CNN Training HW Architecture Design Using C2RTL SoC Synthesis/Verification Framework

Tsuyoshi Isshiki

Dept. of Communications and Computer Engineering
Tokyo Institute of Technology

MPSoC ‘19
July 11th, 2019

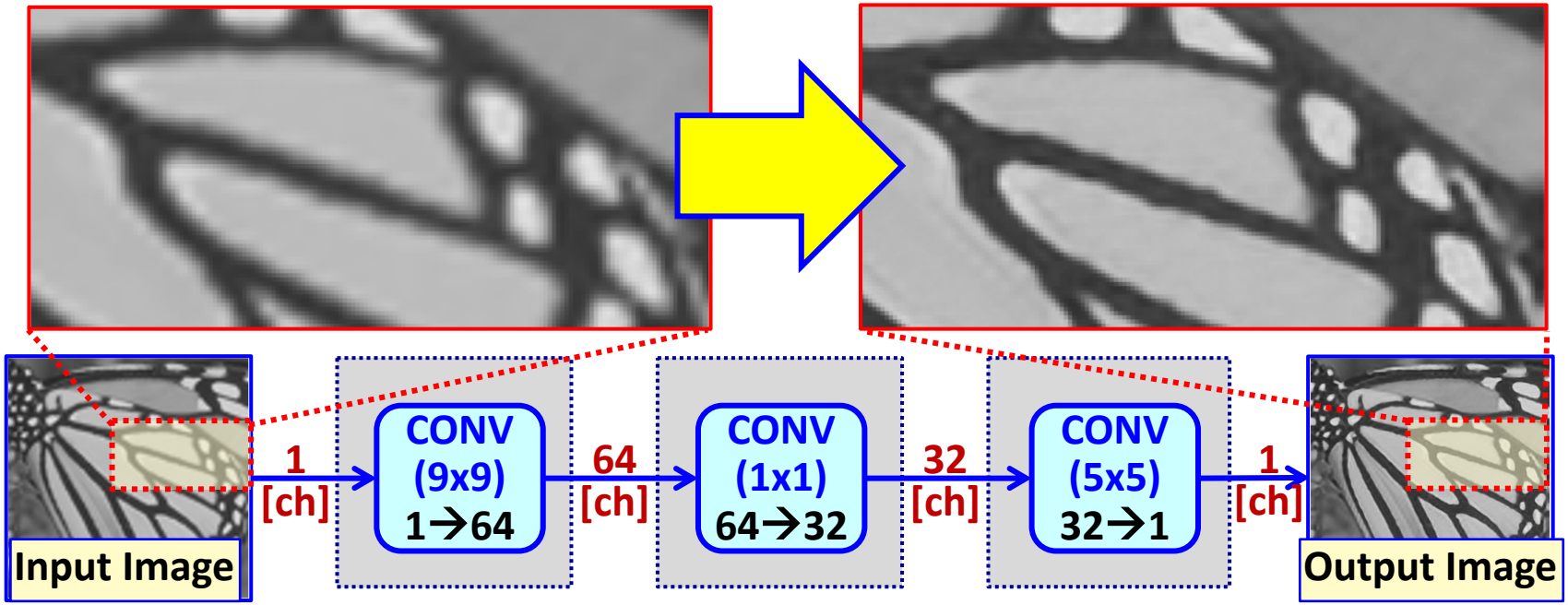
Deep Learning Hardware Frameworks

- **Inference hardware frameworks : many solutions . . .**
  - **CPU/GPU : Rich tools (Tensorflow, etc), easy setup**
  - **FPGA : Binary/Ternary Networks, xDNN/ML-Suite (Xilinx)**
  - **Custom HW : Servers (TPU), edge devices (chips, IPs)**
- **Training hardware frameworks : limited to CPU/GPU . . .**
  - **Floating-point arithmetics (FP32, FP16) → difficult to implement high density/low latency FPUs on FPGA**
  - **Evolving DNN topologies and use-cases → DNN Training workload demands will continue to increase dramatically**

→ ***Opportunities for FPGAs on Deep Learning TRAINING ?***

→ ***Deep Learning Training algorithm(C++) & HW design using C2RTL Design Framework***

Super-Resolution CNN (SRCNN)



