Extracting INT8 Multipliers from INT18 Multipliers

Martin Langhammer, Bogdan Pasca, Gregg Baeckler, Sergey Gribok

Extracting INT8 Multipliers from INT18 Multipliers

Martin Langhammer
Intel

Bogdan Pasca
Intel

Gregg Baeckler
Intel

Sergey Gribok
Intel

***Abstract*—With the advent of machine learning as perhaps the most high-profile application area for FPGAs, there is a compelling reason to improve the provision of smaller precision arithmetic on these devices. INT8 is commonly used for AI inferencing, and along with some additional soft logic for exponent handling, can be an effective solution for training as well. This paper describes techniques for efficiently extracting INT8 multipliers from commonly available INT18 multipliers found in many modern FPGAs. A small amount of soft logic - as little as 7 ALMs per INT8 multiplier - is required to provide pre or post multiplier correction to calculate two INT8 multiplies from a single 18x18 multiplier. We present two configurations for both signed and unsigned representations where two multiplications share one input operand. In addition to the individual INT8 variants, we present full device cases of 22,400 INT8 multipliers organized as DOT32 product arrays, with the soft logic tightly bound to the INT18 based DSP Blocks. A majority of the soft logic and routing in the device is left untouched, and available for application development.**

I. Introduction

Earlier Altera Stratix devices contained DSP Blocks where 18x18 multipliers were decomposable into INT9 multipliers [1], [2]. In the past, application areas like video and broadcast were important markets for FPGA, and these precisions aligned with the required widths for video and image processing [3], [4]. With newer, much larger FPGAs being introduced, the hardware cost and complexity of designing and providing smaller multipliers was largely made redundant by the provision of many thousands of 18x18 multipliers. These could act down as INT8 or INT9 multipliers, or alternately, allow for local wordgrowth within a series of calculations.

In [5], Xilinx showed methods to extract two INT8 multipliers from the Ultrascale and Ultrascale+ DSP48E2 Blocks [6], which contain a 27x18 multiplier. Both signed and unsigned representations are described, but in all cases, the two multipliers share an input. The application note explains that many AI inferencing implementations would not be disadvantaged by this restriction. Only multiplier pairs are described, and systems (where many multipliers would be used) are not implemented, and only presented by calculation. One of the main assertions of the document is that a minimum of a 27x18 multiplier precision is required to support two INT8 multipliers. Implementing multiply-accumulate (MACC) operations (up to 7 for each of the two INT8 multipliers) could be done inside the DSP48E2 Block, and other DSP Blocks can be used to support more accumulations.

In contrast, we describe two methods to extract smaller multipliers from larger multipliers, where the smaller multipliers are not fully arithmetically separated in the larger multiplier. We use the extraction of INT8 multipliers from the 18x18 multiplier as our motivating example. Normally, if the four input operands of the INT8 multipliers are arranged so that they are mapped to the two inputs of the 18x18 multiplier, the overlap in the partial products leads to considerable errors (cross contamination) in the two smaller multiplier outputs, making the results unusable. We introduce two methods for correcting these using a modest amount of soft logic.

One advantage of our approach is that virtually all of the datapath of the larger multiplier contribute to the calculated result. This will result in increased system computational density, and considerably reduced power consumption on a multiplier by multiplier basis. (In cases where arithmetic seperation is used [5], most or all of the multiplier datapath switches, but a considerable portion of the output must be ignored).

The 18x18 multiplier is relatively more gate efficient that the 18x27 multiplier - a large contribution of this is that the carry-propagate adder (CPA) on the output side is smaller. The internal compressors, being in redundant form, have relatively the same efficiency. The 18x18 multiplier is less than 60% the gate count of the 27x18 multiplier (also taking into account the weaker gate drive requirements for timing closure of the shallower logic of the smaller multiplier, and the much smaller span of the CPA), so using the 18x18 yields a 50% greater computational density and 50% improved power density in the hard logic portion of our system. In addition, we describe several further methods to mitigate the cost of the soft logic correction, such as when used in a DOT product, as is typically the case for many AI implementations.

