HBDC A: A Toolchain for High-Accuracy BRAM-Defined CNN Accelerator on FPGA with Flexible Structure

Zhengjie LI (a), Nonmember, Jiabao GAO (b), Student Member, and Jinmei LAI (b), Member

SUMMARY In recent years FPGA has become popular in CNN acceleration, and many CNN-to-FPGA toolchains are proposed to fast deploy CNN on FPGA. However, for these toolchains, updating CNN network means regeneration of RTL code and re-implementation which is time-consuming and may suffer timing-closure problems. So, we propose HBDCA: a toolchain and corresponding accelerator. The CNN on HBDCA is defined by the content of BRAM. The toolchain integrates UpdateMEM utility of Xilinx, which updates content of BRAM without re-synthesis and re-implementation process. The toolchain also integrates TensorFlow Lite which provides high-accuracy quantization. HBDCA supports 8-bits per-channel quantization of weights and 8-bits per-layer quantization of activations. Upgrading CNN on accelerator means the kernel size of CNN may change. Flexible structure of HBDCA supports kernel-level parallelism with three different sizes (3 × 3, 5 × 5, 7 × 7). HBDCA implements four types of parallelism in convolution layer and two types of parallelism in fully-connected layer. In order to reduce access number to memory, both spatial and temporal data-reuse techniques were applied on convolution layer and fully-connect layer. Especially, temporal reuse is adopted at both row and column level of an Input Feature Map of convolution layer. Data can be just read once from BRAM and reused for the following clock. Experiments show by updating BRAM content with single UpdateMEM command, three CNNs with different kernel size (3 × 3, 5 × 5, 7 × 7) are implemented on HBDCA. Compared with traditional design flow, UpdateMEM reduces development time by 7.6–9.1X for different synthesis or implementation strategy. For similar CNN which is created by toolchain, HBDCA has smaller latency (9.97µs–50.73µs), and eliminates re-implementation when update CNN. For similar CNN which is created by dedicated design, HBDCA also has the smallest latency 9.97µs, the highest accuracy 99.14% and the lowest power 1.391W. For different CNN which is created by similar toolchain which eliminate re-implementation process, HBDCA achieves higher speedup 120.28X.

key words: toolchain, TensorFlow Lite, UpdateMEM, CNN, FPGA

1. Introduction

In recent years, more DNN (Deep Neural Network) related applications moving towards edge device, such as smart cities, smart homes, automated driving and industrial applications [1]. DNN is compute-intensive, while edge computing platforms are resource-limited. Therefore, LW (Light Weight) CNN models, such as MobileNetV1 [2], have been proposed oriented for mobile devices. LW CNN uses depth-wise convolution to replace standard convolution, which reduces computation and parameters largely. MobileNetV2 [3] and MobileNetV3 [4] further reduce parameters. LW CNN are deployed on mobile phones, such as Pixel 2 or Pixel 2XL [5], which still have 4GB memory. However, some IoT edge device, such as microcontrollers, are even more memory-limited. Tiny machine learning (TinyML) [6] is proposed to expand the scope of AI applications. For example, Lin et al. [7] propose TinyML model which at most requires 2Mbytes Flash. FPGA has less power than GPUs, higher throughput than CPU, and shorter time-to-market than ASIC. Reconfigurable computing is being considered for inference on edge due to its hardware flexibility that allows for the easy adaption of the target computing platform to the CNN model [8]. So, many FPGA-based accelerators [9]–[16] have been proposed to accelerate CNN inference. However, FPGA is complex for CNN software engineer. So, toolchain was developed to ease the problem [17]–[24]. Toolchain of [18]–[20] integrate HLS flow. Toolchain of [21]–[23] integrates Verilog flow. Toolchain of [24] integrates hybrid HLS-Verilog flow. But, there are two problems for these toolchains when accelerator needs to update CNN. One problem is time-consuming, toolchains of [18]–[24] needs regeneration of RTL code and re-implementation which may takes hours. The other problem is timing-closure problems. These toolchains just can modify hardware parameter, when timing-closure is not meet, it has to decrease the operating frequency, which leads to performance deterioration. So, Yu et al. [25] propose Light-OPU, its toolchain eliminates the re-implementation process. But it doesn’t consider the TinyML model situation that all parameters are stored on-chip memory. Loading model parameters from off-chip memory will increase power and latency.

Quantization is widely used in inference, 16/18-bits quantization was adopted by [11], [13]–[15] which utilizes more logic resources. BNN [26], [27] use 1-bit weight and activation which dramatically decrease hardware consumption, but it also decreases the accuracy. 8-bits quantization was adopted by [16], [21], [25] which use their own scheme and may deteriorate the accuracy for different CNNs. Currently, TF (TensorFlow) Lite adopts per-channel quantization of weights and per-layer quantization of activations to 8-bits, which produces accuracy loss within 2% compared with floating-point networks for a wide variety of CNN [28]. TF Lite does not support FPGA directly. Though Shadyduk et al. [9] applies similar quantization scheme of TF Lite, but they don’t consider round-to-nearest when quantize the data, which may decrease accuracy.