MAC : Multiply-Accumulate

#1 CONV Layer :
1(IN)→ 64(ON)
Filter size : 9x9
MACs : 5,184

#2 CONV Layer :
64(IN) → 32(OUT)
Filter size : 1x1
MACs : 2,048

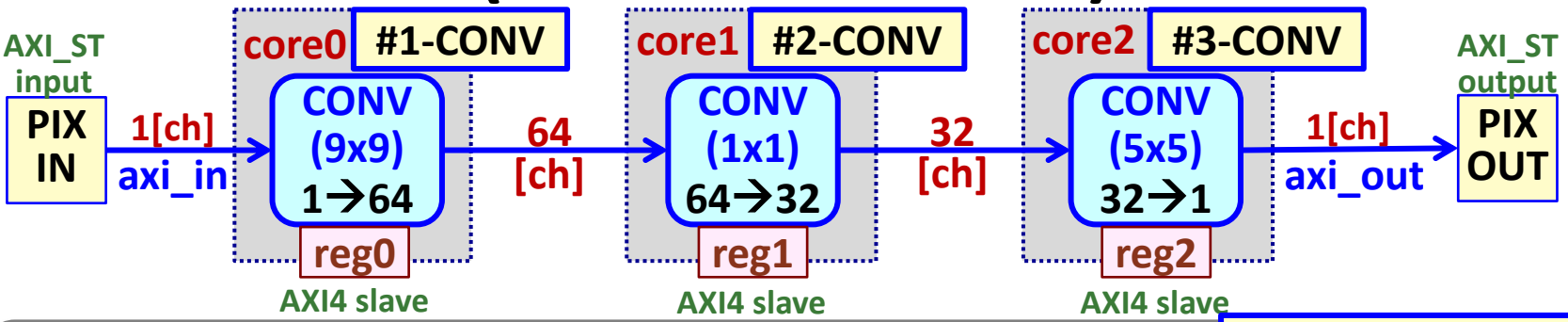
#3 CONV Layer :
32(IN) → 1(OUT)
Filter size : 5x5
MACs : 800

 # MACs per pixel (Inference) = 8,032 MACs = 16,064 Ops

 # operations per sec @150MHz = 2.4 TeraOPs/sec

"Image super-resolution using deep convolutional networks", Dong et al (2016)

SRCNN C++ Resource Description (Inference ONLY)




```
template < typename T, typename CT, int ISZ, int F1SZ, int L1SZ, int F2SZ,
           int OSZ, int SB1, int SB2, int SB3, int CBW, int BBW, int P1BW,

struct SRCNN_TOP {
  SRCNN_REG_AXI4L <CT, ISZ, F1SZ, L1SZ, CBW, BBW>    reg0; /// AXI4 slave
  SRCNN_REG_AXI4L <CT, L1SZ, F2SZ, L2SZ, CBW, BBW>   reg1; /// AXI4 slave
  SRCNN_REG_AXI4L <CT, L2SZ, F3SZ, OSZ, CBW, BBW>    reg2; /// AXI4 slave

  CNN_CONV_CORE   <T, CT, ISZ,   F1SZ, L1SZ, SB1, CBW, BBW, P1BW>  core0;
  CNN_CONV_CORE   <T, CT, L1SZ, F2SZ, L2SZ, SB2, CBW, BBW, P2BW>  core1;
  CNN_CONV_CORE   <T, CT, L2SZ, F3SZ,  OSZ, SB3, CBW, BBW, P2BW>  core2;

  AXI_ST::SlaveFSM    in_sif;      /// AXI_ST input (slave interface)
  AXI_ST::MasterFSM out_mif;     /// AXI_ST output (master interface)

  void Run(AXI4L::CH *axi_rif0, AXI4L::CH *axi_rif1, AXI4L::CH *axi_rif2,
     AXI_ST::CH *axi_in, AXI_ST::CH *axi_out); /// SRCNN top function
};
```

CNN Model Params (# channels, filter size, bit-widths)

