Exact Fused Dot Product Add Operators

Orégane Desrentes, Benoît Dupont de Dinechin, Florent de Dinechin

Exact Fused Dot Product Add Operators

Orégane Desrentes
Kalray S.A., Montbonnot, France &
Univ Lyon, INSA Lyon, Inria, CITI, France
odesrentes@kalray.eu

Benoît Dupont de Dinechin
Kalray S.A.
Montbonnot, France
bddinechin@kalray.eu

Florent de Dinechin
Univ Lyon, INSA Lyon, Inria, CITI
Villeurbanne, France
florent.de-dinechin@insa-lyon.fr

***Abstract*—This article explores architectures of exact (correctly rounded) fused dot product and add operators suitable for the FP32 and FP64 binary floating-point representations with subnormal support, and other representations with a wide dynamic range such as bfloat16. The exact summation of terms before rounding requires a full-size accumulator, and this work discusses techniques to compress the identical bits of this accumulator. This requires the computation of the relative shift amounts of the terms, which is formulated as a parallel prefix algorithm, allowing for a low-latency implementation. Architectural options for the exact fused dot product and add operators with up to 16 products for FP32, FP64 and mixed-precision BF16 to FP32 are evaluated using the TSMC 16FFC technology node.**

***Index Terms*—dot product, BF16, FP32, FP64, three-term sum**

I. Introduction

This work addresses the floating-point computation of

$$R = \circ(X_0 \times Y_0 + \ldots + X_{N-1} \times Y_{N-1} + Z)$$

with a single rounding $\circ$ of the exact dot product plus addend. Here $(X_i)_{i\in[0,N-1]}$, $(Y_i)_{i\in[0,N-1]}$, Z and R are floating-point numbers in a representation with a wide dynamic range such as IEEE 754 binary 64 (FP64), binary 32 (FP32), and bfloat16 (BF16). The multiplicands $X_{i\in\{0;N-1\}}$ and $Y_{i\in\{0;N-1\}}$ have e_{in} exponent bits and m_{in} significand bits. The output R and the addend Z have a possibly wider format with e_{out} and m_{out} bits. The corresponding operator is called FDPNA, possibly suffixed with the format, e.g. FDPNA$_{\text{FP32}}$ for homogeneous operators and FDPNA$_{\text{BF16}\to\text{FP32}}$ for mixed-precision ones.

A. Motivations

The main applications of FDPNA operators are accumulations of partial dot products in machine learning and linear algebra applications. In addition, the operator where $N = 2$ (called ExSdotp in [1]) effectively supports complex floating-point arithmetic, as a complex fused multiply and add $R = XY + Z$ is implemented with the best possible accuracy in only two operations: Noting $j^2 = -1$,

$$R = R_{\text{re}} + jR_{\text{im}} \text{ with } \begin{cases} R_{\text{re}} = \circ(X_{\text{re}}Y_{\text{re}} - X_{\text{im}}Y_{\text{im}} + Z_{\text{re}}) \\ R_{\text{im}} = \circ(X_{\text{re}}Y_{\text{im}} + X_{\text{im}}Y_{\text{re}} + Z_{\text{im}}) \end{cases}.$$

An illustration of practical importance is the on-line computations of the twiddle factors of Fast Fourier Transforms (FFT) [2]. The twiddle factors of a N-point DFT are defined as $W_P^k = e^{\frac{-2j\pi k}{P}}$, with P some power of 2 smaller than N and $k \in [0, P-1]$. They can be computed offline with the results stored in a table, or online using the recurrence $e^{j(k+1)\theta} = e^{jk\theta} \times e^{j\theta}$ with $\theta = -\frac{2\pi}{P}$. Such a sequence of n multiplications by $e^{j\theta}$ may lead to $O(n)$ error [3], so a twiddle factor recurrence should be implemented carefully. Rewriting $e^{j(k+1)\theta} = e^{jk\theta} + e^{jk\theta}(e^{j\theta} - 1)$ with $e^{j\theta} - 1 = -2\sin^2\frac{\theta}{2} + j\sin\theta$ avoids the cancellation in $cos\theta - 1$ [4] since θ is small for large-N FFTs. Computing $e^{j(k+1)\theta}$ then becomes a complex multiply-add that is accurately implemented by the FDP2A operators.

Fig. 1 displays the maximum and average errors for the FMA-based and the FDP2A-based twiddle factor recurrences in $\log_{10}$ scale, for N spanning successive powers of two. The reference values are twiddle factors computed using the `libm` standard `cos` and `sin` functions, rounded to FP32 from FP64. The error is defined as the modulus of the difference between a value and the baseline using FP64 arithmetic. Since twiddle factors are roots of unity, these errors are both absolute and relative. The FMA-based recurrence errors grow up to two orders of magnitude larger than those of the FDP2A-based recurrence errors.

B. Related Work and Previous Implementations

Fused sums or sums of products have been studied before [5], [6], [7], [8], [9], [10], [11], [1]. However, only [6], [8], [1] are exact. All the others either truncate the term summation or compress the smaller magnitude terms into sticky bits, which leads to inexact results in some cases of cancellation.

1) Complex Multiply-Add Implementations: The Arm SVE2 Floating-point Complex Multiply Accumulate (FCMLA) instruction [11] multiplies a complex number by the real or imaginary part of the second multiplicand with optional negation, then adds a complex addend. A full complex multiply-add requires only two FCMLA instructions, however each FCMLA instruction is implemented as two parallel FMA operations.

2) Inexact Fused Add3 Operators: The fused sum of three floating-point numbers has been studied in [5], [7]. Here the

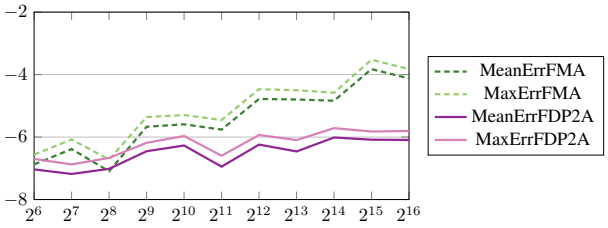

Fig. 1. Base-10 logarithm of errors for the FMA-based and the FDP2A-based radix-2 twiddle factor FP32 recurrences depending on the FFT size.

term significands are aligned in a way similar to the sum of two numbers, and if not overlapping the smaller term is shifted out into a sticky bit. In case of a full cancellation of the two larger terms, the result should be the third (smaller) term, but it cannot be recovered, so this approach is not exact.

