Synthesis of Floating-point Addition Clusters on FPGAs Using Carry-Save Arithmetic

Amit Verma[1] Ajay K. Verma[2] Hadi Parandeh-Afshar[2] Philip Brisk[3] Paolo Ienne[2]

[1]National Institute of Technology
Rourkela, India

[2]Ecole Polytechnique Fédérale de Lausanne
School of Computer and Communication Sciences
Lausanne, Switzerland

[3]University of California, Riverside
Department of Computer Science and Engineering
Riverside, CA, USA

***Abstract*—A new method to synthesize clusters of floating-point addition operations on FPGAs is presented. Similar to Altera's floating-point data path compiler, it performs normalization once, at the output of the cluster operation. All significands in the clustered operation are denormalized in parallel with respect to the largest exponent: a fixed-point compressor tree then sums the aligned significands, followed by normalization and rounding. Compared to Altera's floating-point datapath compiler, our method reduces the critical path delay by as much as 20%, and area by as much as 29% on Altera Stratix III FPGAs.**

Keywords-Floating-point addition, carry-save arithmetic, FPGA

I. Introduction

Researchers from Altera recently published a floating-point FPGA datapath compiler, which included some non-standard transformations [1, 2]. One of the most effective ones was to fuse together addition clusters of up to sixteen inputs. The significand is extended with four overflow and three underflow bits, permitting up to sixteen inputs to be added and subtracted without overflow or underflow and eliminating intermediary normalization steps inside the cluster; this significantly reduces area and allows more floating-point operators to be synthesized on a fixed-size FPGA, increasing the throughput of the design.

This paper presents a new approach to synthesize floating-point addition clusters. All significands are denormalized with respect to the largest exponent among all input operands, and a compressor tree [3] sums the denormalized significands; the result is normalized, rounded, and packed into a floating-point number along with the resulting exponent and sign bit. Compared to Altera's compiler [1, 2] our approach improves critical path delay up to 20% and area up to 29%.

II. Preliminaries

A. IEEE-754 Floating-point Standard

An IEEE format number is represented as a triple (S, E, F), where S is a single bit representing the *sign*, E is the *exponent*, and F is the *significand*; taken together, S and F represent the significand in sign-magnitude format. S is a single bit; e and f respectively denote the bitwidth of the exponent and significand. For IEEE-754 single-precision format, used throughout this paper, $e = 8$ and $f = 23$.

A *normalized* significand includes a hidden '1' that is not explicitly encoded, i.e., it has the form $1.F$; if not, i.e., the significand has the form $0.F$ and is *denormalized*, as discussed below. The range of a normalized significand is

$$1 \leq 1.F \leq 2 - 2^{-f}\ . \qquad (1)$$

The exponent, E, has a *base* of 2 and a *bias* of $B = 2^{e-1} - 1$; for single-precision format, $B = +127$. Thus, the value represented by an IEEE-754 floating-point number (S, E, F) is

$$(-1)^S \times 1.F \times 2^{E-(B-1)}\ . \qquad (2)$$

The IEEE-754 format includes several "special" numbers that are represented explicitly, including positive/negative zero, positive/negative infinity, denormalized numbers, and not-a-number (NaN), which, occurs, for example, when the square root of a negative number is taken.

The number $(S, 0, 0)$ represents zero; because of the sign bit, both positive and negative zeroes exist. As a consequence, the value 1.0×2^{-B} cannot be represented.

Any floating-point number of the form $(S, 0, F \neq 0)$ is denormalized. Denormalized significands have the form $0.F$, and represent values of the form

$$(-1)^S \times 0.F \times 2^{-(B-1)}\ . \qquad (3)$$

The maximum exponent value is $E_{max} = 2^e - 1 = 2B + 1$. The floating-point representation for NaN is $(S, E_{max}, F \neq 0)$ and for positive/negative infinity is $(S, E_{max}, 0)$.

