Journal of Systems Architecture 118 (2021) 102198

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

In-circuit tuning of deep learning designs

Zhiqiang Que [c,*], Daniel Holanda Noronha [a], Ruizhe Zhao [c], Xinyu Niu [b], Steven J.E. Wilton [a], Wayne Luk [c]

[a] *University of British Columbia, Canada*
[b] *Corerain Technologies Ltd., China*
[c] *Imperial College London, UK*

ARTICLE INFO

Keywords:
FPGA
EDA
Deep learning
In-circuit tuning
Reconfigurable computing

ABSTRACT

This paper presents OTune, a novel overlay-based approach for rapid in-circuit debugging and tuning of Deep Neural Network (DNN) designs targeting Field-Programmable Gate Array (FPGA). We first propose overlay-based instruments that provide hardware profiling information to FPGA-based DNN developers for tuning and debugging their designs. Our instrumentation is optimized to take advantage of characteristics of the DNN application domain and traces useful information for in-circuit domain-specific development. Besides, a light-weight overlay-based DNN processing engine is implemented to support rapid word length tuning, which allows adjusting each DNN layer's datapath without time-consuming FPGA compilation. Furthermore, our approach enables tuning of FPGA-based DNN designs for edge systems, which would benefit developing adaptive learning systems. Evaluation results show that OTune can tune a fixed-point design to the same accuracy as a floating-point one with less than 4% added FPGA area.

1. Introduction

The emerging edge computing in recent years has seen successful development in various fields given its potential in reducing latency, protecting privacy and saving energy. Edge computing supports data processing at the edge of a network, which is close to data sources allowing reduction in data transmission. Although GPU solutions are widely adopted in the cloud for DNN training and inference, they may not be suitable for edge data processing due to their high power consumption and cost. Besides, edge nodes should be able to serve multiple DNN computation requests at a time, which makes them impractical to target CPUs and GPUs [1,2]. In contrast, Field Programmable Gate Array (FPGA) technology has shown promise in implementing DNNs for edge nodes. Over the last decade, FPGAs have gained increasing popularity in DNN deployment [3–5].

Nevertheless, it is challenging to implement such FPGA accelerator designs, especially when in-circuit debugging and tuning are involved. Software simulators can help addressing this challenge, but they have major limitations: software simulation is insufficient to find the root cause of many functional and performance bugs, largely due to the fact that some hardware behaviors cannot be correctly or rapidly simulated, e.g., non-deterministic DRAM accesses; running DNN workloads that normally consist of millions of images on software simulators is time consuming; and some third-party hardware libraries may not have accurate simulation models. Therefore, tuning and debugging of DNN acceleration cannot be supported easily by software simulation. The only method to find the cause of some types of function and/or performance bugs is to run the hardware in a realistic environment at the actual speed with representative workloads.

Besides, there is a significant gap between the software-oriented representation of DNN models and their actual form of hardware execution on FPGAs. DNN models are often trained and fine-tuned on CPUs and/or GPUs using Python, C++, or C, and they are often manually converted to hardware descriptions. This conversion process is error-prone and time-consuming, and it may take several rounds until DNN models are finally well tuned, which increases the cost to deploy an optimal system on FPGAs. Therefore, it is preferable to build the hardware once and carry out further tuning on that compiled version, rather than converting a software-based DNN model to hardware every time it needs tuning. However, it is still challenging to develop such a hardware system that is able to fine-tune itself based on the hardware profiling information. Furthermore, the resources of edge nodes are often limited. Hence it is preferable for edge nodes to be engaged in DNN inference rather than DNN training.

This paper addresses these limitations by proposing OTune, a novel overlay-based rapid in-circuit tuning and debugging framework which

* Corresponding author.
E-mail addresses: z.que@imperial.ac.uk (Z. Que), danielhn@ece.ubc.ca (D.H. Noronha), ruizhe.zhao15@imperial.ac.uk (R. Zhao), xinyu.niu@corerain.com (X. Niu), stevew@ece.ubc.ca (S.J.E. Wilton), w.luk@imperial.ac.uk (W. Luk).

https://doi.org/10.1016/j.sysarc.2021.102198
Received 31 December 2020; Received in revised form 26 April 2021; Accepted 31 May 2021
Available online 4 June 2021

allows users to adjust the word length to find the most suitable precision for a trained DNN model without re-compilation. Besides, instead of retraining the deployed hardware models when accuracy decreases, it fine tunes the deployed hardware models for various inputs to get better accuracy for edge systems. The system of OTune has the following components: a light-weight (less than 4% added area in our current implementation) overlay-based DNN processing engine that allows running network layers in various levels of precision without recompilation; an overlay-based overflow instrument that provides in-circuit overflow information for tuning and debugging; and a prototype toolflow that rapidly explores the design space of word length through a novel tuning algorithm. Different from normal word-length optimization that aims to find the minimum number of bits to support a given accuracy, our tuning algorithm optimizes the representations within a fixed budget of bits to maximize accuracy, which is more hardware friendly and efficient.

Edge nodes would need to cover various deep learning functions, and in-circuit tuning would allow their optimization without time-consuming re-compilation. In-circuit debugging and tuning can take place at design time on a large FPGA, and the final version without the related instruments can then target a small FPGA to optimize for speed, energy consumption and cost at run time. For applications that allow run-time monitoring, the edge nodes can send monitored data to server nodes which can provide updated configurations, possibly based on in-circuit tuning, to the edge nodes to adapt to run-time conditions.

To the best of our knowledge, this is the first overlay-based in-circuit tuning approach that can optimize a DNN-based FPGA design.

We make the following contributions in this paper:

(1) A novel overlay-based overflow instrument for in-circuit tuning and debugging.
(2) A novel light-weight overlay-based processing engine for running various DNN layers using mixed precision which can be changed without recompilation.
(3) A framework for rapid in-circuit accuracy tuning of DNN designs using mixed precision for different layers based on a novel tuning algorithm.

