FIE : A Flexible Inference Engine for Deep Learning

LEI CHENG,

ACM Reference Format:
Lei Cheng. 2020. FIE : A Flexible Inference Engine for Deep Learning. 1, 1 (December 2020), 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 PRELIMINARIES

This section clarifies the terminology and notation used in the paper. A DNN model consists of many layers. Each layer performs a specific function. We briefly review the operations supported by the FIE accelerator. For a detailed description of these operations, please refer to [13, 39]. Table 1 summaries the parameters and notations used in this paper.

Convolution. A convolution layer takes a 3-D feature tensor (or feature map) A [1] and a 4-D weight tensor F as inputs, and produces another 3-D feature tensor B for the next layer. The three dimensions of A are denoted as channel (C), width (W), and height(H). The three dimensions of B are denoted as channel (M), width (W'), and height(H'). The weight tensor is a collection of M filters (or kernels). Each filter is used to produce one channel for the output tensor, and it has a shape of $C \times K \times K$.[2] A filter slides over the input feature tensor with a stride of S_x over the width direction, and with a stride of S_y over the height direction.

Pooling. A pooling operation downsamples an input feature tensor by performing maximum or average operations to its non-overlapping windows. The shape of the pooling window is $P_x \times P_y$, and the strides of the pooling operation are S_x and S_y.

Normalization. One problem of DNN is that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. *Batch normalization* [21] is an effective method for this problem. In batch normalization, every input data x is scaled and shifted according to Eq.(1), where parameters $\gamma, \beta, \sigma, \mu$ are all learned from training. ϵ is a small constant to avoid numerical problems. These parameters do not change for inference, we can rewrite the formula as $y = \beta_1 x + \beta_0$, where $\beta_1 = \gamma/\sqrt{\sigma^2 + \epsilon}$ and $\beta_0 = \beta - \gamma\mu/\sqrt{\sigma^2 + \epsilon}$.

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \beta \tag{1}$$

Activations. Matrix multiplication is a linear function. Without non-linearity, a DNN will collapse into a linear regression model. As a result, DNN models use non-linear activation functions. *Relu* is a popular activation function, it is defined in Eq.(2). Our accelerator has an efficient implementation for *Relu* and *LeakyRelu* (Eq.(3)) functions. For other

[1]For simplicity, this paper uses a batch size of 1.
[2]Again for simplicity, we assume a filter has the same width and height.

Author's address: Lei Cheng, , lcheng2@gmail.com.

Table 1. DNN Parameters and Notations

Parameters	Description
C	♯ of input feature tensor channels
W, H	input feature tensor width and height
A	input feature tensor data, $A \in \mathbb{R}^{C \times W \times H}$
M	♯ of output feature tensor channels
W', H'	output feature tensor width and height
B	output feature tensor data, $B \in \mathbb{R}^{M \times W' \times H'}$
K	filter width and height
F	weight data, $F \in \mathbb{R}^{M \times C \times K \times K}$
P_x, P_y	pooling window width and height
S_x, S_y	convolution or pooling strides
β_0, β_1	batch normalization parameters
a_i, b_i, $i = 0,\ldots,15$	piecewise linear interpolation parameters
$\mathbb{Z}_p$	p-bit quantization range
$\lfloor \cdot \rceil$	rounding to the nearest integer
$[\cdot]_{\mathbb{Z}_p}$	saturation to $\mathbb{Z}_p$ domain

activation functions, such as *tanh*, *sigmoid* , we use a piecewise linear interpolation for approximation. Piecewise linear interpolation function $f(x)$ is defined as $f(x) = a_i x + b_i$, where $x \in [x_i, x_{i+1})$ and where $i = 0, \ldots, 15$. Piecewise linear interpolation is accurate enough for DNN tasks, and it is used in [10].

$$Relu : y = \begin{cases} x & x > 0 \\ 0 & x <= 0 \end{cases} \tag{2}$$

$$LeakyRelu : y = \begin{cases} x & x > 0 \\ \alpha x & x <= 0 \end{cases} \tag{3}$$

Residual Operation. Residual operation is first proposed in Resnet [19]. It is implemented by adding the outputs of two layers, and it helps model training by back propagating much deeper.

Fully Connected Layer. Fully connected layers are implemented by matrix multiplications. Matrix multiplication is a special case of the convolution with $K = 1$, thus our accelerator supports fully connected layers.

1.1 Quantization

For energy efficiency, inference accelerators use quantization [7, 23] to convert the weight and feature data into the low precision fixed-point representation. Global quantization uses one set of quantization parameters for a tensor. Per-axis quantization [15] allows different quantization parameters with respect to the quantized dimension, and greatly improves the inference performance. Kernel-wise quantization [7] for weight data is a per-axis quantization with respect to the output channel. In kernel-wise quantization, every filter of the weight data has its own set of quantization parameters.

Eq. 4 is a popular p-bit quantization scheme for data set X. Let $\mathbb{Z}_p$ denote the range of the p-bit quantization, and $\lfloor \cdot \rceil$ denote rounding to the nearest integer. In Eq. 4, δ is the quantization offset, α is the quantization scale, and $[X]_{\mathbb{Z}_p} = \min(\max(\lfloor X \rceil, \min(\mathbb{Z}_p)), \max(\mathbb{Z}_p))$ denotes saturation to the $\mathbb{Z}_p$ domain.