Some accelerators just optimized to one CNN [11],
[16], [21], or one certain kernel size [11], [25]. However, upgrading CNN on edge device means accelerator should support more CNNs and the kernel size of different CNN may change. PE array optimized for certain kernel size may not fit other sizes well. This is why some accelerators only optimize their architecture on 3x3 kernel size [29]. PE array of [25] just support 3x3 kernel, and larger kernel is decomposed into several 3x3 kernels which increase computation time.

Data reuse technique reduces the access number of off-chip and large on-chip memory and decrease power. There are mainly two types of data reuse: spatial reuse and temporal reuse [10]. Spatial reuse is widely adopted to generate multiple Output Feature Map (OFM). Temporal reuse isn’t fully explored. Temporal reuse doesn’t adopt by [21], [25]. Temporal reuse of [12], [22] just adopts a row or a column in IFM (input feature map) at convolution (CONV) layer, not both, and doesn’t adopt temporal reuse at fully-connected (FC) layer.

To deal with these problems above, we propose HB-DCA: a toolchain and corresponding accelerator. For TinyML model, CNN on accelerator is defined by content of BRAM. We use XPM (Xilinx Parameterized Macros) [30] to design BRAM, after implementation we use UpdateMEM [31] to upgrade CNN by changing the content of BRAM. UpdateMEM generates new bitstream without re-implementation. We integrate TF Lite in our toolchain, and implement function of TF Lite and two round-to-nearest functions of gemmlowp library to get high-accuracy result. We also design flexible PE arrays, which support kernel-level parallelism with three different size (3, 5, 7). Besides spatial reuse, temporal reuse is adopted at CONV layer (both row and column level of an IFM) and FC layer. Data just be read once from BRAM and reused for the following clock.

The rest of the paper is organized as follows. Section 2 describes the toolchain. Section 3 describes accelerator’s flexible hardware architecture. Section 4 describes the experiment results. Section 5 concludes the paper.

2. Toolchain Description

The overall toolchain is shown in Fig. 1. One input of our toolchain is Definition of CNN & Dataset. We use Keras of TensorFlow 2.3 and TensorFlow Lite to train and quantize the model. Quantization Parameters Extractor gets quantized parameters, such as scales, zero-points, quantized weights and bias, from quantized model.

The other input of our toolchain is FPGA Available Resources, combined with several CNNs information, we can do design exploration to find best hardware configuration information. Then, Code Generator generates single set of Verilog files. Quantization BRAM Content Generator and Instruction Generator generates several Memory files [31] which define content of XPM BRAM. Vivado gets one set of Memory files and single set of Verilog files to do implementation and finally generates a bitstream file and an MMI (Memory Map Information) file [31].

Finally, if we want to update CNN, CNN Updater can generate a new bitstream file by using UpdateMEM, this process excludes re-implementation process.

Modules in grey color of Fig. 1 will discussed further.

2.1 Quantization Parameters Extractor

For two square \( N \times N \) matrices of real numbers, \( r_1 \) and \( r_2 \), the result of their matrix multiplication is \( r_3 \), so \( r_3=r_1r_2 \). In Eq. (1), \( q_i \) \( (i = 1, 2, 3) \) is corresponding quantized integer matrices. The entries of real and quantized integer matrices are \( r_i^{(jk)} \) and \( q_i^{(jk)} \), \( (i = 1, 2, 3; 1 \leq j, k \leq N) \). The scale and zero-point of each matrix is \( S_i \) and \( Z_i \) \( (i = 1, 2, 3) \). The quantized \( q_i^{(jk)} \) is calculated by Eq. (2). The scalars ratio \( M \) in Eq. (3) is the only float-point and in the interval (0, 1). \( M \) can be converted to an approximated fixed-point equivalent by doubling \( M \) until it is in the interval [0.5, 1) which could be represented by a truncated integer multiplier \( MULT \) with an appropriate right \( SHIFT \), yielding Eq. (4).

\[
\begin{align*}
q_i^{(jk)} &= S_i (q_i^{(jk)} - Z_i) \quad (i = 1, 2, 3) \quad (1) \\
q_3^{(jk)} &= Z_3 + \frac{S_1 S_2}{S_3} \sum_{j=1}^{N} (q_1^{(j)} - Z_1)(q_2^{(k)} - Z_2) \quad (2) \\
M &= \frac{S_1 S_2}{S_3} \approx 2^{-SHIFT} MULT \quad (3) \\
q_3^{(jk)} &= Z_3 + MULT \sum_{j=1}^{N} (q_1^{(j)} - Z_1)(q_1^{(k)} - Z_2) \quad (4)
\end{align*}
\]
Equation (4) is part of function of TF Lite. After analyzing TF Lite model, we can get scale, zero-point, quantized weights and bias of each layer of CNN. We store all the quantized parameters in NumPy files, see Fig. 2 (a).

