Towards In-Circuit Tuning of Deep Learning Designs
invited paper

Zhiqiang Que*, Daniel Holanda Noronha†, Ruizhe Zhao*, Steven J.E. Wilton†, Wayne Luk*
*Dept. of Computing, School of Engineering, Imperial College London, UK {*z.que, ruizhe.zhao15, w.luk*}*@imperial.ac.uk*
†Electrical and Computer Engineering, University of British Columbia, Vancouver, BC, Canada {*danielhn, stevew*}*@ece.ubc.ca*

***Abstract*—This paper presents InTune, a novel approach for in-circuit tuning of deep learning designs targeting implementations in field-programmable gate array technology. This approach combines two promising techniques: domain-specific adaptation and in-circuit tuning. Domain-specific adaptation exploits domain-specific information in adapting pre-trained models to specific application domains, replacing standard convolution layers with efficient convolution blocks; the effects of such adaptation are then assessed by in-circuit tuning instruments to provide information to application builders for tuning the design. This approach is illustrated by its deployment in tuning deep neural networks, and its potential for a new generation of domain-specific tools with tight integration of synthesis and in-circuit tuning is explored.**

I. Introduction

Field-Programmable Gate Array (FPGA) has become a popular and promising technology for implementing Convolutional Neural Network (CNN) [1, 2] in recent years. Most CNN applications on FPGA are domain-specific involving the detection and classification of objects from a narrow range of classes. It has been shown that transfer learning [3] can be applied to CNN models pre-trained on general datasets, e.g. ImageNet, to support efficient fine-tuning [4] for specific domains; this approach has been adopted in developing domain-specific CNN designs for FPGAs [5].

Although FPGA implementation of CNNs is promising, tuning and debugging a Deep Neural Network (DNN) mapped onto an FPGA are still challenging. Software simulators are important parts of a Machine Learning debug and performance tuning ecosystem, but software simulation may be insufficient to find the root cause of many types of functional and performance bugs. Software simulation often requires long run time with multiple training and inference sessions in locating hardware bugs. Besides, it may not exactly capture the hardware implementation because the environment can not be adequately described, e.g. non-deterministic DRAM accesses. Furthermore, some third-party hardware libraries may not have accurate simulation models. The only method to find the cause of these types of bugs is to run the hardware in a realistic environment at the actual speed with representative workloads.

This paper describes InTune, a novel in-circuit tuning approach for domain-specific deep learning designs. The proposed tuning instrumentation enables visualisation of the patterns of overflow exceptions and supports precision tuning of CNN layers. To the best of our knowledge, this is the first in-circuit tuning approach that can automatically fine-tune a domain-specific FPGA design using information from hardware.

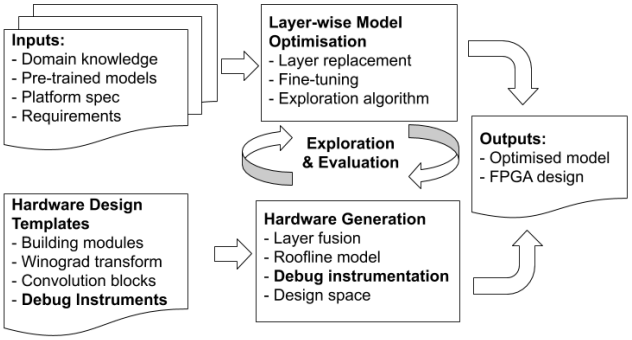

Fig. 1. The InTune design flow.

Our contributions are as follows:

1) Novel map and statistics instruments for in-circuit detection of overflow in DNN designs.
2) An approach for fine-tuning DNN accuracy based on such map and statistics instruments.
3) Evaluation of the proposed approach showing, for example, that it can be significantly faster than software simulation for tuning DNN designs on FPGAs.

Figure 1 shows the development flow for domain-specific CNN designs [5]. InTune enables in-circuit tuning in the Exploration-and-Evaluation step for optimising such designs.