The default rounding mode is round to nearest, and to even when there is a tie; the user can also specify three alternative rounding modes: round toward positive or negative infinity, or round toward zero (truncation).

The IEEE-754 standard specifies six numerical operations: addition, subtraction, multiplication, division, remainder, and square root. The standard also specifies rules for converting to and from the different floating-point formats (e.g short/integer/long to/from single/double/quad-precision), and conversion between the different floating-point formats.

The standard includes five exceptions involving special numbers; they set a flag and permit computation to continue. The exceptions are as follows: (1) if an operation overflows,

1946-1488/10 $26.00 © 2010 IEEE
DOI 10.1109/FPL.2010.15

IEEE computer society

the result is positive or negative infinity, depending on the sign bit; (2) underflow (when a rounded value is so small that it cannot be represented); (3) division by zero; (4) inexact results, i.e., when the result of an infinite precision operation cannot be represented in the fixed-precision format being used; and (5) invalid (the result of the operation is NaN). There are also additional rules involving arithmetic involving positive and negative infinities and NaN, e.g., $x\ op\ NaN = NaN$. The interested reader should consult the IEEE standard for details.

B. Floating-point Addition and Subtraction

This section describes the concepts underlying floating-point addition. The discussion focuses on a naïve, but small, implementation of a floating-point adder. More complex implementations, such as dual-path adders [4], are generally poor choices on which to base fused FPGA operators, because they consume too much area.

The inputs are floating-point numbers $N_1 = (S_1, E_1, F_1)$ and $N_2 = (S_2, E_2, F_2)$, along with a single bit, *op* that indicates whether addition $(op = 0)$ or subtraction $(op = 1)$ is to be performed. The output is a normalized IEEE-754 floating-point number $N_3 = (S_3, E_3, F_3) = N_1\ op\ N_2$. The five key steps, as follows, are illustrated in Figure 1(a). Special cases (zero, infinity, NaN) are handled externally; if the result is a special case, then the main datapath will be preempted.

Max.Exp: One cannot add or subtract two significands whose exponents differ. The first step of the floating-point addition operator is to compute the maximum of the two exponents.

Denormalize: The second step is to shift the significand corresponding to the smaller exponent right by the exponent difference, $\Delta E = |E_1 - E_2|$, effectively dividing the significand by $2^{\Delta E}$; implicitly, this is compensated by adding ΔE to the smaller exponent, equalizing it with the larger exponent. The right shift denormalizes the smaller significand if $\Delta E > 0$.

Let f be the precision of the initial significand; the denormalized significand has precision $f+3$. The bit in position 3 is called the *guard bit (G)*; the bit in position 2 is called the *round bit (R)*; the bit in position 1, which is the logical-or of all of the remaining bits that have been shifted off, is called the *sticky bit (T)*; these bits are required later for rounding.

Eff.Op: With equal exponents, it is now safe to add the significands. The *effective operation (EOP)* refers to the operation that is performed (either addition or subtraction), and depends on the input bit *op* and the signs of the two operands. Prior work has shown that $EOP = op \oplus S_1 \oplus S_2$, where $EOP = 0$ indicates addition and $EOP = 1$ indicates subtraction [4].

Normalize: The result of the effective operation may not be normalized. The next step is normalization. Different steps must be taken, depending on the effective operation.

First, consider effective addition $(EOP = 0)$. Assume that $E_1 = E_2$, meaning that no alignment shift is required. In this case, the inputs to add are normalized, having the form $1.F_1$ and $1.F_2$, yielding a result of the form $10.F_3$, which indicates an overflow of one bit position. In this case, we shift $10.F_3$ right by one to normalize and increment the exponent. If the result of the effective operation is normalized, then no shift is required.

