electronics

MDPI

Article

Approximate Floating-Point Multiplier based on Static Segmentation

Gennaro Di Meo *, Gerardo Saggese, Antonio G. M. Strollo, Davide De Caro and Nicola Petra

Department of Electrical Engineering and Information Technology, University of Naples Federico II, 80131 Naples, Italy

* Correspondence: gennaro.dimeo@unina.it

Abstract: In this paper a novel low-power approximate floating-point multiplier is presented. Since the mantissa computation is responsible for the largest part of the power consumption, we apply a novel approximation technique to mantissa multiplication, based on static segmentation. In our approach, the inputs of the mantissa multiplier are properly segmented so that a small inner multiplier can be used to calculate the output, with beneficial impact on power and area. To further improve performance, we introduce a novel segmentation-and-truncation approach which allows us to eliminate the shifter normally present at the output of the segmented multiplier. In addition, a simple compensation term for reducing approximation error is employed. The accuracy of the circuit can be tailored at the design time, by acting on a single parameter. The proposed approximate floating-point multiplier is compared with the state-of-the-art, showing good performance in terms of both precision and hardware saving. For single-precision floating-point format, the obtained *NMED* is in the range 10^{-5}–7×10^{-7}, while *MRED* is in the range 3×10^{-3}–1.7×10^{-4}. Synthesis results in 28 nm CMOS show area and power saving of up to 82% and 85%, respectively, compared to the exact floating-point multiplier. Image processing applications confirm the expectations, with results very close to the exact case.

Keywords: floating-point multiplier; approximate computing; static segment method (SSM); low power

Citation: Di Meo, G.; Saggese, G.; Strollo, A.G.M.; De Caro, D.; Petra, N. Approximate Floating-Point Multiplier based on Static Segmentation. *Electronics* **2022**, *11*, 3005. https://doi.org/10.3390/electronics11193005

Academic Editors: Marcello Traiola, Elena-Ioana Vătăjelu and Angeliki Kritikakou

Received: 26 August 2022
Accepted: 19 September 2022
Published: 22 September 2022

1. Introduction

Multipliers are the most-used arithmetic blocks in many digital signal processing applications, being the basic elements for operations as filtering, correlation, de-noising, and domain transformation. Thanks to the favorable hardware features, fixed-point implementation is extensively exploited in a wide range of electronic systems including transceivers [1,2], FPGA accelerators [3,4], digital phase locked loops and spread spectrum clock generators [5–7]. The fixed-point arithmetic employs a fixed number of bits for representing the integer and the fractional parts of the signals. The bit-length of the integer part is related to the range of representable numbers, while the bit-length of the fractional part affects the accuracy of the operations. Therefore, the designer must properly choose the signal bit-widths and resolutions to manage numerical range and precision.

Floating-point (FP) arithmetic, while complex from the point of view of hardware implementation, offers a flexible way of performing numerical computations, providing at the same time a large range of representable values and high precision. It is therefore routinely used in applications such as scientific computing, digital signal processing, and computer graphics. The standardized IEEE 754 format [8] is ubiquitous in most computing platforms, from CPUs to GPUs and microcontrollers. According to this standard, a FP number consists of sign S, exponent E and mantissa M. The value encoded in the FP format is given by: $A = (-1)^S \cdot (1 + M) \cdot 2^{E\text{-}bias}$, with the mantissa M in the range [0, 1).

A FP multiplication involves a fixed-point adder to sum up the exponents, a fixed-point multiplier for the mantissa processing, and a normalization logic for the result. It

follows that in most applications the FP multiplication dissipates the largest part of the overall power consumption [9]. In recent years, approximate computing techniques have been introduced in FP multipliers to save power and area, at the cost of introducing some (small) errors in the result. Approximate computing is a general paradigm that aims at improving the hardware performance by intentionally introducing approximations into the design [10,11]. The authors of [12] devise a piecewise linear approximation for the mantissa multiplication and introduce a tunable error compensation technique. The work [13] involves gate-level pruning and inexact carry propagate adder (CPA) for the computation of the product, whereas [14] skips the multiplication and transfers one of the input mantissas to the output. In [15], the mantissa multiplier is substituted by an adder, and an adaptive control logic is introduced for alleviating the approximation error. The paper [16] scales the accuracy of the multiplier, freezing some of its partial products, whereas in [17] an approximate addition performs the product in the logarithmic number system.