II. Background and Related Work

A. Convolution Layer

A convolution layer performs multi-dimensional convolution computation between an input feature map and a filter. It extracts features from an input feature map and generates a new feature map. More specifically, given an input tensor $x \in R^{N_x \times N_y \times N_c}$ (e.g. a 2D image with N_c channels), a weight tensor $w \in R^{k_x \times k_y \times N_c \times N_f}$, and a bias term $b \in R^{N_f}$, a convolution layer f is defined as:

$$f(x,w,b)_{lmn} = \sum_{i=l-k_x/2}^{l+k_x/2} \sum_{j=m-k_y/2}^{m+k_y/2} \sum_{k=0}^{N_f} w_{ijkn} x_{ijk} + b_n \quad (1)$$

Weights in convolution layers are often called *convolutional kernels*. Convolutional layers have hyper-parameters such as

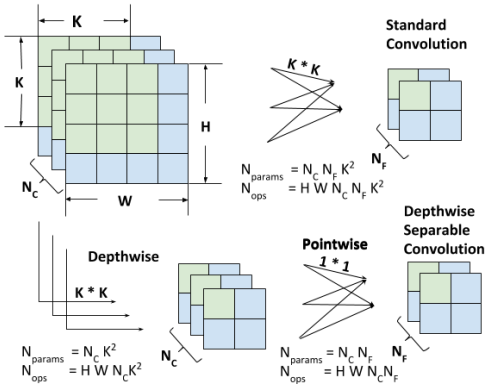

Fig. 2. (a) Standard and (b) Depthwise Separable convolution layers.

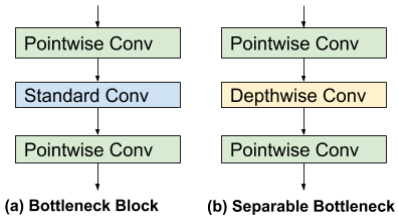

Fig. 3. Bottleneck (left) and Separable Bottleneck (right) convolution blocks.

kernel width (k_x, k_y), number of filters N_f, *stride* and *dilation* factors. Compared with a fully-connected layer in feed-forward networks applying independent weights to all input features, a convolutional layer has fewer parameters as its weights are shared among all elements within a channel of a feature map. This property also allows convolution layers to extract spatially local features.

Besides the standard convolution, there are some other types of convolution. Depthwise convolution [6, 7, 8] and pointwise convolution [6] are both lightweight building blocks of modern CNNs. Figure 2 illustrates the comparison among standard, depthwise, and pointwise convolution. Compared with the standard convolution, depthwise convolution only applies one filter on each channel, which significantly decreases the amount of computation and parameters and is relatively efficient. A depthwise separable convolution layer stacks the depthwise and pointwise convolution, which extracts spatial and cross-channel correlation using depthwise and pointwise convolution respectively. There are also different types of convolution block. The bottleneck block (Figure 2), in ResNet-50 [9], covers residual learning. A separable bottleneck block is used in MobileNet V2 [10] to improve efficiency.

B. Domain-specific Debug Instrumentation

On-chip debugging can be preferable to software simulation. First, many bugs only manifest when the design is being executed at-speed, on-chip. These bugs are commonly related to timing issues or real-time IO patterns that are hard to simulate (e.g. random DRAM accesses or real-time input data from sensors). Second, simulating large circuits can take a long time, while on-chip debugging is fast.

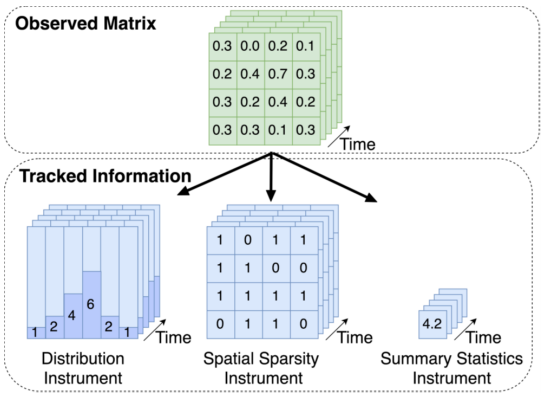

Fig. 4. Instruments overview.

Machine learning circuits can be difficult to debug. To understand their overall behaviour, we need to take even longer execution time compared with other designs to iterate over multiple training or inference samples. Moreover, those applications are commonly compiled from high-level programming languages and frameworks, such as C and TensorFlow [11], to RTL. The lack of human-readability in generated hardware designs makes debugging more difficult.