3) Inexact Fused Dot Product Add Operators: Likewise, the fused floating-point four-term dot product unit of [12] is inexact as significands of low magnitude are compressed into a sticky bit.

The Intel Nervana Neural Network Processor-T [10] implements an inexact 32-product BF16 dot product FP32 add operator. The products are aligned to the maximum product exponent then truncated to an internal 37-bit datapath before being summed. A similar operator with up to 32 products is presented in [13], focusing on optimizing the maximum exponent and global alignment computation which becomes a bottleneck for large numbers of products. Reverse-engineering of the NVIDIA V100 GPU tensor cores [9] reveals they also adopted a similar approach for dot product add operators.

4) Exact Fused Dot Product Add Operators: The ExSdotp fused dot product add operator of [1] first sorts the three terms based on their exponents (the exponent of each product being the sums of the multiplicand exponents). The two larger terms are added, and if a full cancellation is detected, the smaller term is restored for the result. This works for the sum of three floating-point numbers. However, with products involving subnormal multiplicands, this approach may result in incorrect results with directed rounding. For example, consider ExSdotp FP16→FP32 with rounding up; $X_0 = Y_0 = X_1 = 1$; Y_1 a small positive FP16 subnormal; Z a FP32 such that $Z = -2X_1Y_1$. Obviously $X_0Y_0 + X_1Y_1 + Z = 1 - X_1Y_1 < 1$, therefore, R should be 1. However, the exponent of X_1Y_1 is larger than that of Z due to Y_1 being subnormal. Therefore the sort, based on the exponents only, will use X_1Y_1 for the addition instead of Z, and the result returned will be $R = 1 + 2^{-23}$. This issue also also affects the inexact flag in round to nearest.

Kulisch [14] advocated the use of a fixed-point accumulator large enough to hold any exact product of floating-point terms. Many-term exact fused dot product add operators using a Kulisch-like full-size accumulator have been proposed in [8], [15]. This approach will be studied quantitatively in this paper. An alternative is to compress identical bits inside the full size accumulator. Here we build on Tao et al. [6], improving their work with subnormal support, managing the addend Z, and mixed-precision. We also explore alternative techniques for several sub-problems, including sorting networks and a parallel-prefix computation of the significand shifts.

C. Outline

This article is organized as follows. Section II exposes the operating principles of the proposed exact fused dot product add operators. Section III focuses on the key insight, which is how to suitably align the sorted term significands before their exact summation. Section IV discusses implementation and validation details. Section V reports on the synthesis results.

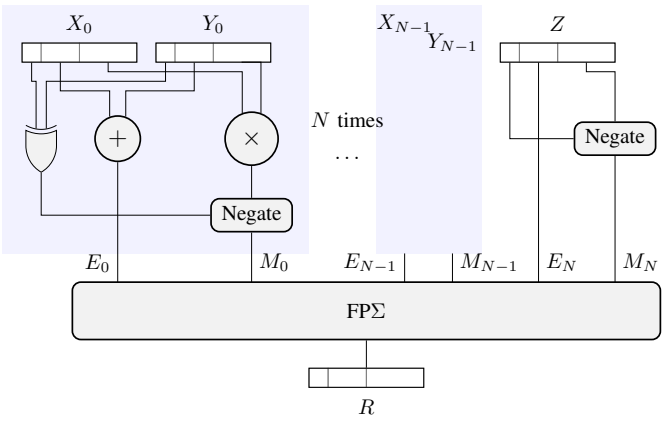

Fig. 2. High-level architecture of the FDPNA operator.

Section VI presents some conclusions with respect to the relevance domain of the techniques studied.

II. Operating Principles

A. Architecture Overview

Fig. 2 describes the high-level architecture of our dot product add operators. Prior to each multiplication, the implicit bit is added to the fraction of each input, with a value 0 for a subnormal and 1 otherwise. The significands are multiplied together and negated if needed, while the exponents are added. Each product X_iY_i is now represented with E_i its exponent and M_i its signed, non-normalised significand. The addend Z is also split into its exponent E_N and signed significand M_N.

Then, a component denoted FPΣ aligns the significands and sums the $N+1$ floating-point terms. Following the summation, a Leading Zero Count (LZC) retrieves the exponent of the result. Finally, the sum is normalized or subnormalized, rounded, and output as a floating-point number. Overflow, underflow, NaN and IEEE 754 flags are managed within FPΣ as well.

B. Full-Size FPΣ Implementation

The FPΣ component can be implemented by first converting all terms into a common two's complement fixed-point representation [14], [8]. To ensure that this conversion is exact, the fixed-point representation is defined by the worst-case Most Significant Bit (MSB) and Least Significant Bit (LSB) of a shifted product significand. These values can be deduced from the parameters e_{in}, m_{in}, e_{out} and m_{out} of the in/out floating-point formats. For example, in the homogeneous case, $\text{LSB} = -2 \times (2^{e_{\text{in}}-1} - 2 + m_{\text{in}})$ and $\text{MSB} = 2 \times (2^{e_{\text{in}}-1} - 1)$. This defines the full-size in bits w_{full} of the common fixed-point representation. Values for the floating-points formats of interest are summarized in Table I.

TABLE I
FIXED-POINT PRODUCTS IN THE FULL-SIZE FPΣ OPERATOR.

Format	Product			
	mult. size	LSB	MSB	Full-Size w_{full} (bits)
FP16	11×11	-48	30	80
BF16	8×8	-266	254	522
FP32	24×24	-298	254	554
FP64	53×53	-2148	2046	4196

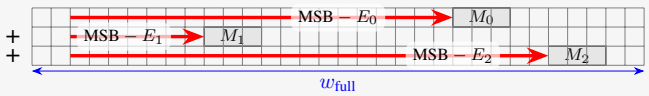

Fig. 3. Example of alignment for the sum using a full-size fixed-point format.

This conversion to a common fixed-point format (or alignment) requires shifting left M_i by E_i bits, or alternatively shifting right M_i by MSB $- E_i$ bits, in both cases with sign extension. Fig. 3 uses right-shifting for consistency with later figures. Once converted to fixed-point, the terms can be summed using integer addition into a full-size accumulator (Fig. 3), extended with $\lceil \log_2(N+1) \rceil$ extra bits on the left to guard against possible overflows. Multiplications, conversions, and additions are all exact. The result may then be normalized and rounded (only once) to the output format.