We describe two cases: (1) the extraction of two unsigned INT8 multipliers with a shared operand, using some post processing correction, and (2) the extraction of two signed INT8 multipliers with a shared operand, using both pre-adders embedded in the DSP Blocks, along with some additional post correction soft logic. We analyze the resource utilization of our proposed multipliers in isolation, and also in the context of dot-product - where correction operations can be delayed and merged with the dot-product adder tree. Finally we extend our study to chip-filling designs where we show the scalability of our multipliers by packing 700 32-element dot-products (97% of the total DSPs) in a Stratix 10 device, and we obtain push-button a frequency of over 350MHz, and over 400MHz with a small amount of direction.

Bit weight	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
P									**b7**	**b6**	**b5**	**b4**	**b3**	**b2**	**b1**	**b0**	0	0	**c7**	**c6**	**c5**	**c4**	**c3**	**c2**	**c1**	**c0**
Q									0	0	0	0	0	0	0	0	0	0	**a7**	**a6**	**a5**	**a4**	**a3**	**a2**	**a1**	**a0**
											y15	y14	y13	y12	y11	y10	y9	y8	y7	y6	y5	y4	y3	y2	y1	y0
	z15	z14	z13	z12	z11	z10	z9	z8	z7	z6	z5	z4	z3	z2	z1	z0										
O=PxQ	o25	o24	o23	o22	o21	o20	o19	o18	o17	o16	o15	o14	o13	o12	o11	o10	o9	o8	o7	o6	o5	o4	o3	o2	o1	o0

Fig. 1: Two unsigned 8x8 multiplier with one shared input connection patterns

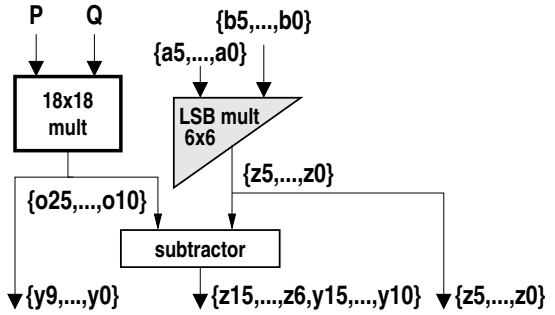


Fig. 2: Architecture of two unsigned 8x8 multipliers using one shared input

II. Multiplier Methods

All of the four following methods assume that an embedded 18x18 multiplier is available in the FPGA. Later in this paper, we will report results for Intel Stratix 10 [7], where a DSP contains either two 19x18 signed multipliers, or two 18x18 unsigned multipliers. Our methods only require 18x18 functionality, which will allow this method to be ported to a wide variety of commercial and academic FPGA architectures. In the case of Intel FPGAs, the two embedded multipliers per DSP can be accessed completely independently; four INT8 multipliers can therefore be extracted from a single DSP Block.

Most of our correction methods will use what we refer to as an LSB multiplier. We will need an n-bit LSB result from an $n \cdot n$ multiplication. This will be inexpensive, in terms of both area and depth: first, the LSB multiplier will only operate on a small precision, typically in the range of 3 to 6 bits, and secondly, calculating only the LSBs will require a small subset of the full multiplication. We do not discuss the structure of these multipliers in this paper, but they can be built by several methods: (i) the smallest precisions can be mapped directly to look up tables, (ii) standard soft multiplier IP can be used, relying on the synthesis tools to optimize away the unneeded portions of the operation, or (iii) specific multiplier circuits can be designed, knowing that carries into the upper bits will never be needed, giving more optimization points to consider.

We use several types of operations in our methods: **Create**, which generates input vectors to the circuits, usually by appending smaller vectors and individual bits together, **Multiply**, using the 18x18 or specified LSB multipliers (*e.g.* **Multiply 6LSB6**, the self-explanatory **Subtract** and **Add**, and finally **Assemble**, which generates the output values by concatenating smaller processed vectors and individual bits.

The methods are presented purely combinatorially. Typically, pipelining (and additional balancing registers) would be used in a real-world application. The resource cost of registers will be included in the reported results in the next section.