While only a few papers deal with approximate FP multipliers, approximate fixed-point multipliers have been extensively studied by using several techniques. In [18–21], the partial products generation stage is approximated by truncating some of the less-significant partial products and mitigating the truncation error with a suitable correction function. The papers [22–29] approximate the compression of partial product matrix: the papers [22,23] use simple OR gates for the compression, whereas [24–29] use more complex approximate compressors. The works [30–35] design low-power multipliers by assembling small elementary approximate multipliers in a hierarchical fashion. The use of logarithmic number system is investigated in [36–39]. The static and the dynamic segment methods reduce the size of the multiplier by selecting portions of the inputs for the product [40–43]. While the dynamic segmentation allows a finer selection at the cost of introduction of additive logic (as leading one detectors and barrel shifters), the static segmentation alleviates the hardware burden by choosing between fixed portions of the inputs.

In this paper, we propose the design of a novel approximate static segmented floating-point multiplier (SSFPM) with static segmentation applied to the mantissa product. The proposed circuit is designed for single precision floating-point arithmetic.

In the proposed approach, the inputs of the mantissa multiplier are properly segmented so that a small inner multiplier can be used to calculate the output. To further improve performance, we introduce a novel segmentation-and-truncation approach which allows us to eliminate the shifter normally present at the output of the segmented multiplier. In addition, as in [41], a simple compensation term for reducing the approximation error is employed. The accuracy of the SSFPM can be accurately tailored at the design time, by acting on a single parameter.

Analysis of the error metrics shows that the proposed SSFPM is competitive with the state-of-the-art. The power consumption and the area occupation, obtained by synthesis in TSMC 28 nm CMOS technology, demonstrate remarkable performance. Finally, the results of signal processing applications such as JPEG compression, image filtering, and tone mapping of high dynamic range (HDR) images [44] show the effectiveness of the proposed SSFPM.

The paper is organized as follows: in Section 2 we recall the steps used to perform the standard floating-point multiplication. In this section we also describe the optimized normalization logic, the proposed static segment method, and the correction technique. Section 3 shows the results in terms of error metrics and hardware performances, as well as discusses the behavior of the SSFPM in image processing applications. Section 4 compares the results with the state-of-the-art, while Section 5 concludes the paper.

2. Floating-Point Multiplication

2.1. Single Precision Floating-Point Multiplier

In the following we consider the IEEE 754 single precision standard. According to this standard, a FP number consists of sign S, exponent E and mantissa M.

The value encoded in the FP format, in the case of normalized representations, is given by: $X = (-1)^S \cdot (1 + M) \cdot 2^{E\text{-}bias}$ (for the sake of simplicity, we do not consider de-normalized representations).

The mantissa M is in the range [0, 1). The '1' bit added to M is the so-called implicit bit. The mantissa M is represented with 23 bits (with MSB and LSB of weights 2^{-1} and 2^{-23}, respectively). The exponent E is an 8-bit integer, while the exponent bias is 127. Thus, the exponent value E–*bias* is in the range [–127, 128].

The Figure 1 shows an example, where the decimal number −13.140625 is represented according to the IEEE 754 single precision standard. This number can be written as: $-2^3 \times (1 + 1/2 + 1/8 + 1/64 + 1/512)$. Thus: the sign bit is $S = 1$; the exponent value is $E = 3 + bias = 130$, corresponding to binary 10000010; the mantissa is M = 1/2 + 1/8 + 1/64 + 1/512, corresponding to binary 10100100100000000000000.

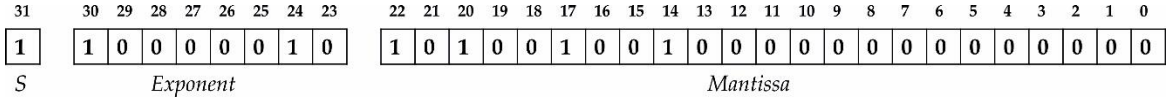


Figure 1. Representation of the decimal number −13.140625 according to the IEEE 754 single precision standard. The numbers from 0 to 31 highlight the bit position in the digital string.