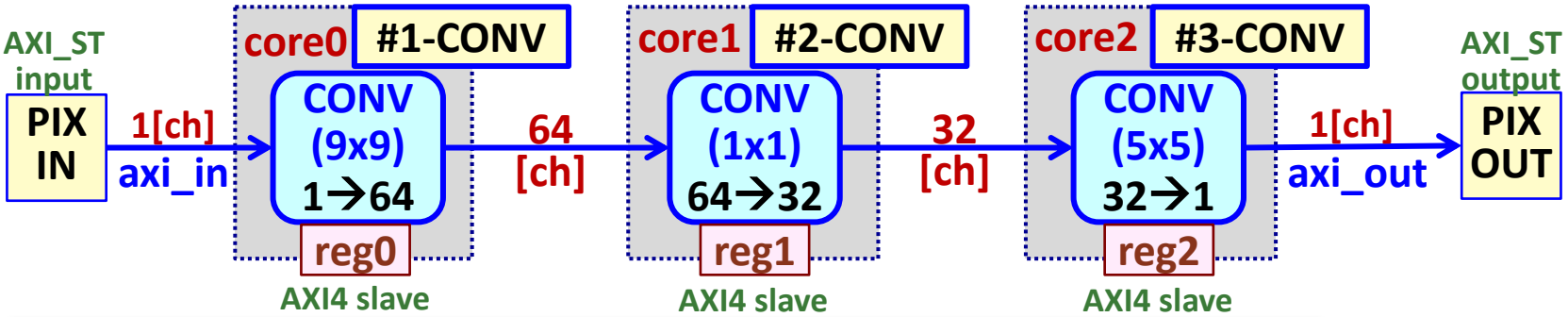
CONV coefs Register I/F

CONV core functions

AXI-Stream IN/OUT I/F

SRCNN Top Function

SRCNN Top C++ Dataflow Description




```
template < . . . > void SRCNN_TOP< . . . > :: Run (
AXI4L::CH *axi_rif0, AXI4L::CH *axi_rif1, AXI4L::CH *axi_rif2,
AXI_ST::CH *axi_in, AXI_ST::CH *axi_out) {
      int reset = reg0.reset && reg1.reset && reg2.reset;
      int initialized = reg0.initialized && reg1.initialized && reg2.initialized;
      T v0[ISZ], v1[L1SZ], v2[L2SZ], v3[OSZ];
    BIT vld0 = in_sif.read( axi_in, initialized, &v0 );
    BIT vld1 = core0.Run( v0, v1, reg0, reset && !initialized, vld0, 0 );
    reg0.fsm(axi_rif0);
    BIT vld2 = core1.Run( v1, v2, reg1, reset && !initialized, vld1, 0 );
    reg1.fsm(axi_rif1);
    BIT vld3 = core2.Run( v2, v3, reg2, reset && !initialized, vld2, 0 );
    reg2.fsm(axi_rif2);
    out_mif.write(axi_out, vld3, v3[0] );
  }
};
```

PIX Input

#1-CONV

#2-CONV

#3-CONV

PIX Output

CONV-Core C++ Dataflow Description

```
template < typename T, typename CT, int ISZ, int FSZ, int OSZ, int SBITS,
           int CBW, int BBW, int PBW >   struct CNN_CONV_CORE
{
  VectorLineBuffer <T, ISZ, FSZ, MAX_WIDTH, PBW>  lbuf;
  VectorShiftRegWindow <T, ISZ, FSZ, FSZ, PBW>     srw;
  int Kernel(T *sr, T *weight);
  BIT Run ( T din[], T dout[], SRCNN_REG_AXI4L<CT, ISZ, FSZ, OSZ, CBW, BBW> &reg,
       int init, int invalid ) {
    ...
    lbuf.UpdateData101(din, lb1, reg.width, reg.height, lbvld);
    srw.UpdateData101(lb1, lbvld, inValid, sr1, srvld, px, reg.width);
    outValid = srvld[FSZ*FSZ / 2];
    for (int i = 0; i < OSZ; i++) {   /// OUT channel for-loop
     if (outValid) {
        T sum = 0;
        for (int j = 0; j < ISZ; j++) {    /// IN channel for-loop
         sum += Kernel(&sr1[j*FSZ*FSZ], &reg.weight[(i*ISZ + j)*FSZ*FSZ]);
        }
        sum = DESCALE(sum, SBITS) + reg.bias[i];
        dout[i] = ReLU(sum);  /// ReLU
     }
     else { dout[i] = 0; }
    }
  }
};
```

CONV (FSZxFSZ) ISZ→OSZ

Line Buffer Shift Register

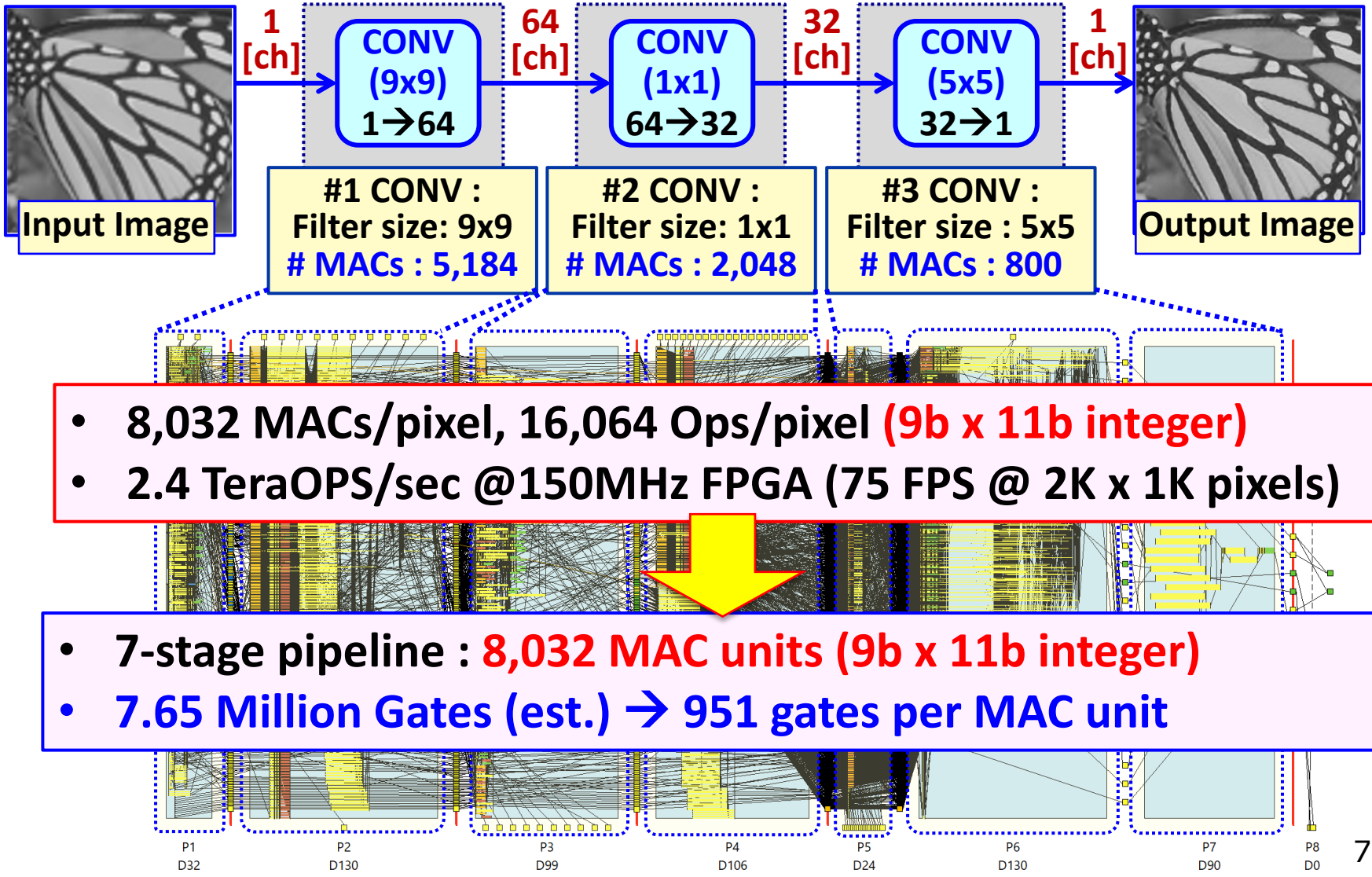
ISZ : # input channels
OSZ : # output channels
FSZ : filter size