This paper extends our previous work described in a conference paper [6] and in an extended abstract [7]. The in-circuit tuning framework, InTune [6], aims to address the aforementioned challenges by exploiting hardware profiling information. While searching for an optimal design, a user may wish to try out various configurations to observe how the results evolve with the user's understanding of how the circuit operates. An important limitation of InTune is that, every time a user proposes a new tuning configuration, the circuit concerned will be modified and recompiled. Recompilation is slow, and this limits tuning productivity. Meanwhile, InTune tunes DNNs under the constraint of identical word length across all layers. Unlike the instruments in [6], the instrument proposed in this work is based on an overlay which can be reconfigured without re-compilation. Besides, a light-weight spatial processing engine is proposed to support CNN layers with various data representations within a fixed word length. Combining the overlay-based instruments and the flexible hardware engine, our approach can optimize an FPGA design to achieve high accuracy for DNNs.

2. Background and preliminaries

2.1. Convolution neural networks (CNNs)

A typical CNN consists of N layers including convolution, ReLU, pooling, and Fully-Connected (FC) layers, where N denotes the total number of layers.

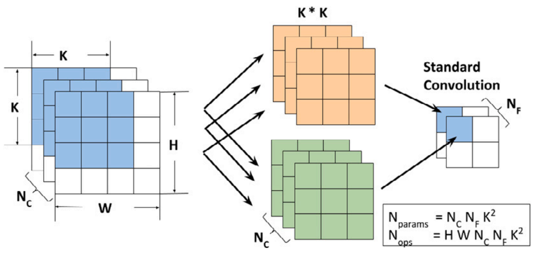

Fig. 1. Standard convolution layers.

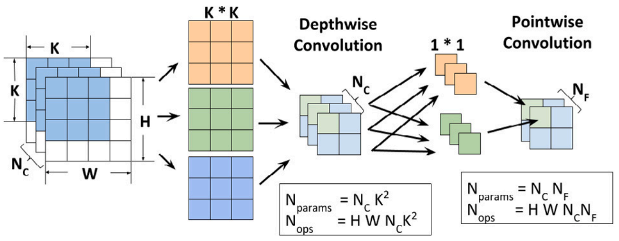

Fig. 2. Depthwise and Pointwise convolution layers.

2.1.1. Convolution layers

Convolution layers perform multi-dimensional convolution computation between an input feature map and a filter, as shown in Fig. 1. They extract features from an input feature map and generate 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 standard convolution layer f is defined as:

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

Generally, weights in convolution layers are called *convolutional kernels*. Convolutional layers have hyper-parameters such as kernel width (k_x, k_y), number of filters N_f, *stride* and *dilation* factors.

Besides standard convolution, there are other types of convolution. Depthwise convolution [8–10] and pointwise convolution [8] are both lightweight building blocks of modern CNNs, as shown in Fig. 2. Compared with standard convolution, depthwise convolution only applies one filter on each channel, which significantly decreases the amount of computation and parameters and is relatively efficient. However, it only filters input channels and does not combine them to create new features. So a pointwise convolution which has kernel size of 1×1, is used to generate new features via a linear combination of the output of the depthwise layers.

2.1.2. Fully-Connected (FC) layers

FC layers are an affine transformation of the input feature vector. It contains a single matrix–vector multiplication followed by a bias offset.

2.1.3. Rectified Linear Unit (ReLU)

Relu is one of the many non-linear activation functions which bring non-linearity to the CNNs. To acquire non-linearity, CNN usually contains activation layers. It computes an activated value $f(x)$ as:

$$f(x_{ij}) = ReLU(x_{ij}) = max(0, x_{ij})$$

2.1.4. Max-pooling

The forward 2D max-pooling layer performs a non-linear sub-sampling approach that takes only a summary statistic of the nearby region, such as the maximum value of each small region, in the input feature map. These regions can be constructed by performing sliding window operations on the 2D input feature map. This layer can significantly reduce the dimensions of feature maps and enhance the translation-invariance property of CNNs.

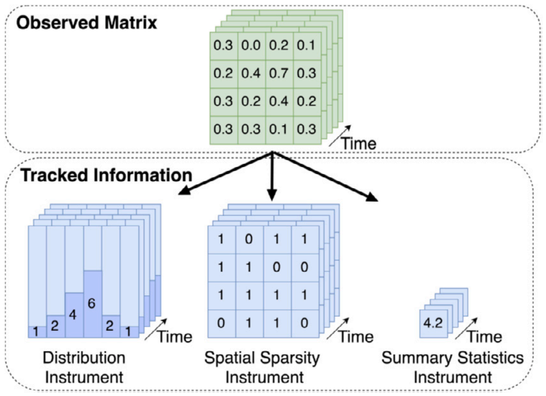

Fig. 3. Instruments overview [6].

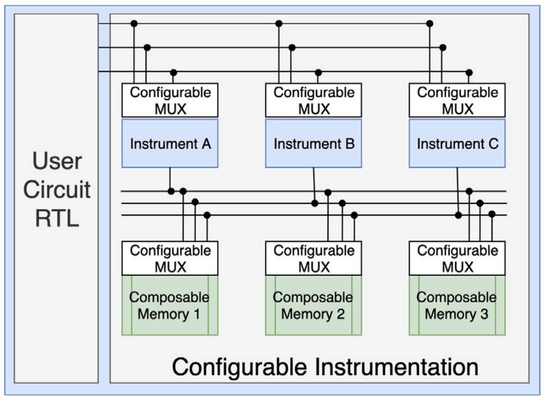

Fig. 4. Overlay-based Instrumentation Architecture [13].

2.2. *Domain-specific debug instrumentation*

When compared to circuits from other domains, machine learning circuits can be substantially more difficult to debug. The CNN circuits can require very long run times (such as computing multiple training or inference samples) before their overall behavior can be understood. Bugs can also be partially or completely masked due to the resilience of neural networks. Moreover, applications are increasingly compiled from high-level programming languages and frameworks, such as C and TensorFlow [11], which means that having visibility in terms of the automatically generated hardware may not be as useful.