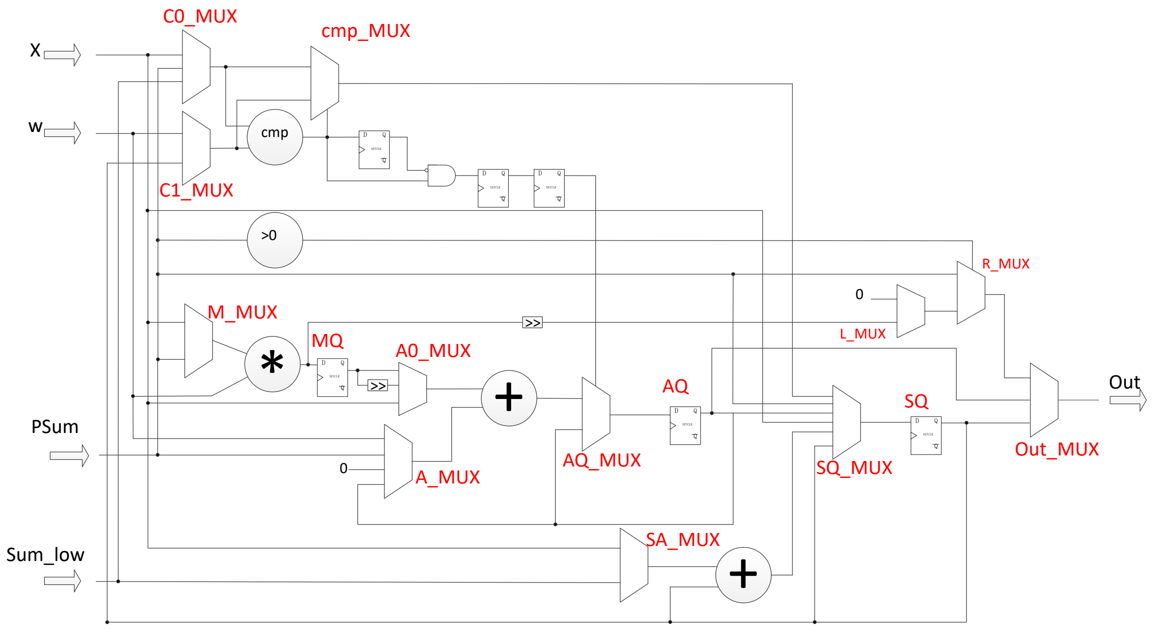


Fig. 1. Unified Computing Logic. This unit can implement convolution, pooling, batch normalization, and various activation functions. x is the feature data, w is the weight data (convolution weight, β_0, β_1, or piecewise linear interpolation weight for activation functions), PSum is the convolution partial sum of all previous layers, Sum_low is the partial sum from other UCL rows (to be explained). Since we use $8-$bit quantization, the bit width for weight and feature data is 8. We use $20-$bit for accumulations, which is the same bitwidth used for accumulations in [5].

$$\hat{X} = \alpha \Big[\frac{X-\delta}{\alpha}\Big]_{\mathbb{Z}_p} + \delta, \quad \delta = min\{T\}, \quad \alpha = \frac{max\{T\}-\delta}{2^p-1} \tag{4}$$

Our accelerator supports 8-bit kernel-wise quantization for weight data and 8-bit global quantization for feature data. Quantization range $\mathbb{Z}_8 = \{-128, \ldots, 127\}$. Quantization with 8-bit provides enough accuracy for inference, and is widely used [5, 14, 30].

2 COMPUTING ARCHITECTURE

In all current DNN accelerator implementations, each operation type has its own processing unit. There are different processing units for convolutions and activations, for example. However, ratios of the different processing units cannot be pre-determined, because ratios of different operations vary from layer to layer. As a result, some processing units have low utilizations.

Most DNN operations involve multiplications, and multipliers are precious resources because they take lots of area and power consumption both in ASIC and FPGA implementations. If we could reuse multipliers for different DNN operations as much as possible, we could improve the system efficiency. In FIE, we use multiplexers to config the processing unit to implement different DNN operations. With reconfigurability, multipliers, and some other units such as adders, can be used for many operations.

Unified Computing Logic (UCL), our proposed processing unit, can be configured to implement various operations, such as convolution, pooling, batch normalization, and activations. Our processing units can achieve 100% utilization

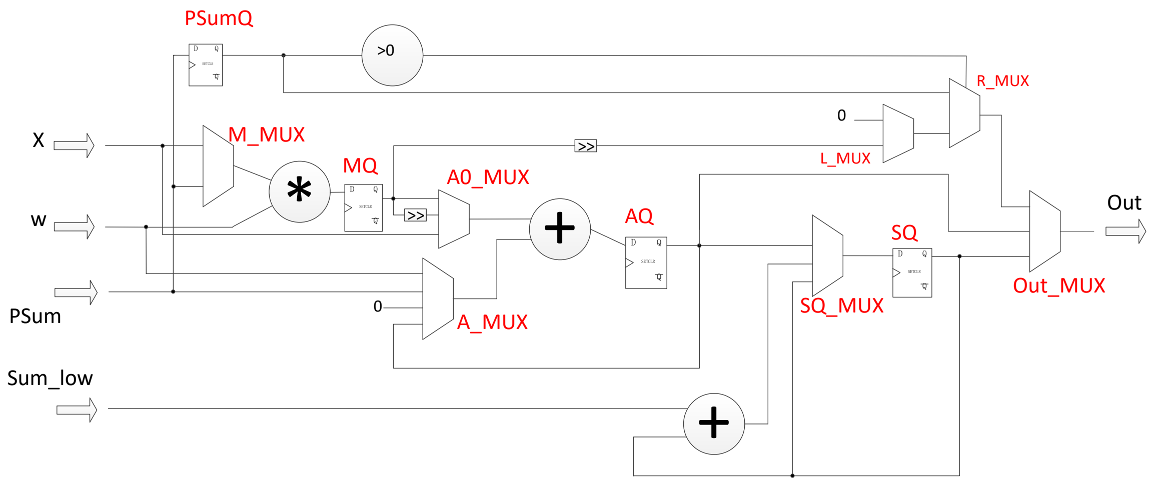


Fig. 2. Light Weighted Unified Computing Logic. This unit can implement convolution, average pooling, batch normalization, *Relu*, and *LeakyRelu*.

when there are enough memory bandwidth. Fig. 1 shows the UCL architecture. If a DNN model only uses *Relu* or *LeakyRelu* as activation functions, we can use a light-weighted UCL as in Fig. 2.

n UCLs form a **Shift Computing Group (SCG)**. *m* SCGs form a **Federated Computing Bank (FCB)**. One FCB consists of a matrix of UCLs, and each SCG is one row of the matrix. Feature data of one SCG can be shifted left, and they can also be shifted up to the next SCG. All UCLs in one SCG share a common set of weight data. We will show how the proposed architecture implements different operations in the following subsections.

2.1 Convolution Operation

Without loss of generality, this subsection illustrates how to compute the first output data B_{000} with UCLs (please refer Table 1 for notations). We assume zero padding and $K = 3$. B_{000} can be computed by Eq.(5). Let P_c denote the partial sum of B_{000} before input layer c, $P_0 = 0$, and Eq.(6) compute P_c recursively. By definition, $B_{000} = P_C$.

$$B_{000} = \sum_{c=0}^{C-1} \sum_{0 \le i,j<3} A_{cij} * F_{0cij} \tag{5}$$

$$P_{c+1} = P_c + \sum_{0 \le i,j<3} A_{cij} * F_{0cij}, c \ge 0 \tag{6}$$

Fig. 3 provides a step by step example of how to compute P_c for B_{000}. Within 3 clock cycles, all products of feature data and weights are computed. From the example, we can see that the computation has a pattern of shifting feature data left $K - 1$ times, then shifting them up once. Every shifting up operation starts a new partial sum computation. Shifting up operations have separate storage, so that the shifting up data do not mess up with the shifting left data (note from the example that the data of cycle 4 is a shift up of data from cycle 1). UCLs in the same SCG share a common set of weight data. The weight data are stored in SCGs, and broadcasted to UCLs each cycle.