For effective subtraction $(EOP = 1)$, the result may be denormalized, and the number of leading zeroes before the first one cannot be determined statically. The case where the result is zero is handled as a special case. Assume that the significand resulting from the second effective operation has the form $0.F_3$; we must first count the number of leading zeroes in $0.F_3$ using a component called a *leading zero detector (LZD)*; supposing that there are L leading zeroes, we shift $0.F_3$ left by L (effectively multiplying it by 2^L) and subtract L from the resulting exponent to compensate; the result is normalized.

Round: This final step accounts for the error incurred due to the bits that were shifted right during significand alignment. A value of 1 is added to the significand based on the rounding mode specified and the values of the guard, round, and sticky bits, G, R, and T, as discussed earlier; this may once again cause overflow, such that the result has the form $10.F$; this case is handled identically to normalization for effective addition: the significand is shifted right by one, normalizing it, and the exponent is incremented.

C. Clustered Addition in Altera's Floating-point Datapath Compiler

Altera's floating-point datapath compiler removes internal normalization and rounding operations within a cluster; it performs these operations only at the output, as illustrated in Fig. 1(b). Traditional IEEE-754 floating-point operators use a sign-magnitude representation of significands inside a cluster; Altera's compiler converts the significand to a two's complement integer instead, and extends the significand from 24 bits (including the hidden '1') to 32 bits (which includes the sign bit). Four additional bits are provided for overflow and three additional bits are provided for underflow. The hidden '1' no longer remains in the most significant position.

Within each internal two-input operator, the exponents must still be aligned before the effective operation can be performed. The denormalized significand corresponding to the smaller exponent must still be shifted right by the exponent difference, ΔE. After the final fixed-point addition operation in the cluster, the signed significand is converted back to sign-magnitude form; this is required for normalization. Altera's papers do not explicitly state which rounding modes are supported, whether or not normalization uses a sticky bit, or how the sticky bit is computed (beyond the extended significand format) if one is used.

With four additional overflow bits, up to 16 normalized positive numbers can be effectively added before overflow; overflow occurs when the signifcand has the form 10000.z. This permits one normalization operation to be performed after adding 16 floating-point numbers; however, the length of the right shift required to normalize the significand will range from 0 to 4; in the floating-point adder described in the preceding section, the length of the shift ranged from 0 to 1.

The three underflow bits help to mitigate negative wordgrowth during effective subtraction; however, these bits may be lost due to the right shifting during significand alignment (denormalization). For large shift values during denormalization and effective subtraction, the result may be less accurate than using IEEE-754 operators.

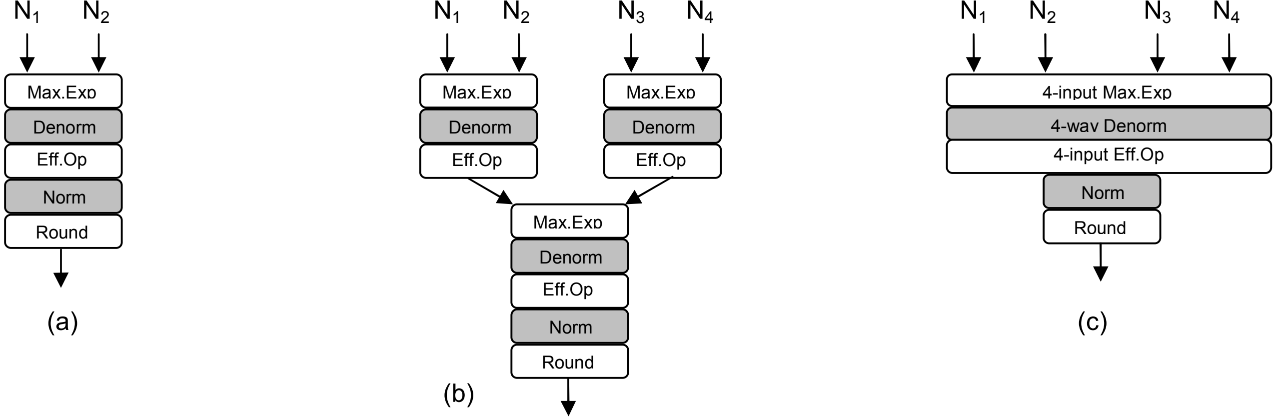

Figure 1. The five main stages of a traditional IEEE-754 floating point adder/subtractor (a); a four-input floating-point addition cluster as generated by Altera's floating-point datapath compiler (b) and our improved method (c).