A. Unsigned INT8, Correction Method (Shared Inputs)

Here we compute the two products $Y = A \cdot C$ and $Z = A \cdot B$ using an 18x18 multiplier, a 6x6 LSB multiplier, and a soft logic subtractor. All three inputs A, B and C are 8-bit unsigned numbers, and the 18x18 multiplier is configured as an unsigned multiplier as well. The 8 bits corresponding to each of the 3 inputs (A, B and C) are applied on the input ports P and Q of the 18x18-bit multiplier, which then generates the result $O = P \times Q$. Figure 1 depicts the connection patterns. We can then extract Y and Z, which are shown as virtual internal DSP Block signals in the figure. We obtain the bits $\{y9,...,y0\} = \{o9,...,o0\}$ directly. In order to recover the bits $\{y15,...,y10\}$ we use make the observation:

$$\begin{aligned}\{o25,...,o10\} &= \{y15,...,y10\} + \{z15,...,z0\} \\ &= \{z15,...,z6,y15,...,y10\} + \{z5,...,z0\}\end{aligned}$$

Therefore:

$$\{z15,...,z6,y15,...,y10\} = \{o25,...,o10\} - \{z5,...z0\}$$

Summarizing, the steps of this algorithm are as follows:

- **Create** two 18 bit input vectors P and Q.
  - $Q = \{a7,...,a0\}$ (zero extended)
  - $P = \{c7,...,c0\}$,"00",$\{b7,...,b0\}$
- **Multiply**
  - $O = P \times Q$
- **Multiply LSB6**:
  - Use an LSB multiplier to obtain the lower 6 bits of z. $\{z5,...,z0\} = \{a5,..,a0\}\{c5,..,c0\}[5:0]$
- **Extract**
  - The lower 10 bits of y are obtained directly from the 18x18 multiplier: $\{y9,..,y0\} = \{o9,..,o0\}$
- **Subtract**

  $$\{z15,...,z6,y15,...,y10\} = \{o25,...,o10\} - \{z5,...,z0\}$$
- **Assemble** $Y = \{y15,...,y10\},\{y9,...,y0\}$.

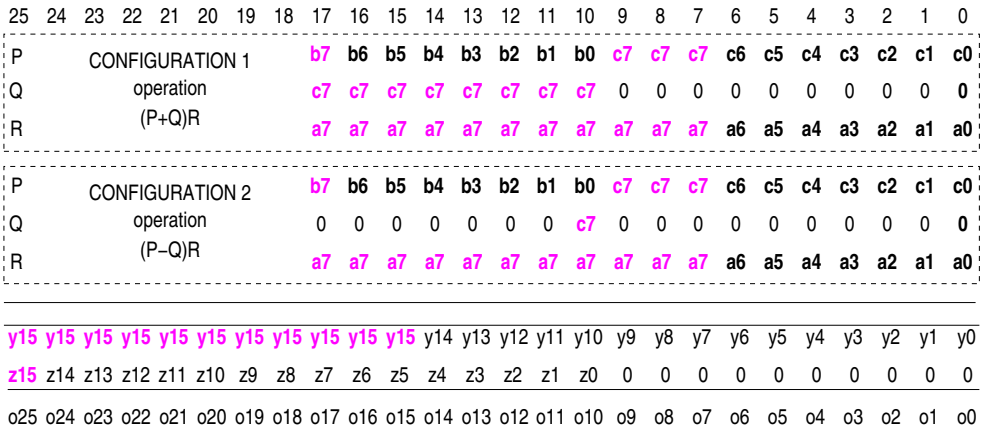


Fig. 3: Connection Patterns for two signed 8x8 multipliers with one shared input

- **Assemble** $Z = \{z15,...,z6\},\{z5,...,z0\}$

The architecture is depicted in Figure 2. The vectors P and Q feed the 18x18-bit unsigned multiplier, with an output $\{o25,...,o0\}$. The bottom 10 bits $\{o9,...,o0\}$ correspond directly to $\{y9,...,y0\}$, and are therefore forwarded to the output. The LSB multiplier (which produces the correct 6 LSBs of Z) operates in parallel with the 18x18 hardened multiplier. The computed correction bits $\{z5,...,z0\}$ are then subtracted from $\{o25,...,o10\}$ to produce both the upper bits of Y, $\{y15,...,y10\}$ and the upper bits of Z, $\{z15,...,z6\}$. The correction bits then form the 6 lower bits of Z.