Figure 2 shows the block diagram of the single precision floating-point multiplier.

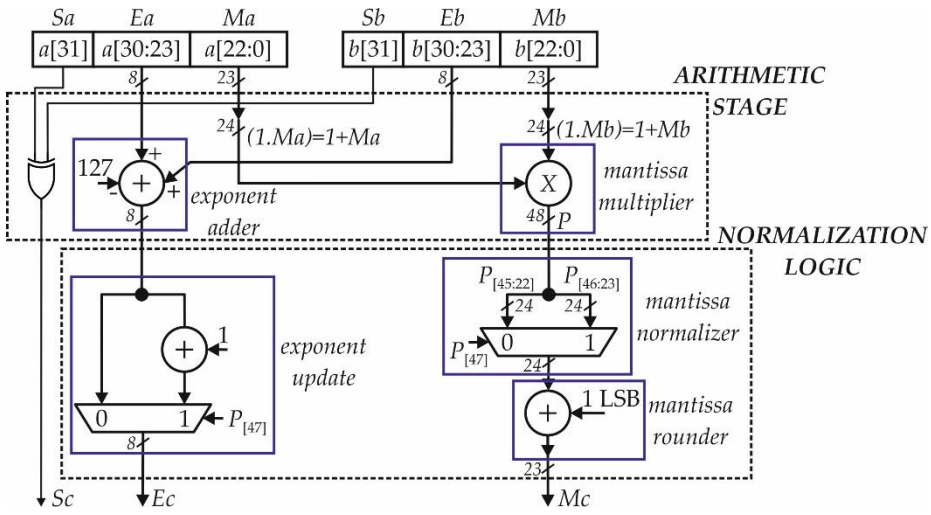


Figure 2. Block diagram of the single precision floating-point multiplier.

As detailed above, the input a is expressed on 32 bits, with $n_e = 8$ bits dedicated to the exponent and $n_m = 23$ bits dedicated to the mantissa. More precisely, the most significant bit (MSB) of a, $a[31]$, is the sign Sa, while the portions $Ea = a[30:23]$ and $Ma = a[22:0]$ are the exponent and the mantissa, respectively. The same scheme applies also to the input b, with sign $Sb = b[31]$, exponent $Eb = b[30:23]$, and mantissa $Mb = b[22:0]$. Therefore, the signals a and b are represented as follows:

$$\begin{aligned} a &= (-1)^{Sa} \cdot 2^{Ea-127} \cdot (1 + Ma) \\ b &= (-1)^{Sb} \cdot 2^{Eb-127} \cdot (1 + Mb) \end{aligned} \tag{1}$$

Similarly, the product $c = a \cdot b$ is expressed in the form:

$$c = (-1)^{Sc} \cdot 2^{Ec-127} \cdot (1 + Mc) \tag{2}$$

where Sc, Ec, and Mc are the sign, exponent and mantissa of c, respectively.

The mantissas Ma, Mb, and Mc are in the range [0, 1) with MSB and LSB of weights 2^{-1} and 2^{-23}, respectively. Therefore, they constitute the fractional part of the quanti-

ties (1 + *Ma*), (1 + *Mb*), and (1 + *Mc*) (also indicated with (1.*Ma*), (1.*Mb*) and (1.*Mc*) in the following).

The arithmetic stage of Figure 2 computes *Sc*, *Ec*, and the mantissa product *P*. As shown, the XOR between *Sa* and *Sb* allows us to obtain the sign *Sc*. The sum between *Ea* and *Eb* computes the exponent *Ec*, while the subtraction by 127 considers the exponent bias. In the mantissa multiplier, a bit '1' is explicitly concatenated to *Ma* and *Mb* at the most significant position (see (1.*Ma*) = (1 + *Ma*) and (1.*Mb*) = (1 + *Mb*) in the figure)), to compute:

$$P = (1 + Ma)\cdot(1 + Mb) \tag{3}$$

It is worth noting that (1 + *Ma*) lays in the range [1, 2). As consequence, *P* is in the range [1, 4), that means that 2 MSBs are involved for representing its integer part (namely *p*[47] and *p*[46]). We also underline that *P* is expressed on 48 bits.