This approach is quite efficient for formats with small full-size accumulators, so has been used in particular for the exact sequential accumulation of FP16 products [8] and a FDP8A$_{\text{FP16}\rightarrow\text{FP32}}$ operator [15]. In the mixed-precision case, it is better to perform the sum of products separately in an accumulator corresponding to the small format, then perform the last addition of Z as in a classical FMA. As the sum of products is exact, it is associative and can be parallelized without any consideration of the exponent values.

C. Compressed FPΣ Principles

When (as is the case in Fig. 3) the accumulator size is much larger than the sum of the widths of the $N+1$ significand terms to add, one observes that most of the summation adds zeros (or sign bits, which are similarly easy to manage). Intuitively, these parts of the summation can be saved, and the worst-case size of actual addition needed should be roughly N times the size of a term significand. What we call compression in this article is the suppression of the columns of predictably identical bits in Fig. 3. It is called realignment in [6].

The core idea is to use a single fixed-point multi-operand adder of size $w_{\text{compressed}} < w_{\text{full}}$. The shift values needed to align the M_i before their summation are no longer trivially deduced from the E_i as in the full-size case: they now need a more complex computation to skip the compressed bits. The main issue is the combinatorial explosion of alignment situations because it has to be implemented in hardware.

To address this explosion, we first sort the terms by their exponent E_i. Section IV-A discusses our approach for this. Let us note $(E_0^*, M_0^*), (E_1^*, M_1^*), ...(E_N^*, M_N^*)$ the renumbered (exponent, significand) pairs such that $E_0^* \geq E_1^* \geq \ldots \geq E_N^*$. Our sorting approach requires that the products X_iY_i and the addend Z be represented in the same way. The significand size w that fits all is the maximum width of all M_i:

$$w = \max(2 + m_{\text{out}}, 2(1 + m_{\text{in}}))$$

As illustrated in Fig. 4, the significand M_N of the addend Z is extended with an extra MSB corresponding to the overflow bit of a product. In the homogeneous case, M_N is extended with extra LSBs. In the mixed-precision case M_N usually also has more precision to the right than the other M_i (e.g. 22 bits for BF16 products versus 24 bits for FP32). In the sequel, all M_i are considered to have the same size w. The exponent E_N of Z must be similarly updated, we do not detail it further.

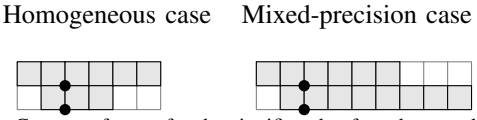

Fig. 4. Common format for the significands of products and addend.

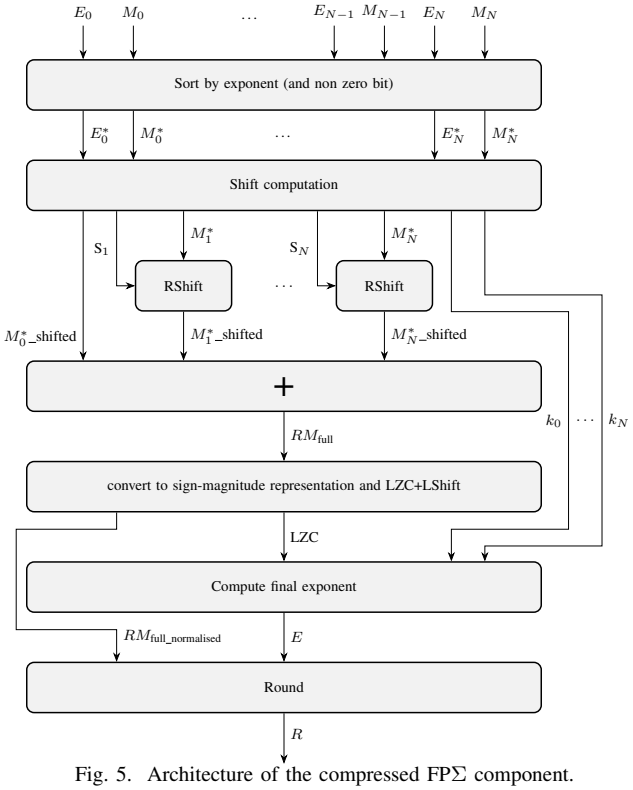

Fig. 5. Architecture of the compressed FPΣ component.

Accordingly, the architecture we propose is sketched in Fig. 5. The (exponent, significand) pairs are first sorted by their exponent, then processed by a component that determines the adjusted shift values S_i. The sorted significands are then shifted before being summed into an adder tree. The sum RM_{full} is converted back to a (sign,magnitude) representation, normalized, then rounded to produce the final significand.

Compared to the full-size FPΣ, this architecture involves one fewer shift, and if $w_{\text{compressed}} < w_{\text{full}}$ the shifts are smaller, as are the adder tree and the LZC. However, it also requires a sorting component, more exponent pre-processing, and some exponent post-processing as well. This trade-off is evaluated quantitatively in Section V.

D. Introductory Considerations

What is the smallest number of bits $w_{\text{compressed}}$ such that all the $N+1$ terms can be added using integer adders of size at most $w_{\text{compressed}}$ and the exact sum can be recovered for rounding as if it were computed on w_{full} bits? To address this question while introducing notions needed in the sequel, first consider the trivial case $N = 1$ then the simple case $N = 2$.

1) Two Terms ($N = 1$): Fig. 6 shows a fixed-point addition of two terms. There are two cases: (case 1) the two significands

are far apart, and the bits between them can be omitted, as the result will be M_0^* or one of its immediate FP neighbours (one round bit must be kept to the right of M_0^*); or (case 2) M_1^* overlaps M_0^*, and there will be an addition of size at most $2w$ bits. In both cases the w_{full} bits of the exact sum can be compressed in $2w+2$ bits. This observation is exploited in classic FP adders, but with an additional trick: the bits in the blue zone of Fig. 6 can be compressed into a sticky bit and a guard bit [16] in a way that keeps enough information for a correctly rounded addition.

2) Three Terms ($N = 2$): There is now the possibility of a complete cancellation of the two leading terms M_0^* and M_1^*. In such a case, the exact result is M_2^*, therefore we must keep all its bits, so it is incorrect to compress them into a sticky bit. This will be the case for all $N > 1$, so we will no longer mention sticky bits. Fig. 7 shows the four alignment cases of fixed-point addition for three terms ($N = 2$).

In case 1, the three significands are fully separated and all the bits between them can be compressed. We have three colored zones in this figure (from left to right : orange, blue and magenta), and the architecture will need to remember the exponent of each significand and associate it with the corresponding zone to emulate the full-size accumulator. In all the sequel, zone i is associated with the exponent E_i^* and a priori contains the significand M_i^*.