Then we use scale parameters to calculate MULT and SHIFT parameters of each layer. Especially, for CONV layer, each channel has different MULT and SHIFT. For TF Lite, MULT in Eq. (3) and biases are 32-bit, we will adopt proper precision for MULT and biases, which can balance precision and hardware resources, see Fig. 2 (b).

2.2 Quantization BRAM Content Generator

The Memory (MEM) file with .mem extension describes contiguous blocks of data. They define content of XPM BRAMs. The MEM files have to add to the Vivado project as design source. The format is discussed in [31]. After Design Exploration, we get the BRAM size. Quantization BRAM Content Generator uses size information and writes quantized weights, bias and other parameters (MULT, SHIFT, zero-point) into .mem files. Figure 3 shows three examples of how data is organized in MEM files.

For CONV layer, the number of output channel and input channel are $C_{out}$ and $C_{in}$. The size of kernel is $K_h \times K_w$.

The number of PE arrays is $N_{pe\_array}$, and one PE array generates one channel of OFMs. For Fig. 3 (a), it shows weights organization of a CONV layer when $C_{in} = 1$, $C_{out} = 6$, $N_{pe\_array} = 4$. For Fig. 3 (b), it shows biases organization of all CONV layer. The number of output channel of CONV1, CONV2 and CONV3 are $C_{out\_CONV1}$, $C_{out\_CONV2}$ and $C_{out\_CONV3}$, respectively. In Fig. 3 (b), $C_{out\_CONV1} = 6$, $C_{out\_CONV2} = 4n$, $N_{pe\_array} = 4$. For Fig. 3 (c), it shows weights organization of a CONV when $C_{in} \neq 1$, $C_{out} = 4n$, $N_{pe\_array} = 4$.

2.3 Instruction Generator

With JSON file and hardware configuration information, Instruction Generator translates each CNN layer of JSON file into series of instruction. Accelerator analyzes instructions and operates accordingly. Instructions are stored in MEM files.

2.4 Code Generator

The proposed hardware architecture is shown in Fig. 8. Code Generator will generate single set Verilog files which include all the modules in Fig. 8. The details of each module will be discussed in Sect. 3.3. Especially, BRAM uses XPM memory template [30]. Attribute MEMORY_PRIMITIVE set to “block”; attribute MEMORY_INIT_PARAM set to “”; attribute MEMORY_INIT_FILE set to a Memory (MEM) file. Once BRAMs are designed XPM memory template, after Implementation, an MMI (Memory Map Information) file will be automatically generated. The (MMI) file is a text file that describes how individual BRAMs on the Xilinx device are grouped together to form a contiguous address space called an address block [31].

2.5 CNN Updater

CNN Updater uses UpdateMEM utility, which is an update_mem [31] commands. With MMI file, the update_mem command updates contiguous blocks of data across multiple BRAMs. By this way, new bitstream file will be generated without re-implementation process. One example is shown below.

```
update_mem -meminfo xpm.mmi -data ins_k3.mem -procctl/ins_buffer/xpm_memory_sprom_inst/
```
xpm_memory_base_inst -bit cnn_top.bit -out k3_ins.bit -force -debug
Above command updates content of Instruction BRAM (ins_buffer) with new instructions (ins_k3.mem). Along with more -data and -proc pairs, it updates content of all BRAMs and generates a new bitstream file. By downloading the new bitstream file, the new CNN network in updated on FPGA accelerator.

3. Flexible Hardware

CONV layer and FC layer are two main layers of a typical CNN network. Flexible PE (Processing Element) arrays can implement four types parallelism in CONV layer and two types parallelism in FC layer.

3.1 Flexible PE Arrays for CONV Computation and Tiling

For CONV layer, the computation is shown in Fig. 4, the number of output channel and input channel are Cout and Cin. The height and width of OFM and IFM are Hout, Wout, H_in and W_in respectively. The size of kernel is Kh × Kw. The stride of kernel is s, the default value of s is 1. Our accelerator implements four types of parallelism (line 1, 2, 3, 6 in Fig. 4): numbers of OFM channel (ToChan), numbers of rows of an OFM (ToRow), numbers of columns of an OFM (Tocol), and size of kernel’s width (Kw).

The PE arrays compute a tiling of IFMs at one time, see Fig. 5, the width and height of tiling IFM are Ticol and Tiro. The width, height and channel of tiling OFM are Tocol, Toro, Ton, respectively. The maximum value of acceleration and tiling parameters has relationship with hardware resources. In Fig. 5, number of PE arrays is Npe_array.

The Eqs. (6)–(10) are the value of ToChanmax, Tocolmax, Torowmax, Tirowmax, Ticolmax, Tiro.

\[
\begin{align*}
ToChanmax &= N_{pe	ext{-}array} \\
ToRowmax &= \left\lfloor \frac{PE_{row}}{K} \right\rfloor \\
Tocolmax &= PE_{col} \\
Tirowmax &= T_{rowmax} + K - 1 \\
Ticolmax &= T_{colmax} + K - 1
\end{align*}
\]

3.2 Flexible PE Arrays for FC Computation and Tiling

For FC layer, the computation is shown in Fig. 6, the total output neurons (ONs) are Ton, the total input neurons (INs) are Tin, we implement parallelism of ONs: Pon, and parallelism of INs: Pin (line 1, 2 in Fig. 6).

