# High Density and Performance Multiplication for FPGA

Martin Langhammer
*Intel UK*

Gregg Baeckler
*Intel US*

***Abstract*—Arithmetic based applications are one of the most common use cases for modern FPGAs. Currently, machine learning is emerging as the fastest growth area for FPGAs, renewing an interest in low precision multiplication. There is now a new focus on multiplication in the soft fabric - very high-density systems, consisting of many thousands of operations, are the current norm. In this paper we introduce multiplier regularization, which restructures common multiplier algorithms into smaller, and more efficient architectures. The multiplier structure is parameterizable, and results are given for a continuous range of input sizes, although the algorithm is most efficient for small input precisions. The multiplier is particularly effective for typical machine learning inferencing uses, and the presented cores can be used for dot products required for these applications. Although the examples presented here are optimized for Intel Stratix 10 devices, the concept of regularized arithmetic structures are applicable to generic FPGA LUT architectures. Results are compared to Intel Megafunction IP as well as contrasted with normalized representations of recently published results for Xilinx devices. We report a 10% to 35% smaller area, and a more significant latency reduction, in the range of 25% to 50%, for typical inferencing use cases.**

## I. Introduction

The emergence of Artificial Intelligence (AI), Machine Learning (ML) and Deep Learning (DL) - which for brevity we will simply refer to as ML in this paper - is driving architectural change and innovation across a wide range of devices: CPU, GPU, ASIC/ASSP, and of course, FPGA. There are two main classes of machine learning, training and inference. Inference, with its many arrays of typically lower precision multipliers (usually arranged as large groups of DOT product operators), appears to be a good match for FPGA architectures. Modern FPGAs [1][2] now have over a million LUT/register combinations, which suggests that they are well suited to implementing these types of arrays. Training, by contrast, requires higher precision arithmetic, typically FP. Although high performance embedded FP is now supported by FPGAs [3], this paper will only examine soft logic arithmetic suitable for inference.

A recent commercial FPGA based system, Microsoft Brainwave [4], illustrates this. The reported performance of 90 TOPs at a 500MHz clock rate in an Intel Stratix 10 device implies that 180K operators - and therefore 90K multipliers - are present in the device. Details of the arithmetic format are not disclosed other than an indication that it is a proprietary FP representation. The soft logic comprises 933K ALMs (a brief description of FPGA logic resources is given in the following section) on the target device, *i.e.* about 10ALMs per operator (also, all the DSP Blocks are used, which suggests that they are implementing some of the multiplications), implying a dense, efficient soft logic structure.

Virtually all modern FPGAs contain embedded multipliers, typically based on 18 bit precision [5][6], including low end devices [7][8]. Smaller multipliers can be extracted from larger ones [9], but this can introduce other inefficiencies. Extracting subsets still powers up the entire multiplier datapath of the DSP Block - the Xilinx 27x18 has about four times the arithmetic density of the two extracted 8x8s, so the power consumption is that of the entire datapath, plus additional soft logic support, although only half of the arithmetic capability is used. In addition, the FPGA resource mix is compartmentalized, where similar functions are grouped together. The connections between the DSP Block and soft logic needed to support the DOT operator and accumulators require a significant amount of routing, which is further complicated by the fact that the busses are restricted because of the relative placement of bits. This further reduces device flexibility, and increases the power consumption per operation. We can therefore see that the design of efficient and high performance soft logic arithmetic operators for machine learning inference is of great importance.

In this paper, we introduce a new method and approach for mapping integer multipliers to current FPGAs. Arithmetic in FPGAs typically uses the embedded carry chain in the soft logic structures. The new techniques will more fully utilize the soft logic which precedes the carry chains, including the case of summation of the generated partial products. Routing density will be carefully considered and optimized. In the presented constructs, any required logic which cannot be mapped entirely to the look up tables supporting the carry chain will be calculated out-of-band, and only the single bit results of the out-of-band function routed to the carry chain logic.

The rest of the paper is organized as follows. Section II will review recent work on FPGA soft logic multiplication algorithms. Section III will present the bulk of the new algorithms and approaches, with the specific examples illustrated in a series of figures. This will be followed by results in section IV, where areas of different multipliers will be tabulated. Finally, future work will be introduced in section V, and the conclusions presented in section VI.

## II. Related work

Both Intel and Xilinx have provided multipliers as part of their IP offerings for many years. We will compare this work to Intel and Xilinx (in the Xilinx case, based on two recent academic works) qualitatively. The resources required by these new algorithms are often significantly smaller than the previously published results, but just as importantly, use the routing resources of the FPGA more effectively. A quantitative analysis of routing stress is beyond the scope of this paper, but we will attempt to highlight some of the key issues informally by discussion.

The soft logic resources in FPGAs are comprised of many look up tables (LUTs), which can often be decomposed into a series of smaller LUTs. A 6 input LUT (6LUT) is the norm for modern FPGAs; the six inputs can implement a function of $2^6$ combinations. The 6LUT can be decomposed: four 4LUTs in the case of the Intel device (supporting two bits of arithmetic), and two 5LUTs for the Xilinx device, with a single bit of arithmetic. Recent work for Xilinx devices more fully uses the 5LUT structure. In this paper we show improved mappings to the smaller 4LUT subset - but wider arithmetic - of the Intel 6LUT. The Intel 6LUT structure is known as an ALM, a term we will use in this paper.

Kumm et. al [10] described a new multiplier approach which uses an array multiplier architecture [11], where the partial products of each level are calculated using modified Booth's encoding [12][13], and then summed to the partial products of the previous levels. This work relies on the features of the Xilinx soft logic, in particular, that each bit of the embedded adder is fed by two 5LUTs. The array multiplier architecture builds a linear rather than a logarithmic depth structure, which will decrease performance for a combinatorial implementation, or increase the latency for a pipelined design. Although there is minimal logic cost in pipelining as registers available on each bit of the adder, the large number of registers - and therefore the switching rate for this linear array will likely cause an increased power consumption.