The debug toolflow proposed in [12] addresses those problems by using domain-specific characteristics of machine learning circuits to store more useful information on-chip. Similar to common hardware-oriented debug flows, debug instrumentation is added to the design in order to trace information and store it on-chip for later interrogation. Rather than focusing on individual signals and wires, [12] focuses on tracing large matrices produced in machine learning workloads, e.g., activations and weights. Moreover, instead of storing the raw values of those matrices on-chip, they cover statistic properties to reduce memory footprint. This is done by tracing data at every cycle, retrieving their statistics, and reporting results after processing a whole workload input, e.g., an image. This technique allows users to trace data for a longer period, while still stores information that could be useful to potentially find the root cause of a bug. Although the raw values are not observable using this technique, the information stored can be useful to allow the designer to have a good overall understanding of the circuit in the first few debug iterations, accelerating the debug process.

Figure 4 shows some of the instruments proposed in [12]. The distribution instrument, for example, stores a history of the distribution of an observed matrix over time. The spatial sparsity instrument stores an indicator of whether each particular element in a given matrix is zero or not, based on a predetermined threshold. Finally, the summary statistics instrument summarises one kind of statistics over a single frame, e.g. sparsity, average, or standard deviation.

III. Debug and Tuning Instrumentation

A. Debug of Fused Convolution Blocks

Typically, convolution blocks consume most of the operations in a convolutional neural network [5] and should be well-

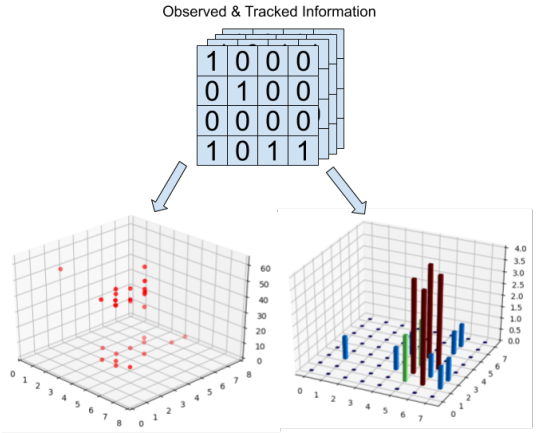

Fig. 5. Overflow map instrument overview.

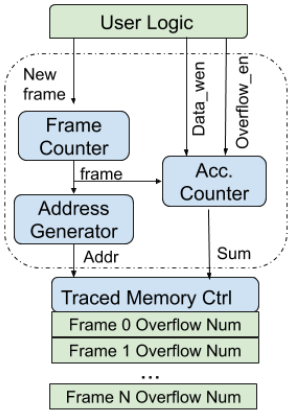

Fig. 6. Overflow statistics instrument architecture.

optimised for performance improvement. Generally, a baseline accelerator for the convolution block is mainly based on layer-by-layer execution which incurs significant external memory access and consequently cannot fully exploit the potential of pipelined CNN layers. To address this issue, the standard convolutions can be fused with a uniform kernel [13, 14], and various convolution types can be fused automatically [5]. However, it makes the debug of the system more complex because the data of some internal layers will not show at the output. Although RTL simulation may help to find the bugs of the fused layers, it requires long run-times. If techniques for capturing raw variable values [15] are used, it would be possible to record all values in an array, and then perform the analysis off-line. However, for large arrays, this may result in very inefficient use of trace buffer memory; every change to every element in the array would consume an entry in the trace buffer. For debugging fused convolution blocks, we propose to use novel debug instruments to monitor overflow. In addition, the distribution instrument [12] can be used to monitor all words in a specified array and to aggregates the values into a histogram per frame, as shown in Figure 4. It can detect outliers or errors causing activations to clamp at minimum/maximum values, which suits our purpose.

B. Overflow of CNNs on FPGA

Multiplications and accumulations in convolutional layers can lead to overflow, which can cause severe computation accuracy loss. Generally, an overflow occurs when an arithmetic operation attempts to create a numerical value that is outside the range that can be represented with a given number of bits. In some FPGA designs [2, 16], the word lengths of intermediate data are extended to avoid overflow. However, if the input and output data have small word lengths, then overflow can occur. Unanticipated arithmetic overflow is a common cause of system errors. Such overflow bugs may be hard to discover and diagnose because they may manifest themselves only after long run-time. Sometimes, the overflow may not make a neural network design failed but may cause accuracy loss, especially when deploying a neural network on an FPGA with small word lengths. The FPGA system works fine but has lower accuracy than the corresponding floating-point model. This type of accuracy loss due to overflow can be hard to debug on FPGAs, since there can be many potential sources of overflow in various layers.