If a tree of IEEE-754 operators is used to perform clustered addition, each normalization and rounding step at one level will be followed immediately by a denormalization at the next step; this normalization/denormalization is wholly redundant for effective addition when the expanded significand width is employed, because normalization is only used to fix overflow; extending the significand with additional overflow bits eliminates the need to normalize after every effective addition. In the general case, extending the significand by k bits permits effective addition of up to 2^k operands followed by one normalization operation while guaranteeing that overflow will not occur; underflow, in contrast, cannot be eliminated.

III. A New Approach to Floating-point Addition Cluster Synthesis on FPGAs

Figure 1(c) depicts a simplified version of our approach to floating-point clustered addition for four input operands. First, the maximum of the four exponents is found: second, the three significands corresponding to the non-maximal exponents are denormalized relative to the maximal exponent; third, the effective operation is applied to the four significands using carry-save addition (they are converted to two's complement and sign-extended to 32 bits before the effective operation, and back to sign-magnitude afterward); lastly, normalization and rounding are applied. Fig. 2 shows a detailed description of the multi-operand floating-point adder, including the sign extension and two's complement conversion/deconversion operations discussed in the preceding paragraph. The key features of this clustered adder are discussed in the following subsections.

A. Max.Exp

The most straightforward way to compute the maximum of a set of k numbers is to use a tree of 2-input *MAX* operators. If n is the bitwidth of the input operators, the delay complexity of the *MAX* operator is $O(\log n)$, assuming the use of a fast prefix adder to perform subtraction. A tree of 2-input *MAX* operators that computes the maximum of k operands has a height of $O(\log k)$. Therefore, the delay complexity of a tree of two-input *MAX* operators is $O(\log n \times \log k)$.

Our implementation of the *MAX* operator uses a different approach, which shares some principle similarities to leading one's counters. Our input is a set of k n-bit unsigned numbers (exponents are unsigned). Intuitively, the maximum unsigned integer has the longest string of consecutive '1's in the most significant positions; however, a situation may exist wherein all of the unsigned integers have at least one leading zero.

Step 1. The first step is to compute a *domination table*: if all bits in some position are '0' for all integers, we flip these bits to '1'. This guarantees that at least the maximum unsigned integer among the inputs begins with a string of consecutive '1's at the most significant position. If this occurs at bit position i, this effectively adds 2^i to each of the integers, which does not affect their relative ordering.

Step 2. Next, we count the number of leading ones in each bit position. For an n-bit unsigned integer, the number of leading ones itself is a $\lceil \log(n+1) \rceil$-bit unsigned integer. Thus, a list of k n-bit unsigned integers is reduced to a list of k $\lceil \log(n+1) \rceil$-bit unsigned integers. We then compute the maximum unsigned integer in the second list recursively. Termination is eventually guaranteed, since the bitwidth of the unsigned integers is reduced logarithmically at each stage.

Figure 3 shows a list of four 8-bit unsigned integers: *{22, 59, 62, 61}*. The maximum among them, *62*, is computed in three recursive steps; numbers in parenthesis are the base-10 representation of the input integers and number of leading ones in the domination table. In the example, the domination table differs from the integer only at the first stage.