CONV kernel code

```
int Kernel(T *sr, T *weight) {
   T sum = 0;
   for (int i = 0; i < FSZ*FSZ; i++) {
      sum += sr[i] * weight[i];
   }
   return sum;
}
```

All loops are completely unrolled in HW implementation
→ ISZ x OSZ Instances of CONV circuits

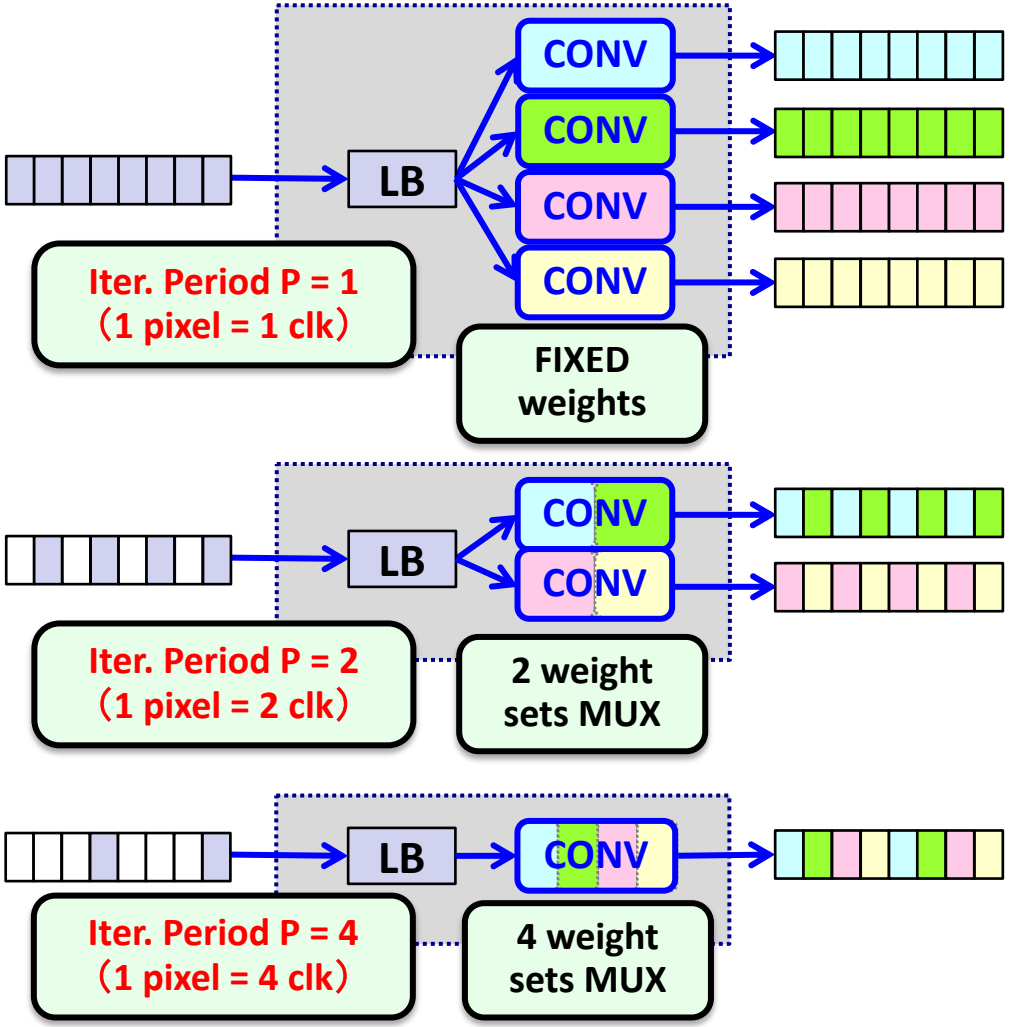
SRCNN Inference Hardware Synthesis Results