The overflow statistics information from our proposed instruments targeting hardware can quickly detect overflow issues in FPGA systems from thousands of values in a dataset. A high-level fixed-point software model of a hardware design may not capture all the details, while a low-level fixed-point software model is difficult to develop and can be slow to run.

C. Overflow Instruments

A fixed-point computation unit targeting FPGAs can be carefully designed without accuracy loss using large word length intermediate data and calculation. However, the output data need to be down-scaled into small word length, for example, 8-bit and then output to memory. Thus, an overflow may happen in this step. Our debug instruments are designed to track this sort of overflow. The debug instruments do not need to check and trace every addition/multiplication but only the final down-scaling. Thus, there could be only one possible overflow for each output pixel, which means only one bit will be needed to store the overflow status for each output pixel.

In addition, typically the FPGA performs down-scaling in saturation mode, which means the debug instruments do not need to perform comparisons but just track the overflow bit in the original design, which can save FPGA resources.

1) Overflow Map Instrument: This instrument monitors the overflow status of each output pixel. Instead of storing all the values of the outputs data, it stores a Boolean indicating overflow has occurred. This provides information about the overflow status of the output tensor, and a 3D map can be reconstructed from the tracked information (Figure 5).

2) Overflow Statistics Instrument: Tracking and storing one bit for each output pixel can be expensive, since many trace buffers are needed when the output tensor is large. In contrast, this overflow statistics instrument focuses on calculating the summary statistics to assist debugging machine learning circuits. Rather than storing one bit for each output data, this instrument stores summary statistics for each output channel or just stores one summary for a whole output tensor.

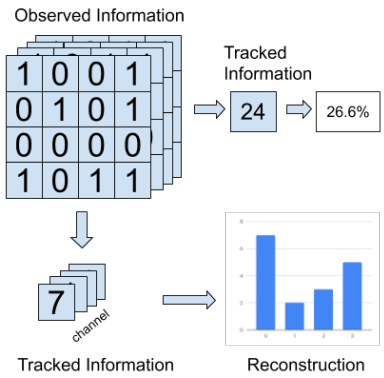

Fig. 7. Overflow statistics instrument overview.

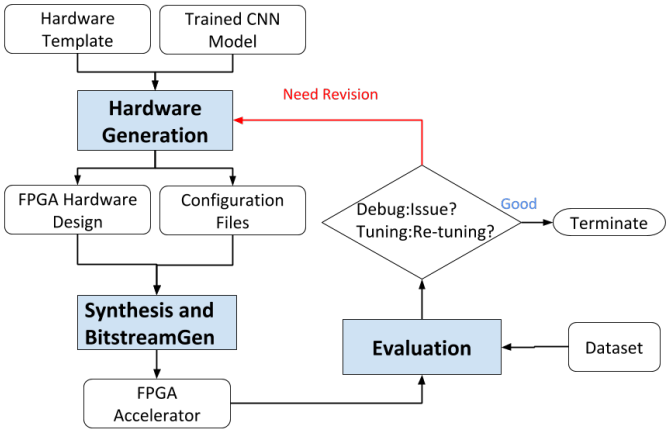

Fig. 8. CNN in-circuit tuning flow.

While the overflow map instrument traces every overflow case, the overflow statistics instrument producing a histogram or a summary does not retain location information of the output pixel causing the overflow. Figure 6 shows the architecture of the overflow statistics instrument, while an overview of its operation is shown in Figure 7. A counter is used to track the number of overflow cases in the output.