Each bit of the domination table is computed independently in time $O(\log k)$, while the table area is $O(nk)$. The delay of an n-bit leading one detector is $O(\log n)$ and its area is $O(n)$. Since we count the leading '1's of k integers in parallel, the space complexity is $O(nk)$. To account for the subsequent stages of the recursion, we use the following recurrence relations, $T(n, k)$ and $A(n, k)$ for time and space complexity respectively:

$$T(n,k) = O(\log k + \log n) + T(\log n, k) \quad (4)$$

$$A(n,k) = O(nk) + T(\log n,\ k) \quad (5)$$

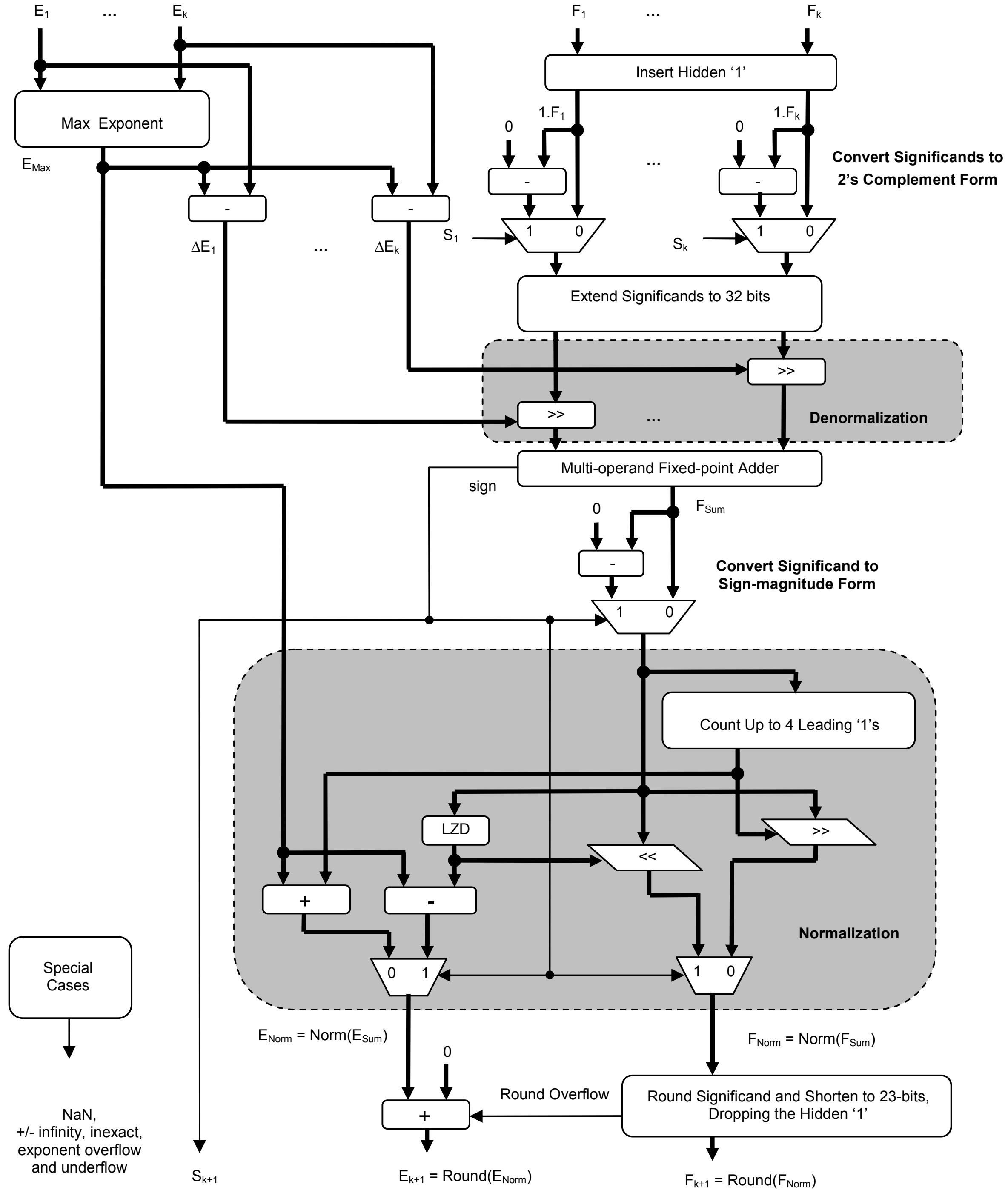

Figure 2. Illustration of our multi-operand floating-point addition cluster

The solutions to the two recurrence relations are as follows; we use the shorthand *log*n* to denote *log(log(… (logn)...))*:

