Manuscript version: Author’s Accepted Manuscript
The version presented in WRAP is the author’s accepted manuscript and may differ from the published version or Version of Record.

Persistent WRAP URL:
http://wrap.warwick.ac.uk/128884

How to cite:
Please refer to published version for the most recent bibliographic citation information. If a published version is known of, the repository item page linked to above, will contain details on accessing it.

Copyright and reuse:
The Warwick Research Archive Portal (WRAP) makes this work by researchers of the University of Warwick available open access under the following conditions.

Copyright © and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners. To the extent reasonable and practicable the material made available in WRAP has been checked for eligibility before being made available.

Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

Publisher’s statement:
Please refer to the repository item page, publisher’s statement section, for further information.

For more information, please contact the WRAP Team at: wrap@warwick.ac.uk.
Abstract—Implementations of hardware accelerators for neural networks are increasingly popular on FPGAs, due to flexibility, achievable performance and efficiency gains resulting from network optimisations. The long compilation time required by the backend toolflow, however, makes rapid deployment and prototyping of such accelerators on FPGAs more difficult. Moreover, achieving high frequency of operation requires significant low-level design effort. We present a neural network overlay for FPGAs that exploits DSP blocks, operating at near their theoretical maximum frequency, while minimizing resource utilization. The proposed architecture is flexible, enabling rapid runtime configuration of network parameters according to the desired network topology. It is tailored for lightweight edge implementations requiring acceleration, rather than the highest throughput achieved by more complex architectures in the datacenter.

I. INTRODUCTION

The increasing efficacy of Neural Networks (NNs) has attracted interest not only in improving their accuracy, but also in accelerating this class of workloads for real time performance. The inherent parallelism and computational regularity in NNs have been taken advantage of in highly parallel computing platforms, such as multicore CPUs and GPUs, and in custom computing architectures on FPGAs and ASICs. The ease of accelerating NNs in highly parallel computing platforms, through the availability of a number of frameworks, coupled with their fast compilation, have driven wider use of such platforms. Custom computing architectures offer additional advantages in terms of datapath and numerical representation optimisations, offering improved energy efficiency, which in turn makes them ideal for power-constrained platforms at the edge, where multicore CPUs and GPUs are unlikely to be suitable.

Numerous approaches and methodologies have been proposed in order to bridge the gap between FPGAs and highly parallel computing platforms. High Level Synthesis (HLS) has contributed significantly in reducing accelerator design time, but still requires a lengthy backend toolflow. In order to automate the mapping of Convolutional Neural Networks on FPGAs, a variety of toolflows have been proposed [1]. The majority of these take advantage of the higher abstraction layer offered by HLS to also provide design space exploration, resulting in hardware implementations tailored to user requirements and platform capabilities. The Xilinx Deep Neural Network Development Kit (DNNDK) is an example of a vendor flow for accelerating NN inference on an accelerator architecture on FPGAs.

Most FPGA-based NNs architectures operate at significantly lower frequencies compared to the capabilities of the DSP blocks in modern FPGAs [2]. This results in less energy efficient implementations as leakage is clock independent [3], and has more impact on FPGAs at newer process nodes where leakage current is more dominant. In contrast, FPGA implementations that achieve high operating frequencies do so at the expense of flexibility and thus modifications to network topology or coefficients require a new compilation.

Overlays have been proposed as a way of enabling high level programming with rapid compilation and predictable performance on FPGAs. When designed in an architecture-centric manner, overlays can achieve near the theoretical maximum frequency supported by underlying FPGA architecture, while scaling to large overlay sizes [4]. Meanwhile, compilation does not involve the FPGA backend flow and so can be very fast, lightweight and vendor independent.

II. RELATED WORK

Overlays enhance flexibility in custom computing architectures by forming a coarser grained abstraction on top of the FPGA fabric and, as a result, reduce the long compilation time required by the backend toolflow. Performance can be more predictable as it is closely tied to the fixed performance of functional units, and routing overhead can be reduced by taking into account the regularity of the required data movement [5]. The authors in [6] present a family of overlay architectures and associated design methodology. By using datapath merging, they minimise the added overhead to support various computations while also providing optional adjustable flexibility through a secondary interconnect network. Their experiments demonstrate faster runtime compilation and reduced area utilization, though resulting in reduced operating frequency due to the slower operators occurring in the same context as faster ones. Further performance improvement in overlays can be obtained when tailoring the architecture to heavily take advantage of the high performance DSP blocks, that are abundant in modern FPGAs [4].

More generic NN computing architectures, like the DPU used by the DNNDK, offer a more balanced performance acceleration-to-flexibility and area ratio. DNNDK includes model compression, by using data quantization and pruning.
Moreover, by being able to operate at a relevantly high frequency, the performance overhead is somewhat mitigated. The proposed overlay architecture is depicted in Fig. 1.