B. Signed INT8, Correction Method (Shared Inputs)

The calculations $Y = A \cdot C$ and $Z = A \cdot B$ use the three inputs A, B and C as 8-bit signed numbers. The 18x18 multiplier is now configured as a signed multiplier, with one of the inputs connected to the pre-adder. Refer to Figure 3 for the connection patterns. The algorithm steps are given below:

- **Create**
  - $R = \{a7,...,a0\}$, (sign extended). Connect this to one multiplier input directly.
  - $P = \{b7,...,b0,c7,c7,c7,...,c0\}$ and connect to one pre-adder input.
  - There are two possible configurations of the Q vector depending on the pre-adder configuration:
    1) Configuration 1:
       Create $Q = \{8'c7,10'b0\}$. The pre-adder is configured as an adder.
    2) Configuration 2:
       Create $Q = \{7'b0,c7,10'b0\}$ The pre-adder is configured as a subtractor.
  - There are two possible configuratitons for the multiplication, depending on the pre-adder:
    1) Configuration 1: **Multiply** $O = (P+Q)R$
    2) Configuration 2: **Multiply** $O = (P-Q)R$.
- **Multiply 6LSB**:
  - Use an LSB multiplier to obtain the lower 6 bits of z.

$$\{z5,..,z0\} = \{a5,..,a0\}\{b5,..,b0\}[5:0]$$

- **Extract**
  - The low 10-bits of y can be read directly on the multiplier output.

$$\{y9,...,y0\} = \{o9,...,o0\}$$

- There are two possible output subtract/add types:
  - Type 1: **Subtract**

$$\{z15,..,z6,y15,..,y10\} = \{o25,..,o10\} - \{10'y15,z5,..,z0\}$$

  - Type 2: **Add**

$$\{cOut,y15,...,y10\} = \{0,o15,...,o10\} + \{0,\overline{z5},...,\overline{z0}\}$$
$$\{z15,...,z6\} = \{o25,...,o16\} + \{\overline{y15},...\overline{y15}\} + cOut$$

- **Assemble**:
  - $Y = \{y15,...y10\},\{y9,...,y0\}$
  - $Z = \{z15,...,z6\},\{z5,...,z0\}$

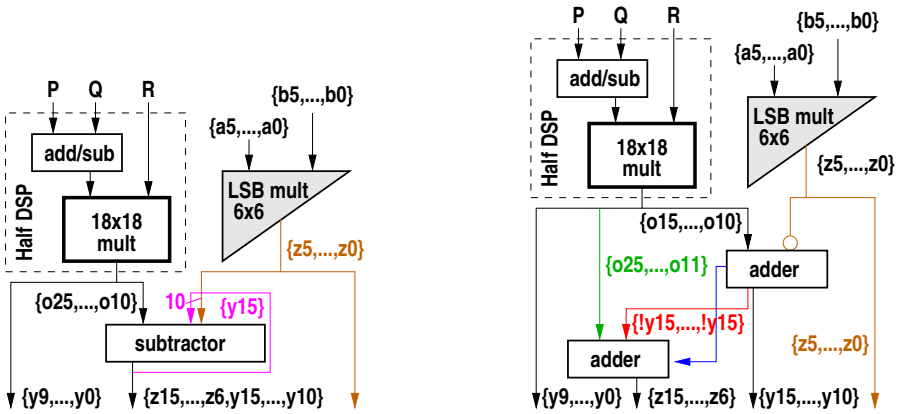
In the first type of output subtraction, bit $y15$ at the output of the subtractor is fed forward to all more significant inputs of the subtractor. This architecture is shown in Figure 4a, with the feedback connections of $y15$ detailed in Figure 5.

Figure 4b shows the second add type, which is instead expressed as a chaining of operations, which converts the feedback of $y15$ into a feedforward - using two adders, each with about half the precision as the other subtraction. The overall cost of the post processing is therefore approximately the same in both cases, although additional pipelining may be required for the second one.

As in the unsigned case, the LSB multiplier computes the low 6 LSBs of $Z = A \cdot B$.