In case 2, the two lower significands M_1^* and M_2^* overlap, which means that their sum may overflow by one bit to the left of M_1^*. The blue zone, in the compressed view, reserves for this overflow one more bit than in Fig. 6 to the left of M_1^*. Besides, there is no longer a magenta zone like in case 1: significand M_2^* belongs to the blue zone: since it must be added to M_1^*, it inherits its exponent E_1^*. The good news is that in the compressed adder tree, the bits necessary to this addition may recycle those of the magenta zone in case 1.

Case 3 is similar, but with M_1^* now in the orange zone due to M_1^* overlapping M_0^*. Note that the magenta zone, associated with the exponent E_2, exists in this case. In such a case E_1^* will be used to shift M_1^* to its proper place in the compressed sum, but it is not associated with a zone.

Finally, case 4 shows the situation where we only have one zone associated with E_0^*, which happens as soon as the compressed accumulator can hold the exact sum. Here neither E_1^* nor E_2^* are associated with a zone.

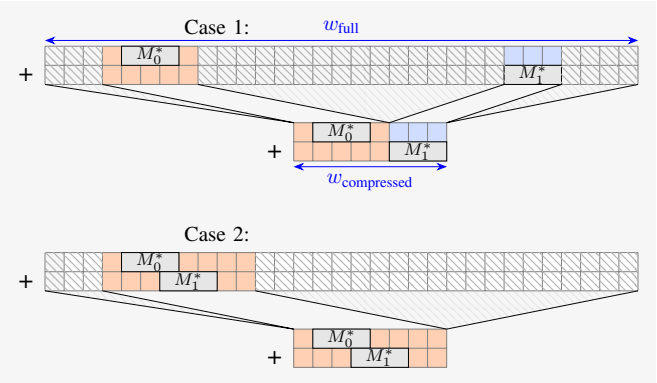

Fig. 6. Compressing the exact addition of two terms.

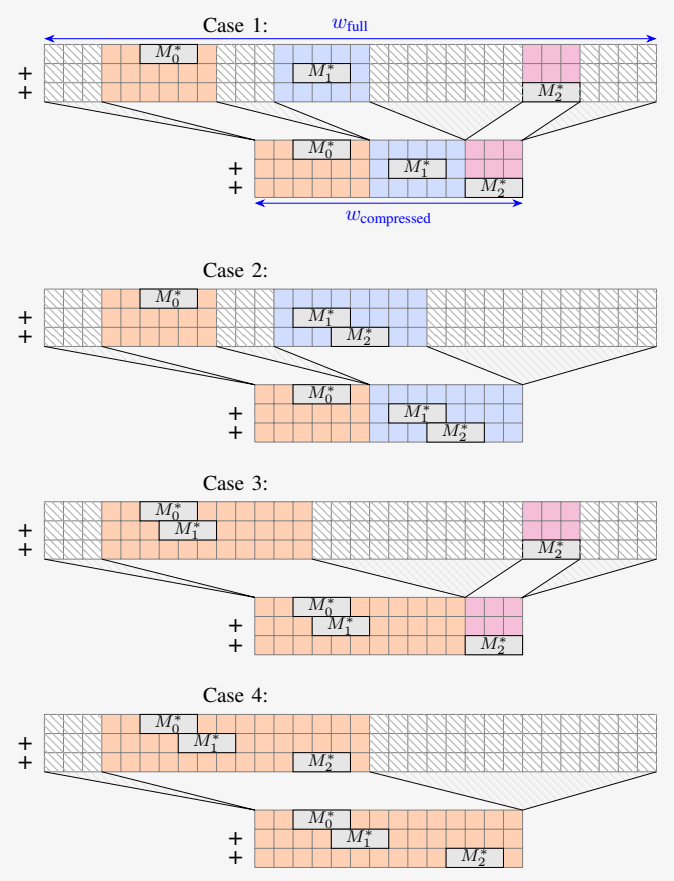

Fig. 7. Compressing the exact addition of three terms.

III. Construction of the Compressed FPΣ

In this section, we first generalize the $N = 1$ and $N = 2$ considerations to formally define the sizes and parameters of the $N+1$ zones of a FDPNA operator. We then define an architecture to shift all the significands to their proper place in the compressed accumulator. Determining those shift values can be implemented as a parallel prefix computation.

A. Compressed FPΣ Parameters for N Terms

In case of N terms, the key is to determine the zones in the summation where each significand M_i^* should be positioned if it does not overlap with other significands. These zones are specified by their boundaries, denoted d_i. Fig. 8 illustrates the d_i values for $w = 3$, in cases $N = 2$ and $N = 4$.

The number of bits between d_i and d_{i+1} should at least be the width w of the significand. An extra bit is added to the right of the significand as a placeholder for a round bit. Besides, p_i protection bits are added to the left of the significand (see Fig. 8) to absorb the worst-case overflow of the significands of lower or same magnitude:

$$\begin{aligned} p_i &= \lceil \log_2(N-i) \rceil \\ d_0 &= 0 \\ d_i &= d_{i-1} + w + 1 + p_{i-1} \quad . \end{aligned}$$

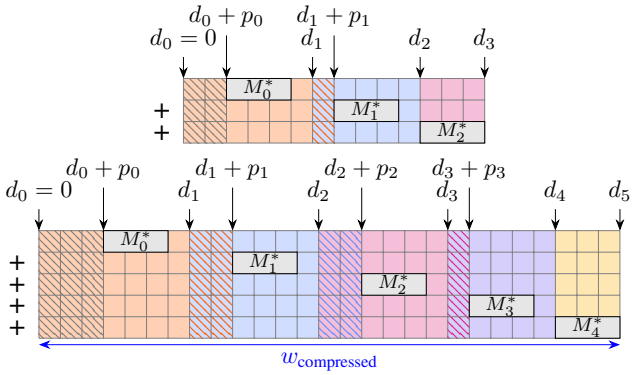

Fig. 8. Position of the zone delimiters in cases $N = 2$ and $N = 4$.