$$T(n,k) = O(\log n + \log^* n \times \log k) \tag{6}$$

$$A(n,k) = O(nk) \tag{7}$$

This MAX circuit ourselves was discovered automatically by an internally-developed logic synthesis tool that uses decomposition methods to restructure arithmetic circuits [5]. In actuality, this circuit may have been discovered in the past and published elsewhere; if so, we remain unaware of its existence.

Integer	Dom.Table	Leading 1's
(22) 0010110	1010110	001 (1)
(59) 0111011	1111011	100 (4)
(62) 0111110	1111110	110 (6)
(61) 0111101	1111101	101 (5)
(1) 001	001	00 (0)
(4) 100	100	01 (1)
(6) 110	110	10 (2)
(5) 101	101	01 (1)
(0) 00	00	0
(1) 01	01	0
(2) 10	10	1 (MAX)
(1) 01	01	0

Figure 3. Illustration of a multi-operand MAX function implemented using a domination table.

B. Conversion to Sign-Magnitude

The output of the multi-operand adder is a two's complement integer; the expanded significand ensures that overflow does not occur if we add sixteen or fewer operands. The conversion back to sign-magnitude representation includes a subtraction operation, which is applied when the sign is negative. For a 16-operand implementation of our adder, the conversions to and from two's complement create a maximal error of at most 1 *unit in the last position (ulp)*. This error is bounded because the significand's precision has been extended by four extra underflow bits in the least significant positions.

The error occurs because right-shifting an integer and then negating it is not the same as negating an integer and then right-shifting it, i.e., $-(A >> k) \neq (-A) >> k$. For example, let $A = 10111$ and $k = 2$; here, the bits shifted in are equal to the sign bit that has been shifted right to preserve the positive/negative status of the shifter integer. In this case, $-(A >> 2) = -(11101) = (00010 + 1) = 00011$; in contrast, $(-A) >> 2 = -(10111) >> 2 = (01000 + 1) >> 2 = 01001 >> 2 = 00010$.

Subtraction produces a carry-out bit; when shifting before subtracting, some of the bits that may affect the value of the carry-out bit are shifted right, and are not considered when performing the subtraction. In other words, if L_k is the one's complement of the k least significant bits of A, that were shifted out in the former case, the following identity holds:

$$-(A >> k) = ((-A) >> k) + \overline{(carry_{out}(1 + L_k))} \qquad (8)$$

Since the value of $carry_{out}(1 + L_k)$ is either 0 or 1, there is at most *ulp* of error if we approximate $-(A >> k)$ as $(-A) >> k$. Since we are using four underflow bits, this error will propagate to the actual least significant bits only if we add 16 integers and all of them have this error. If, for example, we wanted to guarantee a similar error for 32-operand addition, we would need to add a fifth underflow bit to the significand.

C. Rounding

Altera does not specify which rounding modes are supported in their floating-point datapath compiler; nor do they specify a method to compute a sticky bit inside of an addition cluster. One possibility is that they may compute the sticky bit based on the right shift in the denormalization stage of the last 2-input addition in the cluster; however, this sticky bit would not necessarily be accurate, as the input significands are themselves denormalized. This latter possibility would be impossible to implement with our approach, as a compressor tree rather than a tree of adders, is used to perform significand addition. Since we use the same internal significand representation as Altera, we have implemented our own ad-hoc rounding mode, and we have used it in both our design and our implementation of Altera's clustered addition operations. This rounding mode uses the four underflow bits to mimic the IEEE round to +/– infinity, but it does not use a sticky bit. The drawback is that if all non-zero bits were right-shifted out during alignment, then the only way to account for their existence would be to use a sticky bit, which we do not.