Let's see some details of how the above computation is done in our accelerator. Fig. 4 shows the UCL configuration for convolution operations. UCLs can process Fig. 4a and Fig. 4b in parallel. For the above example, in the first 3 cycles,

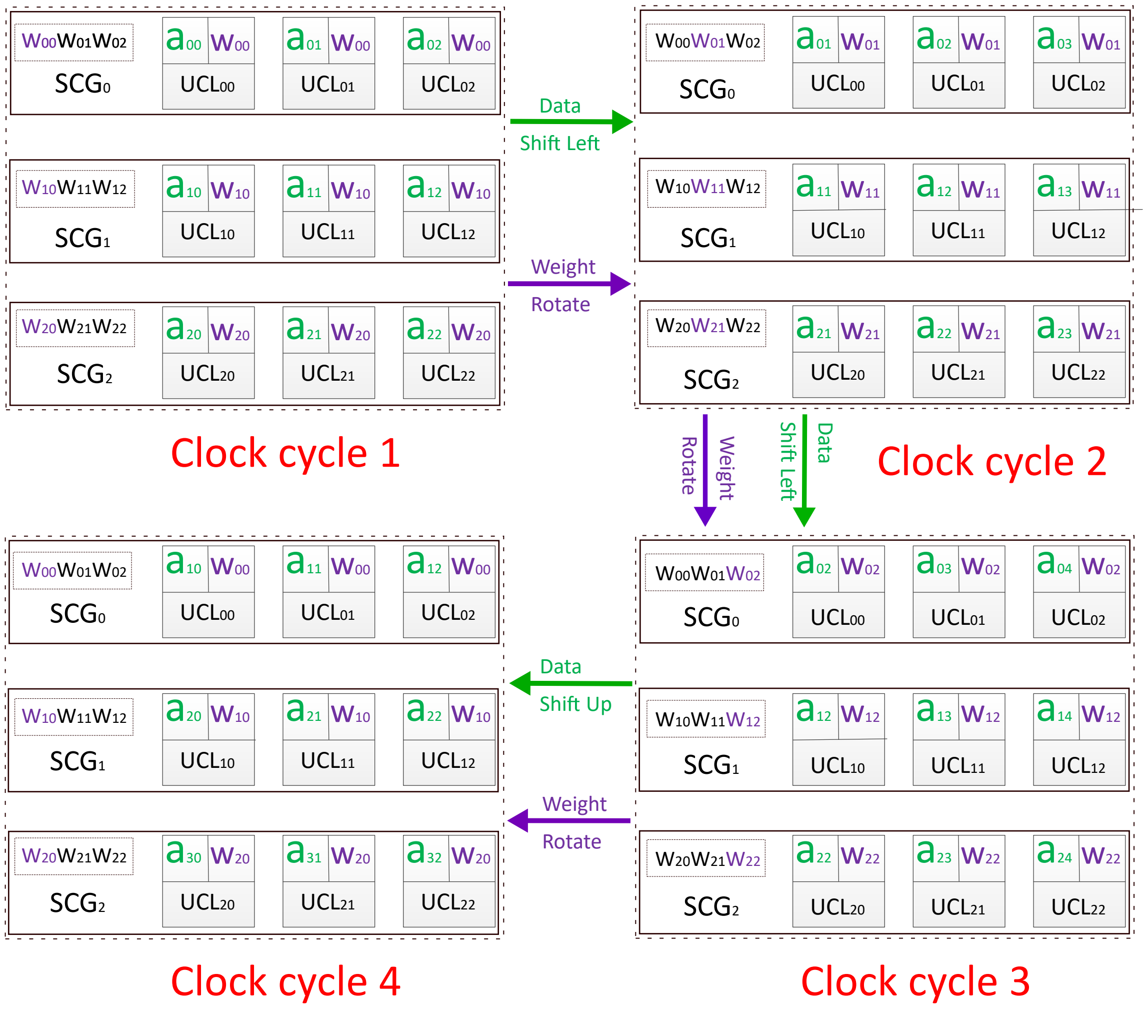
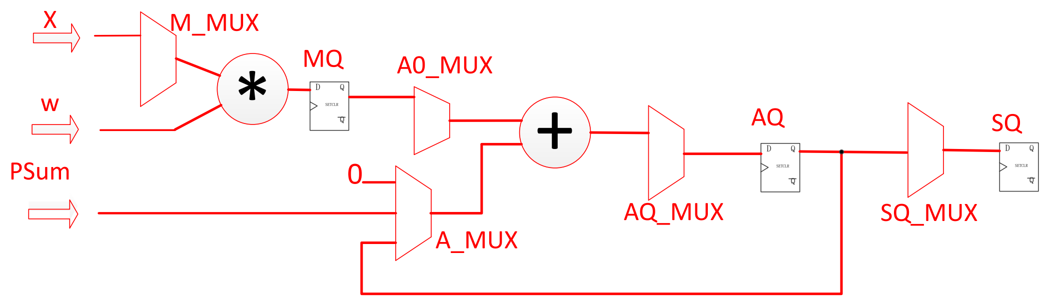


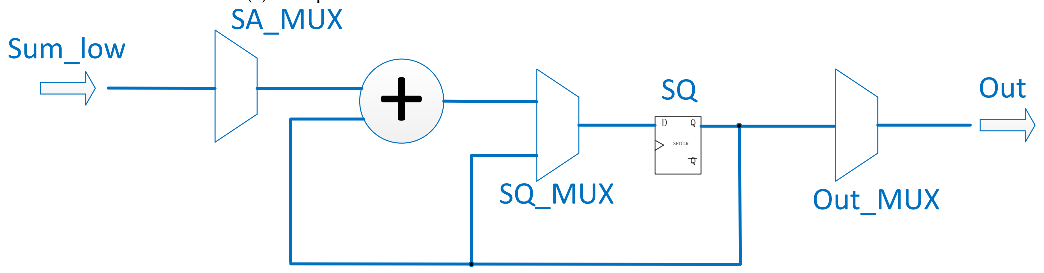
Fig. 3. Convolution Computing Steps. In this example, SCG_0, SCG_1, and SCG_2 are used to compute the first layer of B. Specifically, UCL_{00}, UCL_{10}, UCL_{20} are used to computing the first column of the layer. UCL_{01}, UCL_{11}, UCL_{21} are used to computing the second column of the layer, etc. In the first three cycles, UCL_{00} computes $a_{00} * w_{00}$, $a_{01} * w_{01}$, and $a_{02} * w_{02}$ respectively. UCL_{10} computes $a_{10} * w_{10}$, $a_{11} * w_{11}$, and $a_{12} * w_{12}$ respectively. UCL_{20} computes $a_{20} * w_{20}$, $a_{21} * w_{21}$, and $a_{22} * w_{22}$. Then, UCL_{00} , UCL_{10}, and UCL_{20} will add these products up to get the current iteration partial sum for B_{000}. Clock cycles 4, 5, 6 compute partial sum for B_{010}, etc.

Fig. 4a computes products for B_{000}. In cycles $4 - 6$, Fig. 4a computes products for B_{010}, and Fig. 4b computes partial sums for B_{000}. Table. 2 shows multiplexers' configurations for 7 clock cycles. There is a step 0 computing $a_{00} * w_{00}$ and store the result to MQ. In step 1, A_MUX of UCL_{00} selects Psum, and we set value $p_c + a_{00} * w_{00}$ to AQ. For UCL_{10} and UCL_{20}, A_MUXs' select input 0, and they do not add partial sums.