Kumm also states that this technique would not work on similar Intel devices, as the local logic decomposition on the Intel devices is two 4LUTs per bit, instead of 5LUTs. The Intel 6LUT, however, can be decomposed into two arithmetic bits rather than the single bit of the Xilinx 6LUT. We will show that the former case provides a more efficient target for multiplier implementation that the latter.

Walters [14], [15] presents a similar scheme for Xilinx FPGAs, where a Booth's coding is mapped to the LUTs, and an array multiplier is implemented using the associated carry chains. A tree structure is also described, albeit with a stated caveat that the tree structure is somewhat larger than the linear array structure. The tree structure also appears to require the use of ternary adders, but there is no discussion of the routing density problem of ternary addition, and the possible system impact of filling a device with multipliers containing them. In our experience, ternary addition places great stress on routing, and can only be used for a subset of the device. For current applications requiring many soft multipliers, such as machine learning, the system arithmetic density required may make ternary adders unusable.

## III. Multiplier Regularization

The canonic mapping for typically lower precision multipliers results in disjoint FPGA resource requirements with varying multiplier size. For example, the dimensionless arithmetic complexity of a multiplier can be estimated by the product of the input precisions. A 4x4 multiplier is expected to be about 16 $(4 \cdot 4)$, while a 5x5 multiplier is 25 $(5 \cdot 5)$, or a ratio of 1.56x. In practice, however, the cost in FPGA resources for the larger multiplier is about twice that of the smaller one; in an Intel Stratix 10 device (using the Intel Megafunction IP), the two cores are 11 ALMs and 22 ALMs, respectively. The difference can be largely explained by the granularity of the resources: the 4x4 multiplier maps directly to two pairs of 4-input (although using only two independent inputs) LUT groups, while the 5x5 multiplier requires a third LUT group, containing only a single partial product. The third group must then be added separately to the sum of the first two pairs of partial products. There are therefore multiple inefficiencies in the 5x5 multiplier: the logic of the third group of LUTs is only partially used - and the associated adder is not used at all - and a final adder level is required to add in the third partial product. Not only are the resources of this last adder required, but also the additional level of logic, with its attendant routing, latency, and possible balancing implications, which impact the true cost of increasing the multiplier precision by a bit.

By using the first of the methods described in this paper - restructuring the partial products with out-of-band compression of triple columns - the three sets of partial products in the 5x5 case would be reduced to two pairs, requiring only a single adder to sum, reducing the (dimensionless) system cost to around 20. The 6x6 case would also produce 3 pairs of partial products, but the second method described here - creating an alternate adder in triangular form - reduces the 2 levels of adder to a single adder. In the case of the 7x7 multiplier, both of these methods would be combined to make the near equivalent of the 6x6 case.

It is also possible that multiple stages of these algorithms can be used to change one optimal form of multiplier into another form of multiplier. For example, an 8x8 multiplier, which appears to map directly to the FPGA, can be changed to an 8x6 by applying the partial product compression twice, followed by the adder triangular optimization on the summing of the three partial products.

| **Column** | **5** | **4** | **3** | **2** | **1** | **0** |
|---|---|---|---|---|---|---|
| PP0 | 0 | 0 | 0 | $p_{0,2}$ | $p_{0,1}$ | $p_{0,0}$ |
| PP1 | 0 | 0 | $p_{1,2}$ | $p_{1,1}$ | $p_{1,0}$ | 0 |
| PP2 | 0 | $p_{2,2}$ | $p_{2,1}$ | $p_{2,0}$ | 0 | 0 |

Figure 1. $3 \times 3$ Multiplier $\{a_2, a_1, a_0\} \times \{b_2, b_1, b_0\}$

These techniques can be used for larger multipliers as well, but these can usually be more efficiently (power, performance, area) implemented in embedded (hard) DSP resources in modern FPGAs. We now define multiplier regularization as a combination of up to three methods:

- A general multiplier mapping with some processing in separate combinatorial circuits, where an entire level of partial product is factored out. The combinatorial functions do not form part of the carry chain associated with the soft logic, and are therefore denoted as being out-of-band.
- Multiplier mappings by refactoring arithmetic to implement a subset of the multiplication with the equivalent of ternary addition, while only using a binary arithmetic logic structure. This optimization initiates by forming a triangular shaped group of partial products, often from the first partial product refactorization.
- Using 2:2 compression to introduce gaps in the combination of partial products so that the 1's to 2's complement arithmetic for signed operations can be implemented in the minimum number of levels.

Signed and unsigned multipliers are different cases of these methods, and special techniques for signed multipliers are described later in this section.

We have discussed how modern FPGAs have a dedicated adder structure supported in the logic, typically in some form of ripple carry adder, with each bit position fed by LUTs. An equally important consideration is the routing flexibility of the system, as there are only a limited number of independent inputs for each LUT in a local group of logic.

In one example, Intel Stratix 10 devices [1] groups of 10 logic blocks (ALMs) are arranged together in a structure known as a LAB. Each logic block can be configured into an arithmetic mode, where four 4LUTs can feed two bits of a ripple carry adder. These four LUTs will share 6 independent inputs in a specified way. Each group of 10 ALMs (*i.e.* LAB) will have a total number of independent inputs that in is that the number of independent inputs per ALM, typically 25% of the LAB number. Therefore, a large number of common inputs must be shared across the ALMs if there are more than 2 or 3 inputs per ALM used.

### A. *Out-of-Band Mapping - Simple Case*

We will now illustrate the restructuring of the partial products using the out of band combinatorial functions by an unsigned 3x3 multiplier example. For brevity, we will refer to the combinatorial functions here as auxiliary cells.