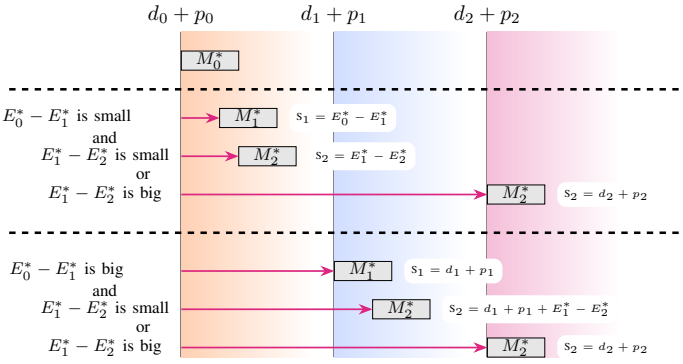

Fig. 9. Determining the shift values.

This recurrence also defines the total size of the compressed accumulator $w_{\text{compressed}} = d_{N+1} - d_0$. The p_i and d_i parameters only depend on the value of N that is set at design time.

B. Definition of the Shift Values S_i

M_0^* is placed in a constant position to the left of the addition: $S_0 = d_0 + p_0$. Determining the subsequent S_i involves another case analysis, illustrated in Fig. 9 for $N = 2$:

- If E_1^* is close to E_0^* then $S_1 = d_0 + p_0 + E_0^* - E_1^*$, and
  - If E_2^* is close to E_0^* then M_2^* is placed in the zone of M_0^* and $S_2 = d_0 + p_0 + E_0^* - E_2^*$
  - Else M_2^* is placed in its own zone: $S_2 = d_2 + p_2$
- Else M_1^* is placed in its own zone: $S_1 = d_1 + p_1$, and
  - If E_2^* is close to E_1^* then $S_2 = d_1 + p_1 + E_1^* - E_2^*$
  - Else M_2 is placed in its own zone: $S_2 = d_2 + p_2$.

Specifically, E_i^* "close to" E_j^* means that M_i^* can be placed in the same zone as M_j^*. Then it aligns with E_j^*, and the bits of zone i are recycled to make space for the addition in zone j. This leads to the following definitions for the shifts:

$$
\begin{aligned}
S_0 &= d_0 + p_0 \\
S_1 &= \min(d_0 + p_0 + E_0^* - E_1^*,\ d_1 + p_1) \\
&\quad \text{if } S_1 = d_1 + p_1 \\
&\qquad \text{then } S_2 = \min(d_1 + p_1 + E_1^* - E_2^*, d_2 + p_2) \\
&\qquad \text{else } S_2 = \min(d_0 + p_0 + E_0^* - E_2^*, d_2 + p_2)
\end{aligned}
$$

C. Parallel Prefix Computation of S_i

The recurrence defining S_i may be evaluated faster by reformulating it into a form suitable for parallel prefix computation.

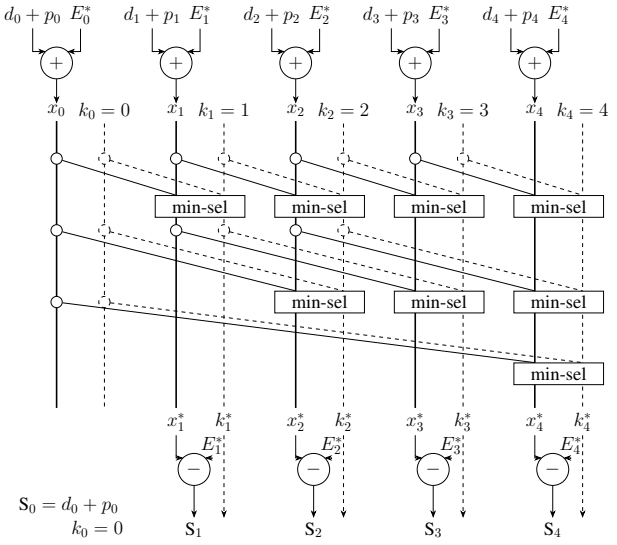

Fig. 10. Example of parallel prefix computation for $N = 4$.

First, each level contains dependencies on all the positions of the previous levels, but this is not necessary for a correct result. Indeed, if E_2^* is close to E_1^* and E_1^* is close to E_0^* then:

$$
\begin{aligned}
& d_0 + p_0 + E_0^* - E_1^* \le d_1 + p_1 \\
\Rightarrow\ & d_0 + p_0 + E_0^* - E_1^* + E_1^* - E_2^* \le d_1 + p_1 + E_1^* - E_2^* \\
\Rightarrow\ & d_0 + p_0 + E_0^* - E_2^* \le d_1 + p_1 + E_1^* - E_2^*
\end{aligned}
$$

The formula for S_2 can be rewritten as:
$S_2 = \min(d_0 + p_0 + E_0^* - E_2^*,\ d_1 + p_1 + E_1^* - E_2^*,\ d_2 + p_2)$.

Second, observe that the last argument of the min can be rewritten as $d_2 + p_2 = d_2 + p_2 + E_2^* - E_2^*$. This leads to:
$S_2 = \min(d_0 + p_0 + E_0^*,\ d_1 + p_1 + E_1^*,\ d_2 + p_2 + E_2^*) - E_2^*$.

Note that all the $d_i + p_i + E_i^*$ can be computed in parallel as soon as the order of the exponents is known. This is also a convenient form for hardware implementation using a Hillis & Steele parallel prefix tree [17] as shown in Fig. 10.

The general formulas for the shifts (pre-processing) are:

$$
\begin{aligned}
S_0 &= d_0 + p_0 \\
S_i &= \min_{j \in \{0,\ldots,i\}} \{d_j + p_j + E_j^*\} - E_i^* \\
k_i &= i \text{ if } (S_i = d_i + p_i) \text{ else } k_{i-1}
\end{aligned}
$$

The index k_i is computed along the min and is the index of the zone to which M_i^* belongs.

D. Computation of Final Exponent E

Once the summation is done, its result is converted back into a sign-magnitude representation and then normalized. Normalization starts with a leading zero counter (LZC) which determines the number L of zeros before the leading bit in the (compressed) sum. This L is compared to all the d_j to determine the index i of the zone of the result. Then the index k_i is used to retrieve the exponent $E_{k_i}^*$ actually associated with this zone. The result exponent verifies $L - (d_{k_i} + p_{k_i}) = E_{k_i}^* + 1 - E$. Here the +1 captures the fact that our M_i^* have two bits, not one, to the left of the binary point (see Fig. 4).