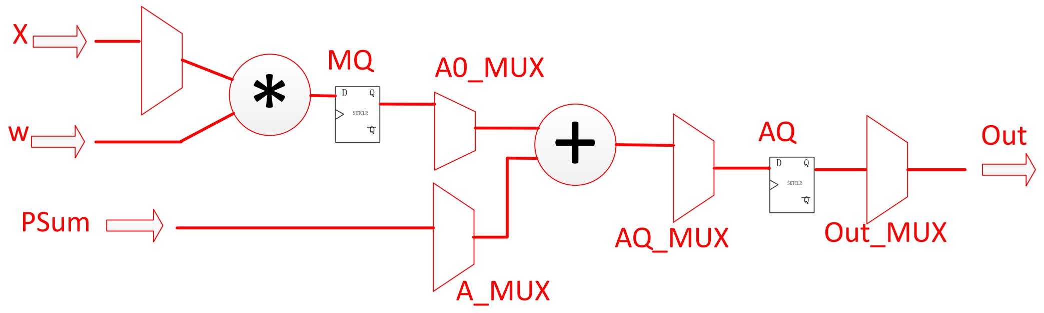
In step 2, A_MUX selects AQ. We set value $p_c + a_{00} * w_{00} + a_{01} * w_{01}$ to AQ. Similarly, in step 3, we set value $p_c + a_{00} * w_{00} + a_{01} * w_{01} + a_{02} * w_{02}$ to AQ. In step $4 - 6$, UCL_s computes products for B_{010}, and store them to AQ. In step 4, SQ_MUX selects AQ, so UCL_{00} will set value $p_c + a_{00} * w_{00} + a_{01} * w_{01} + a_{02} * w_{02}$ to SQ, UCL_{10} will set value $a_{10} * w_{10} + a_{11} * w_{11} + a_{12} * w_{12}$ to SQ, and UCL_{20} will set value $a_{20} * w_{20} + a_{21} * w_{21} + a_{22} * w_{22}$ to SQ. In step 5, UCL_{10}



(a) Computation Phase. First Phase of Convolution with $K > 1$.



(b) Summation Phase. Second Phase of Convolution with $K > 1$.



(c) Convolution Configuration for $K = 1$

Fig. 4. UCL Convolution Configuration. In the first phase, *UCLs* use K cycles to compute products of features and weights. In the second phase, *UCLs* use $K - 1$ cycles to compute the partial sums, and the port *Sum_low* is connected to the *Out* port of the UCL below. Convolution for $K = 1$ is simpler, and only has one phase.

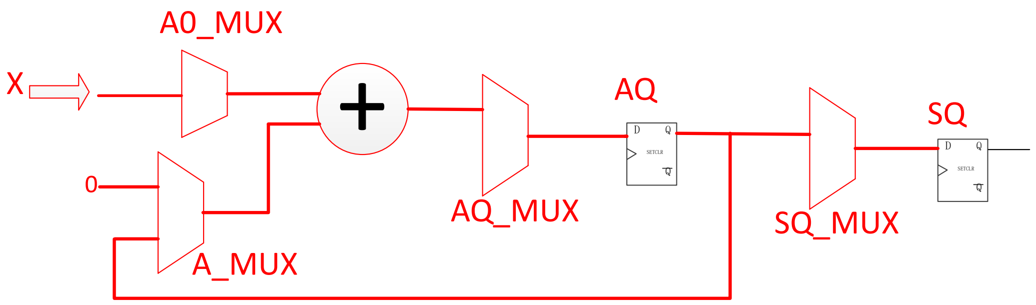
will add sum_low, which is SQ value of UCL_{20}, to its SQ, and UCL_{00} keeps SQ unchanged. In step 6, UCL_{00} will add sum_low (SQ of UCL_{10}) to its SQ , this is the result of the partial sum P_{c+1}. In step 7, UCL_{00} outputs the partial sum from SQ through *Out_MUX*. The computation of the products and the sums are processed in a pipeline mode, and the related components in the UCL are utilized fully. We omit the detailed computation steps for $K = 1$. They are simpler, and easy to figure out from Fig. 4c.

2.2 Pooling Operation

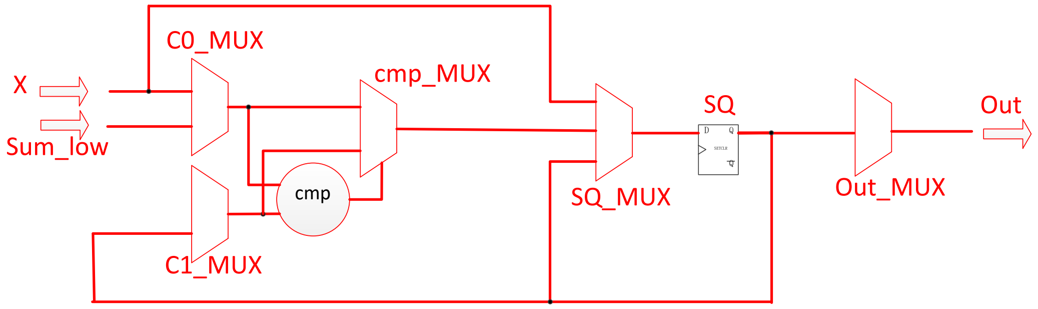
Fig. 5 shows UCL pooling operation configurations. Like convolution operations, average pooling operations are performed with two phases. In the first phase, each UCL adds up its input feature data, and store the sums into *AQ*. The second phase of average pooling is the same as that of the convolution operation (Fig. 4b), and it starts by transfering

Table 2. Convolution Multiplexer Details for UCL_{00}

step	A_MUX	SQ_MUX	SA_MUX	Out_MUX
1	Psum			
2	AQ			
3	AQ			
4	Psum0	AQ		
5	AQ	SQ		
6	AQ	SADD	Sum_low	
7	Psum0	AQ		SQ



(a) First Phase of Average Pooling. The second phase is the same as that of convolution (Fig. 4b).



(b) Maximum Pooling. There are also two phases. In the first phase, port x is used. In the second phase, port Sum_low is used.

Fig. 5. UCL Pooling Configuration.

data from AQ to SQ, then adds the SQ from the UCL below through port Sum_low. The average pooling operation only computes the sum of the pooling window, and it does not divide the sum by the number of elements. The division is transferred to parameters of other operations by the compiler. For example, the compiler can divide β_0, β_1 of the related batch normalization layer by the number of elements in the pooling window.

The max pooling operation is supported with the UCL configuration as shown in Fig. 5b. The x port is the input data, and the Sum_low port is connected to the Out port of the UCL below. The max pooling operation is also a two-phase operation. In the first phase, it computes the maximum of the same row. In the second phase, computes the maximum of P_y rows. In the first phase, port Sum_low is not used, and $C0\_MUX$ only selects x. In the first iteration, SQ_MUX selects x, and sends it to SQ. In the following iterations, SQ_MUX selects the maximum of x and SQ from the cmp_MUX

output. In the second phase, port x is not used, and $C0\_MUX$ only selects Sum_low. SQ_MUX selects the maximum of its SQ and Sum_low. After $P_x + P_y$ cycles, the pooling result is presented on the Out port.