III. Results

We will analyze the area and performance at three levels. First, the relationship between individual INT8 multipliers, DSP Blocks, and soft logic will be examined. Next, we will



(a) a (b) b

Fig. 4: Extraction Architecture for two signed INT8 multipliers with one shared input

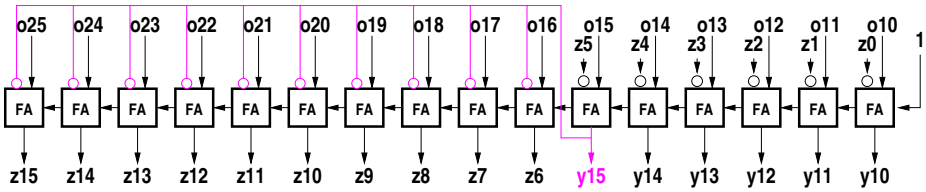


Fig. 5: Application of the Y15 bit for Figure 4a.

construct a single DOT product for a performance baseline, and understand the additional soft logic required. Finally, we will show some large system designs, where a large FPGA is filled with DOT products. This should give the reader confidence that our methods scale, maintaining a reasonable full chip performance, and managing routing stress so that additional functions can also be implemented in the remainder of the device.

A. Multiplier Sizes

The soft logic cost of an INT8 multiplier can be considered in multiple ways: first, the cost of an individual multiplier - or in this case, a pair of multipliers, second, the cost of extracting 4 multipliers per DSP Block, and finally the average cost of implementing a typical use-case kernel, such as a INT8 DOT32 function. As the number of multipliers in a local group increases, the correction circuit can itself be extracted, and shared between all operators in the group. This amortization of the correction circuit can lead to significant average logic savings. We will illustrate this by analyzing Case A from Section II (Unsigned INT8, Correction Method, Shared Inputs).

There are two soft-logic components of the multiplier pair structure: the LSB multiplier, and the subtractor to apply it to the left multiplier in the pair. The LSB multiplier requires 8 ALMs, and the 16 bit subtractor also needs 8 ALMs. In most cases, however, several (or many) multipliers are summed together, and the correction can be amortized over the entire reduction operation. Only the correction to the upper bits of $a \cdot c$ is needed at the output of the embedded multiplier, and the correction to $a \cdot b$ can be accumulated separately, and subtracted as a single value at the end of the DOT product. The immediate, or local cost of this method is therefore 12 ALMs (8 for the LSB multiplier and 3 for the ac correction). An ALM is required to calculate the carry in to the $a \cdot b$ upper bit correction, and on average, another ALM to bring this value forward to the next level, by adding it to the adjacent carry forward. For a typical application, 7 ALMs are needed to extract an INT8 multiplication from an 18x18 embedded multiplier. For high performance (i.e. high clock frequency) designs, we also need to pipeline appropriately. This requires additional balancing registers. These add another 19 ALMs per DSP Block.

Table I shows the resource utilization for all our proposed shared-input multipliers, both for signed and unsigned configurations. Since we were unable to find multipliers built with comparable techniques, we list independent-input soft-core multipliers constructed for the same sizes. We use information from [8] since it normalizes the data for Virtex6 [9] on the architectures from [8], [10] and [11]. We also list [12] which gives the cost of an 8x8 soft-logic multiplier for LUT6-based FPGAs and [13] which shows area for a soft-logic 8x8 multiplier on Stratix 10.

B. DOT Product Sizes

We used a pair of 16-element INT8 dot-products (pair-of-dot16s) as an instance for our system-level designs. A simplified structure for a pair-of-dot4s using two DSP Blocks is illustrated in Figure 6. The compute kernel is that of Figure 2 but with changes for making the dot-product more efficient -