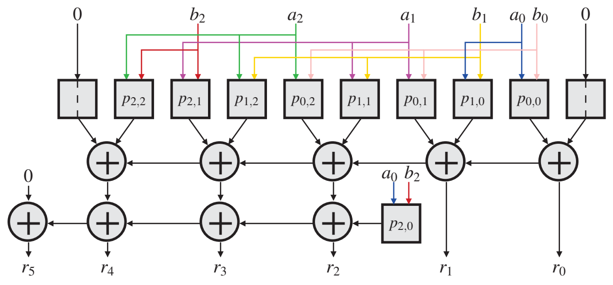


Figure 2. Known 3x3 Mapping

We define the multiplicand as the three bits $\{a_2, a_1, a_0\}$ (from most significant to least significant bit) and the multiplier as $\{b_2, b_1, b_0\}$. Figure 1 shows the partial product generation and alignment. For example, the term $p_{1,0}$ is the bit 0 of the multiplicand (the least significant bit) ANDed with bit 1 of the multiplier. Although ternary addition is supported by many modern FPGA devices, the independent number of inputs required (three input bits per output bit) is larger than supported by the routing architecture of some FPGAs, and can therefore not universally applied across the FPGA. Even if ternary addition is supported, the partial products must be generated and processed first. The typical architectural case where some logic capability exists before the dedicated ripple carry adder will support the calculation and addition of two partial products, but the third will have to be produced separately.

Figure 2 shows a structure of the known method of multiplier mappings to the Intel FPGA architecture. The small 3x3 multiplier case can also be decoded efficiently directly with 5 ALMs; 4 in 6LUT mode, and 1 decomposed into two 5LUTs.

We will now show how to transform this into the sum of two partial products with a binary (two input) adder, with the assistance of the out-of-band auxiliary cells. First, we extract two entries in column 2 with the function ($p_{0,2} \oplus p_{1,1}$), which is the redundant sum compression of those input bits. These four constituent input bits ($a_2$,$a_1$,$b_1$, and $b_0$) can be used in the same ALM used to calculate the redundant carry function ($a_2 \cdot b_0 \cdot a_1 \cdot b_1$). Moving the redundant carry function into column 3 creates another column of three bits, but in this case, the redundant carry can be expanded to include the redundant sum calculation for column 3, or $(a_2 \cdot b_0 \cdot a_1 \cdot b_1) \oplus (a_2 \cdot b_1)$.

There is, however, another condition that needs to be considered. If $a_2 \cdot b_0$, $a_1 \cdot b_1$, and $a_2 \cdot b_1$ are all '1', then a carry will be produced in column 4. Another way of looking at this is that if the redundant sum in column 3 is $(a_2 \cdot b_0 \cdot a_1 \cdot b_1) \oplus (a_2 \cdot b_1)$, the following redundant carry will be $(a_2 \cdot b_0 \cdot a_1 \cdot b_1) \cdot (a_2 \cdot b_1)$. To avoid creating another auxiliary cell, we notice that the redundant sum can be XORed with $a_2 \cdot b_1$ in the cell of column 4, generating

| Column | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|
| PP0 | 0 | $p_{2,2}$ | $p_{2,1}$ | $p_{2,0}$ | $p_{0,1}$ | $p_{0,0}$ |
| PP1 | 0 | $AUX_2 \oplus p_{1,2}$ | $AUX_2$ | $AUX_1$ | $p_{1,0}$ | 0 |

Figure 3. $3 \times 3$ Multiplier in Two Levels with Auxiliary Functions

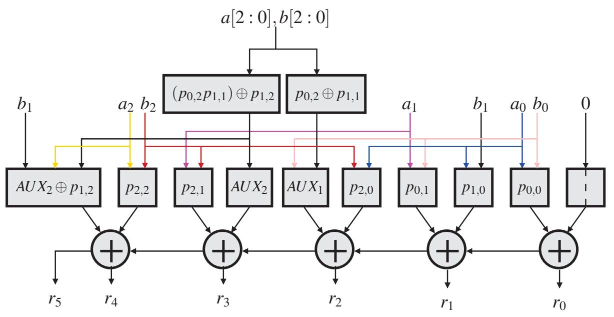


Figure 4. New 3x3 Mapping

$(a_2 \cdot b_0 \cdot a_1 \cdot b_1) \oplus (a_2 \cdot b_1) \oplus (a_2 \cdot b_1)$, or $(a_2 \cdot b_0 \cdot a_1 \cdot b_1)$. The redundant carry would be $(a_2 \cdot b_0 \cdot a_1 \cdot b_1 \cdot a_2 \cdot b_1)$, which simplifies to $(a_2 \cdot b_0 \cdot a_1 \cdot b_1)$, which is what we just generated in column 4.

An unsigned 3x3 multiplier therefore reduces to 4 ALMs: 3 ALMs in an unbroken carry chain, plus 1 ALM for the two auxiliary functions ($AUX_x$). This is illustrated in Figure 3. Grouping each pair of columns into an ALM shows that a maximum of 4 independent inputs is required per ALM, with a total of 6 independent inputs over 4 ALMs. We are therefore well below the maximum level of inputs per ALM, and at only 1.5 independent inputs averaged over a group of ALMs, expect that this will be a very low stress system routing problem. Consequently, this multiplier should fit into the device with high density, while maintaining high speed.

Figure 4 shows this method mapped to the Intel Stratix logic. Note that the arithmetic depth has also decreased from two levels to a single level.

## B. Out-of-Band Mapping - General Case

The method for the 3x3 example can be expanded for larger multipliers, such as 5x5 and 7x7, where an odd number of partial products exist. An even number of partial products will create a balanced first level addition of partial products, and therefore our goal is to transform the odd number of partial product to an even number by removing one of the initial partial products.

Figure 5 shows the partial products of an unsigned 5x5 multiplier. We can decompose this into two sets of partial products, one with two partial products (but with one column of 3 bits) and the other consisting of two partial products. Figure 6 and Figure 7 show these two sets, respectively.