Tiling in FC layer is shown in Fig. 7, PE row transfers to different PE arrays simultaneously, this complete data spatial reuse. One PE array generates one channel of OFMs, so Npe_array PE arrays generate Npe_array channels of OFMs, Npe_array equals to ToChanmax. One column of a PE array generates one column of an OFM, so PE_col columns of a PE array generate PE_col columns of an OFM, PE_col equals to Tocolmax.

PE array supports kernel-level parallelism with three different size (3×3, 5×5, 7×7). PE array is divided by the size of kernel in the row direction, and each separated part we call Line. Each Line generates one row of an OFM. So total Torowmax Lines generate Torowmax rows of an OFM. In Fig. 5, K = 3, from Eq. (5), it also can be 5 or 7.

\[
K_n = K_h = K, \quad K \in (3, 5, 7)
\]

The numbers of rows and columns of a PE array are PE_row, PE_col. IFM transfers to different PE arrays simultaneously, this complete data spatial reuse. One PE array generates one channel of OFMs, so Npe_array PE arrays generate Npe_array channels of OFMs, Npe_array equals to ToChanmax. One column of a PE array generates one column of an OFM, so PE_col columns of a PE array generate PE_col columns of an OFM, PE_col equals to Tocolmax.

PE array supports kernel-level parallelism with three different size (3×3, 5×5, 7×7). PE array is divided by the size of kernel in the row direction, and each separated part we call Line. Each Line generates one row of an OFM. So total Torowmax Lines generate Torowmax rows of an OFM. In Fig. 5, K = 3, from Eq. (5), it also can be 5 or 7.

K_n = K_h = K, \quad K \in (3, 5, 7)

The Eqs. (6)–(10) are the value of ToChanmax, Tocolmax, Torowmax, Tirowmax, Ticolmax.

\[
\begin{align*}
ToChanmax &= N_{pe	ext{-}array} \\
ToRowmax &= \left\lfloor \frac{PE_{row}}{K} \right\rfloor \\
Tocolmax &= PE_{col} \\
Tirowmax &= T_{rowmax} + K - 1 \\
Ticolmax &= T_{colmax} + K - 1
\end{align*}
\]
3.3 Overall Architecture

The proposed hardware architecture is shown in Fig. 8. It includes six modules: Ping-Pong Buffers, PE Arrays, Register Bank, Control Unit, Instruction BRAM, and Quantization BRAMs.

**Ping-Pong Buffers** stores image and intermediate activation values of each CONV and FC layer.

**PE Arrays** do CONV and FC layer computation in pipeline manner. In Fig. 9, one PE Array is composed of PEs (such as R0C0, R3C0, etc.), Accumulator (such as “A 0,0”, “A 1,0”, etc.) and POOL. The POOL completes max pooling operation. The accumulation computation of lines 7–8 in Fig. 4 is calculated by Accumulator. The quantization of activation is also calculated in Accumulator. The result of convolution is multiplied by $MULT$ and followed right $SHIFT$, as described in Eq. (4). Which is quantization function of $MultiplyByQuantizedMultiplierSmallerThanOne$ of TF Lite. Accumulator also implements round-to-nearest function $SaturatingRoundingDoublingHighPartialRoundingDivideByPOT$ of gemmlowp library when quantizes activations.

**Register Bank** allocates IFMs or INs, weights to PE Array, which completes data temporal reuse at row dimension. It includes Delay line to pipeline all data.

**Control Unit** processes instructions from Instruction BRAM, and coordinate each module to completes all the CONV, POOL and FC layer operation.

**Instructions BRAM** stores the instructions generated by Instruction Generator.

**Quantization BRAMs** stores the weights, bias and other parameters (MULT, SHIFT, zero-point) of all layers.

3.4 Mechanism of Convolutional Computation

The PE array supports parallelism of three different kernel size (3, 5, 7). For simplicity, Fig. 9 shows the case when $K_w = 3, C_{out} = 1, C_{in} = 1, PE_{row} = 9, PE_{col} = 4$. According to Eqs. (7)–(10), $T_{row} = 3, T_{col} = 4, T_{irow} = 5, T_{icol} = 6$. $T_{row} = 3$ means the PE array is divided into 3 Lines, each Line has 3 rows of PEs and computes one row of OFM. $T_{icol} = 6$ means PE array generates 4 columns of OFM. So, Accumulators of Line 1, Line 2, Line 3 generate the 3 rows and 4 columns of pixels in OFM.