- 8,032 MACs/pixel, 16,064 Ops/pixel (9b x 11b integer)
- 2.4 TeraOPS/sec @150MHz FPGA (75 FPS @ 2K x 1K pixels)

- 7-stage pipeline : 8,032 MAC units (9b x 11b integer)
- 7.65 Million Gates (est.) → 951 gates per MAC unit

Time Multiplexing CNN Hardware



Repeated CONV operation pattern allows simple speed/area tradeoff using time multiplexing with minimal HW overhead !!

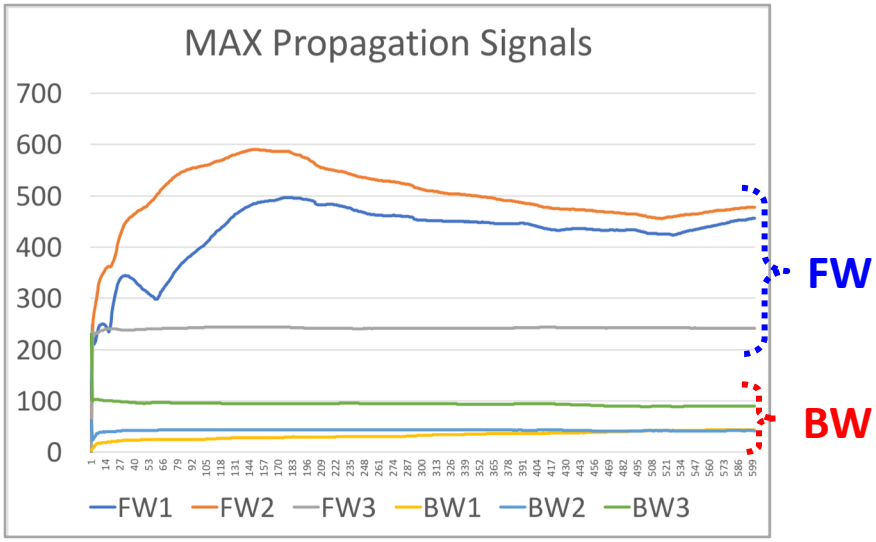
Iteration Period	# MACs	# Gates
P = 1	8,032	7.647 M
P = 2	4,016	3.734 M
P = 4	2,008	1.923 M
P = 8	1,004	1.002 M

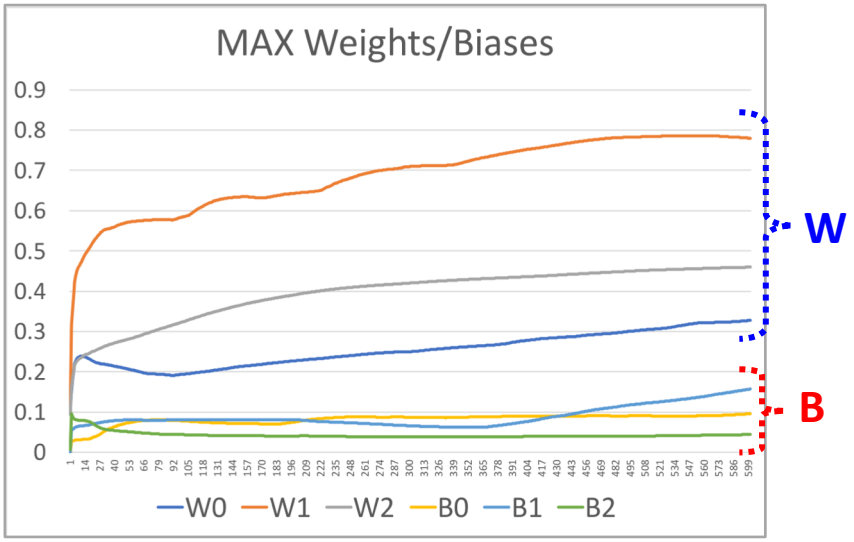
Numerical Issues in Deep Learning Training