The normalization logic extracts the mantissa *Mc* from *P* and consequently adjusts the exponent *Ec*. If $P < 2$ (i.e., if *p*[47] is low), the product *P* is in the form (1.*Mc*). Therefore, the extraction of *Mc* simply requires to select the fractional part of *P*, that is *p*[45:0]. Actually, only 24 bits of *P* (that is *p*[45:22]) are sent to the next rounding stage.

In the case $P \geq 2$ (i.e., if *p*[47] is high), we need to right-shift *P* of one position in order to express the product in the form (1.*Mc*) before to apply the rounding. Therefore, the segment *p*[46:23] is sent to the rounding stage. In this case, moreover, the exponent is incremented by one to compensate for the shift on *P*, as shown in exponent update block in Figure 2.

The mantissa rounding block, finally, rounds the mantissa to 23 bits, as required by the standard, with the help of a 24-bit adder.

2.2. Proposed Optimization for the Mantissa Computation

In the proposed approach, instead of computing *P* as in (3), we compute:

$$P' = P - 1 \tag{4}$$

as follows:

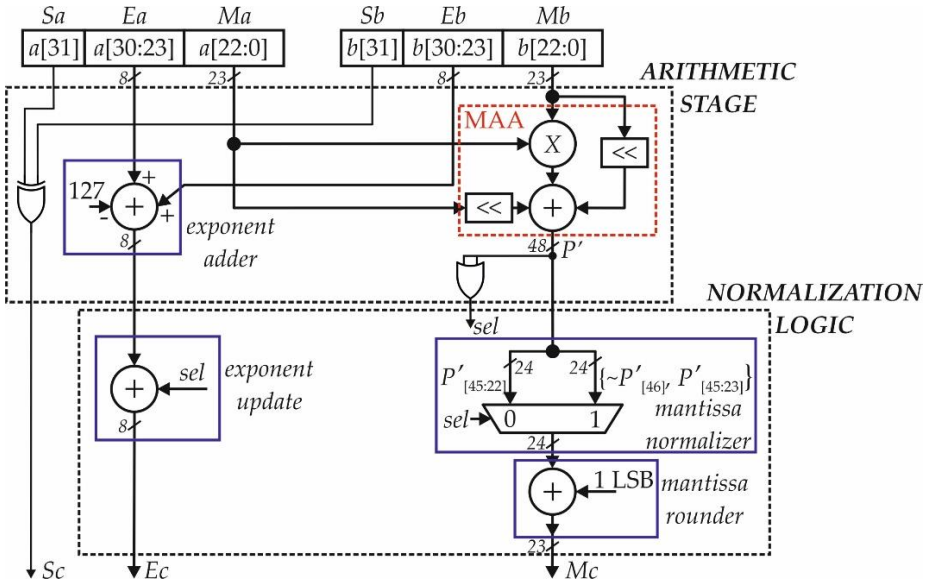
$$P' = (1 + Ma)\cdot(1 + Mb) - 1 = Ma\cdot Mb + Ma + Mb \tag{5}$$

Please note that (3) requires a 24 bit × 24 bit multiplier, due to the bit '1' explicitly concatenated to *Ma* and *Mb*. On the other hand, the calculation of *P*′ in (5) requires a smaller 23 bit × 23 bit multiplier. Equation (5) opens the way to a static segmentation of the multiplication: in fact in the multiplication operation described by (3) the bit '1' does not allow to segment the multiplier inputs, while, on the other hand, the multiplicands contained in (5) does not include any stuck at '1' bit. Figure 3a shows the structure of the proposed floating-point multiplier and highlights in the dashed red rectangle the multiply-and-add unit (MAA) that implements Equation (5).

As shown in the figure, since the LSBs of the mantissas and of the product *Ma*·*Mb* have weights 2^{-23} and 2^{-46}, respectively, both *Ma* and *Mb* are added with a hard-wired left-shift of 23 positions, to properly align their LSBs to *Ma*·*Mb*.