A PE is implemented by DSP module, each PE of a column uses $PCIN$ and $PCOUT$ ports of DSP to cascaded from top to bottom. The function of PEs of the first row of a Line (PEs of R0, R3 and R6 in Fig. 9) is defined by Eq. (13), because they don’t have cascaded input. The function of the rest PEs is defined by Eq. (14). W and IFM are weight and IFM at line 7 in Fig. 4.

$$PCOUT = W \times IFM$$

By this cascading method, partial convolution result is sent to Accumulator. For example, PE R0C0, R1C0 and R2C0 of C0 column of Line 1 propagate the partial convolution result to Accumulator “A 0,0”. So, Accumulator “A 0,0” gets result of $3 \times 3$ convolution through pipeline computation, and generates one pixel (1st row, 1st column) of OFM if there is no POOL operation. Computation detail will be discussed in Sect. 3.5.

$T_{icol} = 5$ means accelerator reads 5 rows of IFMs from Pong buffer. $T_{icol} = 6$ means grey color PEs receive data from 6 columns of IFMs at different clock. The detail is shown in Fig. 10.

After PE Array computes one tiling ($T_{irow} = 5, T_{icol} = 6$) of IFMs, PE Array generates one tiling ($T_{row} = 3$, $T_{col} = 4, T_{irow} = 5, T_{icol} = 6$) of OFMs.
of OFMs. If there is no POOL operation, just as Fig. 9 shows, Ping buffer receives one tiling of OFMs directly. Otherwise, these data will first be processed by POOL module, then send pooling result to Ping buffer.

3.5 Data Temporal Reuse

In Fig. 5 IFM transfers to different PE arrays simultaneously, this complete data spatial reuse. The detail of data temporal reuse of CONV layer is shown in Fig. 10.

At first, the grey color PEs reads one column IFMs at each clock. For example, for PEs of Line 1 in Fig. 10, R0C0 reads ‘1-1’ of 1st IFM column at clk0; R1C0 reads ‘1-2’ of 2nd IFM column at clk1; R2C0 reads ‘1-3’ of 3rd IFM column at clk2.

Delay line of Register Bank in Fig. 9 is used to do pipeline operation when grey color PEs of column C1, C2 and C3 read BRAM data. So, for PEs of Line 1 in Fig. 10, R2C1 reads ‘1-4’ of 4th IFM column at clk4, R2C2 reads ‘1-5’ of 5th IFM column at clk6.

After reading one data from BRAM, the grey color PEs in Fig. 10 turn into the blue color, and blue color PEs enter into 1st shift stage and receive shift data from corresponding blue color PEs of next Line. For example, for PEs of Line 1 in Fig. 10, at clk1, R0C0 reads ‘2-1’ of Line 2; at clk2, R1C0 reads ‘2-2’ of Line 2; at clk3, R2C0 reads ‘2-3’ of Line 2.

After 1st shift stage, the blue color PEs in Fig. 10 turn into the yellow color, and yellow color PEs enter into 2nd/final (= K – 1) shift stage and receive shift data from corresponding yellow color PEs of next Line. For example, for PEs of Line 1 in Fig. 10, at clk2, R0C0 reads ‘3-1’ of Line 2; at clk3, R1C0 reads ‘3-2’ of Line 2; at clk4, R2C0 reads ‘3-3’ of Line 2.

Description above shows data movement of PEs in Line 1, PEs of Line 2 and Line 3 work in the same way. Data temporal reuse of grey color PEs at row dimension can be represented by Eq. (15).

\[ R_{rC_c} = R_{r+kC_c}, \quad k \in (3, 5, 7) \] (15)

At the same time, upper right PEs receive data from lower left PEs. For example, for PEs of column C1 of Line 1 in Fig. 10, R0C0 reads ‘1-2’ from R1C0 at clk2; R1C0 reads ‘1-3’ from R2C0 at clk3. PEs of Line 2 and Line 3 work in the same way. Data temporal reuse at column dimension can be represented by Eq. (16).

\[ R_{rC_{c+1}} = R_{rC_c} \] (16)

So, data temporal reuse can be at both row and column dimension. PE array reads IFMs just once from BRAM and reuse the data in following computation. For example, R8C0 reads ‘3-3’ at clk2. In row dimension, R5C0 reuses ‘3-3’ at clk3, R2C0 reuses ‘3-3’ at clk4. In column dimension, R7C1 reuses ‘3-3’ at clk3, R6C2 reuses ‘3-3’ at clk4.

In Fig. 9, \( K_w \times K_h \) (9) numbers of weights are reused by each Line spatially. In Fig. 11, for Line 1, \( K_w \times K_h \) (9) numbers of weights of are reused from C0 column to C2 columns temporally. It takes two clocks move from one column to next column for pipeline operation. For example, R0C0 reads weight ‘1-1’ from BRAM at clk0; R0C1 reads weight ‘1-1’ at clk2; R0C2 reuses weight ‘1-1’ at clk4.

