 layer and parameter categories and found on average a 47% LUT overhead for the best HLS implementation when latency is near-equal. This is very similar to the previously

	Conv2D (16 bit)			Conv2D (8 bit)		
	hls4ml (latency)	hls4ml (resource)	Custom Verilog	hls4ml (latency)	hls4ml (resource)	Custom Verilog
LUTs	820	514	437	787	531	429
FFs	1552	1455	242	1214	1063	471
DSPs	25	25	25	0	0	0
Latency (cycles)	271	3835	260	268	3069	260
Est. maximum frequency	142 MHz	157 MHz	162 MHz	153 MHz	162 MHz	157 MHz

Table 4: Area, latency, and timing results for the 2D convolution layer implementations (using 16 or 8-bit fixed point with a size 5x5 kernel and 16x16 input items).

measured HLS LUT overhead of 41% [5], giving some doubt to the hypothesis that HLS for machine learning is more effective than average. I also attempted optimising the HLS implementations in hls4ml but with rather limited success, partially because I believe the hls4ml implementations are already fairly efficient.

Importantly, the DSP usage in all hls4ml implementations was equal to the Verilog implementations, showing that HLS does not waste DSP blocks. This is significant, as most practical 16-bit FPGA inference implementations will likely be bottlenecked by DSP usage, with LUTs left over (if the FPGA is used only for inference), as in my dense layer experiments, hls4ml was consuming 50% of the available DSPs on the PYNQ-Z2 FPGA, yet only 20% of the available LUTs. In such a scenario, the LUT overheads of HLS will cause no negative effect. For lower-precision inference, however, the LUT overheads could cause a practical performance impact.

I also explored the reasons for anomalous behaviour in some HLS implementations. Issues could be caused by the usage of pragmas without understanding the general paradigms of efficient hardware design, thereby forcing HLS into generating inefficient designs. Tuning HLS designs can also be more frustrating, due to the additional layer of abstraction with limited low-level controls.

5.1 Limitations

The key limitation in my work is a question of inductive reasoning – whether the results from comparing my custom Verilog implementation against the hls4ml HLS layer implementations will generalize to comparing average neural network Verilog and HLS implementations. I have taken steps to improve the generalizability by not significantly optimising the custom Verilog implementations, and ensuring that the hls4ml implementations are reasonably efficient, but any potential reader should still carefully consider this aspect.

Also, it was difficult to decompose and understand the area overheads of HLS, due to the generated Verilog code being quite complex and not very readable. It could be that there are tools or techniques that provide more insight into the HLS compiler, which I am not aware of.

5.2 Further work

As further work, it would be interesting to explore if custom Verilog implementations can be integrated into hls4ml to be used instead of the pre-written HLS layers. There is some support for RTL black-boxing in Vivado HLS, but integrating it smoothly and efficiently may also present challenges. An alternative would be to create a library that doesn't use HLS at all and converts a PyTorch model directly into Verilog by chaining pre-written Verilog layer implementations. However, more development time might be required, especially to create Verilog layers that can function well for all model configurations.

References

[1] N. Kapre and S. Bayliss, “Survey of domain-specific languages for fpga computing,” in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016, pp. 1–12. DOI: 10.1109/FPL.2016.7577380.

[2] Y. Guan *et al.*, “Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates,” in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017, pp. 152–159. DOI: 10.1109/FCCM.2017.25.

[3] S. Mittal, “A survey of fpga-based accelerators for convolutional neural networks,” en, *Neural Computing and Applications*, vol. 32, no. 4, pp. 1109–1139, Oct. 2018. DOI: 10.1007/s00521-018-3761-1. [Online]. Available: http://dx.doi.org/10.1007/s00521-018-3761-1.

[4] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, “[dl] a survey of fpga-based neural network inference accelerators,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 12, no. 1, Mar. 2019, ISSN: 1936-7406. DOI: 10.1145/3289185. [Online]. Available: https://doi.org/10.1145/3289185.

[5] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, “Are we there yet? a study on the state of high-level synthesis,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 898–911, 2019. DOI: 10.1109/TCAD.2018.2834439.

[6] J. Ngadiuba *et al.*, “Compressing deep neural networks on fpgas to binary and ternary precision with hls4ml,” *Machine Learning: Science and Technology*, vol. 2, no. 1, p. 015 001, Dec. 2020. DOI: 10.1088/2632-2153/aba042. [Online]. Available: https://dx.doi.org/10.1088/2632-2153/aba042.

[7] T. Aarrestad *et al.*, “Fast convolutional neural networks on fpgas with hls4ml,” *Machine Learning: Science and Technology*, vol. 2, no. 4, p. 045 015, 2021. DOI: 10.1088/2632-2153/ac0ea1.

[8] S.-E. Chang *et al.*, “Mix and match: A novel fpga-centric deep neural network quantization framework,” in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 208–220. DOI: 10.1109/HPCA51647.2021.00027.

[9] F. Fahim *et al.*, *Hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices*, 2021. arXiv: 2103.05579 [cs.LG].

[10] D. Carini, “Comparing hls4ml and vitis ai for cnn synthesis and evaluation on fpga: A comprehensive study,” 2022.

[11] N. Ghielmetti *et al.*, “Real-time semantic segmentation on fpgas for autonomous vehicles with hls4ml,” *Machine Learning: Science and Technology*, vol. 3, no. 4, p. 045 011, 2022. DOI: 10.1088/2632-2153/ac9cb5.

[12] E. E. Khoda *et al.*, “Ultra-low latency recurrent neural network inference on fpgas for physics applications with hls4ml,” *Machine Learning: Science and Technology*, vol. 4, no. 2, p. 025 004, 2023. DOI: 10.1088/2632-2153/acc0d7.

[13] J.-X. Hu, *Hls4ml progress update*. [Online]. Available: https://indico.cern.ch/event/1027582/contributions/4465213/attachments/2289134/3891487/0729%20HLS4ML%20Progress%20Update.pdf.

[14] Xilinx, *Vitis AI*. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html.