IV. Experimental Results

In this section, we compare our proposed technique to synthesize single-precision clustered floating-point operands to the technique employed in Altera's floating-point datapath compiler [1, 2]; we also compare with a tree of IEEE-compliant two-input floating-point adders as well. We consider clusters of 2, 4, 8, and 16 operands.

We implemented the three clustered floating-point adder designs in all four cluster sizes in VHDL and synthesized them on an Altera Stratix III FPGA, performing synthesis with Altera's Quartus II software. We synthesized each design as a purely combinational circuit as to enable a fair comparison. Delay, in both cases, were reported in nanoseconds (ns); area is reported in *adaptive LUTs (ALUTs)*. It is important to note that ALUTs are an *Altera*-specific metric, and that area usage on a different vendor's FPGA may be different due to architectural differences, and variations in CAD flows targeting them.

Fig. 4(a) and (b) show the respective delay and area. Trees of IEEE 2-input floating-point adders were non-competitive compared to Altera's floating-point datapath compiler [1, 2] for more than two operands. Two implementations of our synthesis method are shown: one which builds the multi-operand adder as a tree of 3-input adders (with each ALM configured in *Shared Arithmetic Mode*), and one that implements it using a carry-save-based compressor tree [3]; the former achieves tigher area bounds, while the latter improves critical path delay.

For 4, 8, and 16-operand adder trees, the respective improvements in critical path delay compared to Altera's floating-path datapath compiler were 9%, 8%, and 6% for our approach using adder trees, and 16%, 17%, and 20% using compressor trees; similarly, the improvements in area were 21%, 29%, and 26% using adder trees, and 17%, 25%, and 20% using compressor trees.

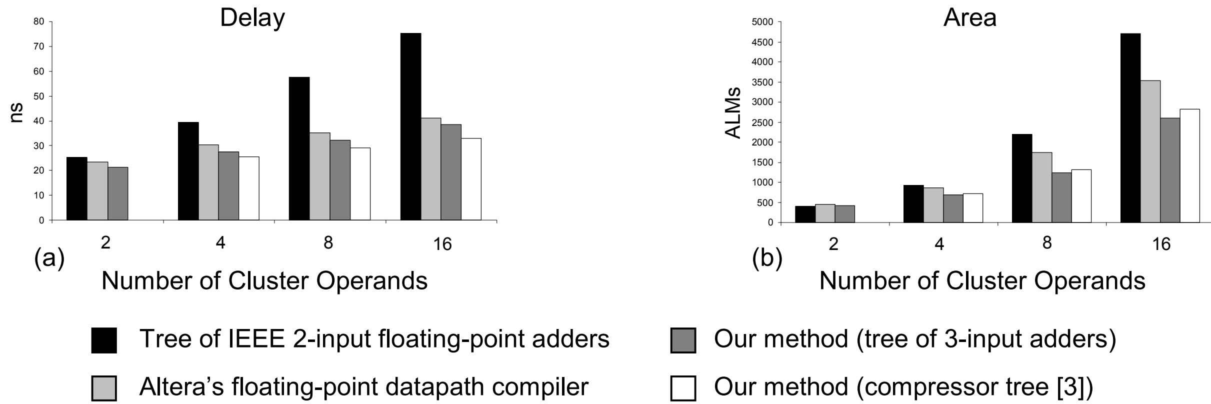

Figure 4. Experimental results that compare and contrast our method for clustered single-precision floating-point addition synthesis with Altera's method and a tree of IEEE 2-input floating-point adders for cluster sizes of 2, 4, 8, and 16 synthesized on an Altera Stratix III FPGA. For cluster sizes of 4, 8, and 16, our method has the lowest delay and consumes the smallest area in comparison to the others.