From IFM data movement in Fig. 10, weights data movement in Fig. 11, PE function of Eq. (13) and Eq. (14), R0C0 gets result \( P_{\text{COUTRICO}} = W_{1-1} \times IFM_{1-1} \) at clk0; R1C0 gets result \( P_{\text{COUTRICO}} = W_{1-2} \times IFM_{1-2} + P_{\text{CINRICO}} \) at clk1; R2C0 gets result \( P_{\text{COUTRICO}} = W_{1-3} \times IFM_{1-3} + \)
First, we use our toolchain in Fig. 1 to implement one CNN with traditional Vivado design flow on Xilinx XC7Z100 in a customized board, see Fig. 13. Clock frequency is 100MHz.

3.6 Design Exploration

Because CONV layer is compute-intensive, we aim to minimize the cycles of convolution computation. The number of cycles to calculate a row of OFM (Cycle\textsubscript{row}) in Fig. 9 is shown in Eqs. (17)–(19). Cycle\textsubscript{small} and Cycle\textsubscript{large} means the number of cycles to calculate a row of OFM when PE\textsubscript{col} is small or large.

\[
\text{Cycle}_{\text{small}} = \left( \left\lfloor \frac{W_{\text{out}}}{PE_{\text{col}}} \right\rfloor - 1 \right) \times (K - PE_{\text{col}}) + W_{\text{in}} \tag{17}
\]

\[
\text{Cycle}_{\text{large}} = \left( \left\lfloor \frac{W_{\text{out}}}{PE_{\text{col}}} \right\rfloor \times K + PE_{\text{col}} - 1 \right) \tag{18}
\]

\[
\text{Cycle}_{\text{row}} = \max \left( \text{Cycle}_{\text{small}}, \text{Cycle}_{\text{large}} \right) \tag{19}
\]

W\textsubscript{out}, W\textsubscript{in}, PE\textsubscript{col}, PE\textsubscript{row} and N\textsubscript{pe\_array} here are identical to those in Fig. 5. We iterate PE\textsubscript{col} from \([K, W_{\text{out\_row}}]\) interval. W\textsubscript{out\_row} is W\textsubscript{out} of CONV1, which is largest compared with W\textsubscript{out} of CONV2 or CONV3. For a specific PE\textsubscript{col}, we calculate Cycle\textsubscript{row} for all CONV layer of all CNNs and add all together to get total cycles (Cycle\textsubscript{total\_col}) which is calculated by Eq. (20). So, we can get a set of total cycles for different PE\textsubscript{col}, and find PE\textsubscript{col} which achieves minimum Cycle\textsubscript{total\_col} from the set by Eq. (21). Then, we use Eqs. (22)–(24) to get PE\textsubscript{row} and N\textsubscript{pe\_array}. ACC\textsubscript{dsp} means the number of DSPs used as Accumulator in a PE array. DSP\textsubscript{avail} is the number of available DSPs of a FPGA.

![Data temporal reuse of input neurons in FC layer.](image)

### 4. Experiments

![Customized board.](image)

| Design Runs (Strategy)                  | Traditional\textsuperscript{[a]} | Update MEM\textsuperscript{[b]} |
|----------------------------------------|-----------------------------------|----------------------------------|
| Synthesis (Default)                    | 11                                | 0                                |
| Implementation (Default)               | 55                                | 0                                |
| Synthesis (Flow\_PerfOptimized\_high)  | 10                                | 0                                |
| Implementation (Performance\_NetDelay\_high) | 58                     | 0                                |
| Incremental Synthesis (Flow\_PerfOptimized\_high) | 11                     | 0                                |
| Incremental Implementation (Performance\_NetDelay\_high) | 45                     | 0                                |
| Bitstream generation                   | 5                   | 8                                |
| Total (Synthesis + Implementation + Bitstream) | 61/71/73                     | 8                                |
| Execution time (Compared to UpdateMEM) | 7.6X/8.9X/9.1X | 1X                               |

\textsuperscript{[a]} We use “get_property STATS.EIAPSED [get-runs impl_*/synth_*]” command to get the runtime, and round to the nearest minute.

\textsuperscript{[b]} UpdateMEM doesn’t support command in [a]. We use “clock seconds” to get start time and end time of UpdateMEM, and calculate the runtime from the difference.
Table 2  Three different CNNs, ($K(3, 5, 7)$).

| Layer  | $K = 3$ | $K = 5$ | $K = 7/5$ |
|--------|---------|---------|-----------|
|        | $H_{out}$ | $W_{out}$ | $C_{out}$ | $H_{out}$ | $W_{out}$ | $C_{out}$ | $H_{out}$ | $W_{out}$ | $C_{out}$ |
| CONV1  | 28      | 28      | 8        | 28      | 28      | 6        | 22      | 22      | 4        |
| POOL1  | 14      | 14      | 8        | 14      | 14      | 6        | 11      | 11      | 4        |
| CONV2  | 12      | 12      | 32       | 10      | 10      | 16       | 5       | 5       | 16       |
| POOL2  | 6       | 6       | 32       | 5       | 5       | 16       | -       | -       | -        |
| CONV3  | 4       | 4       | 64       | 1       | 1       | 120      | 1       | 1       | 120      |
| POOL3  | 2       | 2       | 64       | -       | -       | -        | -       | -       | -        |
| FC1    | 1       | 1       | 128      | 1       | 1       | 84       | 1       | 1       | 84       |
| FC2    | 1       | 1       | 10       | 1       | 1       | 10       | 1       | 1       | 10       |
| Operations | 1458314 | 845862  | 468854   |