A debug toolflow [12] addresses those problems by using domain-specific characteristics of machine learning circuits to store useful information on-chip. Similar to common hardware-oriented debug flows, debug instrumentation is added to the design in order to store a history of how signals change over time for later interrogation. In order to make efficient use of the trace buffer space, different statistics about these large matrices are recorded [12] instead of recording their raw values on-chip. Three types of instruments are proposed [12]. The distribution instrument 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 summarizes one kind of statistics over a single frame, e.g., sparsity, average, or standard deviation.

Unlike the previous approach, it has been shown that the instrumentation circuitry can be reconfigured at debug-time without recompilation [13], as shown in Fig. 4. After the circuit has been compiled, but before it is run, the instrumentation can be configured in the three ways described below:

2.2.1. *Selective matrix tracing*

This allows to select, at debug-time, which matrices should be traced, given that these matrices have been instrumented at compile-time.

2.2.2. *Selective compression*

The selective compression refers to the capability to choose the way in which data will be compressed (which statistics will be computed) at debug-time.

2.2.3. *Flexible trace buffer*

Instead of using a centralized memory as a trace buffer, multiple memories can be dynamically combined to act as a single trace buffer. This enables the user to store different signals for different amounts of time. This can be used in tracing a signal that gives context of the execution for long periods of time (e.g., the loss during training), while storing less critical signals for a shorter time.

After the instrumentation has been configured (by writing the configuration through the JTAG port), the circuit is then run, and the instrumentation records information regarding each identified matrix into the trace buffer. As users refine their understanding of the behavior of the circuit, they may wish to modify the debug scenario. This iterative process does not require recompilation and it is repeated until the root cause of the bug is determined.

3. Debug and tuning instrumentation

3.1. *Debug of fused convolution blocks*

Typically, convolution blocks consume most of the operations in a convolutional neural network [5] and should be well-optimized for performance improvement. Generally, a baseline accelerator for the convolution block is mainly based on layer-by-layer execution which involves 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 [14, 15], and various convolution types can be fused automatically [5]. However, it makes the debug of the system difficult 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 [16] 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 aggregate the values into a histogram per frame, as shown in Fig. 3. It can detect outliers or errors causing activations to “clamp” at minimum/maximum values, which suits our purpose.

3.2. *Overflow of CNNs on FPGA*

Fixed-point operations, which are common in FPGA designs, can produce results with fewer digits than the inputs. Thus, information loss is possible. For example, the result of a fixed-point multiplication could potentially have as many digits as the sum of the number of digits in the two input operands. If the word-length of the result is less than the number of this sum, then this result needs to be truncated, rounded and/or saturated. Fractional digits lost below this value represent a precision loss which is common in fractional multiplication. However, if any valid integer digits are lost, the value will be radically inaccurate. Unanticipated arithmetic overflow is a common cause of system errors. Thus, this work focuses on overflow issues. In convolutional

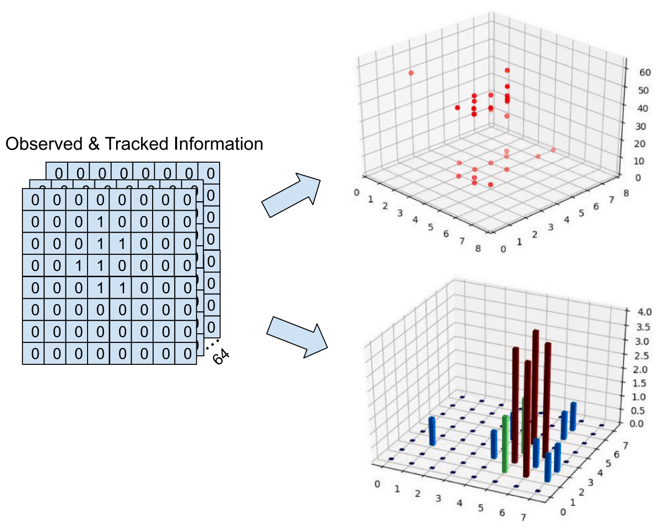

Fig. 5. Overflow map instrument overview.

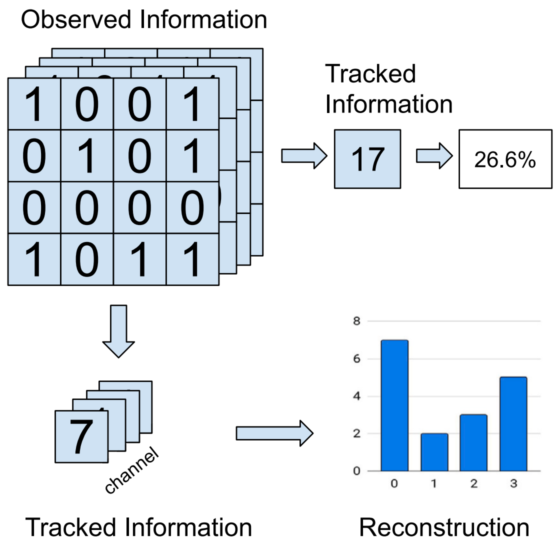

Fig. 6. Overflow statistics instrument overview.

layers, multiplications and accumulations are common and may lead to overflow, which can cause severe computational accuracy loss.

In many FPGA-based DNN designs [3,17,18], the word lengths of the intermediate data are extended to avoid potential overflow. However, if the final output data have a small word length, then overflow can still occur. Such overflow bugs are hard to diagnose since they may manifest themselves only after long run-time when, for example, errors begin to accumulate. Sometimes, a neural network design may not fail due to overflow but only suffers loss of accuracy, especially when it is deployed on an FPGA using small word lengths. The FPGA system may work fine but has lower accuracy than the floating-point counterpart. This type of accuracy loss due to overflow can be difficult to debug on FPGAs, since there can be many potential sources of overflow accumulated 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.

3.3. The novel 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 even do not need to perform comparisons but just track the overflow bit in the original design, which can save FPGA resources.

3.3.1. Overflow map instrument

This instrument monitors the overflow status of each output pixel. Instead of storing all the values of the output 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 as shown in Fig. 5.