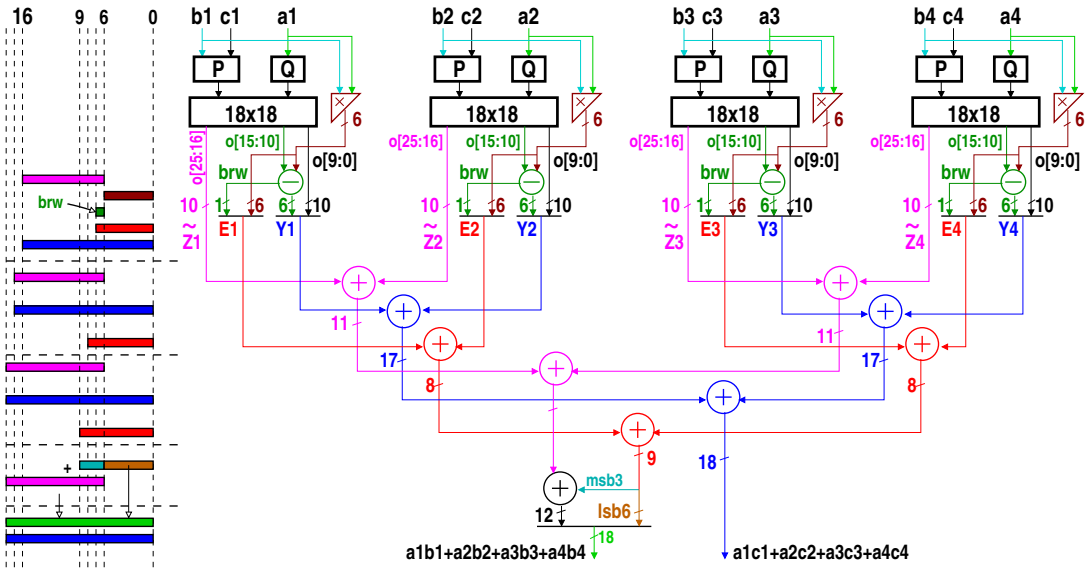


Fig. 6: Two 4-element shared dot-products using eight 8x8u multipliers

Arch.	Case	Type	18x18 (two)	DSP (four)	ALMs/ int8
ours	II.A	Standalone	16 ALMs	32 ALMs	8
		Local	12 ALMs	24 ALMs	6
		Scaled	14 ALMs	28 ALMs	7
	II.B (a)	Standalone	16 ALMs	32 ALMs	8
	II.B (b)	Standalone	17 ALMs	34 ALMs	8.5
[8]	8x8	Standalone	15 SLICES, 0 DSPs		60 LUT6/int8
[10]	8x8	Standalone	18 SLICES, 0 DSPs		72 LUT6/int8
[11]	8x8	Standalone	21 SLICES, 0 DSPs		84 LUT6/int8
[12]	8x8	Standalone	43/44 LUT6, 0 DSPs		43/44 LUT6/int8
[14]	8x8	Standalone	36 LUT6, 0 DSPs		36 LUT6/int8
[13]	8x8	Standalone	36 ALMs, 0 DSPs		36
[5]	8x8	Standalone	2 8x8 per 27x18 DSP		0

TABLE I: Resource utilization of our proposed architectures. [8], [10], [11] give results for 8x8 multipliers on Xilinx Virtex6

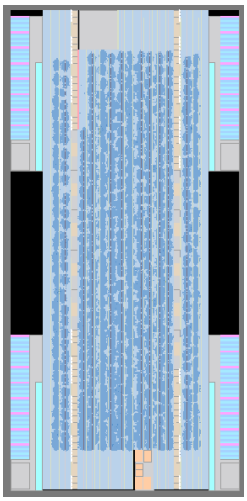
these will be explained below. The reduction flow consists of three adder trees, which are then combined at the output level. The first adder tree (pink) sums the potentially polluted upper 10 bits of $Z_i = a_i b_i$ that we denote by $\widetilde{Z}_i$. These bits are available directly at the 18x18 multiplier output. The second adder tree (blue) sums the $Y_i = a_i c_i$ products. The lower 10 bits of Y_i correspond to bits $\{o9,..,o0\}$ from the 18x18 multiplier output. The upper 6 bits of Y_i are obtained by subtracting the overlapping $\{z5,..,z0\}$ bits from $\{o15,..,o10\}$ (see Figure 2). The borrow bit that is potentially produced by this subtraction indicates that the overlap has contaminated $\widetilde{Z}_i$. This bit has weight $-2^6 \cdot brw$ and should added to $\widetilde{Z}_i$. We show next how this can be integrated with the third adder tree with a minimal cost. Finally, the third adder tree (red) sums the bottom 6 bits of Z_i which are computed using the LSB multiplier. We integrate the *brw*-bit to each Z_i by concatenating it to the left of this 6 bits to produce a 2s complement value (on 7 bits). One additional adder is required to sum the overlapping contribution of the red tree over the pink tree (the alignment is shown on the left of Figure 6, bottom left).