The overflow rate defined in Equation 2, based on the overflow information for each output tensor related to CNN layers, can be useful for debugging machine learning applications. In our InTune approach, the overflow rate from hardware based on the overflow statistics instrument can be used to find the best configuration of datapath word length.

$$Overflow\_rate = \frac{Overflow\_Num}{Whole\_Tensor\_Pixel\_Num} \quad (2)$$

IV. In-circuit Tuning

A. InTune: Overview

InTune is a novel approach for in-circuit tuning of domain-specific FPGA designs. It accepts a fresh model or a pre-trained CNN model using a large-scale dataset, replaces the selected standard convolution layers with various convolution blocks (Figure 2 and 3), fine-tunes and evaluates the layer-wise optimised model. After a floating-point based model is generated, it fine-tunes the required word length of each CNN layer first based on a fixed-point software model and then using in-circuit information. At the end, it will output an efficient FPGA design. Figure 8 provides an overview of in-circuit fine-tuning flow using InTune. To efficiently process convolution blocks, InTune generates FPGA designs that fuse their inner convolution layers [5, 13] and inserts different debug instruments for debug and tuning. The trained CNN model includes trained weights and network architecture while the hardware template includes all the necessary hardware components which will be discussed in the next section. After the FPGA processes a given dataset, the debug and tuning instruments provide information for checking if there is any issue in the hardware. In addition, hardware information, such as overflow rate from the tuning instruments will be used to fine-tune the hardware system. If there is any design problem or it requires re-tuning, a new hardware design will be generated using a hardware template. This work focuses on optimising datapath configurations with different word lengths. The tuning strategy for InTune consists of customising the hardware template by systematic word length refinement, compiling the resulting design into a bitstream, and evaluating the effects such as overflow rate to determine whether further tuning is needed. Some bitstreams can be pre-compiled to reduce tuning time.

B. Hardware Template and Layer Fusion

InTune supports a scalable hardware design template as a fundamental component. The template can be configured to generate optimised CNN hardware with various convolution types utilised in recent efficient CNN models. An accelerator for convolution layer or block can be constructed by basic building modules in our template as shown in Figure 9. Most of the arithmetic computations in a typical CNN workload involve dot-product, which is employed in the spatial and cross-channel convolution and also in the fully-connected (FC) layers. Each dot-product module consists of an array of multipliers followed by an adder tree. The dot-product modules are further organised into a higher-level array for parallelisation. This module can be shared among convolution and FC layers when necessary. Each module can be configured regarding the level of parallelism or computation sequence. Modules are connected using data-flow streams with the same input and output width. Outputs from building modules should be consumed immediately to avoid congestion. Our design is by default implemented with fixed-point representation, and its configuration is decided by the data range. In addition, convolution blocks are fused to enable the computation of multi-layers in one launch.

C. Prototype Toolflow

Combining hardware generation with layer fusion, the proposed hardware tuning approach is illustrated in Algorithm 1. This algorithm jointly explores the design space of a CNN model and hardware for efficient inference. Given M is a trained model on a domain-specific dataset D, P is FPGA platform specification while R captures the requirements. This

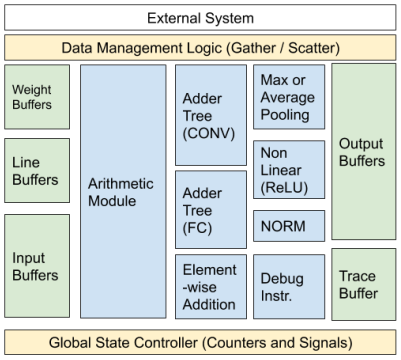

Fig. 9. Our reconfigurable system architecture (Instr. stands for instruments).

Algorithm 1: InTune pseudocode

```
1  Function InTune(M, D, P, R, w_0):
2      h ← InitHardwareGen(M, P, R, w_0);
3      h*, obj*, w ← h, 0, w_0;
4      while OverflowRate(h) ≥ T_ov do
5          acc ← EvaluateAccuracy(h, M, P, D);
6          obj ← ObjectiveFunction(h, acc, w);
7          if obj ≥ obj* then
8              h*, obj* ← h, obj
9          end
10         w ← FineTune(w);
11         h ← HardwareGen(M, P, R, h, acc, w);
12     end
13     return h*, acc*;
14 End Function
```

algorithm is driven by the function $HardwareGen()$ in line 11 which can generate a new hardware design with fine-tuned word lengths from the trained model, the current design, the FPGA platform specification, and profiling information from the current design, such as its overflow rate and model accuracy. We terminate the optimisation process by evaluating the overflow rate of design h (denoted by $OverflowRate(h)$ which is retrieved from the overflow statistics instrument), and checking whether the measured rate is above the threshold T_{ov}. While stepping through the optimisation, we fine-tune the word length w of the hardware design to be generated, and we measure the objective score obj of the current design h by its model accuracy acc and word length w. The objective function should reflect the relative importance of model accuracy and size with regards to user requirements. The design that achieves the highest obj socre will be returned as the best implementation.