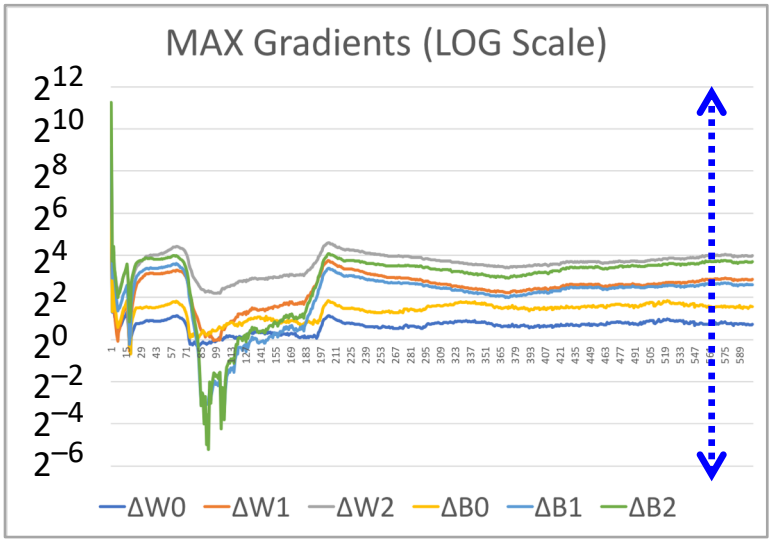
- **Inference : integer (fixed point) operations are sufficient**
  - Narrow dynamic range : FW propagation signals, weights
  - Ideal for FPGAs (INT8 typical in classification problems)
  - SRCNN is more sensitive to bit-width (9b x 11b required)
- **Training : need floating point operations**
  - Large (unpredictable) dynamic ranges : FW/BW prop. signals, weights, gradients → *NEXT PAGE*
  - Fixed point implementation requires larger bit-widths and manual setting of decimal positions and scaling factors
  - On FPGA, standard FPU implementation is very inefficient compared to Fixed Point [1] : 5X DSPs, 11X LUTs, 7.5X latency
- **Approach : *shared exponent* on group of fixed-point data**
  - "Block Floating Point" (BFP) for FFT [2]
  - "Dynamic Fixed Point" for Deep Neural Network Training [3]

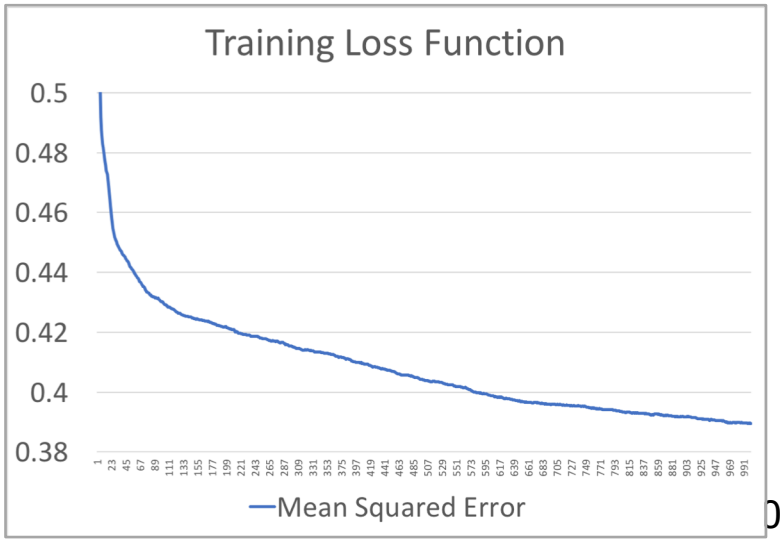
[1] Reduce Power and Cost by Converting from Floating Point to Fixed Point (Xilinx WP, 2017)
[2] A Block Floating Point Implementation for an N-Point FFT on the TMS320C55x DSP (TI WP, 2003)
[3] Training Deep Neural Networks with Low Precision Multiplications (Courbariaux/David, ICLR 2015)

Dynamic Ranges of Training Parameters/Signals









Floating Point vs. Dynamic Fixed Point (DFX)

- **Floating point (FP) : exponent on individual fraction value**

$$V = [2^{e_0} f_0 \;\; 2^{e_1} f_1 \cdots 2^{e_{N-1}} f_{N-1}]$$: Floating point vector

$$V_a \cdot V_b = \sum_{k=0}^{N-1} 2^{(ea_k + eb_k)} \cdot fa_k \cdot fb_k$$

Floating-Point accumulations are EXPENSIVE!

- **Dynamic fixed point (DFX) : common exponent on vector of fractions**

$$V = 2^e \cdot [f_0 \, f_1 \cdots f_{N-1}]$$: Dynamic fixed point (DFX) vector

$$V_a \cdot V_b = 2^{(ea+eb)} \cdot \sum_{k=0}^{N-1} fa_k \cdot fb_k$$

Can use INTEGER MAC (Multiply-Accumulate) operations!!