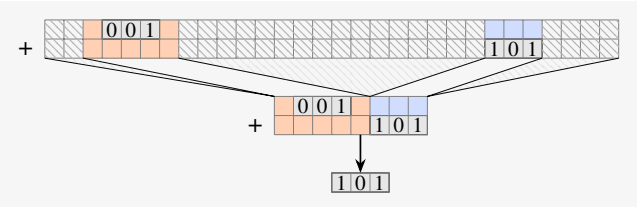

Fig. 11. Result is using bits from 2 different zones, giving an incorrect result.

Finally the normalized result exponent E is computed as:

$$\begin{aligned} \text{If } L = d_n &: \text{result is } R = 0 \\ \text{else if } L < d_1 &: E = E_0^* + 1 - (L - (d_0 + p_0)) \\ \text{else if } L \in [d_i, d_{i+1}[&: E = E_{k_i}^* + 1 - (L - (d_{k_i} + p_{k_i})) \end{aligned}$$

E. Subnormal Management

Fig. 11 illustrates a problematic situation to avoid: due to a cancellation, the normalized result contains bits that belong to two different zones. In the full sum there would be more bits between them, so the resulting significand is incorrect.

This situation cannot happen because of partial cancellations with *normal* inputs, as in this case the zone in which the cancellation occurs has been enlarged by fusing in the bits from zones to the right. See Case 3 of Fig. 7: there are more than w extra bits in the orange zone due to fusing in the blue zone. A cancellation is either full (then the LZC will skip the orange zone), or it cancels at most $w-1$ bits and there remains more than a significand worth in the orange zone to the right of the leading one. This is also true if three or more significands share the same zone, as the zone is correspondingly larger.

However, the problematic situation may occur if there is one subnormal input to a product. Managing this properly requires specific considerations on e_{in}, e_{out}, m_{in}, and m_{out}.

In the following paragraphs, M_0^* will be used to discuss the problematic situations, since they arise when the leading 1 after sum is positioned near the next zone. In general, those situations may happen in any zone, in case of a complete cancellation in the leading zones.

1) Homogeneous Case ($m_{\text{in}} = m_{\text{out}}$ and $e_{\text{in}} = e_{\text{out}}$): In this case remember that the width of the M_i^* is $w = 2 + 2m_{\text{in}}$. If M_0^*, the largest-magnitude term, is a product between a large normal and a subnormal, it cannot have more than m_{in} leading zeros and the width of the zone is greater than $w = 2 + 2m_{\text{in}}$. So the problematic situation in Fig. 11 cannot happen.

If M_0^* is a product of two subnormals, then its exponent is the smallest possible, which means that all terms are products of two subnormals, all the following zones will be merged, and there are no problems either.

2) Mixed-Precision when $e_{\text{in}} \leq e_{\text{out}}$: In the worst case, the product of the two smallest non-zero subnormals has only one bit left, and nevertheless may be larger than Z. This leads to the problematic situation in Fig. 11.

3) FDP$NA_{BF16\to FP32}$ Special Case: The BF16 format does not support subnormals [18]. The operator FDP$NA_{\text{BF16.32}}$ can still support subnormals in its FP32 input Z and output R. A subnormal Z input is not a problem. Even if there is a product of BF16 numbers smaller than Z, the case in Fig. 11 cannot happen as the result will also be subnormal. Subnormalization of the output is managed in all cases by the normalization and rounding of the compressed exact sum.

Since BF16 and FP32 share the same exponent size, it is possible to support BF16 subnormals by taking $w = 32$ instead of $w = \max(2+23, 2(1+7)) = 25$. This modification ensures that the significand of the product of a normal by a subnormal cannot extend to the next zone: this product has a maximum of 8 leading zeros, therefore to ensure 24 significant bits in the zone, we need $w \geq 24 + 8 = 32$.

IV. Implementation and Validation

A. Exponent Sorting Network

To ensure that subnormals sort bigger than 0, a bit is first appended to the LSB of each E_i, equal to 0 iff the significand is 0. These modified exponents are the keys in the (key, payload) pairs processed by the 'Sort by exponent' component in Fig. 5. Several variants of this component were explored and implemented in a fork of FloPoCo [19].

First, we have two options for the payloads: either directly use the significand, or use only their index (which fits on much fewer bits). The second option makes the sorting itself cheaper, but requires an expensive multiplexing step in order to recover the significands from their indices. A detailed evaluation (omitted here for brevity) shows that sorting by indices ends up being almost twice as expensive in terms of area. In isolation, it would also be slower due to the extra multiplexing to recover the significands. However, sorting by indices reduces the overall latency since the sorting can be performed while the significand products are being computed. It is thus the option used in the sequel.

For the sorting algorithm itself, we considered two alternatives. The first is to use textbook sorting networks [20]. For $N = 4$ (5 terms to sort) a bitonic sort has a depth of 5 which is optimal [21]. For 9 terms, the network from [22] has a depth of 7 which is proved optimal in [23]. For 17 terms the sort used is from [24], whose depth of 10 is optimal.

The second alternative is from by Tao et al. [6]. To sort $n = N + 1$ terms, it first performs $n(n-1)/2$ parallel key comparisons. The resulting comparison bits or their complement are input to n counters ("population count") that compute in parallel the rank of each key. Finally a $n \times n$ crossbar completes the sort. This technique obviously has a lower latency than a sorting network. In our experiments, even its area is competitive with sorting networks at least for the values of N considered here, all the more as only the final crossbar moves payloads. This sorting technique is therefore used in the sequel.

The sequential shift computation of [6] is able to recycle the values $E_i^* - E_j^*, \forall i \geq j$ computed in the sort for their sign bit. This saves area in a non-trivial way (it increases the number of multiplexers used at the end of the sort, but reduces the number of adders and saves their latency). The parallel prefix computation of shift values does not allow to use the same recycling trick.

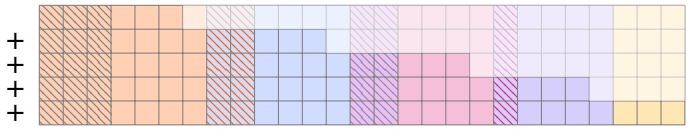

Fig. 12. Some bits can be omitted from RShift and the summation bit heap.

B. Shifter and Bit Heap Sizes

Due to the structure of the summation in the compressed FPΣ component, the 'RShift' and '+' components of Fig. 5 can be simplified. As the significand M_i^* can only be positioned in $\{Zone_0, \ldots, Zone_i\}$, the actual size of the shifters is $\text{max\_shift}_i = d_i + p_i$ (Fig. 12).

Similarly, this structure reduces the actual number of bits to be added. The summation is performed using a compressor tree that input the bit array (or bit heap) represented in Fig. 12. Note that a full-size architecture always requires a rectangular bit array.