In this implementation, the two MSBs of *P*′ are exploited to manage the normalization process. Before to proceed with the discussion, let us observe that the maximum value of *P*′, named P'_{max}, can easily be calculated since the maximum value of *Ma* and *Mb* is equal to $(1–2^{-23})$. We have: $P'_{max} = (1–2^{-23})^2 + 2\cdot(1–2^{-23})$, therefore P'_{max} is slightly less than 3. Thus, $0 \leq P' < 3$ and, in binary representation, the two MSBs of *P*′, which constitute its integer part, can vary between '00', '01', and '10'. This observation helps the calculation of $P = P' + 1$. In fact, the fractional bits of *P* coincide with the fractional bits of *P*′ and only the two MSBs of *P* (the integer part) should be computed from the two MSBs of *P*′. To this purpose, let us consider the various cases reported in the truth table of Figure 3b. When *p*′[47:46] = '00', the two MSBs of *P* are *p*[47:46] = '01'. In this case, the normalization is not required since $P < 2$. Conversely, when *p*′[47:46] = '01' or '10', the two MSBs of *P* are

p[47:46] = '10' or '11'. These cases demand for the normalization since $P \geq 2$. Following the truth table, we can define the signal *sel* as the OR between p'[47] and p'[46], and normalize the mantissa when *sel* = 1 (see also Figure 3a). The truth table also shows that p[46] is inverted with respect to p'[46]. Therefore, the output of the mantissa multiplier is given by {~p'[46], p[45:0]}, where '~' is used to represent the complement operation.



(**a**)

$P'_{[47:46]}$	$P_{[47:46]}$	*sel*
00	01	0
01	10	1
10	11	1
11	NA	NA

(**b**)

Figure 3. (**a**) Block diagram of the improved floating-point multipliers and (**b**) truth table that links the MSBs of P' and P. NA stands for not applicable.

After mantissa multiplier, mantissa normalization and rounding follow, as in Figure 2. The main difference is the use of signal *sel* (instead of p[47]) to perform normalization. Note also that exponent update is implemented without multiplexer, by using *sel* to increment the exponent.

2.3. Static Segmentation Method

The static segmentation [40,41] reduces the size of the multiplier by segmenting the multiplicands before the product. Each segment comprises m bits, with $n_m/2 < m < n_m$. Therefore, an $m \times m$ multiplier is employed for computing the result instead of a $n_m \times n_m$ multiplier. Synthesizable HDL descriptions of fixed-point static-segmented multipliers are available in [45].

Figure 4 shows the segmentation for the mantissa *Ma*, where, for the sake of simplicity, we assume n_m = 8 bits with m = 5. The mantissa *Ma* is divided in a lower portion (*LPa*), given by its m LSBs, and in an upper portion (*UPa*), given by its m MSBs. The (n_m-m) MSBs constitutes the control segment *CSa*, used to decide between *LPa* and *UPa*. When the bits of *CSa* are low, the segment *LPa* is selected for the multiplication. Conversely, if at least one of the bits of *CS* is high, the segment *UPa* is chosen. A similar mechanism is applied also to input *Mb*. Therefore, defining the selection flags α_{Ma} and α_{Mb} as the OR of bits of the control segments *CSa* and *CSb*, respectively, and naming Ma_{ssm}, Mb_{ssm} the segmented signals, the following relations hold:

$$Ma_{ssm} = \begin{cases} Ma[m-1:0] & if\ \alpha_{Ma} = 0 \\ Ma[n_m-1:n_m-m] & if\ \alpha_{Ma} = 1 \end{cases} \quad (6)$$

$$Mb_{ssm} = \begin{cases} Mb[m-1:0] & if\ \alpha_{Mb} = 0 \\ Mb[n_m-1:n_m-m] & if\ \alpha_{Mb} = 1 \end{cases} \quad (7)$$

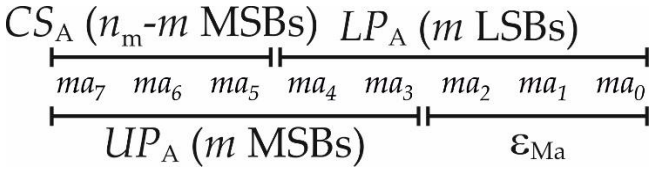


Figure 4. Segmentation of the mantissa *Ma* in the case n_m = 8 bits and m = 5 bits.