We compiled the pair-of-DOT16s into an Intel Stratix 10 1SG280LN2F43E1VG, and achieved 581 MHz for exactly 600 ALMs (and 8 DSP Blocks, as expected). We can analyze the expected resource count using the architecture of Figure 6. There are three reduction trees, starting with 16, 10, and 7 bits respectively, each with 8, 4, 2, and 1 adders per level. Each adder level does grows the adder size by 1 bit. The number of bits per individual tree are: 251 for blue, 161 for pink and 116 for red - which translates to 264ALMs using 1ALM=2bits. Additionally, we count 3.5x16=56ALMs for the 6-bit subtracters with borrow, and 7 ALMs for the final subtracter required for $\sum Z_i$. These total 327 ALMs and the remaining logic up to 600 ALMs is used for pipelining registers.

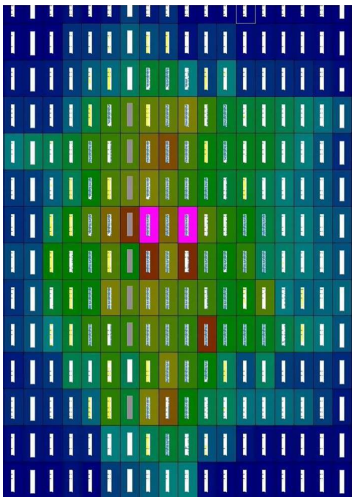
C. Chip Scale Application Examples

To be efficient, these small cores must scale to system level. We demonstrate this by instantiating many into a larger device while maintaining a usable operational frequency. We also show that the remaining logic and routing are left untouched, and are available for other functionality.

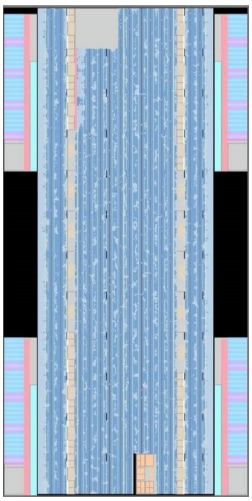
Our first design fit 500 DOT32 cores into the Stratix 10 1SG280LN2F43E1VG device. We used the Quartus 18.1 tool flow, with the Fractal Synthesis directive turned on (in order to pack the arithmetic logic as tightly as possible). We used a completely pushbutton approach, with no floorplanning. This required 4000 DSP Blocks out of the 5760 available (16000 INT8 multipliers). Maximum clock frequency was 457.9 MHz. Not including the virtual pins used for fitting, the 300,347 ALMs represent 32% of the available logic - in other words, the logic required for the arithmetic datapaths as a ratio of



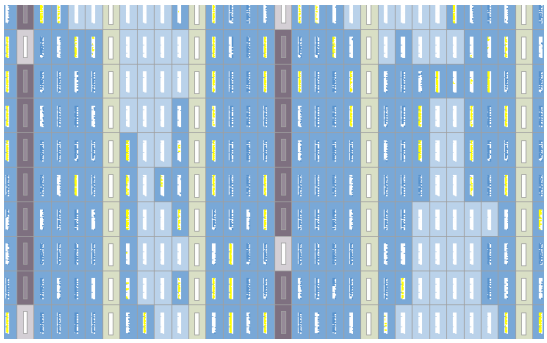
(a) 500 DOT32 Floorplan



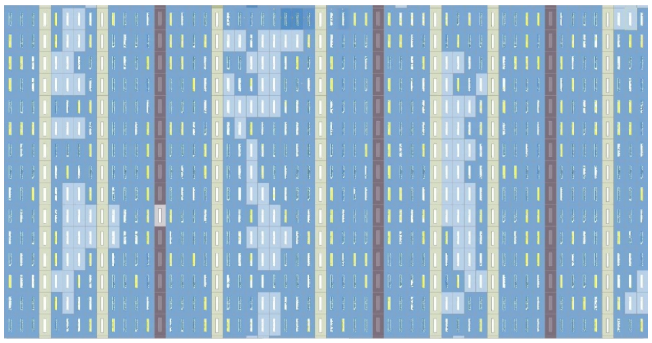
(b) DOT32 Routing Heatmap



(c) 700 DOT32 Floorplan



(d) Detail of 500 DOT32 Design



(e) Detail of 700 DOT32 Design

Fig. 7: Floorplan of various-size dot-product densities

the multiplier count is less than half of the logic available. In arithmetically dense use cases such as machine learning, where we may need the maximum capability of the device, datapath (consisting of a mix of DSP Blocks and logic) compared to application and control (logic only) percentage of the design may be 80% to 20% [15]. This INT8 method will therefore be suitable for these type of designs, with a modest amount of logic required compared to what is available.