The input data along with a valid signal stream into the overlay serially. Each neuron receives a new input which is subsequently passed to the next neuron in the same layer. To allow this serial computing pattern to deal with a number of neurons at one layer that is greater than the number of layer inputs, data flow must be stalled for a number of clock cycles as required by the network topology. Before receiving any inputs, the overlay is configured by setting the number of neurons used at each layer (the reset address for each layer), when to stall and for how many clock cycles, along with the network weights and biases. The weights are stored in LUTRAMs while the rest of the configurations in registers.

After the overlay is configured, the input data flows into the first layer, from neuron to neuron, along with a valid signal that is used to enable the address counter. The address counter increments accordingly and addresses the LUTRAM where the weights are stored, feeding the corresponding weight of each input to the DSP block. The counter resets when it reaches its configured reset address, enabling the proposed overlay to adjust its latency, and as a result its performance, to the topology of the configured network. The address counter is also used to alternate between two DSP opmodes. Instead of resetting the accumulation register at the beginning each iteration, the DSP block OPMODE changes to add the multiplication’s product to the bias (C input of the DSP block). Avoiding, as a result, redundant additions with zero, replacing them instead with the bias additions that would normally take place after all the weighted inputs have been accumulated. For the rest of the computation, a different DSP block OPMODE is used to accumulate the product. When the address counter of a neuron reaches its reset address, meaning that the computation of the neuron has completed, a pulse is generated. The pulse is delayed by three clock cycles for synchronization, and fed to a state machine that generates the enable signal for the first neuron in following layer. The enable signal subsequently propagates from neuron to neuron similarly to the first layer.

Meanwhile, a multiplexer between two layers, addressed
by the counter of the first neuron in the following layer, selects the appropriate input from the previous layer. Each input to the multiplexer is reduced from 48 bit, the output of the DSP block, to 27, the input of the following DSP block, by selecting the appropriate bit range according to the fixed point representation used. The selected output is then passed to another multiplexer that implements the ReLU activation function, by checking whether the MSB is set to 1, and passing the input to the next layer accordingly.

### B. Stall Mechanism

To make the processing and data flow stall for a number of clock cycles, we set accordingly the valid signal. The stall mechanism, shown in Fig. 2, is configured externally before processing takes place. The valid signal is connected to the en_in port, while the rest of the ports, are connected to the external component, i.e. the Arm core. The stall mechanism takes two 5 bit inputs along with their active high configuration signals and stores their values to registers. The total_stall_cycles input takes the total number of clock cycles (processing cycles + stall cycles), while the when_to_stop input takes the number of processing cycles.

The counter increments as long as en_in is active, which means that input data flows to the accelerator. It resets to zero when it is synchronously reset or when it reaches the total_stall_cycles. The counter output is used to detect whether it has reached the point where it has to stall (count==when_to_stop) or whether it has reached the point to stop stalling (count==total_stall_cycles). The output flip-flop inverses its output accordingly, and combined with en_in, controls when to stall the overlay. Where not needed, the stall component can be disabled by setting both, when_to_stop and total_stall_cycles, to the same value. This causes the XOR gate not to generate an active pulse to trigger the T flip-flop.

### C. Case Study

We used the datasets and networks in Table I, trained using Tensorflow [10] to obtain the accuracies shown. These were chosen to represent a range of application domains and to match or exceed the complexity of NNs that have been more widely targetted for acceleration, for instance in [11] and [12]. The proposed overlay, designed for inference, does not implement an activation function at the output layer, since the required comparisons can be more flexibly made in software and raw outputs can used as feedback for fine-tuning.

The proposed overlay is tailored to the features of the DSP48E2 block on the Zynq Ultrascale+ ZU7EV. This DSP block comprises a 27×18 bit multiplier with a 48 bit accumulator/adder. After exploring the networks and datasets in Python, we decided on a representation with 12 fractional bits as it results in no accuracy reduction. The overlay uses 18 bit weights, 48 bit biases, which can be configured externally, and 27 bit inputs. By analysing the topologies of the NNs included in this case study we arrived at an overlay with a 11-12-10-3 configuration and implemented it along with the stall mechanism using Verilog. The proposed architecture has been behaviourally simulated and verified against the expected output in each dataset. The design has then been synthesised and implemented using Vivado 2018.2 and all the results are post place and route.