V. Results and Discussion

A. Experiment Setup

We evaluate InTune on the CIFAR-10 [17] dataset. A custom VGG-like CNN network including 7 standard convolution layers is initially fed to InTune to confirm the in-circuit tuning capability. We can further integrate this tuning capability with TuRF, a domain-specific adaptation framework

TABLE I
Resource Utilisation

	ALM	BRAM	DSP
Available	262.4k	2567	1963
Used	200.2k	1876	1152
Utilisation	76.3%	73.1%	58.7%

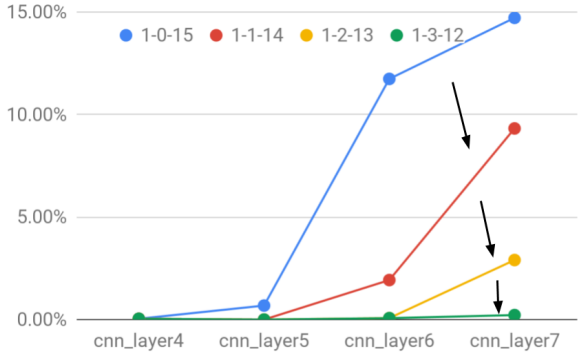

Fig. 10. Overflow rates in $layer4$ to $layer7$. The layers before $layer4$ are not shown as they do not have overflow.

[5]. It works by feeding designs explored by domain-specific adaptations into in-circuit tuning to find the best data type configuration. TensorFlow v1.6 is used to build and train the CNN models. The experimental FPGA platform is Stratix-V 5SGSD8 on a Maxeler MPC-X node, which contains 262.4K adaptive logic modules (ALM), 1963 variable-precision DSP blocks, and 2567 BRAM (M20K). The bandwidth of off-chip data transfer is 38 GB/s. The hardware template prototype is implemented in OpenSPL [18]. MaxCompiler (v2016.1.1) synthesises generated designs. Table I shows the resource utilisation in our design.

B. CNN Accuracy Fine-tuning

Using InTune, we can fine-tune word length configuration for each CNN layer based on hardware information, e.g., overflow rate in output as shown in Figure 10. The accuracy of the current hardware using a fixed-point algorithm (1-0-15 means 1-bit sign, 0-bit integer, and 15-bit fraction) is 71.78%, which shows that there is accuracy loss compared to the counterpart 32-bit floating-point model (Table II). From Figure 10, the blue line capturing the values collected from the overflow statistics instruments shows that the overflow rates of this model after the 4th layer are high. we can optimize this design by fine-tuning the word lengths of CNN layers 5-7 to reduce the overflow rate, reduce the overflow rates, as shown by the red, orange and green lines. The final accuracy of the CNN model increases as shown in Table II. It shows that the InTune approach works because the final accuracy increases to a level that is the same as the floating-point model. Therefore, the debug instrument helps us develop a valid precision fine-tuned neural network system, since accuracy information that used to be hard to obtain can be exploited by our in-circuit tuning method.

TABLE II
ACCURACY BASED ON DIFFERENT WORD LENGTHS

Word Length	float32	16-bit (1-0-15)	16-bit (1-1-14)	16-bit (1-2-13)	16-bit (1-3-12)
Accuracy	72.08%	71.78%	72.05%	72.07%	72.08%

TABLE III
RUNTIME COMPARISON: MODELSIM AND INTUNE

ModelSim	InTune	Speedup
$1615s$	$8.1 * 10^{-5}s$	$2.002 * 10^4$

C. Speedup

We report the speedup provided by InTune over simulation-based methodologies, e.g., hardware/software co-simulation using cycle accurate RTL model. Table III compares the time for executing one input image using ModelSim and our approach to get the debug and profiling information. As shown in Table III, on average InTune achieves a speedup of 20,020 times compared to RTL simulation using ModelSim.