2.3 Batch Normalization

Batch normalization computes function $y = \beta_1 x + \beta_0$. In Fig. 6, the first cycle computes $\beta_1 x$, and stores the result to MQ. In the second cycle, $w = \beta_0$, and the product from MQ is first shifted right by $>>$, then added with β_0. We have to shift $\beta_1 x$ because both values are quantized. β_1 is quantized by left shifting s bits to make it an 8−bit integer (the shifting amount s is determined by the compiler). To make the result of $\beta_1 x$ still be properly quantized, we have to shift result right by s bits. The value of β_0 is quantized the same way as x (the input feature data), so it can add with $\beta_1 x$ directly.

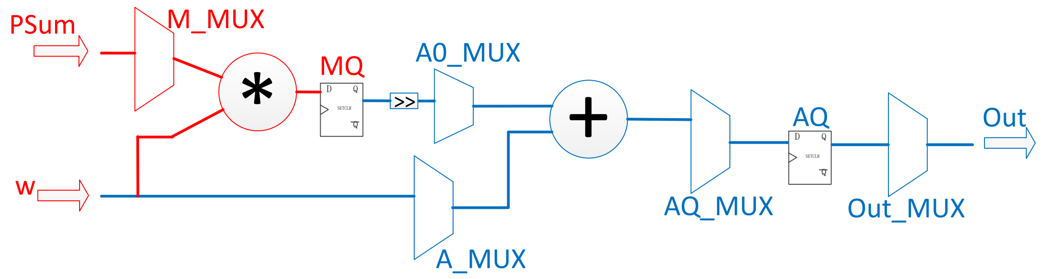


Fig. 6. Batch Normalization. This operation takes two cycles. In the first cycle (path in red), $PSum$ is the input data x, $w = \beta_1$. In the second cycle (path in blue), $w = \beta_0$. The third cycle generates the output through port Out, and starts next batch normalization computation.

2.4 Activation Functions

UCL has dedicated support for Relu and LeakyRelu operations.They are supported with L_MUX, R_MUX, and Out_MUX in Fig. 1.

Other activation functions are supported with the piecewise linear interpolation method (Fig. 7). Each linear function $a_i x + b_i$ takes three cycles. The second and third cycles compute value of $a_i x + b_i$, and these two parts are similar to the batch normalization computation. The first cycle computes whether $x \in [x_i, x_{i+1})$ is true or not. Piecewise linear interpolation implies x_i's are sorted in the ascending order. When they are compared with x, the result will be a series of 0's followed by a series of 1's. The reset value of CQ is 0. Gate $Range\_detect$ detects the pattern of 0 followed by 1, and it can find out the index i such that $x \in [x_i, x_{i+1})$. The value is propagated to the selector input of AQ_MUX two cycles later through two flip-flops. Gate $Range\_detect$ only generates 1 for the correct range, so AQ_MUX only selects $a_i x + b_i$ for the correct range. For all other cases, AQ_MUX selects input from AQ. After 48 $(16 * 3)$ cycles, we propagate results to SQ and to port Out.

2.5 Quantization

For energy efficiency, FIE implements the quantization with shift operators. From Eq. 4, the quantization involves both additions and multiplications, which are all expensive. FIE assumes $\delta = 0$, and $\alpha = 1/2^s$ where s is an integer. With these two assumptions, the equation becomes $\hat{X} = [X \times 2^s]_{\mathbb{Z}_p}/2^s$, and the quantization can be implemented with shift operations. FIE uses the quantized value $[X \times 2^s]_{\mathbb{Z}_8}$, and the interface to FIE is responsible for converting between data and quantized values.

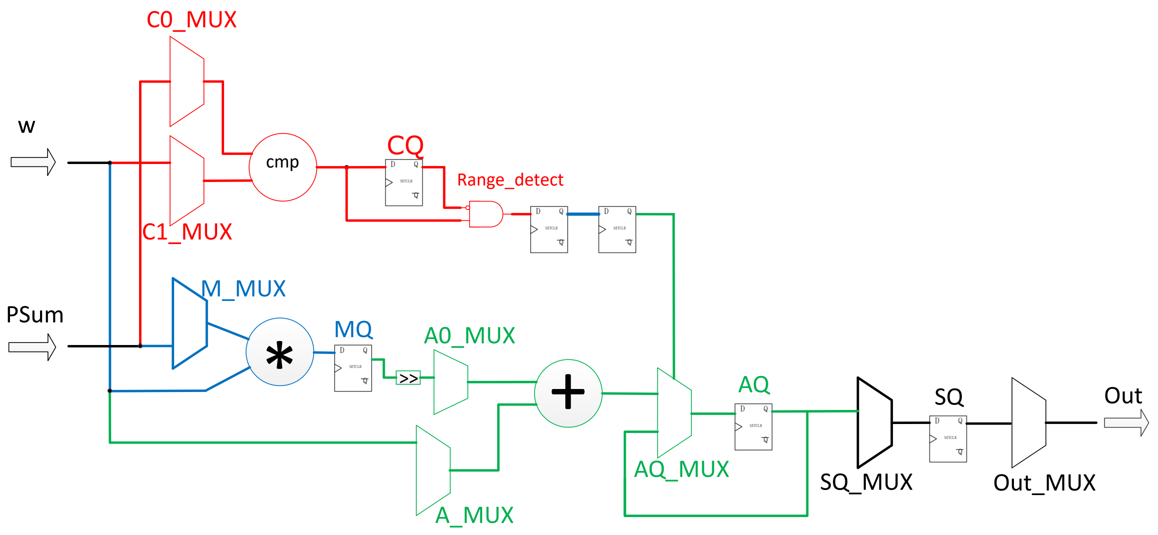


Fig. 7. Piecewise Linear Interpolation for Activation Functions. Each linear function takes three cycles. The first cycle shows in red, and it decides whether or not the current function is used as the result. The second cycle shows in blue, it computes $a_i x$. And the third cycle shows in green, and it computes $a_i x + b_i$. $w = x_i, a_i, b_i$ for these cycles respectively. Input x comes from port $PSum$.

FIE applies 8-bit global quantization for feature data. The input feature tensor A of a convolution operation has a shift value s_A; similarly, the output tensor B, which is the input tensor of the next layer, has a shift value s_B. FIE applies 8-bit kernel-wise quantization for weight tensor F, which means each filter F_i $(0 \leq i < M)$ has a shift value s_i. FIE computes ith layer of B by convoluting over input A and filter F_i. Since the quantization shifts left A by s_A and F_i by s_i, the convolution of A and F_i shifts results left by $s_A + s_i$. After the convolution operation is completed for ith layer of B, the UCL shifts results right by $s_A + s_i - s_B$. As a result, data of B are properly shifted left by s_B.