Table 3  Accuracy of 10000 images and power of three CNNs and average PE utilization.

| TF Lite (%) | Hardware (%) | Power (watt) | Average PE Utilization (%) |
|------------|--------------|--------------|----------------------------|
| CNN ($K = 3$) | 99.07       | 99.04       | 1.495                     | 34.213                     |
| CNN ($K = 5$) | 99.13       | 99.14       | 1.391                     | 64.038                     |
| CNN ($K = 7/5$) | 98.68     | 98.67       | 1.532                     | 53.173                     |

Table 4  Resources utilization of our accelerator.

| Resources | LUT | LUTRAM | FF | BRAM | DSP |
|-----------|-----|--------|----|------|-----|
| Available | 277400 | 108200 | 354800 | 755 | 2040 |
| Used      | 89292 | 3730   | 72190  | 210 | 1376 |
| Utilization | 32.19% | 5.30%  | 13.01% | 27.81% | 68.12% |

Fig. 14  (a) Throughput (GOPS) and (b) latency (µs) of three CNN network

Table 5  Runtime between CPU and FPGA of three CNNs.

| CNN ($K = 3$) | CPU Latency (µs) | FPGA Latency (µs) | Speedup |
|---------------|------------------|-------------------|---------|
| 4500.817      | 50.73            | 88.72             |
| 2000.203      | 18.17            | 110.08            |
| 1199.172      | 9.97             | 120.28            |

From Table 2, for $K = 5$, the output of CONV3 is $H_{out} \times W_{out} \times C_{out} = 1 \times 1 \times 120$, just like the 120 output neurons, the input of CONV3 is $H_{in} \times W_{in} \times C_{in} = 5 \times 5 \times 16 = 400$, just like the 400 input neurons. So, we can transform the computation of CONV3 to the computation of FC. For FC computation, 4 PE arrays read 4 input channels and generate 32 output channels. For CONV computation, 4 PE arrays read 1 input channel and generate 4 output channels. So, in Fig. 14 (b), for $K = 5$, the latency of CONV3 is much shorter than the latency of CONV3+POOL3 for $K = 3$.

We run TensorFlow Lite model on both CPU (AMD Ryzen 7 2700X) and FPGA (XC7Z100), the runtime is shown in Table 5. Speedup is between 88.72X–120.28X.

Table 6 shows performance comparison with other accelerators of similar LeNet. Compared with [21] which also has toolchain, HBDC has smaller latency (batch=1) and eliminates re-implementation and provides power and accuracy result. Compared with accelerator without toolchain [13]–[16], HBDC has smallest latency 9.97µs when $K = 7/5$, highest accuracy 99.14% when $K = 5$; lowest latency of 1.391W when $K = 5$.

Table 7 shows performance comparison with [25] of different CNN. HBDC and [25] both have toolchain and support CNN upgrading without re-implementation. HBDC has higher speedup, and integrates TF Lite which pro-
vides high-accuracy but [25] doesn’t provide accuracy. HB-DCA supports three kernel size ($K = 3, 5, 7$), [25] just support one kernel size ($K = 3$).

### 5. Conclusion

We propose HB-DCA with corresponding toolchain for TinyML model. Toolchain integrates TF Lite which provides high-accuracy, and avoids re-implementation by using UpdateMEM when updates CNN. HB-DCA’s flexible PE arrays support kernel-level parallelism with three different size ($3, 5, 7$). Besides spatial reuse, temporal reuse is adopted for both CONV and FC layer. Especially in CONV layer, temporal reuse is implemented at both row and column level of an IFM. HB-DCA can be configured for different CNNs with different kernel sizes, and three CNNs are implemented on MNIST dataset in experiments. Compared with traditional design flow, UpdateMEM reduces development time by 7.6X–9.1X for different synthesis or implementation strategy. For similar CNN and implemented toolchain, HB-DCA has smallest latency 9.97 µs and eliminates re-implementation. For similar CNN without toolchain, HB-DCA has smallest latency 9.97 µs, highest accuracy 99.14% and lowest power 1.391 W. For other CNNs with similar toolchain, HB-DCA achieves higher speedup (88.72X–120.28X).

### Acknowledgments

This work was supported in part by the National Natural Science Foundation of China under Grant No. U20A20202.

### References

[1] M.P. Véstias, R.P. Duarte, J.T. Sousa, and H.C. Neto, “Moving Deep Learning to the Edge,” Algorithms, vol.13, no.5, 2020 (DOI: 10.3390/a13050125).