3.3.2. 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 and tuning 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. 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. An overview of its operation is shown in Fig. 6. A counter is used to track the number of overflow cases in the output.

The overflow rate defined in Eq. (1), based on the overflow information for each output tensor related to CNN layers, can be useful for debugging and tuning machine learning applications. In our OTune approach, the overflow rate from hardware based on the instruments is used to find the best configuration of datapath word length.

$$Overflow\_rate = \frac{Overflow\_Num}{Whole\_Tensor\_Pixel\_Num} \tag{1}$$

3.4. Overlay-based overflow instruments

One limitation in our former work [6] is that every time the word length of CNN layers needs modification, or every time a new overflow instrument is to be used, the new user logic with the instrumentation needs to be run through the entire FPGA compilation flow, including synthesis, place and routing. This may take hours and significantly limits debug and tuning productivity. Another limitation is that it only supports tuning all the CNN layers using the same word length.

To address these limitations, we propose an overlay-based instrument as shown in Fig. 7. Unlike the instruments described in Section 3.3, this instrument is based on an overlay which can be reconfigured without recompilation. After the user logic has been compiled, but before it is run, this instrument can be configured either as an overflow map instrument or an overflow statistics instrument. When compared to the debugging instruments provided by previous overlay-based DNN debugging system [13], the proposed overlay-based instrument is not only used for debugging but also for tuning. In other words, this work provides controllability (e.g. it is actually changing the circuit) rather than just observability.

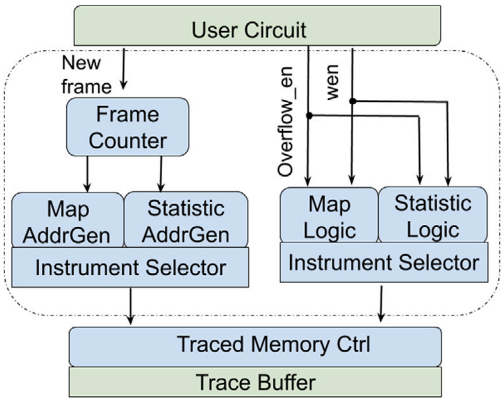

Fig. 7. Overlay-based overflow instrument overview.

3.5. Mixed precision CNNs

Mixed-precision has been used in numerical computations for decades. Generally, there are two methods to support mixed precision CNNs. The first is a temporal architecture. A state-of-the-art temporal design of DNNs accelerator using FPGA is the Bit-Serial Matrix Multiplication Overlay (BISMO) [19]. The proposed bit-serial multipliers in the overlay are fed with one-bit digits from 256 weights and the corresponding activations in parallel at one time. Their partial products are accumulated by shifting over time. The second architecture is a spatial one. The BitFusion architecture [20] is a classic spatial design of a DNN accelerator in which a 2D systolic array of fusion units are employed. These units spatially sum the shifted partial products of two-bit elements from weights and activations.

This work proposes a light-weight spatial architecture which supports CNN layers with various data representations within a fixed word length. The data format $Qx.y$ is used in our design. It consists of one sign bit, x integer bits and y fractional bits. Although there is much previous work using fewer bits for both weights and activations (e.g., between 1 and 8 bits), 16-bit is still a frequently used word length [21]. Thus, we choose the total word-length as 16 bits for the datapath of our design with the values of x and y constrained by the equation $1 + x + y = 16$. One limitation is that when the datapath needs an additional integer bit, it will need to lose a fractional bit. Losing a fractional digit may introduce a precision loss, but losing any valid integer digit will incur large inaccuracy. Different from normal word-length optimization that aims to find the minimum number of bits to support a given accuracy, our tuning algorithm optimizes the representations within a fixed budget of bits to maximize accuracy, which is simpler since the total number of bits do not change.

There are many benefits of this light-weight spatial architecture. First, the FPGA DSP blocks can be utilized efficiently since the multipliers are still 16 × 16 so that the result precision can be kept as much as possible. Second, it can reduce the design cost. Only the output cast unit needs redesigned by inserting a barrel shifter while all the other components, such as the multipliers or accumulators, can remain untouched. Thus, the main datapath can be almost the same as the non-overlay datapath, which reduces the design cost to convert a non-overlay datapath to an overlay-based one to support mixed-precision for different CNN layers.

4. Implementation

4.1. Hardware architecture

Based on the optimization techniques presented above, a light-weight mixed-precision spatial architecture of a CNN is illustrated in Fig. 8. To improve the system scalability, we propose to use the single processing engine architecture [22–24] where a computational engine is designed to run one block or layer at a time, and the whole network is processed by running the computational engine repeatedly, as shown in Fig. 8 (left). It mainly consists of the computational engine, data stream controller, light-weight CPU, on-chip and off-chip memory. In the computational engine, there are several buffers, the Processing Elements Unit, max-pooling and ReLU modules. Each Processing Element (PE) consists of several multipliers and a pipelined adder tree, which is used to perform the multiply-accumulation in convolutional layers. The data stream controller, which consists of several buffers, is dedicated to the data communication between the computational engine and on-chip memory. The data and weights/bias buffers, as shown in Fig. 8 (middle), are used to cache the input/output pixels and weights/bias required by the Processing Elements unit which conducts convolutional operations. The cast unit receives the convolutional results and down-scales the large word length to 16-bit ones. One-bit overflow status is also produced. Since max pooling and ReLU only have the operation of comparison and selection, we can down-scale the engine results before these modules to save FPGA resources. A fixed-point computation unit targeting FPGAs can be carefully designed without accuracy loss using large word length intermediate data and calculation. To keep the accuracy, the result of the multipliers is 32-bit while the accumulator word length is 46-bit to avoid overflow, as shown in Fig. 8 (top right). However, the output data need to be down-scaled into small word