We can now reduce the first set of partial products to two vectors by removing the 3 deep column, using the approach of the 3x3 multiplier case. In column 4, $p_{1,3}$ and $p_{2,2}$ are replaced by $AUX_1 = p_{1,3} \oplus p_{2,2}$. In column 5, $p_{1,4}$

| Col. | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| PP0 | 0 | 0 | 0 | 0 | $p_{0,4}$ | $p_{0,3}$ | $p_{0,2}$ | $p_{0,1}$ | $p_{0,0}$ |
| PP1 | 0 | 0 | 0 | $p_{1,4}$ | $p_{1,3}$ | $p_{1,2}$ | $p_{1,1}$ | $p_{1,0}$ | 0 |
| PP2 | 0 | 0 | $p_{2,4}$ | $p_{2,3}$ | $p_{2,2}$ | $p_{2,1}$ | $p_{2,0}$ | 0 | 0 |
| PP3 | 0 | $p_{3,4}$ | $p_{3,3}$ | $p_{3,2}$ | $p_{3,1}$ | $p_{3,0}$ | 0 | 0 | 0 |
| PP4 | $p_{4,4}$ | $p_{4,3}$ | $p_{4,2}$ | $p_{4,1}$ | $p_{4,0}$ | 0 | 0 | 0 | 0 |

Figure 5. 5x5 Multiplier Partial Product Bits

is replaced by $AUX_2 = ((p_{1,3} \cdot p_{2,2}) \oplus p_{1,4})$, and the carry into column 6 is then $AUX_3 = (p_{2,2} \cdot p_{1,3} \cdot p_{1,4})$. If we define the multiplicand as $a[4:0]$ and the multiplier as $b[4:0]$, it follows that $p_{2,2} = a_2 \cdot b_2$, $p_{1,3} = a_3 \cdot b_1$, and $p_{1,4} = a_4 \cdot b_1$.

Unlike the simpler case of the 3x3 multiplier, there are now five independent variables in the combinatorial functions, and three separate auxiliary functions are required. Instead, we change $AUX_2$ and $AUX_3$ to use $p_{2,3}$ instead of $p_{1,4}$, which results in $AUX_2 = ((p_{1,3} \cdot p_{2,2}) \oplus p_{2,3})$ and $AUX_3 = (p_{2,2} \cdot p_{1,3} \cdot p_{2,3})$. The $AUX_2$ value, $((p_{1,3} \cdot p_{2,2}) \cdot p_{2,3})$, can then be expanded to $((a_3 \cdot b_1) \cdot (a_2 \cdot b_2)) \oplus (a_3 \cdot b_2))$. Similarly, the $AUX_3$ value, $(p_{2,2} \cdot p_{1,3} \cdot p_{2,3})$, is $(a_2 \cdot b_2 \cdot a_3 \cdot b_1 \cdot a_3 \cdot b_2)$, which can then be simplified to $(a_2 \cdot a_3 \cdot b_1 \cdot b_2)$. $AUX_1$ remains $(p_{1,3} \oplus p_{2,2})$, or expressed in the expanded form, $((a_3 \cdot b_1) \oplus (a_2 \cdot b_2))$.

This group of functions now has four independent bits ($a_3$, $a_2$, $b_2$, and $b_1$), and can be implemented in two auxiliary cells. Similarly to the 3x3 case, the $AUX_3$ function is implemented using the available inputs of the last ALM (columns 6 and 7) implementing the generation and addition of the first group of partial products (Figure 6). The $AUX_2$ function is input, and converted to $AUX_3$ by XORing $(a_3 \cdot b_2)$ away. The five ALMs (four in the carry chain, and the one for the out-of-band combinatorial functions) require 8 independent inputs. At less than 2 independent bits per ALM, this is again a very low stress routing problem.

## C. Ternary to Binary Addition Mapping

If the partial products are optimized (such as with our compression of odd to even number of partial products), or not, they still need to be added together. In FPGAs this is most commonly and efficiently done using ripple carry adders. We can take advantage of the logic in front of the ripple carry adders to further optimize the final summation. In particular, we can show how to add three partial products using a binary (two-input adder), with a small number of

| Column | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| PP0 | 0 | 0 | $p_{2,4}$ | $p_{2,3}$ | $p_{0,4}$ | $p_{0,3}$ | $p_{0,2}$ | $p_{0,1}$ | $p_{0,0}$ |
| PP1 | 0 | 0 | 0 | $p_{1,4}$ | $p_{1,3}$ | $p_{1,2}$ | $p_{1,1}$ | $p_{1,0}$ | 0 |
| PP2 | 0 | 0 | 0 | 0 | $p_{2,2}$ | 0 | 0 | 0 | 0 |

Figure 6. $5 \times 5$ Multiplier First Partial Product Set

| Column | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| PP3 | 0 | $p_{3,4}$ | $p_{3,3}$ | $p_{3,2}$ | $p_{3,1}$ | $p_{3,0}$ | 0 | 0 | 0 |
| PP4 | $p_{4,4}$ | $p_{4,3}$ | $p_{4,2}$ | $p_{4,1}$ | $p_{4,0}$ | 0 | 0 | 0 | 0 |

Figure 7. $5 \times 5$ Multiplier Second Partial Product Set

| Col. | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| PP0’ | 0 | 0 | $p_{2,4}$ | $p_{2,3}$ | $p_{0,4}$ | $p_{0,3}$ | $p_{0,2}$ | $p_{0,1}$ | $p_{0,0}$ |
| PP1’ | 0 | 0 | $AUX_2 \oplus p_{2,3}$ | $AUX_2$ | $AUX_1$ | $p_{1,2}$ | $p_{1,1}$ | $p_{1,0}$ | 0 |

Figure 8. Optimized First Partial Product Set

auxiliary cells. We will show how a 6x6 multiplier can be implemented using only two levels of logic: one for the partial product generation (and summing of the first two binary partial products), and one for the summation of the three partial products produced.

This can also be used as a part of the adder tree of larger multipliers, such as when there are 5 partial products (such as an $N \times 10$ multiplier). In that case, three of the partial products would be added with the ternary to binary conversion of this section, and the remaining two partial products with a binary adder. The result is then generated using another binary adder.