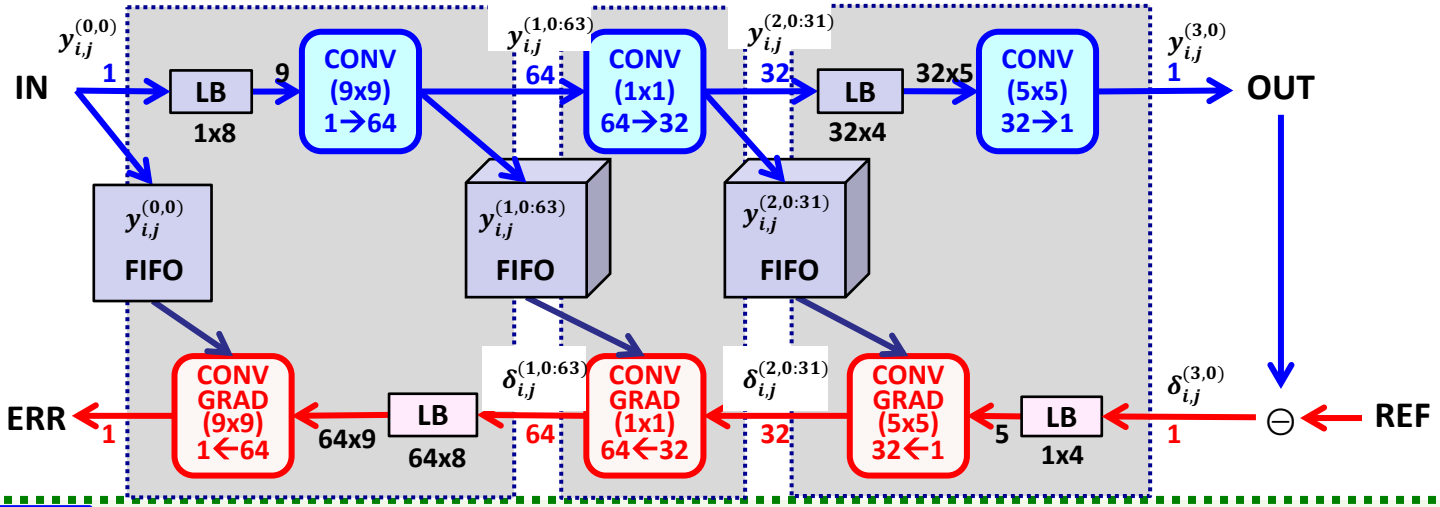
CNN Training DFX Implementation

- **DFX data format : shared exponent at each CONV layer**
  - **FW/BW propagation signals, CONV weight/bias**
  - **Signal saturation in case of fraction overflow**
- **DFX exponent adjustment scheme**
  - **DFX exponents remains FIXED during each batch**
  - **DFX exponent adjustment during weight-update :**
    - **INC DFX-exponent if overflow occurred**
    - **DEC DFX-exponent if Shift-Left does not overflow**
- **Gradients : Floating point format**
  - **Wide dynamic range of accumulated gradients**

→ ***Customized normalization scheme for area efficiency***

All DFX operations and custom FP operations designed on C++ and converted to RTL via C2RTL framework

SRCNN Training Computation



CONV (FW)

$$y_{i,j}^{(k,m)} = \sum_{n=0}^{N^{(k)}-1} \left(W^{(k,n,m)} * y_{i,j}^{(k-1,n)} \right) + b^{(k,m)}$$

$$W^{(k,n,m)} * y_{i,j}^{(k-1,n)} = \sum_{s=0}^{L^{(k)}-1} \sum_{t=0}^{L^{(k)}-1} W_{s,t}^{(k,n,m)} \, y_{i-s,j-t}^{(k-1,n)}$$

CONV (BW)

$$\delta_{i,j}^{(k-1,n)} = \sum_{m=0}^{M^{(k)}-1} \left(\overline{W}^{(k,n,m)} * \delta_{i,j}^{(k,m)} \right)$$

$$\overline{W}_{s,t}^{(k,n,m)} = W_{K-1-s,K-1-t}^{(k,n,m)}$$

DFX computation

GRAD

$$\Delta W_{s,t}^{(k,n,m)} = \delta_{i,j}^{(k,m)} * y_{s-i,t-j}^{(k-1,n)}$$

$$\Delta b^{(k,m)} = \sum \sum \delta_{i,j}^{(k,m)}$$

WEIGHT UPDATE

$$W \leftarrow W + \alpha \cdot \Delta W$$
$$\Delta W \leftarrow \beta \cdot \Delta W$$