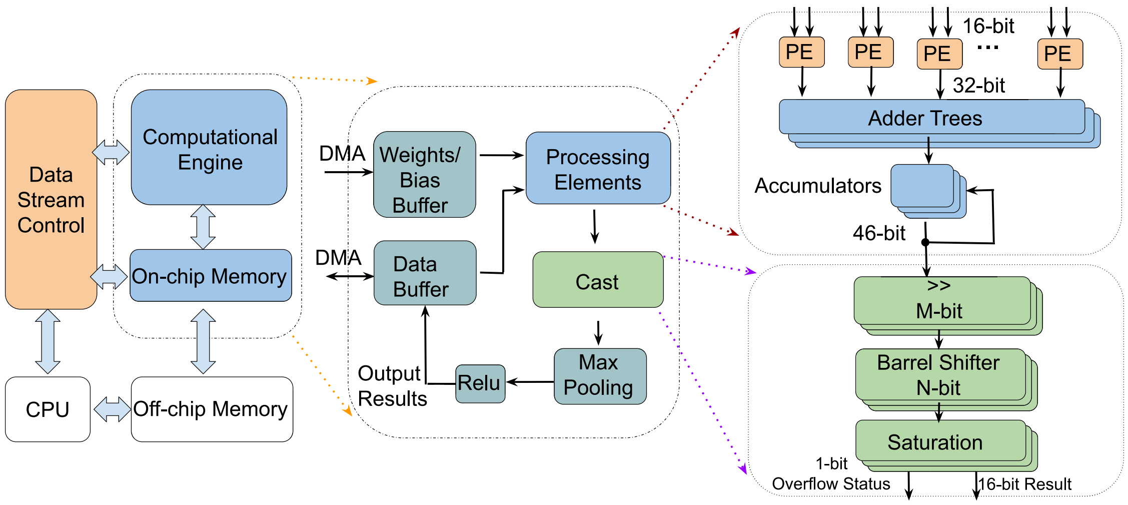

Fig. 8. The light-weight mixed-precision spatial architecture.

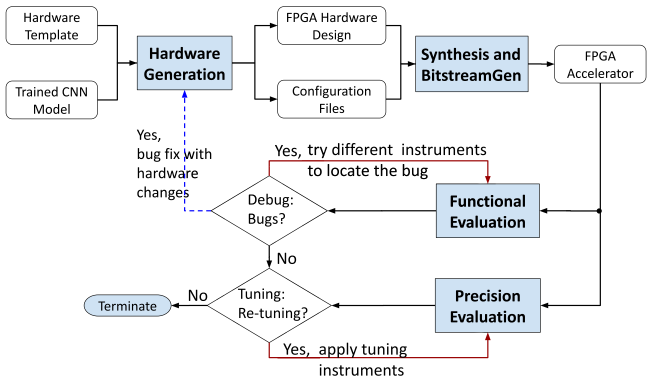

Fig. 9. CNN in-circuit debug and tuning flow.

length, for example, 16-bit and then output to memory. When the input data format is $Qx.y$, the fractional digits of the results of the multipliers and accumulators are $2y$. These results need to be right shifted by y bits using a barrel shifter and then they enter the saturation sub-unit to be down-scaled into 16-bit as the required output.

To support various levels of precision and minimize the resources used, a two-stage custom shifter is proposed in the cast unit. Barrel shifters commonly consume a significant amount of hardware resources. They can shift a data word by a varying number of bits typically within a single clock cycle using only pure combinational logic. Generally, a barrel shifter is split into n stages. Stage s can shift the operand by 2^s bits or leave it unmodified. The number of parallel 2-to-1 multiplexers required for an n-bit barrel shift is $n \log_2 n$ [25]. The proposed two-stage custom barrel shifter consists of a M-bit constant shifter followed by a N-bit barrel shifter as shown in Fig. 8 (bottom right). The constant shifter right shifts the data by a constant number of bits, such as M bits, while the barrel shifter shifts the data by 0 up to $(N-1)$ bits. Thus, this custom shifter can shift the data by M bits up to $(M+N-1)$ bits. Instead of using an $(M+N)$-bit barrel shifter, our custom shifter can significantly reduce the FPGA resource use since the $(M+N)$-bit barrel shifter needs $(M+N)\log_2(M+N)$ multiplexers while our custom one only needs $(M+N)\log_2(N)$ multiplexers. For instance, when $M = N = 8$, our custom shifter can save 25% FPGA resource of a barrel shifter.

After the shifter, the result is down-scaled in saturation mode where the overflow is checked. Thus, our tuning and debugging instruments do not need to calculate and trace every addition/multiplication but only the final down-scaling. There could only be 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, since typically the FPGA performs down-scaling in saturation mode, the tuning and debugging instruments do not need to perform comparisons but just record the overflow bit in the original design, which can save FPGA resources.

4.2. Prototype toolflow of otune

Fig. 9 provides an overview of the in-circuit fine-tuning flow using OTune. First, an FPGA design with flexible instruments is generated. To efficiently process convolution blocks, OTune generates FPGA designs with a light-weight overlay-based processing engine and inserts the flexible instruments for debugging and tuning.

Once the FPGA accelerator processes a given dataset, functional bugs will first be detected by the debugging instrument. When the design is functionally correct, then tuning can begin. The data precision is tuned based on the overflow rate provided by the tuning instrument. One major benefit of using OTune is its support for data precision tuning without design regeneration and recompilation. Because of this

Algorithm 1: OTune pseudocode.

```
1  Function OTune($M, D, P, R, wl_0$):
2      $M$ denotes the model trained on a domain-specific dataset $D$,
       $P$ specifies an FPGA platform, and $R$ is the requirements on
       the design;
3      $wl$ is a set of word lengths configured for each layer, and $wl_0$
       gives the initial values;
4      $hw \leftarrow OverlayHardwareGen(M, P, R, wl_0)$;
5      $of \leftarrow OverflowRate(hw, wl_0)$;
6      $obj^*, wl^*, wl \leftarrow 0, wl_0, wl_0$;
7      while $of \geq T_{ov}$ do
8          if the overflow rate not less than a given threshold $T_{ov}$,
           the exploration continues;
9          $acc \leftarrow EvaluateAccuracy(hw, wl, M, P, D)$;
10         the objective function measures the goodness of the
           hardware based on its task accuracy $acc$ and $wl$;
11         $obj \leftarrow ObjectiveFunction(hw, wl, acc)$;
12         if $obj \geq obj^*$ then
13             $obj^*, wl^* \leftarrow obj, wl$
14         end
15         $of \leftarrow OverflowRate(hw, wl)$;
16         and we will decide how the word length should be
           updated by $of$ and $acc$;
17         $wl \leftarrow FineTune(wl, of, acc)$;
18     end
19     the word length setting $wl^*$ that gives the best objective
       score will be returned;
20     return $hw, wl^*, acc^*$;
21 End Function
```

feature, exploring the vast design space characterized by data precision, which has not been feasible [6], is now a viable objective.