Figure 9 shows the six pencil and paper partial products for the example 6x6 multiplier. These are added pairwise (by the ripple carry adder associated with them) to create three second level partial products, $x$, $y$, and $z$ of Figure 10. The LSB of each partial product pair is denoted with an ’L’ suffix, to indicate that these can be solely calculated by combinatorial logic, and do not have to be tied to the ripple carry adder. The bits $x_L$, $y_L$, and $z_L$ are $p_{0,0}$, $p_{2,0}$, and $p_{4,0}$ respectively. An arithmetically identical arrangement of the bits of Figure 10 is shown in Figure 11.

Although ternary addition is supported by most current FPGAs, it may not be ubiquitously usable because of the input routing stress (independent input flexibility is discussed earlier in Section II). Instead, a combination of 3:2 and 2:2 redundant form compression may be implemented in a portion of the logic associated with the ripple carry adder, again supplemented by some out-of-band (*i.e.* not directly on the carry chain) combinatorial cells. Figure 12 shows the summation of three partial product pairs using two adder levels.

In the construction of Figure 13, columns of 3 bits are reduced to 2 columns of 1 bit each, using 3:2 compression. Because of routing density limitations, 3:2 compression cannot be performed contiguously across the bit vectors. Instead, we perform 3:2 compression on alternate 3 high columns, and otherwise input already compressed bits, which are calculated in the out-of-band auxiliary cells. For smaller multipliers, such as we typically see in machine learning inference applications, the partial triangulation of the partial product pairs will only produce a few 3 bit columns. The number of external combinatorial cells is therefore low.

| Col. | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| PP0 | 0 | 0 | 0 | 0 | 0 | 0 | $p_{0,5}$ | $p_{0,4}$ | $p_{0,3}$ | $p_{0,2}$ | $p_{0,1}$ | $p_{0,0}$ |
| PP1 | 0 | 0 | 0 | 0 | 0 | $p_{1,5}$ | $p_{1,4}$ | $p_{1,3}$ | $p_{1,2}$ | $p_{1,1}$ | $p_{1,0}$ | 0 |
| PP2 | 0 | 0 | 0 | 0 | $p_{2,5}$ | $p_{2,4}$ | $p_{2,3}$ | $p_{2,2}$ | $p_{2,1}$ | $p_{2,0}$ | 0 | 0 |
| PP3 | 0 | 0 | 0 | $p_{3,5}$ | $p_{3,4}$ | $p_{3,3}$ | $p_{3,2}$ | $p_{3,1}$ | $p_{3,0}$ | 0 | 0 | 0 |
| PP4 | 0 | 0 | $p_{4,5}$ | $p_{4,4}$ | $p_{4,3}$ | $p_{4,2}$ | $p_{4,1}$ | $p_{4,0}$ | 0 | 0 | 0 | 0 |
| PP5 | 0 | $p_{5,5}$ | $p_{5,4}$ | $p_{5,3}$ | $p_{5,2}$ | $p_{5,1}$ | $p_{5,0}$ | 0 | 0 | 0 | 0 | 0 |

Figure 9. Partial Products for a 6x6 Multiplier

| Col. | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| PP0+PP1 | 0 | 0 | 0 | 0 | $x_7$ | $x_6$ | $x_5$ | $x_4$ | $x_3$ | $x_2$ | $x_1$ | $x_L$ |
| PP2+PP3 | 0 | 0 | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_L$ | 0 | 0 |
| PP4+PP5 | $z_7$ | $z_6$ | $z_5$ | $z_4$ | $z_3$ | $z_2$ | $z_1$ | $z_L$ | 0 | 0 | 0 | 0 |

Figure 10. Partial Product Pairs for a $6 \times 6$ Multiplier

| Col. | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| PP0+PP1 | $z_7$ | $z_6$ | $z_5$ | $z_4$ | $x_7$ | $x_6$ | $x_5$ | $x_4$ | $x_3$ | $x_2$ | $x_1$ | $x_L$ |
| PP2+PP3 | 0 | 0 | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_L$ | 0 | 0 |
| PP4+PP5 | 0 | 0 | 0 | 0 | $z_3$ | $z_2$ | $z_1$ | $z_L$ | 0 | 0 | 0 | 0 |

Figure 11. Alternate arrangement PP pairs for a $6 \times 6$ multiplier

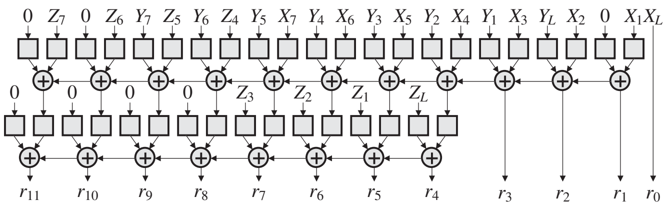


Figure 12. Summation of Three PPs in Two Binary Adder Layers

The 3:2 compression of the middle columns requires that the more significant two bit columns be 2:2 compressed to maintain a 2 bit column height throughout the bit vectors.