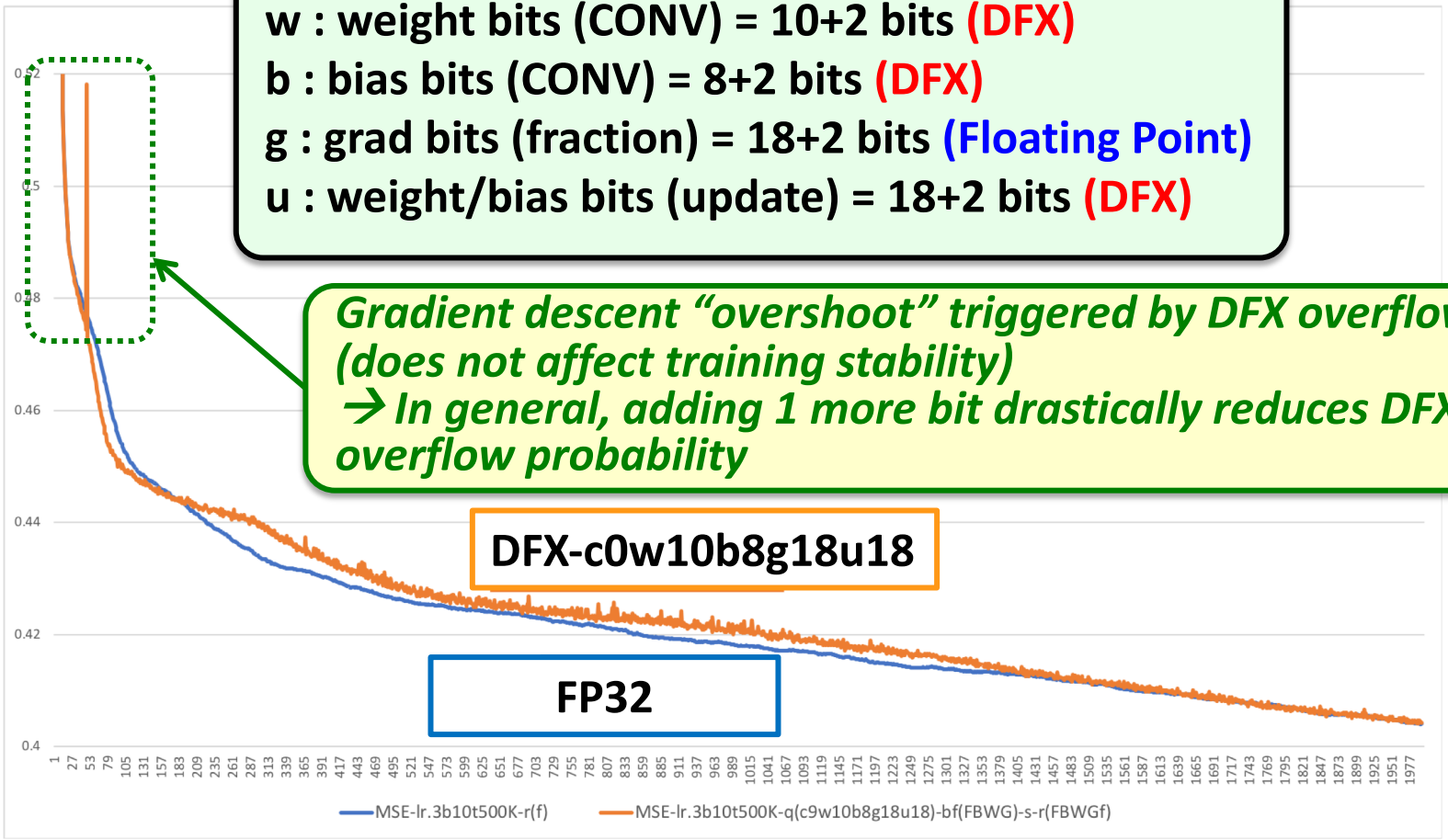
α : learning rate
β : momentum

Floating Point computation

SRCNN Training DFX Accuracy vs. FP32

c : FW/BW channel signal bits = 9+2 bits (DFX)
w : weight bits (CONV) = 10+2 bits (DFX)
b : bias bits (CONV) = 8+2 bits (DFX)
g : grad bits (fraction) = 18+2 bits (Floating Point)
u : weight/bias bits (update) = 18+2 bits (DFX)

Gradient descent "overshoot" triggered by DFX overflow (does not affect training stability)
→ In general, adding 1 more bit drastically reduces DFX overflow probability



SRCNN (Training) RTL Synthesis Results

Design (SRCNN-mini1)	# filters			# MAC units	Gates
	9x9	1x1	5x5		
Inference	16	128	8	1,624 (Fixed-Point)	1.382 M
Training	16	128	8	1,624 * 2 (DFX) + 1,624 (FP)	5.474 M

Verified on Xilinx ZU9EG @ 87MHz (ZU102 board)

9x9	1x1	5x5	# MAC units	Gates
6	24	4	610 * 2 (DFX)+ 610 (FP)	2.088 M

SRCNN Training HW	Clock Freq.	Exe. time	Power
GPU (GTX1080i)	1.6GHz	884 sec (x6.75)	58W (x40.0)
C2RTL/FPGA	50MHz (max:87MHz)	131 sec (x1.00)	1.45W (x1.0)
C2RTL/CPU	3.6GHz	19,467 sec (x148.60)	-

Summary

- **Deep Learning Training hardware : limited to CPU/GPU**
  - **Floating-point arithmetics (FP32, FP16) → difficult to implement high density/low latency FPUs on FPGA**
  - **Evolving DNN topologies and use-cases → DNN Training workload demands will continue to increase dramatically**
- **Dynamic Fixed Point (DFX) for FPGA-based CNN Training**
  - **Shared exponent on vector of fraction data → vector inner-product computation on INTEGER FORMAT!**
  - **Comparable training accuracy vs. FP32 training**
  - **6.75X faster, 40X power efficient than GPU**
- **Future works**
  - **Automatic Training HW generation from DL frameworks using C2RTL framework for CNN/RNN applications**
  - **Design space exploration of CNN/RNN inference engines**

Thank You for Your Attention!

Tsuyoshi Isshiki

isshiki@ict.e.titech.ac.jp

Dept. Communications and Computer Engineering

Tokyo Institute of Technology