Algorithm 1 illustrates how OTune explores data precision. It does not cover the debugging step. The tuning strategy for OTune consists of customizing the hardware template by systematic word length refinement and evaluating the effects such as in-circuit overflow rate to determine whether further tuning is needed. This algorithm is driven by the function $FineTune()$, as shown in blue in line 17 which can fine-tune the word lengths from the profiling information of the current design, such as its overflow rate and model accuracy. The search space is characterized by wl, a set of word length values configured for each layer. A change in wl affects the model accuracy. We devise an objective function that measures the goodness of a wl configuration based on those affected factors in a balanced manner. Initially, wl is configured by a set of lowest possible values. In each optimization step, we fine-tune the position of the binary point of wl if the overflow rate is high or the model accuracy is poor. This optimization terminates when the overflow rate is below a given threshold T_{ov}. During optimization, we keep track of the best objective function score and the corresponding wl value, and the overall best wl^* will be returned in the end.

Compared to our former work [6], the function $OverlayHardware Gen()$, as shown in red in Algorithm 1 , occurs only once outside the while loop. Within the while loop, there is only $FineTune()$, as shown in blue. Please note that this tuning is different from word-length optimization, which aims to find the minimum number of bits to support a given accuracy. Here our tuning is to optimize the representations within a fixed word length to maximize accuracy.

5. Results and discussion

5.1. Experiment setup

To find out the performance and limitations of the proposed overlay-based in-circuit tuning system, we implement the hardware system

Table 1
Architecture of the VGG-7 CNN used in this paper.

Layer	Input Image Size	Num Filters	Kernel Size	Act.Fun.
Conv1–1	$32 \times 32 \times 3$	32	3	ReLU
Conv1–2	$32 \times 32 \times 32$	32	3	ReLU
Maxpool	$32 \times 32 \times 32$	32	2	–
Conv2–1	$16 \times 16 \times 32$	64	3	ReLU
Conv2–2	$16 \times 16 \times 64$	64	3	ReLU
Maxpool	$16 \times 16 \times 64$	64	2	–
Conv3–1	$8 \times 8 \times 64$	128	3	ReLU
Conv3–2	$8 \times 8 \times 128$	128	3	ReLU
Maxpool	$8 \times 8 \times 128$	128	2	–
Dense	2048	128	–	ReLU
Dense	128	10	–	Softmax

Table 2
Resource utilization.

	LUT	FF	BRAM	DSP
Avail.	218600	437200	545	900
Used	54032	46007	301	493
Utili.	24.72%	10.52%	55.23%	54.78%

Table 3
Overhead of overlays.

	LUT	FF	LUT(‰)	FF(‰)
FPGA total resource	437200	218600	–	–
Overlay-based datapath	6791	1100	31.1	2.5
Overflow map instrument	14	91	< 0.1	0.2
Overflow statistics instrument	269	1132	1.2	2.6
Overlay-based overflow instrument	334	1192	1.5	2.7

using Vivado HLS. A VGG-7 CNN model [26–28], as shown in Table 1, is designed to demonstrate our framework with the CIFAR 10 dataset. Although the model demonstrated in this work is small, our OTune framework can be applied to other CNN variants since we focus on utilizing the hardware profiling information to fine tune the designs. The target platform is Xilinx ZC706, which consists of a XC7Z045 FPGA and dual ARM Cortex-A9 processor. 1 GB DDR3 RAM is installed on the platform as the off-chip memory. The DMA word length is 128-bit. Vivado 2018.3 is used for synthesis and implementation. The weights and bias are pre-processed off-line and stored in the off-chip memory.

5.2. Resource utilization and overhead of overlays

Table 2 shows the resource utilization of our design without the overlay-based datapath and overflow instruments. Table 3 shows the resource utilization overhead in our design. This table shows that the proposed instrumentation requires a small amount of resources, even when the overlay-based instrument is included. Compared to [13], our instrument requires fewer FPGA resources. The overhead of the overlay-based processing engine is significant, which costs about 3.11% of the total LUTs on this FPGA. However, once the design is debugged and tuned, one can then produce a final version of the design without the overlay and the instruments, so that the FPGA implementation on the edge nodes would not suffer from any overhead.

5.3. CNN accuracy fine-tuning with mixed precision

Using OTune, we can fine-tune the word length configuration for each convolution layer based on hardware information, e.g., overflow rate in output as shown in Fig. 10. The details of the layers Conv1-1 ~ Conv3-2 are shown in Table 1 while the word length configurations for each convolution layer in each checkpoint are shown in Table 4. The accuracy of the 32-bit floating-point model is 83.21%. As shown

Table 4
Checkpoints of mixed precision tuning of a CNN model.