The equations for the 3:2 compression performed in both the in-line cells and the auxiliary cells are as follows:

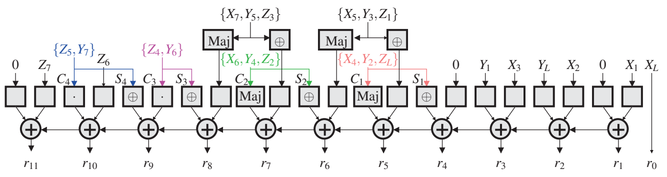
$$\begin{aligned}
s_1 &= x_4 \oplus y_2 \oplus z_L \\
c_1 &= \text{Majority}(x_4, y_2, z_L) \\
&= (x_4 \cdot y_2) + (x_4 \cdot z_L) + (y_2 \cdot z_L) \\
hs_1 &= x_5 \oplus y_3 \oplus z_1 (\text{auxiliary cell}) \\
hc_1 &= \text{Majority}(x_5, y_3, z_1) (\text{auxiliary cell}) \\
s_2 &= x_6 \oplus y_4 \oplus z_2 \\
c_1 &= \text{Majority}(x_6, y_4, z_2) \\
hs_2 &= x_7 \oplus y_5 \oplus z_3 (\text{auxiliary cell}) \\
hc_2 &= \text{Majority}(x_7, y_5, z_3) (\text{auxiliary cell})
\end{aligned}$$

As previously mentioned, some 2:2 compressors are needed to maintain the shift pattern. These can be directly implemented in the logic associated with that section of the carry chain, using a half adder (HA) construction:

$$s_3 = z_4 \oplus y_6 \qquad c_3 = z_4 \cdot y_6$$
$$s_4 = z_5 \oplus y_7 \qquad c_4 = z_5 \cdot y_7$$

| Col. | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Line0 | $z_7$ | $z_6$ | $s_4$ | $s_3$ | $hs_2$ | $s_2$ | $hs_1$ | $s_1$ | $x_3$ | $x_2$ | $x_1$ | $x_L$ |
| Line1 | 0 | $c_4$ | $c_3$ | $hc_2$ | $c_2$ | $hc_1$ | $c_1$ | 0 | $y_1$ | $y_L$ | 0 | 0 |

Figure 13. Redundant Compression of Partial Product Pairs



$\{X_7,Y_5,Z_3\}$
$\{X_5,Y_3,Z_1\}$
Maj
$\oplus$
$\{Z_5,Y_7\}$
$\{Z_4,Y_6\}$
$\{X_6,Y_4,Z_2\}$
$\{X_4,Y_2,Z_L\}$
0 $Z_7$ $C_4$ $Z_6$ $S_4$ $C_3$ $S_3$ $C_2$ $S_2$ $C_1$ $S_1$ 0 $Y_1$ $X_3$ $Y_L$ $X_2$ 0 $X_1$ $X_L$
$r_{11}$ $r_{10}$ $r_9$ $r_8$ $r_7$ $r_6$ $r_5$ $r_4$ $r_3$ $r_2$ $r_1$ $r_0$

Figure 14. Summation of Three PPs in One Binary Adder Layer

At this point, each column has a maximum of two bits, and can be added together with an 11 bit ripple carry adder. The two LSBs form part of the sum without addition, and one bit is needed for overflow from the $z_7$ position. Figure 14 shows how the summation of three partial product pairs can be mapped to a single adder level.

## D. Operand Width Compression

The carry chain in the above example can be reduced to 10 bits, using the existing logic. This will potentially increase system speed (because of a shorter carry chain). System fitting may also be improved, especially if many multipliers are used (as is the case in most machine learning applications). Not only can an additional LSB (in column 2) be calculated combinatorially (thereby given greater flexibility in the placement of the multiplier), this method can be applied when the final carry chain uses a non-integer number of ALMs (in the case of an 11 bit carry chain, 5.5 ALMs are used, restricting the independent utilization of the final LUT).

Figure 15 shows the 2:2 compression on the least significant bits from the compressed three bit columns, i.e. columns 2, 3, and 4.

The logic equations are:

$$s_{L1} = x_2 \oplus y_L \qquad c_{L1} = x_2 \cdot y_L$$
$$s_{L2} = x_3 \oplus y_1 \qquad c_{L2} = x_3 \cdot y_1$$

This may not give a logic reduction in all cases. If we consider each pair of columns mapped to an ALM, the largest number of independent inputs of an ALM would be in the case of a full pair of 3:2 compressors implemented in logic.

One example is the ALM containing $\{s_1, c_{L2}, hs_1, c_1\}$, which expanded out is the set $\{x_4 \oplus y_2 \oplus z_L, x_3 \cdot y_1, hs_1, \text{Majority}(x_4, y_2, z_L)\}$. The independent inputs in this case are $x_4, y_2, z_L, x_3, y_1$, and $hs_1$. This will only be possible if the ALM allowed fully independent inputs to both ripple carry adder bits. In a more typical FPGA, $c_{L2}$ (i.e. $(x_3 \cdot y_1)$) would need to be calculated externally in a auxiliary cell, and then connected to the actual LUT routing. This would somewhat defeat the point of the LSB 2:2 compressors, although the improved placement characteristics, as explained above, may make this worthwhile.

| **Col.** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Line0 | $z_7$ | $z_6$ | $s_4$ | $s_3$ | $hs_2$ | $s_2$ | $hs_1$ | $s_1$ | $s_{L2}$ | $x_{L1}$ | $x_1$ | $x_L$ |
| Line1 | 0 | $c_4$ | $c_3$ | $hc_2$ | $c_2$ | $hc_1$ | $c_1$ | $c_{L2}$ | $c_{L1}$ | 0 | 0 | 0 |

Figure 15. Further Redundant Compression of Partial Product Pairs

In the final adder configuration described in Figure 15, the most densely routed ALM is $\{s_2, hc_1, c_2, hs_2\}$ , or $\{x_6 \oplus y_4 \oplus z_2, hc_2, \text{Majority}(x_6, y_4, z_2), hs_2\}$. Although there are 5 independent variables here, the manner that the routing is mapped to the two LUTs in the ALM is supported.