Given data X (could be A, F_i, or B), we use Mean Squared Error (MSE) method to compute its shift value s. Specifically, we search the shift value s to minimize $\mathbb{E}\|X - \hat{X}\|$. To search s, we estimate an initial value s_0 based on the maximum value of X, then we search around s_0 for 8 numbers, and choose the value with the smallest MSE.

3 MEMORY HIERARCHY

DNN algorithms are notoriously memory hungry. Designing an efficient memory hierarchy is very important for a successful accelerator. We adopt a 4-level memory hierarchy system for the FIE accelerator. Our memory hierarchy improves the system efficiency in the following ways. (1) FIE keeps the data movement inside local hierarchy as much as possible; (2) FIE accesses SDRAMs in large blocks. Our top level hierarchy are UCL data buffers, and they help convolution operations by shifting data between adjacent UCLs; the second level hierarchy are FCB input buffers, and they help to reuse data for different filters of a convolution operation, and they implement prefetching for continuous computations; the third level hierarchy are global buffers, and they are used as caches for SDRAMs to improve power consumptions and access delays; the bottom level hierarchy are SDRAMs, and all FCBs collaborate so that data accesses to SDRAMs are always in large blocks. We will describe them in detail in the following subsections.

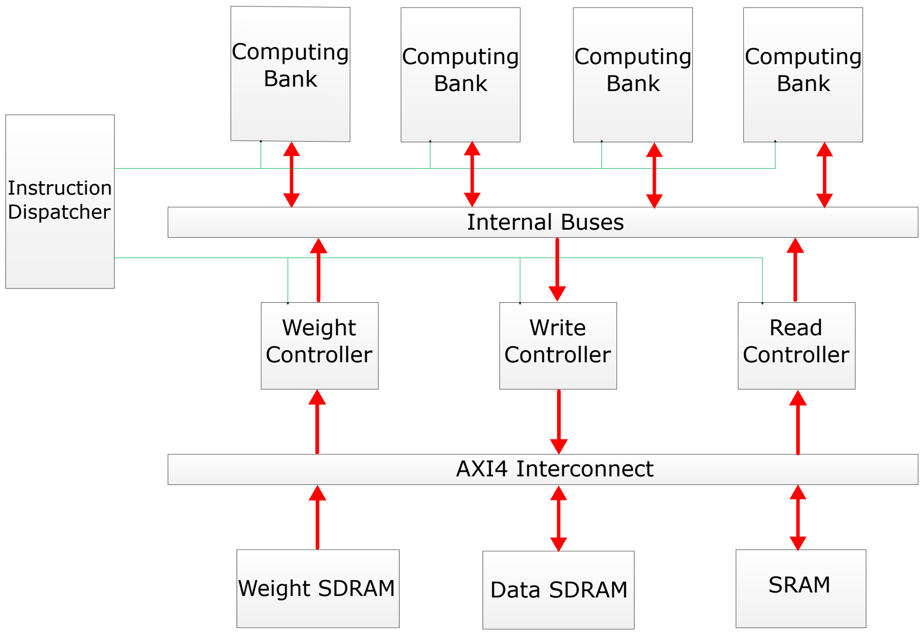


Fig. 8. Top Module Architecture.

3.1 Bottom Level Memories

The bottom level of the memory hierarchy is the external memory. The third level of the hierarchy is the SRAM used as the global buffer (Fig. 8). Whenever the output size of a DNN layer is small enough to fit into the global buffer, the output data will be written into and read out from the global buffer. Both the external memory and the SRAM communicate with the accelerator through the AXI4 protocol. With the AXI4 protocol, memory connections can be very flexible. The SRAM is optional. We can also use multiple SRAMs as global buffers. The accelerator does not know who it is talking to, and it is just accessing AXI4 addresses. Whether it accesses an SRAM or an SDRAM is determined by the DNN model compiler. The compiler has the knowledge of all connected SDRAMs and SRAMs, and it is responsible to work out the memory layout for a DNN model. Data SDRAM and weight SDRAM could also be the same SDRAM. There could be multiple data SDRAMs and multiple weight SDRAMs connected. It is also possible that there is no weight SDRAM or no data SDRAM, but an internet connection instead. This is fine as long as the internet connection speaks AXI4 language. This is a viable option if the DNN model is small enough for its living data to fit into the SRAM.

3.2 Computing Bank Buffers

Computing banks are the core of our inference engine. We can have various number of FCBs. In our current implementation, we use 4 FCBs. Each FCB has an input buffer. These buffers are the second level of the memory hierarchy, and they store the input feature data currently used. All FCBs collaborate to solve a DNN problem.

Recall from the previous section that n is the number of UCLs in one SCG, m is the number of SCGs in an FCB. Let v denote the size of the output buffer in one UCL (to be explained in the next subsection). Fig. 9 shows how our FCBs work together for one layer of a DNN model. FCBs work in different modes according to the shape of the input data.

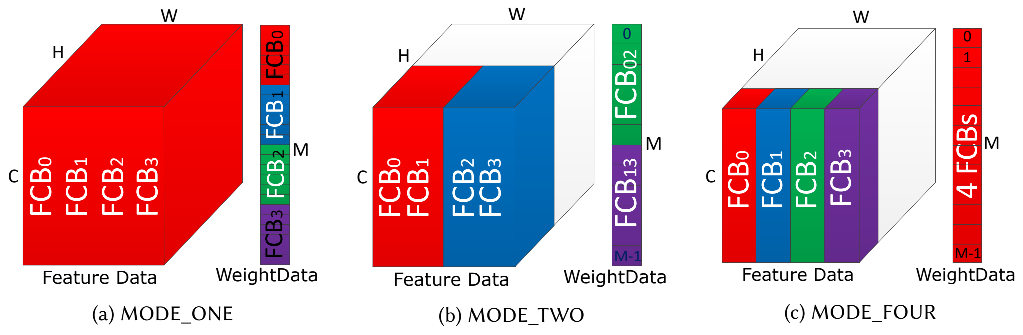


Fig. 9. Computing Bank Working Modes. FCBs work in MODE_ONE when $W \leq n$. FCBs work in MODE_TWO when $n < W \leq 2n$. FCBs work in MODE_FOUR when $2n < W \leq 4n$. These modes guarantee that accesses to SDRAMs are in chunks of $4n$ continous data.

If the width of the layer is no bigger than n, one row of the input feature data can fit into one SCG, FCBs work in MODE_ONE (Fig. 9a). In this mode, all FCBs share the input data. The input buffer of each FCB stores 1/4 of the input data, and all 4 buffers provide data for FCBs. Each FCB processes 1/4 of the weight data, and it produces 1/4 of the output data. For example, if there are 1024 filters in the weight data, each FCB is responsible for processing 256 filters.

FCBs work in MODE_TWO when $n < W \leq 2n$ (Fig. 9b). In this mode, FCB_0 and FCB_1 share input data, each buffer of FCB_0 and FCB_1 stores one half of the data, and each FCB is responsible for one half of the weight data. Note in Fig. 9a that, FCB_0 and FCB_2 share weight data , FCB_1 and FCB_3 share weight data.

FCBs work in MODE_FOUR when $2n < W \leq 4n$ (Fig. 9c). In this mode, each FCB has its own input data, and they share all the weight data. When $W > 4n$, the layer is processed by multiple passes of MODE_FOUR. For a layer with a shape of $C \times W \times H$, we will divide it into data chunks of $C \times n \times v$. As we will see from the next subsection, each FCB is capable of processing a chunk of data with shape $C \times n \times v$ (smaller shapes also work). With these three modes, the accelerator is guaranteed to read in a chunk of $4n$ continous data for each SDRAM access.

From the above description, we can find out the correct size of an FCB input buffer. If the largest layer channel number is l, the FCB buffer size should be $(n+2) * (v+2) * l/4$, where number 2 is for size 1 padding. In practice, padding numbers larger than 1 exist in the early stages of a DNN model. For early layers, C is very small, so early layer storage requirements are not our bottleneck. In our FPGA implementation, we use $l = 1024, n = 7$, and $v = 14$, and the FCB buffer size is $36kB$. Our accelerator is also capable of handling layers with channel numbers larger than $1k$. For example, if the channel number is $2k$, we can divide the computation into two parts. The first part computes for the top $1k$ channels, the second part computes for the bottom $1k$ channels, then we use a residual operation to add them together.

3.3 UCL Local Registers

With feature data stored in FCB input buffers, we explain how data are broadcasted to UCLs in this section.

Each UCL has three 8-bit registers. One register is used for receiving data from the broadcast, and the other two are used for shifting up and shifting left. Each UCL also has two 20-bit register files with v elements each. One register file is for computation, and the other is for output. These register files are used to store partial sums. Given an input feature data and a filter, we compute the convolution channel by channel. We save the partial sums of one input channel to register files, so they can be used for the next input channel computation.

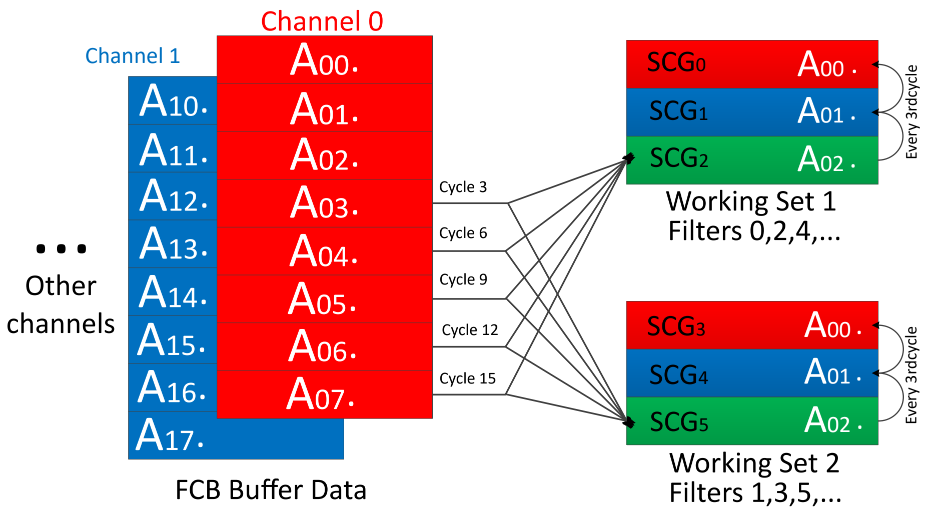


Fig. 10. Working Set Example. $A_{ij\cdot}$ denotes the jth row of the ith channel of the input data A. We show the first two channels of the input data. Working set 1 and working set 2 always receive the same input data, but different weight data.

We know from section 3.1 that we need K SCGs for a convolution operation, and K-SCG is enough for computation. We organize every consecutive K SCGs as a *working set*. Each working set is responsible for processing a subset of filters. Let's use Fig. 10 as an example. We assume $v = 8, m = 6$, and the FCB needs to process 256 filters. In this example, SCG_0, SCG_1, and SCG_2 form one working set. SCG_3, SCG_4, and SCG_5 form the other working set. Working set 1 is responsible for processing filters $0, 2, 4, \ldots, 254$. Working set 2 is responsible for processing filters $1, 3, 5, \ldots, 255$. Filter 0 and filter 1 are processed concurrently by working set 1 and 2 respectively. At the start of the processing, data $A_{00\cdot}$ is loaded into SCG_0 and SCG_3 , data $A_{01\cdot}$ is loaded into SCG_1 and SCG_4, and , data $A_{02\cdot}$ is loaded into SCG_2 and SCG_5. In the first two cycles, each SCG shifts data to the left. On the third cycle, they shift data up. So data from SCG_1 (SCG_4) are fed to SCG_0 (SCG_3), and data from SCG_2 (SCG_5) are fed to SCG_1 (SCG_4). SCG_2 and SCG_5 get data $A_{03\cdot}$ from the bus. Similarly, the FCB broadcasts data $A_{04\cdot}$, $A_{05\cdot}$, $A_{06\cdot}$, $A_{07\cdot}$ on cycles 6, 9, 12, and 15 respectively. Both working sets use cycles 16, 17, 18 to finish the computation for channel 0. And the FCB uses these three cycles to broadcast channel 1's data $A_{10\cdot}$ to SCG_0 and SCG_3, $A_{11\cdot}$ to SCG_1 and SCG_4, $A_{12\cdot}$ to SCG_4 and SCG_5. So the computation for channel 1 can be performed without interruptions. When we are on the last channel, we reload data from channel 0 for the next filter. When we start working on the last filter, we can start prefetching the next data chunk into the FCB input buffer. Specifically, in the first 3 cycles, the buffer storage spaces for data $A_{00\cdot}$, $A_{01\cdot}$ and $A_{02\cdot}$ are available for prefetching. Then after every 3 cycles, one more data storage space is available for prefetching.

Recall from section 3.3 and 3.4, batch normalization and activation functions get inputs from partial sums, instead of from FCB bus broadcasts. This is because we implement them as *fused* functions. After we finish the convolution operation, we immediately process batch normalization and activation functions before we write outputs to SRAMs or SDRAMs.

3.4 Weight Data

Weight data processing is much simpler for inference since they do not change. We know the order each FCB accessing weight data. The compiler sorts the weight data exactly in the order they are requested beforehand. As the weight data accesses are always consecutive, each FCB uses a FIFO to receive and broadcast the weight data. The weight controller has a small buffer to store weight data for the first few layers.

REFERENCES

[1] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, Max Baker, Tom Hawkins, Andrew Bell, John Thompson, Temesghen Kahsai, Garrin Kimmell, Jennifer Hwang, Rebekah Leslie-Hurd, Michael Bye, E. R. Creswick, Matthew Boyd, Mahitha Venigalla, Evan Laforge, Jon Purdy, Purushotham Kamath, Dinesh Maheshwari, Michael Beidler, Geert Rosseel, Omar Ahmad, Gleb Gagarin, Richard Czekalski, Ashay Rane, Sahil Parmar, Jeff Werner, Jim Sproch, Adrian Macias, and Brian Kurtz. 2020. Think Fast: A Tensor Streaming Processor (TSP) for Accelerating Deep Learning Workloads. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020.* IEEE, 145–158. https://doi.org/10.1109/ISCA45697.2020.00023

[2] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* 40, 4 (2020), 10–21. https://doi.org/10.1109/MM.2020.2996616

[3] William Chan, Navdeep Jaitly, Quoc V. Le, and Oriol Vinyals. 2015. Listen, Attend and Spell. *CoRR* abs/1508.01211 (2015). arXiv:1508.01211 http://arxiv.org/abs/1508.01211

[4] Y. Chen, J. Emer, and V. Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA).* 367–379.

[5] Y. Chen, T. Yang, J. Emer, and V. Sze. 2019. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 2 (2019), 292–308.

[6] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie. 2016. PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA).* 27–39.

[7] Y. Choukroun, E. Kravchik, F. Yang, and P. Kisilev. 2019. Low-bit Quantization of Neural Networks for Efficient Inference. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW).* 3009–3018.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

[9] Ben Dickson. [n.d.]. *The untold story of GPT-3 is the transformation of OpenAI.* https://bdtechtalks.com/2020/08/17/openai-gpt-3-commercial-ai/

[10] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA).* 92–104.

[11] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun. 2011. NeuFlow: A runtime reconfigurable dataflow processor for vision. In *CVPR 2011 WORKSHOPS.* 109–116.

[12] LLC Gisselquist Technology. [n.d.]. *FPGAs vs ASICs.* https://zipcpu.com/blog/2017/10/13/fpga-v-asic.html

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning.* MIT Press. http://www.deeplearningbook.org.

[14] Google. [n.d.]. *Edge TPU performance benchmarks.* https://coral.ai/docs/edgetpu/benchmarks/

[15] Google. [n.d.]. *TensorFlow Lite 8-bit quantization specification.* https://www.tensorflow.org/lite/performance/quantization_spec

[16] Cong Hao, Xiaofan Zhang, Yuhong Li, Sitao Huang, Jinjun Xiong, Kyle Rupnow, Wen-Mei Hwu, and Deming Chen. 2019. FPGA/DNN Co-Design: An Efficient Design Methodology for IoT Intelligence on the Edge. *CoRR* abs/1904.04421 (2019). arXiv:1904.04421 http://arxiv.org/abs/1904.04421

[17] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. 2020. Momentum Contrast for Unsupervised Visual Representation Learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).*

[18] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross B. Girshick. 2017. Mask R-CNN. *CoRR* abs/1703.06870 (2017). arXiv:1703.06870 http://arxiv.org/abs/1703.06870

[19] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* 770–778.

[20] M. Imani, M. Samragh Razlighi, Y. Kim, S. Gupta, F. Koushanfar, and T. Rosing. 2020. Deep Learning Acceleration with Neuron-to-Memory Transformation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA).* 1–14.

[21] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR* abs/1502.03167 (2015). arXiv:1502.03167 http://arxiv.org/abs/1502.03167

[22] ipu. [n.d.]. *GraphCore Intelligence Processing Unit IPU.* https://www.graphcore.ai/products/ipu

[23] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. 2017. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. *CoRR* abs/1712.05877 (2017). arXiv:1712.05877 http://arxiv.org/abs/1712.05877

[24] Redmon Joseph and Farhadi Ali. 2018. YOLOv3: An Incremental Improvement. *arXiv* (2018).

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1097–1105. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[26] I. Kuon and J. Rose. 2007. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26, 2 (2007), 203–215.

[27] M. Lenjani, P. Gonzalez, E. Sadredini, S. Li, Y. Xie, A. Akel, S. Eilert, M. R. Stan, and K. Skadron. 2020. Fulcrum: A Simplified Control and Access Mechanism Toward Flexible and Practical In-Situ Accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 556–569.

[28] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie. 2017. DRISA: A DRAM-based Reconfigurable In-Situ Accelerator. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 288–301.

[29] Warren Mcculloch and Walter Pitts. 1943. A Logical Calculus of Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics* 5 (1943), 127–147.

[30] NVIDIA. [n.d.]. *Jetson AGX Xavier: Deep Learning Inference Benchmarks*. https://developer.nvidia.com/embedded/jetson-agx-xavier-dl-inference-benchmarks

[31] NVIDIA. [n.d.]. *Jetson Nano: Deep Learning Inference Benchmarks*. https://developer.nvidia.com/embedded/jetson-nano-dl-inference-benchmarks

[32] NVIDIA. [n.d.]. *Jetson Nano Developer Kit and Modules*. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/

[33] Wei Ping, Kainan Peng, Andrew Gibiansky, Sercan Ömer Arik, Ajay Kannan, Sharan Narang, Jonathan Raiman, and John Miller. 2017. Deep Voice 3: 2000-Speaker Neural Text-to-Speech. *CoRR* abs/1710.07654 (2017). arXiv:1710.07654 http://arxiv.org/abs/1710.07654

[34] Alec Radford and Ilya Sutskever. 2018. Improving Language Understanding by Generative Pre-Training. In *arxiv*.

[35] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 779–788.

[36] F. Rosenblatt. 1958. The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain. *Psychological Review* (1958), 65–386.

[37] Williams Samuel, Waterman Andrew, and Patterson David. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52 (04 2009), 65–76. https://doi.org/10.1145/1498765.1498785

[38] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar. 2016. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 14–26.

[39] V. Sze, Y. Chen, T. Yang, and J. S. Emer. 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proc. IEEE* 105, 12 (2017), 2295–2329.

[40] Yu Wang, Gu-Yeon Wei, and David Brooks. 2019. Benchmarking TPU, GPU, and CPU Platforms for Deep Learning. *CoRR* abs/1907.10701 (2019). arXiv:1907.10701 http://arxiv.org/abs/1907.10701

[41] Pengfei Xu, Xiaofan Zhang, Cong Hao, Yang Zhao, Yongan Zhang, Yue Wang, Chaojian Li, Zetong Guan, Deming Chen, and Yingyan Lin. 2020. AutoDNNchip: An automated DNN chip predictor and builder for both FPGAs and ASICs. In *FPGA 2020 - 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2020 - 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays)*. Association for Computing Machinery, Inc, 40–50. https://doi.org/10.1145/3373087.3375306 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2020 ; Conference date: 23-02-2020 Through 25-02-2020.

[42] Hanchen Ye, Xiaofan Zhang, Zhize Huang, Gengsheng Chen, and Deming Chen. 2020. HybridDNN: A Framework for High-Performance Hybrid DNN Accelerator Design and Implementation. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE.

[43] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen Mei Hwu, and Deming Chen. 2018. DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs. In *2018 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2018 - Digest of Technical Papers*. Institute of Electrical and Electronics Engineers Inc., United States. https://doi.org/10.1145/3240765.3240801 37th IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2018 ; Conference date: 05-11-2018 Through 08-11-2018.