[2] A.G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, et al., “MobileNets: Efficient convolutional neural networks for mobile vision,” arXiv: 1704.04861, 2017.

[3] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C Chen, “MobileNetV2: Inverted Residuals and Linear Bottlenecks,” CVPR, pp.4510–4520, 2018 (DOI: 10.1109/CVPR.2018.00474).

[4] A. Howard, M. Sandler, B. Chen, W. Wang, L.-C. Chen, M. Tan, G. Chu, V. Vasudevan, Y. Zhu, R. Pang, H. Adam, and Q. Le, “Searching for MobileNetV3,” ICCV, pp.1314–1324, 2019 (DOI: 10.1109/ICCV.2019.00140).

[5] S. Alyamkin, M. Ardi, A.C. Berg, A. Brighton, B. Chen, Y. Chen, H.-P. Cheng, Z. Fan, C. Feng, B. Fu, K. Gauen, A. Goel, A. Goncharenko, X. Guo, S. Ha, A. Howard, X. Hu, Y. Huang, D. Kang, J. Kim, J.G. Ko, A. Kundratyev, J. Lee, S. Lee, S. Lee, Z. Li, Z. Liang, J. Liu, X. Liu, Y. Lu, Y.-H. Lu, D. Malik, H.H. Nguyen, E. Park, D. Repin, L. Shen, T. Sheng, F. Sun, D. Svitov, G.K. Thiruvathukal, B. Zhang, J. Zhang, X. Zhang, and S. Zhuo, “Low-Power Computer Vision: Status, Challenges, and Opportunities,” IEEE J. ETCAS, vol.9, no.2, pp.411–421, 2019 (DOI: 10.1109/JETCAS.2019.2911899).

### Table 6 Performance comparison of similar CNN.

| Year | 2018 | 2018 | 2020 | 2020 | 2016 | 2021 |
|------|------|------|------|------|------|------|
| Paper | SLIP [13] | DSP [14] | ITNEC [15] | ICCIA [16] | FPT [21] | This Paper (HB-DCA) |
| Used DSP | 50 | 1120 | 247 | 571 | 484 | 1376 |
| Freq. (Mhz) | 117 | 150 | 150 | 50 | 100 | 100 |
| Re-implement | Yes | Yes | Yes | Yes | Yes | No |
| Toolchain | No | No | No | No | Yes | Yes (TF Lite) |
| Precision | 16b | 16b | 18b | 8b | 8-16b | 8-16b |
| #CONV-#FC | 2C2F | - | 2C2F | 2C1F | 3C2F | 3C2F |
| Latency (µs) | 175.7 | 86 | 17.6 | 526 | 1318 | 9.97–50.73 |
| Power (Watt) | - | - | - | 14.13 | - | 1.391–1.532 |
| Accuracy % | 97.6 | 99 | 97.57 | - | - | 99.14–99.67 |

### Table 7 Performance comparison of different CNN.

| Year | 2020 | 2021 |
|------|------|------|
| Paper | FPGA [25] | This Paper (HB-DCA) |
| Used DSP | 704 | 1344 |
| Freq. (Mhz) | 200 | 100 |
| Network | LW CNNs | LeNet, Ke(3, 5, 7) |
| Precision | Fixed (8b) | Fixed (8-16b) |
| Toolchain | Yes | Yes (TF Lite) |
| Kernel Size | 3 × 3 | 3 × 3, 5, 7 × 7 |
| Re-implementation | No (off-chip) | No (On-chip) |
| Data Reuse | Spatial | Spatial and temporal |
| Power efficiency (GOPS/W) | 9.9 – 52.5 | 19.23 – 33.47 |
| Speedup | 22.0 – 51.9 (ARM) | 88.72 – 120.28 (AMD) |
| Accuracy (%) | - | 98.67 – 99.14 |

IEICE TRANS. INF. & SYST., VOL.E104-D, NO.10 OCTOBER 2021
Binarized neural network on FPGA,” Neurocomputing, vol.275, pp.1072–1086, 2018 (DOI: 10.1016/j.neucom.2017.09.046).

[28] R. Krishnamoorthi, “Quantizing deep convolutional networks for efficient inference: A whitepaper,” arXiv:1806.08342v1, 2018.

[29] Y. Yu, C. Wu, T. Zhao, K. Wang, and L. He, “OPU: An FPGA-Based overlay processor for convolutional neural networks,” VLSI, vol.28, no.1, pp.35–47, 2020 (DOI: 10.1109/VLSI.2019.2939726).

[30] Xilinx: UG953-Vivado Design Suite 7 Series FPGA and Zynq-7000 SoC Libraries Guide, https://www.xilinx.com/, 2020.

[31] Xilinx: UG898-Vivado Design Suite User Guide Embedded Processor Hardware Design, https://www.xilinx.com/, 2020.

[32] Xilinx: UG997-Vivado Design Suite Tutorial Power Analysis and Optimization, https://www.xilinx.com/, 2020.