We consider the improvement in area to be more significant that the improvement in delay for two reasons: the delays and latencies (number of cycles) for our approach and Altera's are likely to be comparable after pipeline registers are inserted; moreover, the critical path of a complete design that includes a pipelined floating-point datapath may occur elsewhere in the design, such as the control path or the interface to memory. In contrast, the reduction in area is not subject to pipelining or the context in which the addition cluster is deployed.

V. RELATED WORK

Tenca [6] introduced a 3-input floating-point adder that achieves reduced error compared to two IEEE floating-point adders; however, this approach requires the precision of the effective operation to increase from f to $2f+5$; this approach is unlikely to scale for more than three or four operands. The FloPoCo compiler [7] can also generate effective 3-input operations when necessary, and can eliminate effective subtraction when all input operands are known to be positive, e.g., when computing $x^2 + y^2 + z^2$. Our approach, in contrast, takes the same approach as Altera's floating-point datapath compiler [1, 2], and scales up to 16 operands, while improving upon both area and delay.

It is impossible to parallelize floating-point addition while maintaining IEEE compliance due to non-associativity. Kapre and DeHon [8] developed a speculative approach to parallel floating-point accumulation that maintains compliance. Operands are added two at a time, in parallel, using IEEE-754-compliant operators under the optimistic assumption that all operators are associative for the given inputs. If a detection circuits asserts that this assumption is violated, the summation process is serialized automatically. The drawback of this approach is that each operator performs normalization.

VI. CONCLUSION

This paper has introduced a new method to synthesized fused floating-point addition clusters on FPGAs, which improves the area and critical path delay compared to the methods currently used in Altera's floating-point datapath compiler [1, 2]. It takes advantage of advances in FPGA synthesis for multi-operand additions using carry save arithmetic [3], and includes a new approach to computing the maximum value among a set of k unsigned integers.

REFERENCES

[1] M. Langhammer, "Floating-point Datapath Synthesis for FPGAs," *Proc. Intl. Conf. Field Programmable Logic and Applications (FPL '08)*, pp. 355-360, Sept. 2008.

[2] M. Langhammer and T. VanCourt, "FPGA Floating-point Datapath Compiler," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM '09)*, Apr. 2009.

[3] H. Parandeh-Afshar, P. Brisk, and P. Ienne, "Exploiting Fast Carry-Chains of FPGAs for Designing Compressor Trees," *Proc. Intl. Conf. Field Programmable Logic and Applications (FPL '09)*, Aug.-Sept. 2009.

[4] P.-M. Seidel and G. Even, "Delay-Optimized Implementation of IEEE Floating-Point Addition," *IEEE Trans. Computers*, vol. 53, no. 2, pp. 97-113, February, 2004.

[5] A. K. Verma, P. Brisk, and P. Ienne, "Iterative Layering: Optimizing Arithmetic Circuits by Structing the Information Flow," *Proc. Intl. Conf. Computer-Aided Design (ICCAD '09)*, Nov. 2009.

[6] A. Tenca, "Multi-operand Floating-point Addition," *Proc. 19th IEEE Intl. Symp. Computer Arithmetic (ARITH-19 '09)*, pp. 161-168, June, 2009.

[7] F. de Dinechin, C. Klein, and B. Pasca, "Generating High-Performance Custom Floating-Point Pipeline," *Proc. Intl. Conf. Field Programmable Logic and Applications (FPL '09)*, Aug.-Sept. 2009, and Technical Report LIP-2009-16, Ens-Lyon, Lyon, France, April, 2009.

[8] N. Kapre and A. DeHon, "Optimistic Parallelization of Floating-Point Accumulation." *Proc. IEEE Symp. Computer Arithmetic (ARITH-18 '07)*, pp. 205-216, June, 2007.