D. Linear Quantization and Block Floating Point

Either linear quantization or block floating point can be used to produce a fixed-point design for CNN model size reduction. The overflow analysis can stay largely the same as what we present earlier, with different model reduction methods. In linear quantization, overflow happens when the quantization parameters (scale, zero-point) are set improperly. So using the overflow statistics instrument can improve quantization parameters by reducing the overflow rate. The overflow statistics instrument has similar benefit for block floating point: add more bits when the overflow rate is high.

E. Comparison with Previous Work

Tuning parameters on a circuit-by-circuit basis can be slow since it is difficult to obtain detailed estimates of candidate optimisation choices. [19] presents a novel hashing mechanism that accelerates the inlining optimisation using a two-level hash structure and quantifies its impact on run-time. [20] proposes to use machine learning to auto-tune the performance and power consumption of FPGA designs. Auto-tuning has also been applied to effectively explore the large, high-dimensional space of tool-specific parameters that control FPGA synthesis [21]. Recently, transfer learning is proposed to optimise parameters for FPGA-based applications [22]. These design approaches cover circuit tuning from a different perspective: they do not involve in-circuit tuning and do not address issues in deep learning designs.

VI. CONCLUSIONS AND FUTURE WORK

This paper proposes InTune, a new approach for in-circuit tuning of domain-specific deep learning designs inspired by transfer learning and in-circuit debugging. The novel aspects of InTune include its debug instruments that investigate overflow map and statistics, and its hardware design optimisation technique that makes use of these instruments to improve the performance of deep learning applications on FPGA. Future work involves covering additional optimisations such as model quantisation based on linear transformation and block floating-point arithmetic, improving instruments to support online tuning of deep learning designs, and evaluating the integration between InTune and TuRF [5].

ACKNOWLEDGEMENT

The support of the United Kingdom EPSRC (grant numbers EP/L016796/1, EP/N031768/1, EP/P010040/1 and EP/L00058X/1), NSERC COHESA, Corerain, Maxeler, Intel and Xilinx is gratefully acknowledged.

REFERENCES

[1] R. Zhao *et al.*, "Optimizing CNN-based object detection algorithms on embedded FPGA platforms," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2017.
[2] H. Fan *et al.*, "A Real-Time Object Detection Accelerator with Compressed SSDLite on FPGA," in *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2018, pp. 14–21.
[3] Y. Bengio, "Deep learning of representations for unsupervised and transfer learning," in *Proceedings of ICML workshop on unsupervised and transfer learning*, 2012, pp. 17–36.
[4] "Fine-Tuning TensorFlow Flowers Dataset." [Online]. Available: https://www.tensorflow.org/tutorials/image retraining#training on flowers
[5] R. Zhao *et al.*, "Towards efficient convolutional neural network for domain-specific applications on FPGA," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE.
[6] A. G. Howard *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.
[7] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1251–1258.
[8] F. Mamalet and C. Garcia, "Simplifying ConvNets for fast learning," in *International Conference on Artificial Neural Networks*. Springer, 2012, pp. 58–65.
[9] K. He *et al.*, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.
[10] M. Sandler *et al.*, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.
[11] M. Abadi *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/
[12] D. Holanda Noronha *et al.*, "On-chip FPGA Debug Instrumentation for Machine Learning Applications," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM.
[13] M. Alwani *et al.*, "Fused-layer CNN accelerators," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 2016.
[14] Q. Xiao *et al.*, "Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on FPGAs," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2017, pp. 1–6.
[15] 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.
[16] K. Guo *et al.*, "Angel-Eye: A complete design flow for mapping CNN onto embedded FPGA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2017.
[17] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," Citeseer, Tech. Rep., 2009.
[18] O. Consortium *et al.*, "Openspl: Revealing the power of spatial computing," *Technical report, Dec; 2013.*
[19] D. H. Noronha *et al.*, "Rapid circuit-specific inlining tuning for FPGA high-level synthesis," in *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE.
[20] A. Mametjanov *et al.*, "Autotuning FPGA design parameters for performance and power," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE.
[21] C. Xu *et al.*, "A parallel bandit-based approach for autotuning FPGA compilation," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM.
[22] M. Kurek *et al.*, "Knowledge transfer in automatic optimisation of reconfigurable designs," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE.