WP487 (v1.0) June 27, 2017

8-Bit Dot-Product Acceleration

By: Yao Fu, Ephrem Wu, and Ashish Sirasao

The DSP architecture in UltraScale™ and UltraScale+™ devices enhances convolution and matrix multiplication throughput for neural networks with a scalable performance of INT8 vector dot products. With the approaches described in this white paper, 1.75X–2X throughput can be reached compared to a traditional (naive) usage of DSP resource.

ABSTRACT

The Xilinx DSP48E2 slice can pack two parallel INT8 multiply-add operations to compute two dot products among three vectors, a computational pattern common in convolution and matrix multiplication for artificial intelligence acceleration. By exploiting parallelism inherent in neural networks, each DSP slice on average performs 1.75 (without using LUT) to 2 (with help of LUT resource) multiply-add (Y = A*B+C) or multiply-add-accumulate (Y += A*B+C) operations for INT8 vector dot products. As the rectified linear unit (ReLU) is a common activation function, the Xilinx DSP48E2 slice supports unsigned 8-bit (UINT8) data operands to enable an extra bit of data precision compared to INT8, achieving a 1.78X – 2X speed-up for dot products between a UINT8 data vector and an INT8 weight vector.

Reduced Numerical Precision in Neural Networks

In neural networks, matrix multiplication and convolution are some of the most memory- and compute-intensive operations. While neural-network data and weights have been modeled with the 32-bit single-precision floating-point format (fp32), researchers have successfully used 8-bit fixed-point integers (INT8) to reduce memory storage and bandwidth while maintaining good accuracy [Ref 1][Ref 2]. Not only does the INT8 format reduce memory bandwidth and storage, it also improves dot-product throughput with the Xilinx DSP48E2 slice [Ref 3].

Enhanced dot-product throughput speeds up both matrix multiplication and convolution. Matrix multiplication can be decomposed into a set of dot products, and benefits from faster dot-product computation. Convolution is a special case of matrix multiplication. For instance, two-dimensional convolution in neural networks, actually shorthand for *sum of two-dimensional cross-correlations,* is a dot product between an input data vector and a neural-network weight vector. Figure 1 represents the first stage of an example convolutional neural network (CNN) for image recognition. An RGB input image, separated into three input feature maps (IFMs), generates four output feature maps (OFMs)[(1)] with a set of filter weights. To clearly illustrate the relationship among IFMs, OFMs, and filter weights, the IFMs are laid out vertically, the OFMs are arranged horizontally, and at the intersection of each IFM-OFM pair is a set of dedicated 3x3 filter weights. Figure 1 represents how the 12 convolution filters create four different views (four OFMs) from a single RGB image (three IFMs). IFM elements broadcast horizontally to all OFM columns whereas output values are summed vertically from bottom to top. Each row of filter weights is applied to only one IFM; likewise, each column of filter weights influence only one OFM. Along each OFM column, each element in the OFM is a dot product of two 27-dimensional vectors because each of the three IFMs contributes nine elements to the data vector for a total of 27 elements, and the three 3x3 filters along an OFM column contribute 27 filter weights.

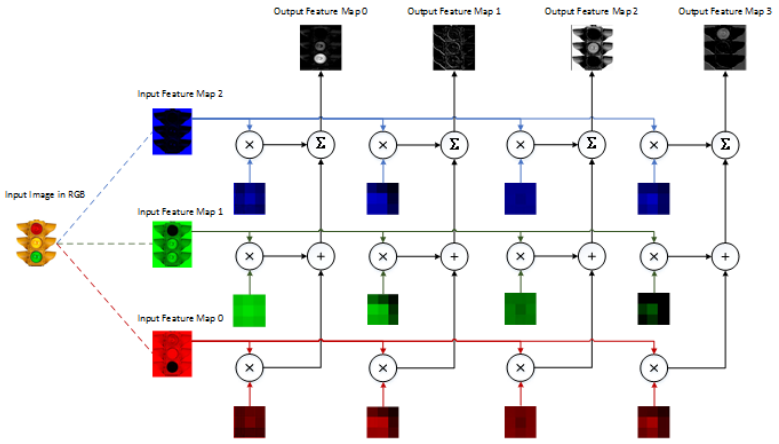

Figure 1: **2D Convolution in Image Recognition**

1. Only four OFMs are shown due to space limitation. Deployed CNNs typically generate 64 or more OFMs in the first convolutional layer.

2D Convolution in Image Recognition

The conv2d() algorithm in CNNs is a sum of 2D convolutions (more accurately cross-correlations), which creates D_2 OFMs from D_1 IFMs with D_1 D_2 filters. Specifically, for Y being a $B \times H_2 \times W_2 \times D_2$ output tensor, X being a $B \times H_1 \times W_1 \times D_1$ input tensor, and W being an $F_y \times F_x \times D_1 \times D_2$ filter weight tensor, Y = conv2d(X,W,S), where for job g in a batch of B jobs,

$Y[g,u,v,d_2]$ is the element at row u and column v in OFM $d_2 \in [1,D_2]$,

$X[g,i,j,d_1]$ is the element at row i and column j in IFM $d_1 \in [1,D_1]$,

W[y,x,d_1,d_2] is the filter element at row y and column x applied to IFM d_1 to produce OFM d_2, and

$S = (S_y,S_x)$, where S_y and S_x are respectively the vertical and horizontal strides.

In the Conv2d() example below, conv2d(), which is a nested loop that computes a sum of 2D convolutions for every OFM element. For each job g in a batch of B jobs, the function conv() called within conv2d() applies one particular set of weights $W[:,:,d_1,d_2]$ to one slice of the input tensor X, namely $X[g,:,:,d_1]$. (The colon symbol (':') denotes all valid values.) Any IFM element outside the $H_1 \times W_1$ range for the input tensor X is zero. Each output slice $Y[g,:,:,d_2]$ is obtained by calling conv() D_1 times (see *Conv() Called from Conv2d()*), once for each input slice, and summing the results.

Conv2d() in Convolutional Neural Networks

$\text{conv2d}(X, W, S)$ {

for g in 1 to B // each job

for $d_2 \in [1, D_2]$ // each OFM

$$Y[g,:,:,d_2] = \sum_{d_1=1}^{D_1} \text{conv}(X[g,:,:,d_1],\ W[:,:,d_1,\ d_2],[S_y,S_x])$$

return Y

Conv() Called from Conv2d()

$\text{conv}(X[g,:,:,d_1],\ H[:,:,d_1,\ d_2],[S_y,S_x])$ {

for c in 1 to W_2

for r in 1 to H_2

$$Z[r,c] = \sum_{u=-F_y/2}^{F_y/2} \sum_{v=-F_x/2}^{F_x/2} X[g, S_x r + u, S_y c + v, d_1] H[u + F_y/2, v + F_x/2, d_1, d_2];$$

return Z;

}

DSP48E2 for Low-Precision Neural Networks

Parallelism

With INT8 operands, each DSP48E2 slice contributes to two dot products in parallel, as opposed to one dot product for, as an example, INT18, provided that the two dot products share a common input vector (Figure 2). Using only three vectors to produce two dot products is reasonable in neural networks due to the high degree of parallelism in matrix multiplication and convolution.

1. For the matrix-matrix multiplication *WX*,
   a. each column vector in *X* produces two dot products with two row vectors in *W* (Figure 3A); alternatively,
   b. each row vector in *W* produces two dot products with two column vectors in *X* (Figure 3B).
2. For the matrix-vector multiplication *Wx*, the column vector *x* produces two dot products with two row vectors in *W*, a special case of 1.a in which the matrix *X* has only one column.
3. For convolution,
   a. one filter weight vector operates with two patches across all IFMs to produce two elements in the same OFM (Figure 4A); alternatively,
   b. one IFM patch goes through two convolution filters to create two elements in two different OFMs (Figure 4B).

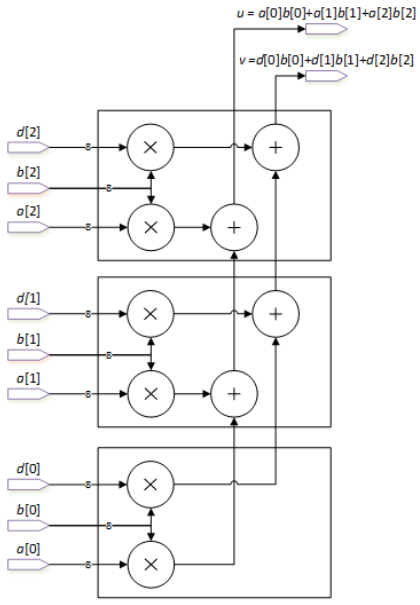

Figure 2: **Conceptual View of Two Parallel INT8 Dot Products Sharing a Common Input Vector**

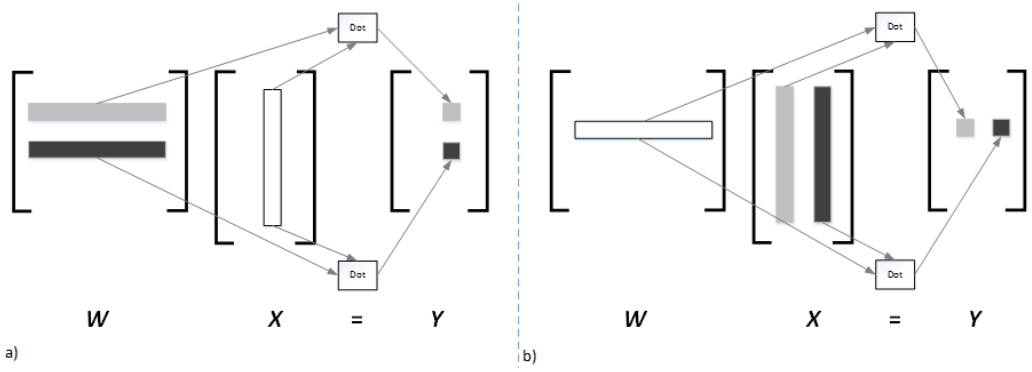

Figure 3: **Matrix-Matrix and Matrix-Vector Multiplication**

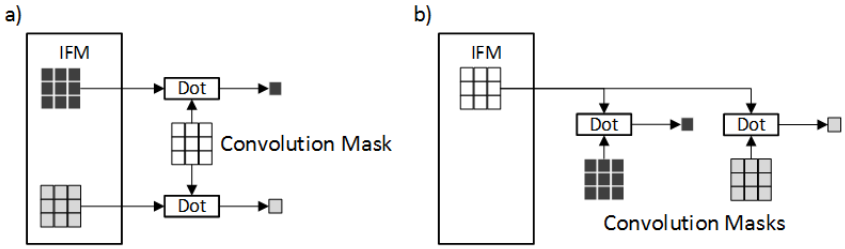

Figure 4: **Parallelism in INT8 Convolution for DSP48E2**

DSP48E2 Mode for INT8

The DSP48E2 slice can function as a twos complement arithmetic unit with a 27-bit pre-adder, a 27x18 multiplier, and a 48-bit post-adder (Figure 5), producing the result

PCOUT = P = (A + D)B + PCIN.

The difference between the output ports P and PCOUT is that P is connected to FPGA interconnects whereas PCOUT is hardwired to the port PCIN of another instance of the DSP48E2 slice in the same DSP48E2 column. The PCIN-PCOUT connection creates a DSP48E2 cascade. Because adjacent DSP48E2 slices are connected by abutment, the operating clock frequency can be maximized.

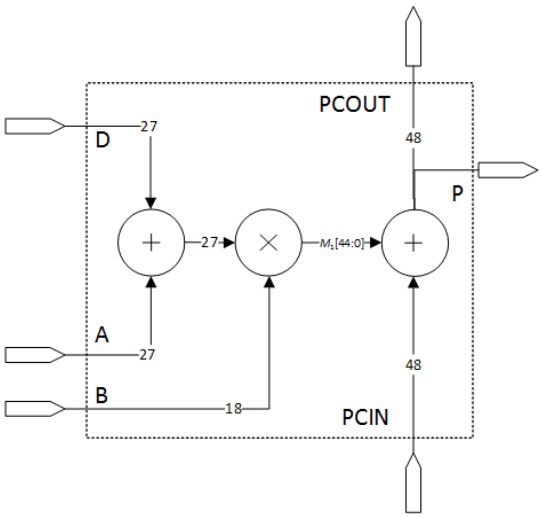

Figure 5: **DSP48E2 Mode Used in 8-Bit Dot-Product Acceleration**

Parallel Input and Packed Output Formats

A DSP48E2 slice can accept three INT8 operands, *a*, *d*, and *b*, to compute *ab* and *db* in parallel. The operands *a* and *d* are elements of two vectors to be multiplied by the common operand *b*, an element from a third vector. To use a DSP48E2 slice for this computation, assign *a*, *d*, and *b* to the ports A, D, and B, respectively, as shown in Figure 6.

- The operand *d* is sign-extended to form a 27-bit word for port D.
- The operand *a* is arithmetically left shifted by 18 bits, sign-extended to 27 bits, and placed onto port A. This operation effectively shifts the operand *a* as far left as possible in the DSP48E2 pre-adder port A while leaving one-bit of headroom, enabling the shifted operand to be added to *d* to generate a 27-bit word without overflow(1). The left-shift amount of *a* being 18 just happens to be the same as the port width of B.
- The shared operand b is sign-extended and placed onto port B.

With these input assignments, the multiplier inside the DSP48E2 slice computes $M = (a2^{18} + d)b$ (Figure 7). Now two dot products among three N-dimensional vectors, *a*, *d*, and *b*, can be packed into the sum of products

$$\sum_{i=0}^{N-1}(a_i 2^G + d_i)b_i,$$

where $G = 18$.

1. The DSP48E2 pre-adder accepts two 27-bit operands and generates a 27-bit, not 28-bit, result.

To make the upper dot product

$$\boldsymbol{a} \cdot \boldsymbol{b} = \sum_{i=0}^{N-1} a_i b_i$$

recoverable from the packed result, the lower dot product

$$\boldsymbol{d} \cdot \boldsymbol{b} = \sum_{i=0}^{N-1} d_i b_i$$

must stay within the lower G = 18 bits, which leads to N≤7 terms.(1)

As a result, given G = 18, two dot products among three seven-dimensional vectors can be computed without the lower dot product bleeding into the upper word.

1. An m-bit twos complement number is in the range $[-2^{m-1}, 2^{m-1}-1]$, and thus the product between two m-bit twos complement numbers is in the range $\left[2^{m-1}-2^{2(m-1)}, 2^{2(m-1)}\right]$. The sum of N such products is therefore in the range $\left[N(2^{m-1}-2^{2(m-1)}), N(2^{2(m-1)})\right]$. To be able to represent this sum as a q-bit twos complement number, which is in the range $[-2^{q-1}, 2^{q-1}-1]$, it must be the case that $-2^{q-1} \leq N(2^{m-1}-2^{2(m-1)})$ and $2^{q-1}-1 \geq N(2^{2(m-1)})$. Because the latter constraint implies the former constraint, rearranging the latter inequality yields $N \leq (2^{q-1}-1)/2^{2(m-1)}$, i.e., $N = \left\lfloor (2^{q-1}-1)/2^{2(m-1)} \right\rfloor$. In the foregoing example, $m = 8$ and $q = 18$, and thus $N = 7$.

The output ports P and PCOUT of the DSP48E2 slice contains a packed word of 2G = 36-bits representing two dot products in a special way (Figure 8). While P[17:0] always represents the dot product ***d·b*** as a twos complement number because the vectors have no more than seven elements, P[35:18] represents(1) ***a·b*** if ***d·b*** is non-negative and ***a·b*** – 1 otherwise. In other words, ***a·b*** = P[35:18]+P[17]. The rest of the bits in P, i.e., P[47:36] are sign-extension bits and are the same as P[35].

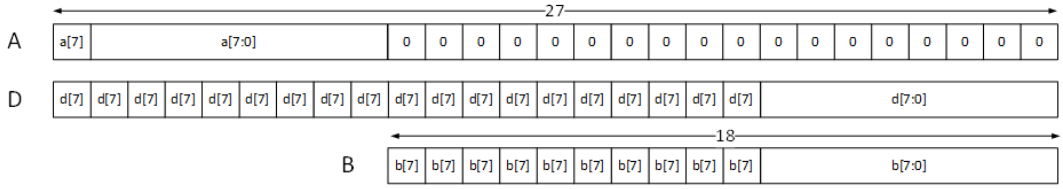

Figure 6: **Input Format**

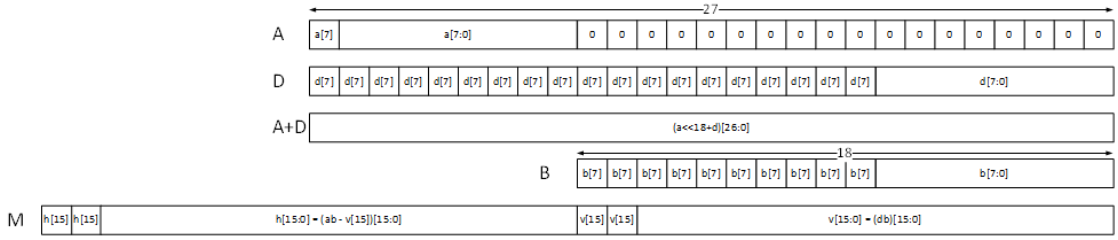

Figure 7: **Two Products in a Packed Word**

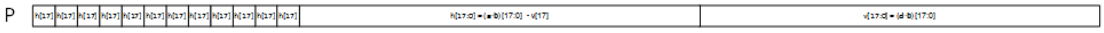

Figure 8: **Output Format for Two Dot Products for Vectors up to 7D**

Up to seven packed words from one or more DSP48E2 slices(2) can be added together—without any intervening adjustment to P[35:18] from each DSP48E2 slice—as shown in Figure 9. When the incrementer at the top of Figure 9 is implemented with LUTs, the dot-product throughput speed-up is 100%.

1. The 18 bits $P[35:18]$ can hold $\boldsymbol{a}\cdot\boldsymbol{b}-1$, which is in the range $\left[N(2^{m-1}-2^{2(m-1)})-1, N(2^{2(m-1)})-1\right]$, where $N=7$ and $m=8$. (See Footnote 1, page 7) The upper word $\boldsymbol{a}\cdot\boldsymbol{b}-1$ is therefore in the range $[-113793, 114687]$, which can be represented by an 18-bit twos complement number.

2. These seven terms do not have to be computed by seven DSP48E2 slices and can be computed by just one DSP48E2 slice when the post-adder is configured as an accumulator. In fact, one to six DSP48E2 slices can compute these seven terms with some slices in the multiply-add mode and others in the multiply-accumulate modes. See [Ref 3] for details.

Note that adjustment should be made only for the packed word that holds the final result.(1) The lower dot product is unaltered in P[17:0]. The upper dot product is unaltered if the lower dot product is non-negative; otherwise, P[35:18] must be incremented by one to recover the upper dot product.

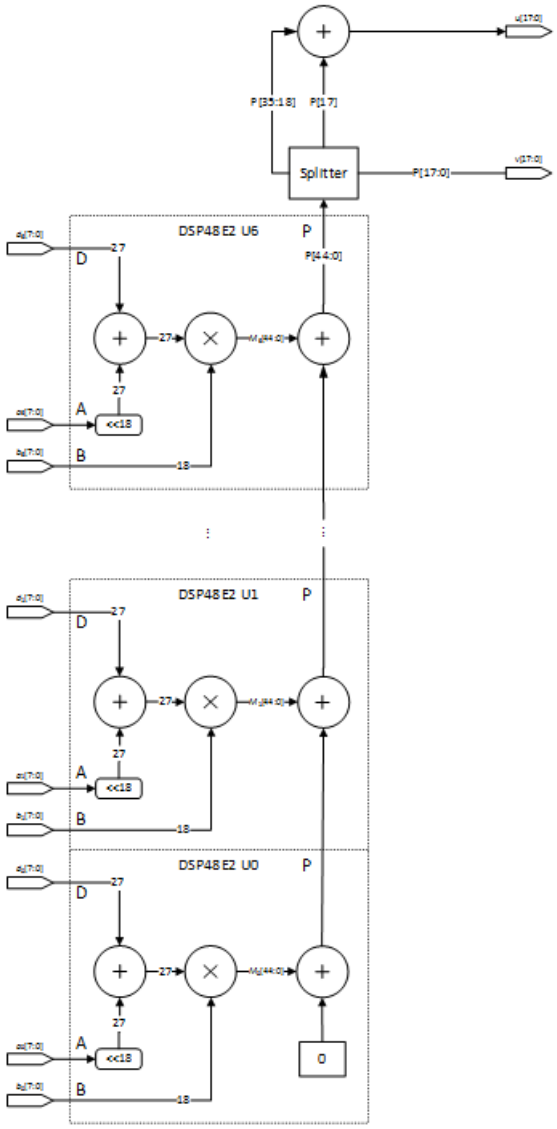

Figure 9: **Recovering the Upper Dot Product**

An Example

Table 1 illustrates an example for seven pairs of dot products, one pair in each row-from one dimensional vectors where i = 0 to seven dimensional vectors where i = 6. To illustrate how the packed word can be added to each other to produce the correct result, these seven dot product

1. For instance, if one DSP48E2 slice in the multiply-accumulate mode computes the seven terms in seven clock cycles, then only the packed word from the seventh cycle should be adjusted.

pairs build on one another as indicated by the column labeled P_i. This column lists the running sum of products, i.e.,

$$P_i = \sum_{j=0}^{i} (a_j b_j 2^G + d_j b_j),$$

which can also be expressed as a recurrence formula where

$P_0 = (a_0 2^G + d_0) b_0$, and $P_i = P_{i-1} + (a_i 2^G + d_i) b_i, i > 0.$

For the DSP48E2 slice, the left-shift amount of a_i is G = 18. Notice that the value listed in the column P_i[35:18] needs to be incremented by one to become the upper dot product if and only if the packed word contains the dot products of interest, and the lower dot product P_i[17:0] is negative, i.e., when P_i[17] = 1. For instance, in last row of Table 1, the upper word P_6[35:18] = 24 and the lower dot product

$P_6[17:0] = \sum_{j=0}^{6} d_j b_j = -1.$

Because the lower dot product is negative, P_6[35:18] needs to be incremented by 1 to yield the correct upper dot product 25. The values of the two dot products now agree with those in the two columns of expected results, i.e.,

$\sum_{j=0}^{i} a_j b_j$ and $\sum_{j=0}^{i} d_j b_j$.

Computing P_i by adding one to P_j[35:18] whenever P_j[17] = 1 for all $j<i$ is unnecessary. The upper 18-bits must be incremented by one for the very last packed-word only when the lower dot product is negative.

For DSP48E2, the left-shift amount for the input a_i is G = 18. To compute a vector dot product with four terms, look up the row where i = 3. The lower dot product is in the second-to-last column, i.e., P_3[17:0] = –1. Because the lower dot product is negative, the upper dot product must be P_3[35:18] + 1 = 2. To compute a dot product with six terms, the lower dot product is P_5[17:0] = 1. Because it is non-negative, the upper dot product is P_5[35:18] = 18 with no adjustment. See Table 1.

Table 1: **Example INT8 Calculations**

i	a_i	d_i	b_i	$a_i b_i$	$d_i b_i$	$P_i = \sum_{j=0}^{i} (a_j 2^G + d_j) b_j$	P_i[35:18]	$\sum_{j=0}^{i} a_j b_j$	P_i[17:0]	$\sum_{j=0}^{i} d_j b_j$
0	1	–4	–2	–2	8	–524280	–2	–2	8	8
1	2	8	–3	–6	–24	–2097168	–9	–8	–16	–16
2	3	17	2	6	34	–524270	–2	–2	18	18
3	4	–19	1	4	–19	524287	1	2	–1	–1
4	5	–1	2	10	–2	3145725	11	12	–3	–3
5	6	4	1	6	4	4718593	18	18	1	1
6	7	–2	1	7	–2	65553599	24	25	–1	–1

Beyond Seven-Dimensional Vectors

The input format (shown in Figure 6) guarantees up to seven terms in the dot product summation without the lower dot product bleeding into the upper word. To sum more than seven terms, the lower dot product and the upper word need to be separated further to form a wider packed word. This separation can be performed with FPGA fabric routing without any LUTs at the output of the DSP48E2 slice that sums the seventh term. Summation of the wider packed word must be performed without any upper-word adjustment until the last wider packed word is obtained. Since the DSP48E2 slice can also be used as a 48-bit two-input adder, the lower and the upper words can be expanded from 18- to 24-bits (Figure 10). Now, the results from multiple seven-DSP48E2 cascades can be summed. Again, adjustment of the upper word based on the sign of the lower dot product is necessary only for the very last, wider packed word. To achieve the maximum clock rate, the 48-bit adders and the 24-bit incrementer in Figure 10 should also be DSP48E2 slices, in which case, every eight DSP48E2 slices compute 14 terms whereas without this technique they compute only eight terms. The throughput improvement is thus 14/8 – 1 = 75%.

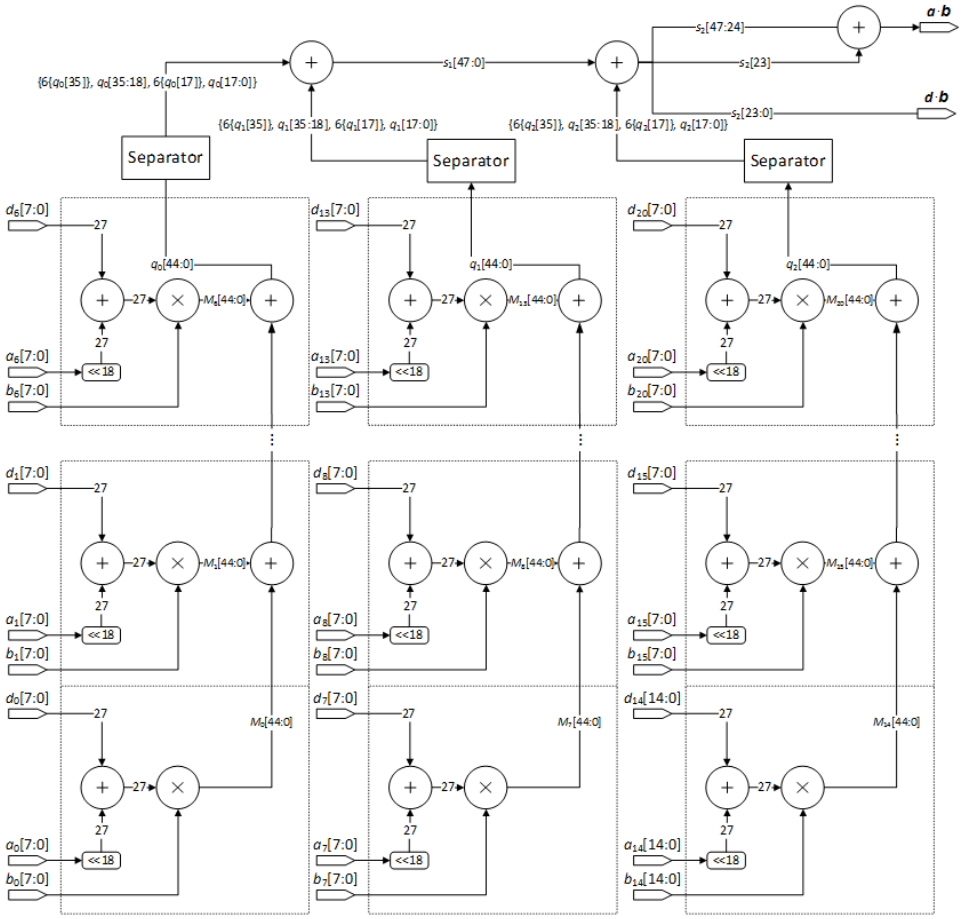

Figure 10: **Packed Word Separation for Dot Products of Higher-Dimensional Vectors**

Unsigned and Signed INT8

A rectified linear unit (ReLU) implements $f(x) = \max(0,x)$, and, as an activation function, has been shown to speed up training in machine learning [Ref 4][Ref 5]. (See Figure 11.) Represented as an INT8 value, the rectified number always has a zero most-significant bit. Representing data as unsigned 8-bit integers (UINT8) is therefore desirable since this format affords an additional bit of precision compared to INT8 with the same amount of memory storage or bandwidth. The DSP48E2 also supports two dot products ***a*·*b*** and ***d*·*b*** when the common vector ***b*** consists of INT8 elements, and the two vectors ***a*** and ***d*** hold UINT8 elements. For convolution, the weight vector is the vector ***b*** because the weights are signed. The rectified data are the vectors ***a*** and ***d***. Using the same dynamic range analysis outlined in Footnote 2, page 6, what follows shows that the number of terms that can be summed without the lower dot product bleeding into the upper dot product is eight, and, as a result, the throughput improvement is 78%.

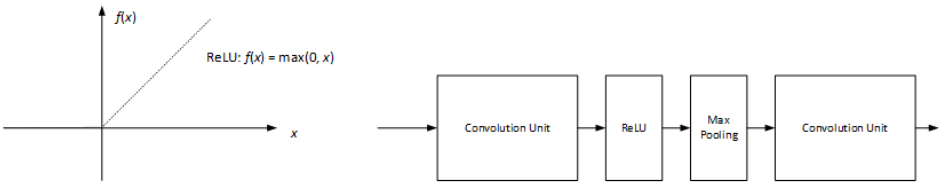

Figure 11: **Rectified Linear Unit (ReLU)**

Because both ***a*** and ***d*** are unsigned, these two vectors can be packed into one 27-bit vector without the DSP48E2 pre-adder. To maximize the gap between the element $a_i \in \boldsymbol{a}$ and the element $d_i \in \boldsymbol{d}$, a_i is left-aligned with the DSP48E2 port A and d_i is right-aligned. In other words, the 27-bit word is formed with the operation $a_i << G+d_i$, where "<<" is the left-shift operator. The shift amount of a_i is G = 19-bits, one more bit than the INT8 case. Because now the most-significant bit of a_i becomes the sign bit of the DSP48E2 port A, whenever $a_i[7] = 1$, the DSP48E2 treats the value in port A as a negative number. When a UINT8 value u is interpreted as an INT8 value x, then

$$x = \begin{cases} u, & u < 128 \\ u - 256, & u \geq 128 \end{cases}.$$

Thus the value of port A becomes

$$A_i = \begin{cases} a_i 2^G + d_i, & a_i < 128 \\ a_i 2^G + d_i - 256 \cdot 2^G, & a_i \geq 128 \end{cases},$$

which can be simplified as

$$A_i = a_i 2^G + d_i - a_i[7] \cdot 256 \cdot 2^G.$$

After multiplication with b_i, the packed output word is therefore

$$M_i = (a_i 2^G + d_i - a_i[7] \cdot 256 \cdot 2^G) b_i.$$

The bias $-256 \cdot 2^G a_i[7] b_i$ must be removed from M_i before summing. Thanks to the wide bias port

C in the post-adder of the DSP48E2 slice, the value $256 \cdot 2^G a_i[7]b_i$ can be added to the product M_i to remove the bias to form $P_{i+1} = P_i + M_i + 256 \cdot 2^G a_i[7]b_i = a_i b_i 2^G + d_i b_i$.

Two dot products for up to *N* dimensions can now be expressed as P_N. The format of this packed word is the same as in the INT8 case, where, for *N*-dimensional vectors such that ***d*·*b*** occupies no more than *G* = 19-bits, if ***d*·*b*** < 0, then P_N[37:19] = ***a*·*b*** – 1; otherwise, P_N[37:19] = ***a*·*b***.

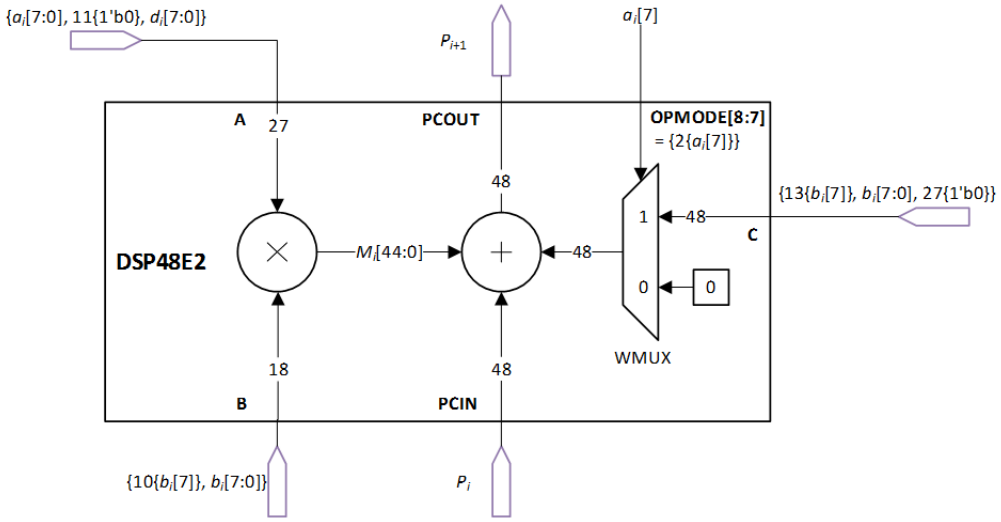

Figure 12: **Two UINT8 Elements Multiplied by an INT8 Element**

Port A receives the unsigned 8-bit operands, a_i and d_i, with the maximum amount of separation. Port B receives the signed operand b_i. The left-shift amount for a_i is G = 19. But whenever a_i > 128, the DSP48E2 slice treats the input at A as a negative number, in effect adding $-256 \cdot 2^G b_i$ to M_i. To remove this bias, the value $256 \cdot 2^G a_i[7]b_i$ is added back into the packed word M_i through the WMUX of the DSP48E2 slice. When and only when $a_i[7] = 1$, the WMUX sends $256 \cdot 2^G b_i = 2^{27} b_i$ through the port C into the post-adder. The two input pins OPMODE[8:7] control the select port of WMUX. When these two bits are both `1`, the value on the port C is selected; otherwise, `0` is selected.

The sum of up to $N = \lfloor (2^{q-1} - 1)/(2^{2m-1} - 1) \rfloor$ products between an *m*-bit signed twos complement number and an *m*-bit unsigned number can be represented as a *q*-bit twos complement number.(1) Here, *m* = 8, and *q* = G = 19. Thus, up to *N* = 8 products can be accumulated in a 19-bit twos

1. The range of an m-bit unsigned number is $[0, 2^m - 1]$ and that of a signed twos complement number is $[-2^{m-1}, 2^{m-1} - 1]$. The range of the product of these two numbers is therefore $[-2^{m-1}(2^m - 1), (2^m - 1)(2^{m-1} - 1)]$, which can be represented by a $2m$-bit twos complement number because its range is $[-2^{2m-1}, 2^{2m-1} - 1]$, wider than the range of the product whenever $m \geq 1$. The range of N such products is now $[-N2^{m-1}(2^m - 1), N(2^m - 1)(2^{m-1} - 1)]$, which can be represented by a q-bit twos complement number, whose range is $[-2^{q-1}, 2^{q-1} - 1]$, when ever $N \leq (2^{q-1} - 1)/(2^{2m-1} - 1)$. Thus, the maximum value of N is $\lfloor (2^{q-1} - 1)/(2^{2m-1} - 1) \rfloor$.

complement number. The same technique in Figure 12 can be used go beyond eight-dimensional vectors. Now every nine DSP48E2 slices perform 8×2 = 16 multiply-add operations whereas without packing they perform only nine such operations. The throughput speed-up is therefore 16/9≈78%.

Conclusion

Massive parallelism in artificial intelligence calls for higher compute density. The Xilinx DSP48E2 slice can pack two parallel 8-bit multiply-add operations and speed up convolution and matrix multiplication by 1.75X to 2X. The 27x18 multiplier in the DSP48E2 slice supports two parallel 8-bit multiplications that share one common operand. Specifically, two INT8 operands are packed into the 27-bit port and then multiplied in parallel by the third, shared INT8 operand in the 18-bit port. For applications that employ the ReLU activation function, the DSP48E2 slice provides the unsigned data with an extra significant bit by multiplying two UINT8 data operands in parallel by one INT8 weight. The output of the DSP48E2 slice in both cases carries two parallel products in a special full-precision dual-product format. The two parallel products represented in this format can be summed as if they were a single twos complement number—up to seven terms for INT8xINT8 and up to eight terms for UINT8xINT8—mathematically proven to preserve full precision. For dot products of higher dimensions, the FPGA routing fabric separates the two parallel products for further summation by either LUTs or additional DSP48E2 slices. This special dual-product format is used throughout the summation without any adjustment. Only the final dot product result might need to be translated: The upper product needs to be incremented by one only when the lower product is negative. The speed-up is thus 2X when the additional summation is performed by the LUTs. The speed-up appears lower at 1.75X for INT8xINT8 but at a higher clock frequency due to a regular DSP array implementation with little fabric routing, and similarly for UINT8xINT8, the speed-up is 1.78X.

References

1. Dettmers, T. 8-Bit Approximations for Parallelism in Deep Learning. In ICLR 2016.
2. Gysel, P., Motamedi, M., and Ghiasi, S. Hardware-oriented Approximation of Convolutional Neural Networks. arXiv preprint arXiv:1604.03168, 2016.
3. UG579, UltraScale Architecture DSP48 Slice User Guide.
4. Glorot, X., Bordes, A., and Bengio, Y., Deep Sparse Rectifier Neural Networks. In JMLR, 15, 315-323, 2011.
5. Krizhevsky, A., Sutskever, I., and Hinton, G., Imagenet Classification with Deep Convolutional Neural Networks. In NIPS, 1097-1105, 2012.

Revision History

The following table shows the revision history for this document:

Date	Version	Description of Revisions
06/27/2017	1.0	Initial Xilinx release.

Disclaimer

Automotive Applications Disclaimer