The proposed architecture can operate at maximum frequency of 770MHz, which is close to theoretical maximum, 775MHz, of the devices’s DSP blocks [13]. The resource utilization of each module is summarized in Table II. It is important to note that the design of the stall mechanism results in an insignificant area overhead, while the total utilization is very small, meaning this architecture could be scaled up significantly on this device.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>NN Topology</th>
<th>Train Entries</th>
<th>Acc. Train</th>
<th>Test Entries</th>
<th>Acc. Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer Churn</td>
<td>11-6-6-1</td>
<td>8000</td>
<td>84.26%</td>
<td>2000</td>
<td>82.95%</td>
</tr>
<tr>
<td>Diabetes</td>
<td>8-12-8-1</td>
<td>768</td>
<td>78.39%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Iris</td>
<td>4-10-10-3</td>
<td>120</td>
<td>98.33%</td>
<td>30</td>
<td>96.67%</td>
</tr>
<tr>
<td>Overlay</td>
<td>11-12-10-3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Module</th>
<th>LUTs</th>
<th>LUTRAM</th>
<th>FFs</th>
<th>DSPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overlay</td>
<td>796</td>
<td>225</td>
<td>2552</td>
<td>25</td>
</tr>
<tr>
<td>Stall Mechanism</td>
<td>24</td>
<td>0</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>819</td>
<td>225</td>
<td>2568</td>
<td>25</td>
</tr>
<tr>
<td>Available</td>
<td>230400</td>
<td>101760</td>
<td>460800</td>
<td>1728</td>
</tr>
</tbody>
</table>
IV. RESULTS AND DISCUSSION

From the simulations, we extracted the number of clock cycles for each network to process the first dataset entry, labelled latency, along with the clock cycles required to process a following entry when the pipeline is saturated, labelled interval. We also provide the number of stall cycles to quantify the stalling overhead. In each case we took into consideration the maximum operating frequency of 770MHz, showing how that translates to actual runtime in Table. III. Compared to other FPGA implementations in the literature, the authors in [11] implement a neural network for gas classification on a Xilinx Zynq XC7Z010T using Vivado HLS v2016.1. The architecture uses fixed point arithmetic and operates at 100MHz, as well as using the more expensive Sigmoid activation function. Parallelism is exploited with pragmas for loop unrolling and pipelining, and they report a latency of 540 ns for their simpler 12-3-1 network topology, which is about 10× slower compared to the proposed overlay for a 8-12-8-1 network that results in a 48.026 ns latency.

### TABLE III
THEORETICAL TIMING RESULTS FOR THE OVERLAY.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Clock Cycles</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Latency</td>
<td>Interval</td>
</tr>
<tr>
<td>Customer Churn</td>
<td>41.536</td>
<td>14.278</td>
</tr>
<tr>
<td>Diabetes</td>
<td>48.026</td>
<td>15.576</td>
</tr>
<tr>
<td>Iris</td>
<td>45.43</td>
<td>12.98</td>
</tr>
</tbody>
</table>

To provide a reference for comparison we processed all three neural networks in software on the Arm Cortex-A53 bare-metal, operating at 1.2GHz, as found in the same Ultrascale+ device. We also processed them on a desktop PC running Ubuntu Linux 18.04 on an Intel Core i7-6700 CPU, at 3.40GHz. Fixed point representation was used for the software, implemented in C. From the execution time measured and the theoretical timings of the overlay, we calculated the inference throughput for each network in Table. IV.

### TABLE IV
INFERENCES PER SECOND ON THE DIFFERENT ARCHITECTURES.

<table>
<thead>
<tr>
<th>Neural Network</th>
<th>Inferences/sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Arm-A53 @1.2GHz</td>
</tr>
<tr>
<td>Customer Churn</td>
<td>0.151 × 10^6</td>
</tr>
<tr>
<td>Diabetes</td>
<td>0.089 × 10^6</td>
</tr>
<tr>
<td>Iris</td>
<td>0.099 × 10^6</td>
</tr>
</tbody>
</table>

The proposed overlay offers a significant performance improvement, compared to the embedded Arm core, able to process the networks in our case study at a significantly greater rate. The proposed overlay is at least 19× faster than the desktop class Intel Core i7-6700.

V. CONCLUSION

In this paper we presented a lightweight streaming neural network overlay that is optimised for the high performance DSP blocks in modern FPGAs and exploits their programmability. The proposed overlay reduces dependency on the backend toolflow and enhances flexibility and programmability of FPGAs in the neural networks domain. The implemented overlay architecture maintains low resource utilization while operating at near the theoretical maximum of the platform. The high operating frequency, that also contributes in minimising the impact of leakage current, coupled with the minimal resource utilization and significant performance improvements, make the proposed overlay ideal for processing at the edge.

In the future, we plan on expanding this work to support deeper and alternative neural network topologies along with a rapid toolflow to automate the configuration and mapping process. This will enable for rapid deployment of neural network accelerators on FPGAs at the edge in a context where compilation can take place on the resource constrained platforms themselves, independently of the vendor backend toolflow.

ACKNOWLEDGEMENT

This work was supported in part by an IBM Faculty Award and the UK Engineering and Physical Sciences Research Council (EPSRC), grant EP/N509796/1.

REFERENCES