## E. Signed Multiplier Analogue

Optimization of the unsigned multiplier is simpler than the signed multiplier, as extensions of partial results can be made with zeroes. Another issue with signed multipliers is the possible generation of the true negative (2's complement) of the multiplicand.

| **Col.** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
|---|---|---|---|---|---|---|---|---|
| PP0 | 0 | 0 | $(p_{0,3})$ | $(p_{0,3})$ | $p_{0,3}$ | $p_{0,2}$ | $p_{0,1}$ | $p_{0,0}$ |
| PP1 | 0 | 0 | $(p_{1,3})$ | $p_{1,3}$ | $p_{1,2}$ | $p_{1,1}$ | $p_{1,0}$ | 0 |
| PP2 | $(p_{2,3})$ | $(p_{2,3})$ | $p_{2,3}$ | $p_{2,2}$ | $p_{2,1}$ | $p_{2,0}$ | 0 | 0 |
| PP3 | $(p_{3,3})$ | $p_{3,3}$ | $p_{3,2}$ | $p_{3,1}$ | $p_{3,0}$ | 0 | 0 | 0 |

Figure 16. $4 \times 4$ Signed Multiplier Pencil and Paper Partial Products

| **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
|---|---|---|---|---|---|---|---|
| $(s_4)$ | $s_4$ | $s_3$ | $s_2$ | $s_1$ | $p_{2,0}$ | 0 | 0 |
| $c_4$ | $c_3$ | $c_2$ | $c_1$ | Comp | 0 | 0 | 0 |

Figure 17. Signed $4 \times 4$ Multiplier Compressed Partial Products

Figure 16 shows a partial product set in the style of Figure 1 for a 4x4 signed multiply, with the sign extensions denoted by (). The first three partial products are calculated by a ANDing of the multiplicand bits with the respective multiplier bit - the same as for the unsigned multiplication, although in this case a sign extension is also performed. The last partial product is negated if the multiplier MSB is '1'. This is normally implemented by inverting the bits of the multiplicand, and then adding a '1' to the LSB position of that partial product. Here there is no obvious place to add the '1' bit. Using a separate adder for the single bit would be inefficient.

One solution is to add the first two partial products normally, but use a 2:2 compression on the second pair, offset by a bit, to create '0' in the place of the $p_{3,0}$. This is shown in Figure 17. The bit 'Comp' is '1' in the case of a negative multiplier, '0' otherwise.

The 2:2 compression here is a half adder analogous to the method of Figure 15. Note that each bit is the logical AND of two inputs bits, but the distribution of the routing in the ALM makes it possible to fully route this pattern. There is also an alternate way of implementing the 1s to 2s complement addition. By inspection of Figure 17, the bit position underneath $p_{2,0}$ bit does not contain a value. Also, the $L$ bit can either be '0' or '1'. Adding a '1' into the $p_{3,0}$ bit can be accomplished by forcing a carry in from the $p_{3,0}$ bit position into the ripple carry adder. Note that the $p_{3,0}$ bit

Table I
COMPARED MULTIPLIER AREA AND LATENCY RESULTS

| Precision | Ours | | Intel | | Xilinx | |
|---|---|---|---|---|---|---|
| | Area | Depth | Area | Depth | Area | Depth |
| 4x4 | 8 | 1 | 11 | 2 | 12 | 2 |
| 5x5 | 13 | 2 | 22 | 3 | 20 | 3 |
| 6x6 | 21 | 2 | 30 | 3 | 24 | 3 |
| 7x7 | 25 | 2 | 34 | 3 | 36 | 4 |
| 8x8 | 36 | 3 | 36 | 3 | 40 | 4 |
| 9x9 | 43 | 3 | 48 | 4 | 55 | 5 |

cannot be changed, but if the ripple carry adder started at $p_{3,0}$ instead of $p_{2,0}$, feeding a 1 into both the empty column at $p_{2,0}$ and as a carry in to $p_{2,0}$ will both leave $p_{2,0}$ unchanged, and force a carry in into the rest of the ripple carry adder.

## IV. RESULTS

To normalize the comparisons, we will describe the resources used in terms of 6LUTs. As described earlier in Section II, Intel and Xilinx devices have different logic structures: the Intel 6LUT (ALM) fractures into two bits, each with two 4LUTs as inputs, and the Xilinx 6LUT contains a single bit, with the 6LUT fracturing into two 5LUTs, which feed a single bit adder. Kumm and Walters both map a Booths coding to the deeper logic structure of the Xilinx architecture while our approach uses the denser arithmetic structure of the Intel devices.

Based on the algorithms of Section III, there are two main components of a regularized multiplier: partial product generation, and the summation of the partial products. The first level summation of the partial products occurs in the same logic where the partial products are created, and the sums of partial product pairs are then added in the second level. Recall that multiplier regularization creates a set of partial product pairs, and that the summation of partial products contains a multiple of pairs.

For an MxN multiplier, the number of ALMs in the partial product pairs is approximately:

$$PP = \left(\frac{M}{2}+2\right)\left\lfloor\frac{N}{2}\right\rfloor + (N \bmod 2)$$

and the number of ALMs in the summation of the pairs:

$$\sum = \left(\frac{M}{2}+2\left(\left\lfloor\frac{N}{2}\right\rfloor-1\right)\right)(\lfloor\log_2 N\rfloor-1)+(\lfloor\log_2 N-1\rfloor \bmod 2)$$

Conceptually, the there will always be an even pair of partial products generated using the out-of-band auxiliary cells. The width of this pair (which incorporates a one bit left shift of the more significant partial product, plus an additional bit for wordgrowth) will be two bits more than the precision of the multiplicand. The number of partial products will also be an even pair, then arranged in a reduction tree. A small additional number of auxiliary cells is required to ensure this.

In Kumm [10], multiplier resources requirements are expressed in slices:

$$Area(slices) = \left\lceil\frac{M}{4}+1\right\rceil\left\lfloor\frac{N}{2}+1\right\rfloor$$

where each slice contains 4 bits, or four 6LUTs.

The [10] and [14], [15] multipliers appear to be almost identical for array multiplication, so we will use the methodology behind the area equation above for the Xilinx numbers. Note that this equation will provide pessimistic results for the lower precisions typically associated with machine learning inference, so we manually construct the architectures for these, producing possibly slightly optimistic results. For example, we compute 40 6LUTs for the 8x8 unsigned case; Walters reports 43 6LUTs.