C. Operator Validation

The general FDPNA operators have been validated with the test bench generator of FloPoCo, which compares outputs to the exact results computed using GNU MPFR. The test bench consists of random tests and some directed tests as well as exhaustive corner-case checks. Special cases and flag behavior are extrapolated from the IEEE standard for fused operations. The directed tests include:

- Testing cancellations by forcing all the inputs to have the same exponent after the product: $E_{X_0} + E_{Y_0} = E_{X_1} + E_{Y_1} = \ldots = E_{X_{N-1}} + E_{Y_{N-1}} = E_Z$, where some products are randomly negative.
- Forcing inputs to have pairs of equal exponents after product: $E_{X_0} + E_{Y_0} = E_{X_1} + E_{Y_1}, \ldots, E_{X_{N-2}} + E_{Y_{N-2}} = E_{X_{N-1}} + E_{Y_{N-1}}$, where one product is negative and another is positive.
- Forcing M_0^* to be a product between normal and subnormal: $E_{X_0} = 0$ and $E_{Y_0} \geq E_{X_1} + E_{Y_1}, \ldots, E_{Y_0} \geq E_{X_{N-1}} + E_{Y_{N-1}}, E_{Y_0} \geq E_Z$.

The FDP2A$_{\text{FP32}}$ and FDP2A$_{\text{FP64}}$ operators have also been validated with a directed framework which tests against a golden model implemented with Sollya [25]. This framework thoroughly explores catastrophic cancellation cases with different significands that multiply to the same number, between Z and a product, different multi-sticky issues, and the use of subnormals. This adds an extra half a million tests.

V. Experimental Results

In this section, we compare three variants of the FDPNA operators: with a Kulisch-like accumulator (denoted *Full*), our approach as per Section III (denoted *Ours*) and a reimplementation of Tao et al. [6] adapted to handle subnormals (denoted *Tao*). The main difference between *Tao* and *Ours* is that the former uses a sequential algorithm to compute the mantissa shift values, whereas *Ours* uses a parallel prefix computation.

All these FDPNA operators have been synthesized with the Synopsys Design Compiler NXT for the TSMC 16FFC node.

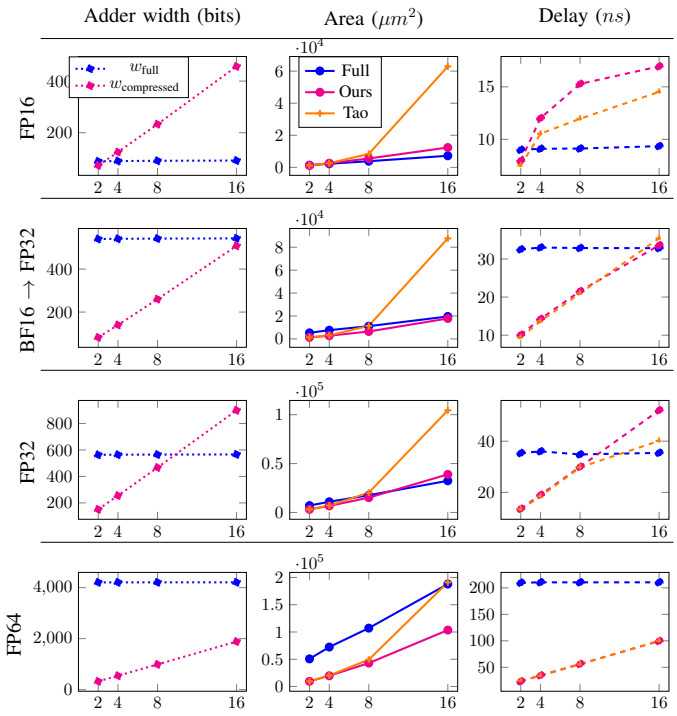

Fig. 13. Combinatorial synthesis results as a function of N.

A. Synthesis Without Pipelining

We first set a target frequency of 1 MHz in order to explore the synthesis trade-off between areas and combinatorial delays. The results appear in Table II and are plotted in Fig. 13, along with the size of the internal additions. There is always a threshold in N above which there is no compression ($w_{\text{compressed}} > w_{\text{full}}$). This threshold is 2 in the homogeneous FP16 case, about 10 in the homogeneous FP32 case, and above 16 for the other formats studied.

In full-size operators, the size of the internal adder does not depend on N. The rest of these operators essentially grows linearly with N. In the compressed operators, the adder sizes increase linearly with N, although extra pre/post-processing may be compensated by the smaller shifters (see Fig. 12).

The parallel prefix computation of shift values is not always beneficial. In particular it is not justified for $N = 2$.

B. Synthesis With Pseudo-Pipelining

We then evaluate the best architecture for each case in the context of a pipelined floating-point unit (FPU). For this purpose, we approximate the synthesis constraints of n-stage operator pipelining by using *pseudo-pipelining*, that is, single-cycle synthesis at $\frac{1}{n}$ the target FPU frequency. The approximation is that the reported results do not account for the cost of the pipeline registers. Results are reported in Table III. The number of pseudo-pipelining cycles n in Table III is selected for each configuration as the area/latency trade-off most relevant for FPU design. Wherever two solutions were close in Table II, both were re-synthesized for the same n, and the best is reported.

TABLE II
SYNTHESIS RESULTS FOR TSMC 16nm (1 MHz), NO PIPELINING. A IS THE AREA IN μm^2, D IS TIMING IN ns.

		FP16				BF16 → FP32				FP32				FP64			
N		2	4	8	16	2	4	8	16	2	4	8	16	2	4	8	16
Full	A	1370	**2247**	**3819**	**7209**	5305	7608	11117	19572	7036	10938	17602	**32411**	50724	72350	107090	187916
	D	9.0	**9.1**	**9.12**	**9.35**	32.51	32.99	32.86	**32.83**	35.34	36.02	34.83	**35.44**	209.72	210.44	210.44	210.49
Ours	A	**1193**	2599	5520	12360	**1298**	**2723**	**6503**	**17972**	**3136**	**6659**	**15128**	39511	**9500**	**19713**	**43078**	**104494**
	D	7.97	12.02	15.28	16.95	10.11	14.26	21.54	33.59	13.57	19.03	29.97	52.05	**23.47**	34.69	56.16	**99.58**
Tao	A	1211	2623	8354	62941	1310	3072	11382	87723	3151	7081	20198	104455	9507	20272	49329	191076
	D	**7.51**	10.54	11.96	14.52	**9.42**	**13.66**	**21.0**	35.39	**13.35**	**18.43**	**29.75**	40.12	24.61	**34.05**	**55.54**	100.05

TABLE III
SYNTHESIS RESULTS FOR TSMC 16nm OF THE AREA IN μm^2, WITH PSEUDO-PIPELINING FOR 1GHz.