Layer	Checkpoint1	Checkpoint2	Checkpoint3	Checkpoint4
Conv1–1	Q0.15	Q1.14	Q1.14	Q1.14
Conv1–2	Q0.15	Q1.14	Q1.14	Q1.14
Conv2–1	Q0.15	Q1.14	Q2.13	Q2.13
Conv2–2	Q0.15	Q1.14	Q2.13	Q3.12
Conv3–1	Q0.15	Q1.14	Q2.13	Q3.12
Conv3–2	Q0.15	Q1.14	Q2.13	Q3.12

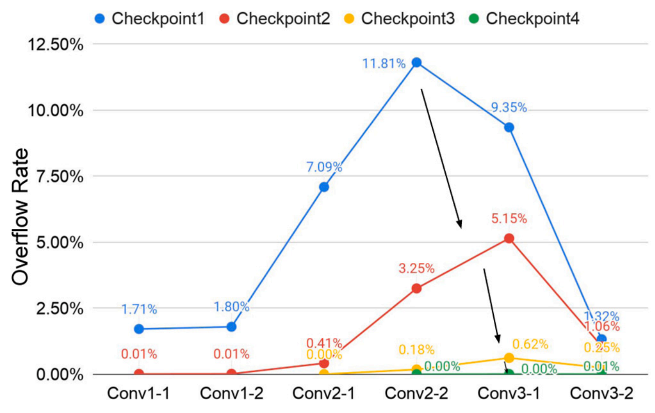

Fig. 10. Overflow rates of each CNN layer in various checkpoints.

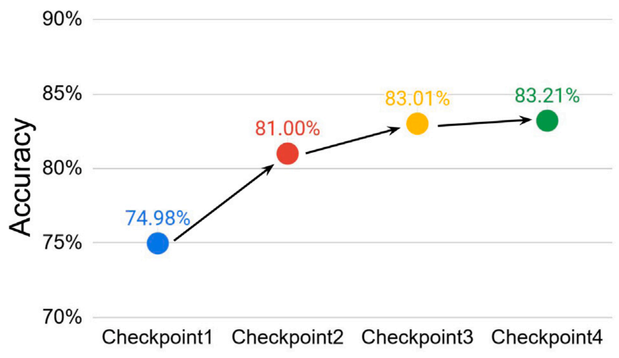

Fig. 11. CNN model accuracy tuning.

in Fig. 11, the accuracy of the hardware using the fixed-point format Q0.15 (shown in blue) is 74.98% which is much lower than the accuracy of the 32-bit floating-point model. Fig. 10 shows that the blue lines capturing the values collected from the overflow statistics instruments indicate that the overflow rates of this model in each layer at this checkpoint are high. Using OTune, the word lengths of each layer in this CNN model can be fine-tuned to reduce the overflow rate, as shown by the red, orange and green lines. Our approach improves the accuracy of the CNN model by 8.23% as shown in Fig. 11. It shows that OTune improves the design to have the same accuracy as the floating-point one. While this work focuses on instrumenting and using overflow information of CNN hardware, the proposed approach can be applied to fine tune systems using some other instruments.

5.4. Speedup

We report the speedup provided by OTune over simulation-based methodologies, e.g., hardware/software co-simulation using cycle accurate RTL model. We compare the time for executing one input image using ModelSim and our approach on a Zynq 7045 FPGA to get the debug and profiling information. On average OTune achieves a speedup of 1.01×10^6 times compared to RTL simulation using ModelSim. The comparison against a ModelSim simulation of the hardware design is reasonable since only the RTL model has all the details of the real hardware system. While a high-level software model of a hardware design may have faster simulation than a low-level one, it may not be able to capture all the details of the hardware and its environment.

When compared to InTune [6], it is estimated that a designer would make about 5 re-compilations for this case study. OTune removes these re-compilations. If an average re-compilation takes one hour, then OTune can save 5 hours.

5.5. 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 optimization choices. [29] presents a novel hashing mechanism that enables rapid evaluation of different inline expansion decisions for HLS design generation. DNNTune [30] can find the optimal mobile-cloud coordinated deployment strategy across a variety of software frameworks. [31] proposes using 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 [32].

Fune [33] and Zac [34] enable fine-grained customization for CNNs by pre-building multiple designs beforehand and dynamically reconfiguring FPGA during run-time, which is beneficial only when the pre-built options are in a small known set, and reconfiguration will not be triggered frequently; otherwise, the effort of pre-building will be enormous and the dynamic reconfiguration overhead will not be negligible. Even with Xilinx Integrated Logic Analyzer (ILA), time-consuming recompilation is required, which makes OTune promising. In addition, the optimal configurations of the CNN engine for running the various CNN layers may be different, which can introduce frequent dynamic configuration of FPGAs to run even a single multi-layer CNN model and bring long latency.

The Hardware-aware Automated Quantization (HAQ) proposed in [35] leverages reinforcement learning (RL) to automatically determine the quantization policy. It takes the feedback from a high-level hardware simulator to an RL agent. Nevertheless, as previously mentioned (Section 1), the hardware simulator may not capture all the details of hardware systems. Besides, it does not support debugging which involves instrumenting the design to increase observability.

Nevertheless, this RL-based HAQ technique can be complementary to our instrumentation-based approach to explore data precision in the tuning process. Moreover, the approaches mentioned earlier cover design tuning from a different perspective: they do not involve in-circuit tuning and do not address issues in debugging deep learning designs. Furthermore, in simulation, it is difficult to obtain accurate performance information of DDR memories, the interaction among layers, and some third-party hardware libraries. Therefore, tuning and debugging of DNN acceleration cannot be accomplished easily by software simulation.

When compared with a previous overlay-based DNN debugging system [13], one of the key differences between [13] and this work is that we are also providing controllability (e.g. we are actually changing the circuit and tuning it into an optimal one) rather than just providing observability. [36] also provides controllability by enabling rapid functional changes through an overlay for debugging. Both [13,36] only focus on debugging. However, our proposed approach can cover both debugging and tuning of CNN designs.

6. Conclusions and future work

This paper presents OTune, a new approach which facilitates in-circuit fine-tuning of fixed-point representation of weights in a CNN. The novel aspects of OTune include the overlay-based instruments and the light-weight overlay-based DNN processing engine. A new tuning algorithm is also proposed, which has shown promise in our experiments. Future work includes extending OTune to cover variable word length, quantifying the benefits and the trade-offs of the proposed approach for specific applications, exploring additional tuning techniques such as those for block floating point representations, and studying the use of our instruments for run-time tuning to support, for example, adaptive federated learning in resource constrained edge computing systems [37].