We will use the Intel Megafunction results (in the Quartus tool), without ternary addition, for the Intel numbers. We will not report Xilinx Logicore numbers, as both Kumm and Walters report that their algorithms are significantly smaller that the Xilinx IP. Our numbers are rounded up to the nearest ALM; in several of these multipliers, the actual area is one half ALM smaller, which can be reused for other logic in some cases. Logic depth, rather than latency, is given to focus on the arithmetic core, rather than the complication of normalizing the cost of any pipeline balancing. The comparison of area and delay of the three methods is given in Table I.

Note that our 4x4 implementation de-regularizes the multiplication first (by splitting it into a 4x3 and 4x1) and then regularizes it (by collapsing the 4x3 into a 4x2 plus a auxiliary cell), and then adding the result of the 4x2 multiply to the unused adder in the 4x1 calculation. This multiplier has therefore a deeper combinatorial path internally. This is unlikely to impact system speed, as the span of this logic is very low.

The area and latency of the Intel Megafunction IP compares closely to the Kumm and Walters results. Our results are typically 10% to 35% smaller, but the latency is significantly reduced, with a 25% to 50% reduction.

For completeness, we also report area and latency for non-symmetric multipliers in Table II, which show that the method of this paper applies to non-square multipliers as well.

Table II
ASYMMETRIC AREA AND LATENCY RESULTS

| Precision | Area | Depth |
|---|---|---|
| 4x3 | 6 | 1 |
| 5x4 | 11 | 2 |
| 5x3 | 7 | 1 |
| 6x5 | 16 | 2 |
| 6x4 | 12 | 2 |
| 6x3 | 8 | 1 |
| 7x6 | 23 | 2 |
| 7x5 | 19 | 2 |
| 7x4 | 14 | 2 |
| 7x3 | 9 | 1 |

Both our reported results, and the current Intel Megafunction multipliers, were synthesized and fit using Quartus 17.1, targeting a Stratix 10 10S280ES-2 device. We do not report speeds here. For both the new multipliers and the Megafunction multipliers, we were able to achieve over 700MHz by pipelining the multipliers fully, for all of the reported precisions. This, however, is not a true indication of multiplier performance. The implementation of the machine learning systems that are likely to use these operators will typically use many thousands in a single design. These system level designs will reveal additional characteristics about the advantages and disadvantages of both the new and known algorithms, such as the impact of ALM input routing density, ALM packing efficiency, and the effect of heterogenous routing paths, such as the placement effects of the logic on the carry chain compared with the out-of-band auxilary logic.

## V. Future Work

The true impact of these new multipliers can only be seen when aggregated into large datapaths, such as the dot product structures typically used in machine learning applications. We have some early results, where device filling designs on the Stratix 10 10S280ES-2 device achieve in excess of 500MHz. There are numerous additional steps in packing such a large number of operators efficiently, and as stated previously, the analysis of routing stress is beyond the scope of this paper. Our next steps will be to construct a number of designs, across a wide set of precisions, and formalize the methods needed to achieve consistent placement efficiencies and performance.

## VI. Conclusions

In this paper, we have described a set of new methods to improve the area and latency (logic depth) for low precision FPGA soft logic multipliers, such as those increasingly being used for machine learning applications. Compared to recent published results, our area - using a FPGA 6LUT metric - is up to 35% smaller. In addition, our logic depth is greatly reduced, with 25% to 50% fewer levels. Our methodology optimizes the Intel arithmetic soft logic structure, which fractures to two bits per ALM. We also briefly analyse FPGA routing density, with the goal of managing the number of independent routes to a logic group. Although ternary addition is available on Intel devices, we specifically architect our multipliers to use only binary adders, and try to limit the average number of independent routes to a logic group to near this level.

## References

[1] *Intel Stratix 10 High-Performance Design Handbook*, 2017, https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/stratix-10/s10_hp_hb.pdf.

[2] *UltraScale Architecture and Product Data Sheet: Overview*, 2018, https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf.

[3] M. Langhammer and B. Pasca, "Floating-point DSP block architecture for FPGAs," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: ACM, 2015, pp. 117–125. [Online]. Available: http://doi.acm.org/10.1145/2684746.2689071

[4] *Microsoft unveils Project Brainwave for real-time AI*, 2017, https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/.

[5] *UltraScale Architecture – DSP Slice. User Guide*, 2017, https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf.

[6] *Arria10 Device Overview*, 2014, http://www.altera.com/literature/hb/arria-10/a10_overview.pdf.

[7] *7 Series DSP48E1 Slice User Guide*, 2016, https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf.

[8] *CycloneV Device Handbook*, 2015, https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/cyclone-v/cyclone5_handbook.pdf.

[9] *Deep Learning with INT8 Optimization on Xilinx Devices*, 2017, https://www.xilinx.com/support/documentation/white_papers/wp486-deep-learning-int8.pdf.

[10] M. Kumm, S. Abbas, and P. Zipf, "An efficient softcore multiplier architecture for xilinx fpgas," in *2015 IEEE 22nd Symposium on Computer Arithmetic*, June 2015, pp. 18–25.

[11] C. R. Baugh and B. A. Wooley, "A two's complement parallel array multiplication algorithm," *IEEE Transactions on Computers*, vol. C-22, no. 12, pp. 1045–1047, Dec 1973.

[12] A. D. Booth, "A signed binary multiplication technique," *The Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951. [Online]. Available: +http://dx.doi.org/10.1093/qjmam/4.2.236

[13] O. L. Macsorley, "High-speed arithmetic in binary computers," *Proceedings of the IRE*, vol. 49, no. 1, pp. 67–91, Jan 1961.

[14] E. G. Walters, "Partial-product generation and addition for multiplication in FPGAs with 6-input LUTs," in *2014 48th Asilomar Conference on Signals, Systems and Computers*, Nov 2014, pp. 1247–1251.

[15] ——, "Array multipliers for high throughput in xilinx FPGAs with 6-input LUTs," *Computers*, vol. 5, no. 4, 2016. [Online]. Available: http://www.mdpi.com/2073-431X/5/4/20