Figure 7a shows a floorplan of the 500 DOT32 design. Figure 7d zooms into show the detail of a small number of DOTs. The unused logic is clearly visible, and large groups of contiguous LABs are completely untouched. The routing heatmap of Figure 7b shows that there is no routing required outside the immediate area of any DOT, with only some routing congestion over some of the DSP Blocks.

We then increased the number of DOT32s to 700, which required 5600 out of the 5760 DSP Blocks on the device (97%). A first pushbutton compilation yielded 356MHz, which was largely due to a small number of routes spanning discontinuities (I/O regions) on the device. We then added a single level of registers immediately after the DSP Blocks, and introduced a floorplanning constraint to force each dot product to stay within a sector boundary. While this undoubtedly added to local routing stress and congestion, performance increased to 416.1 MHz. The additional registers increased the size of each DOT to 645 ALMs, for a total logic use of 452K ALMs on the device - still less than half the logic available. High density AI applications should therefore be easily realizable. We believe that with more careful floorplanning, 500MHz performance is readily achievable. Figure 7c shows the device floorplan with 700 DOTs, and Figure 7e a local detail of this design. Empty LABs are visible. Note that about 20% of the used LABs contain virtual pins, which in an actual design would be available for logic.

IV. Conclusion

We have demonstrated that INT8 multipliers can be very efficiently extracted from commonly available FPGA 18x18 multipliers, using only a small amount of soft logic. This soft logic, which can be as little as 7 ALMs per INT8 on average, is much smaller than the die area required for the additional multiplier datapath logic - a 50% increase - to support a direct INT8 extraction. In a typical application scenario, where DOT products are assembled with soft logic, the 7 ALMs forms only a small portion of the total resources required. In terms of flexibility, power consumption, and cost, the error correction

methods can be considered the best for INT8 implementations in current FPGAs.

REFERENCES

[1] *StratixV Device Handbook*, 2011, http://www.altera.com/literature/hb/stratix-v/stratix5_handbook.pdf.

[2] *StratixIV Device Handbook*, 2011, http://www.altera.com/literature/hb/stratix-iv/stx4_5v1.pdf.

[3] A. Corporation, "Broadcast video infrastructure implementation using FPGAs," *Altera White Paper*, Mar. 2007, https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-brdcst0306.pdf.

[4] ——, "Video and image processing design using FPGAs," *Altera White Paper*, Mar. 2007, https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-video0306.pdf.

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

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

[7] *Intel Stratix 10 GX/SX Device Overview*, 2018, https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/s10-overview.pdf.

[8] 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.

[9] *Virtex-6 FPGA Configurable Logic Block User Guide*, 2009, http://www.xilinx.com/support/documentation/user_guides/ug364.pdf.

[10] H. Parandeh-Afshar and P. Ienne, "Measuring and reducing the performance gap between embedded and soft multipliers on FPGAs," in *2011 21st International Conference on Field Programmable Logic and Applications*, Sep. 2011, pp. 225–231.

[11] *LogiCORE IP Multiplier v11.2*, 2011, https://www.xilinx.com/support/documentation/ip_documentation/mult_gen_ds255.pdf.

[12] 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.

[13] M. Langhammer and G. Baeckler, "High density and performance multiplication for FPGA," in *25th IEEE Symposium on Computer Arithmetic, ARITH 2018, Amherst, MA, USA, June 25-27, 2018*, 2018, pp. 5–12. [Online]. Available: https://doi.org/10.1109/ARITH.2018.8464695

[14] E. G. Walters, "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

[15] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger, "A configurable cloud-scale DNN processor for real-time AI," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 1–14.