	FP16				BF16 → FP32				FP32				FP64			
N	2	4	8	16	2	4	8	16	2	4	8	16	2	4	8	16
n	3	3	3	3	3	4	4	5	3	4	5	6	4	5	5	6
Best	Tao	Full	Full	Full	Tao	Ours	Ours	Full	Tao	Ours	Ours	Full	Tao	Ours	Ours	Ours
Area	1674	2949	4971	9202	1975	3789	11507	25464	4980	9240	19873	40227	12507	24524	56061	132049

VI. CONCLUSIONS

This article explores architectures of exact fused dot product add operators with $N+1$ floating-point operands. Our results show that operators with a compressed accumulator perform better than those with a Kulisch-like full-size accumulator when the floating-point range is large and N is small.

The FP16 format has a large precision with respect to its range, so the full size approach performs better except for $N = 2$. The FP32 format has comparatively less precision with respect to its range, so the full size approach only becomes relevant for $N > 8$. The FP64 formats has a very large range with respect to their precision, therefore a compressed approach performs better even for $N = 16$.

Acknowledgment

This work received funding from the European High-Performance Computing Joint Undertaking (EPI SGA2) and the ANR Imprenum project. The authors wish to thank Maël Feurgard for enlightening discussions, as well as anonymous reviewers for their feedback.

REFERENCES

[1] L. Bertaccini, G. Paulin, T. Fischer, S. Mach, and L. Benini, "MiniFloat-NN and ExSdotp: An ISA Extension and a Modular Open Hardware Unit for Low-Precision Training on RISC-V Cores," in *29th Symp. on Computer Arithmetic (ARITH)*, pp. 1–8, IEEE, 2022.

[2] C. S. Burrus, M. Frigo, S. G. Johnson, M. Pueschel, and I. Selesnick, *Fast Fourier Transforms*. Connexions, Rice University, 2008.

[3] M. Tasche and H. Zeuner, "Improved roundoff error analysis for precomputed twiddle factors," *J. of Computational Analysis and Applications*, vol. 4, pp. 1–18, 01 2002.

[4] R. C. Singleton, "On computing the fast fourier transform," *J. of the ACM*, vol. 10, no. 10, 1967.

[5] Y. Tao, G. Deyuan, F. Xiaoya, and R. Xianglong, "Three-operand floating-point adder," in *2012 IEEE 12th Int. Conf. on Computer and Information Technology*, pp. 192–196, IEEE, 2012.

[6] Y. Tao, G. Deyuan, F. Xiaoya, and J. Nurmi, "Correctly rounded architectures for floating-point multi-operand addition and dot-product computation," in *2013 IEEE 24th Int. Conf. on Application-Specific Systems, Architectures and Processors*, pp. 346–355, IEEE, 2013.

[7] J. Sohn and E. E. Swartzlander, "A fused floating-point three-term adder," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 61, no. 10, pp. 2842–2850, 2014.

[8] N. Brunie, "Modified fused multiply and add for exact low precision product accumulation," in *IEEE 24th Symp. on Computer Arithmetic (ARITH)*, 2017.

[9] B. Hickmann and D. Bradford, "Experimental analysis of matrix multiplication functional units," in *IEEE 26th Symp. on Computer Arithmetic (ARITH)*, 2019.

[10] B. Hickmann, J. Chen, M. Rotzin, A. Yang, M. Urbanski, and S. Avancha, "Intel Nervana Neural Network Processor-T (NNP-T) Fused Floating Point Many-Term Dot Product," in *IEEE 27th Symp. on Computer Arithmetic (ARITH)*, 2020.

[11] "Arm®A64 Instruction Set Architecture Armv9, for Armv9-A architecture profile," November 2018 – 2021.

[12] J. Sohn and E. E. Swartzlander, "A fused floating-point four-term dot product unit," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 63, no. 3, pp. 370–378, 2016.

[13] H. Kaul, M. Anders, S. Mathew, S. Kim, and R. Krishnamurthy, "Optimized fused floating-point many-term dot-product hardware for machine learning accelerators," in *IEEE 26th Symp. on Computer Arithmetic (ARITH)*, 2019.

[14] U. W. Kulisch, *Advanced Arithmetic for the Digital Computer: Design of Arithmetic Units*. Springer-Verlag, 2002.

[15] N. Brunie, "Towards the basic linear algebra unit: replicating multi-dimensional FPUs to accelerate linear algebra applications," in *54th Asilomar Conf. on Signals, Systems, and Computers*, pp. 1283–1290, IEEE, 2020.

[16] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres, *Handbook of Floating-Point Arithmetic, 2nd edition*. Birkhauser, 2018.

[17] W. D. Hillis and G. L. Steele Jr, "Data parallel algorithms," *Communications of the ACM*, vol. 29, no. 12, pp. 1170–1183, 1986.

[18] Intel, "BFLOAT16 – Hardware Numerics Definition Revision 1.0," November 2018.

[19] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, 2011.

[20] S. W. A.-H. Baddar and K. E. Batcher, *Designing sorting networks: A new paradigm*. Springer, 2012.

[21] R. W. Floyd and D. E. Knuth, "The bose-nelson sorting problem," in *A survey of combinatorial theory*, pp. 163–172, Elsevier, 1973.

[22] R. Zeno, "A reference of the best-known sorting networks for up to 16 inputs.." https://www.angelfire.com/blog/ronz/Articles/999SortingNetworksReferen.html, 2002. Accessed: 2023-04-04.

[23] I. Parberry, "A computer-assisted optimal depth lower bound for nine-input sorting networks," *Mathematical systems theory*, vol. 24, no. 1, pp. 101–116, 1991.

[24] T. Ehlers and M. Müller, "New bounds on optimal sorting networks," in *Evolving Computability: 11th Conf. on Computability in Europe, (CiE 2015)*, pp. 167–176, Springer, 2015.

[25] S. Chevillard, M. Joldeș, and C. Lauter, "Sollya: An environment for the development of numerical codes," in *Mathematical Software – ICMS 2010*, pp. 28–31, Springer Berlin Heidelberg, 2010.