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

The support of the United Kingdom EPSRC (grant numbers EP/L016796/1, EP/N031768/1, EP/P010040/1, and EP/S030069/1), Corerain, Xilinx and Intel is gratefully acknowledged.

References

[1] X. Wang, Y. Han, V.C. Leung, D. Niyato, X. Yan, X. Chen, Convergence of edge computing and deep learning: A comprehensive survey, IEEE Commun. Surv. Tutor. 22 (2) (2020) 869–904.
[2] J. Chen, X. Ran, Deep learning with edge computing: A review, Proc. IEEE (2019).
[3] 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.
[4] Z. Que, T. Nugent, S. Liu, L. Tian, X. Niu, Y. Zhu, W. Luk, Efficient weight reuse for large LSTMs, in: 2019 IEEE 30th International Conference on Application-Specific Systems, Architectures and Processors (ASAP), Vol. 2160, IEEE, 2019, pp. 17–24.
[5] R. Zhao, et al., Towards efficient convolutional neural network for domain-specific applications on FPGA, in: 28th International Conference on Field Programmable Logic and Applications, FPL, IEEE, 2018.
[6] Z. Que, et al., Towards in-circuit tuning of deep learning designs, in: International Conference on Computer-Aided Design, ICCAD, IEEE, 2019.
[7] Z. Que, D.H. Noronha, R. Zhao, S.J. Wilton, X. Niu, W. Luk, Towards overlay-based rapid in-circuit tuning of deep learning designs, in: 2020 International Conference on Field-Programmable Technology, FPT, IEEE, 2020.
[8] A.G. Howard, et al., Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017, ArXiv Preprint ArXiv:1704.04861.
[9] 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.
[10] F. Mamalet, C. Garcia, Simplifying convnets for fast learning, in: International Conference on Artificial Neural Networks, Springer, 2012, pp. 58–65.
[11] M. Abadi, et al., Tensorflow: Large-scale machine learning on heterogeneous systems, 2015, Software available from tensorflow.org, http://tensorflow.org/.
[12] D.H. Noronha, et al., On-chip FPGA debug instrumentation for machine learning applications, in: Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ACM, 2019.
[13] D.H. Noronha, R. Zhao, Z. Que, J. Goeders, W. Luk, S. Wilton, An overlay for rapid FPGA debug of machine learning applications, in: International Conference on Field-Programmable Technology, ICFPT, IEEE, 2019.
[14] M. Alwani, et al., Fused-layer CNN accelerators, in: The 49th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Press, 2016.
[15] 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.
[16] J. Goeders, S.J. Wilton, Signal-tracing techniques for in-system FPGA debugging of high-level synthesis circuits, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 36 (1) (2016) 83–96.
[17] K. Guo, et al., Angel-eye: A complete design flow for mapping CNN onto embedded FPGA, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. (2017).
[18] R. Zhao, et al., Optimizing CNN-based object detection algorithms on embedded FPGA platforms, in: International Symposium on Applied Reconfigurable Computing, Springer, 2017.
[19] Y. Umuroglu, et al., BISMO: A scalable bit-serial matrix multiplication overlay for reconfigurable computing, in: 28th International Conference on Field Programmable Logic and Applications, FPL, IEEE, 2018.

[20] H. Sharma, et al., Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network, in: 45th Annual International Symposium on Computer Architecture, ISCA, IEEE, 2018.

[21] T. Posewsky, D. Ziener, Throughput optimizations for FPGA-based deep neural network inference, Microprocess. Microsyst. 60 (2018) 151–161.

[22] J. Misra, I. Saha, Artificial neural networks in hardware: A survey of two decades of progress, Neurocomputing (2010).

[23] Z. Que, H. Nakahara, E. Nurvitadhi, H. Fan, C. Zeng, J. Meng, X. Niu, W. Luk, Optimizing reconfigurable recurrent neural networks, in: IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM, IEEE, 2020, pp. 10–18.

[24] Z. Que, et al., A reconfigurable multithreaded accelerator for recurrent neural networks, 2020 International Conference on Field-Programmable Technology (ICFPT), IEEE, 2020.

[25] D. Kroening, O. Strichman, Decision Procedures, Springer, 2016.

[26] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, 2014, ArXiv Preprint ArXiv:1409.1556.

[27] F. Li, B. Zhang, B. Liu, Ternary weight networks, 2016, ArXiv Preprint ArXiv:1605.04711.

[28] S. Tridgell, et al., Unrolling ternary neural networks, Trans. Reconfigurable Technol. Syst. (2019).

[29] D.H. Noronha, et al., Rapid circuit-specific inlining tuning for FPGA high-level synthesis, in: International Conference on ReConFigurable Computing and FPGAs, ReConFig, IEEE, 2017.

[30] C. Xia, et al., DNNTune: Automatic benchmarking DNN models for mobile-cloud computing, ACM Trans. Archit. Code Optim. (2019).

[31] A. Mametjanov, et al., Autotuning FPGA design parameters for performance and power, in: IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines, IEEE, 2015.

[32] C. Xu, et al., A parallel bandit-based approach for autotuning FPGA compilation, in: Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ACM, 2017.

[33] Q. Xiao, Y. Liang, Fune: An FPGA tuning framework for CNN acceleration, IEEE Des. Test (2019).

[34] Q. Xiao, et al., Zac: Towards automatic optimization and deployment of quantized deep neural networks on embedded devices, in: International Conference on Computer-Aided Design, ICCAD, IEEE, 2019.

[35] K. Wang, et al., HAQ: Hardware-aware automated quantization with mixed precision, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2019.

[36] A.-S. Jamal, J. Goeders, S.J. Wilton, An FPGA overlay architecture supporting rapid implementation of functional changes during on-chip debug, in: 2018 28th International Conference on Field Programmable Logic and Applications, FPL, IEEE, 2018, pp. 403–4037.

[37] S. Wang, T. Tuor, T. Salonidis, K.K. Leung, C. Makaya, T. He, K. Chan, Adaptive federated learning in resource constrained edge computing systems, IEEE J. Sel. Areas Commun. (2019).