Please note that an approximation error is introduced when the upper portion of the mantissa is multiplied (i.e., when the selection flag is high) since the less significant part is discarded (namely ε_{Ma} in the figure). On the contrary, no approximation is introduced if the lower portion is selected.

After the multiplication, a left-shift is required to extend the result from $2{\cdot}m$ bits to $2{\cdot}n_m$ bits. Then, the approximate product K_{ssm} is computed as follows:

$$K_{ssm} = (Ma_{ssm}{\cdot}2^{LSHa}){\cdot}(Mb_{ssm}{\cdot}2^{LSHb}) = (Ma_{ssm}{\cdot}Mb_{ssm}){\cdot}2^{LSH} \tag{8}$$

with *LSHa* and *LSHb* defined as

$$LSHa = \begin{cases} 0 & if\ \alpha_{Ma} = 0 \\ n_m - m & if\ \alpha_{Ma} = 1 \end{cases} \quad LSHb = \begin{cases} 0 & if\ \alpha_{Mb} = 0 \\ n_m - m & if\ \alpha_{Mb} = 1 \end{cases} \tag{9}$$

The term *LSH* in (8) is given by: *LSH* = *LSHa* + *LSHb* and is the number of positions for the overall left-shift:

$$LSH = \begin{cases} 0 & if\ \alpha_{Ma} = 0,\ \alpha_{Mb} = 0 \\ n_m - m & if\ \alpha_{Ma} = 0,\ \alpha_{Mb} = 1\ or\ \alpha_{Ma} = 1,\ \alpha_{Mb} = 0 \\ 2{\cdot}(n_m - m) & if\ \alpha_{Ma} = 1,\ \alpha_{Mb} = 1 \end{cases} \tag{10}$$

2.4. Static Segmentation Applied to the Mantissa Product

In this paragraph, we apply the segmentation to the inputs of the MAA unit to employ an $m \times m$ multiplier and an m-bits adder for the mantissa computation. By assuming that Ma_{ssm} and Mb_{ssm} have LSB of weight 2^0, we write the approximate product P'_{apprx} as follows:

$$P'_{apprx} = (Ma_{ssm}{\cdot}Mb_{ssm}){\cdot}2^{-2{\cdot}nm+LSH} + Ma_{ssm}{\cdot}2^{-nm+LSHa} + Mb_{ssm}{\cdot}2^{-nm+LSHb} \tag{11}$$

Figure 5 depicts the circuit that implements Equation (11).

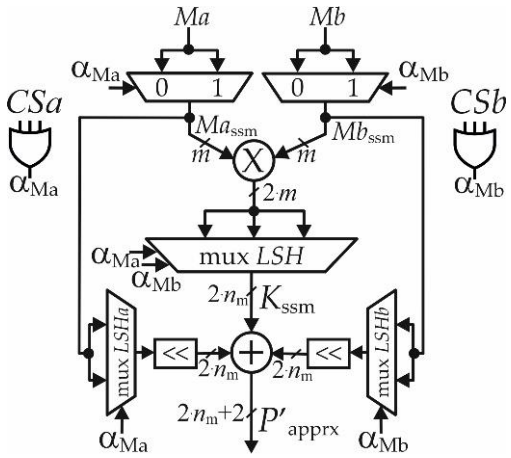


Figure 5. Segmented MAA that implements Equation (11).

The multiplexer mux *LSH*, used for the left-shift of the multiplier output, does not allow to merge the multiplier and the adder in a fused PPM, and leads to the usage of

two cascaded CPAs, one for computing the product $Ma_{ssm} \cdot Mb_{ssm}\ 2^{-2nm+LSH}$ and one for computing P'_{apprx}. Furthermore, $Ma_{ssm} \cdot\ 2^{-nm+LSHa}$ and $Mb_{ssm} \cdot\ 2^{-nm+LSHb}$ involve up to $2 \cdot n_m$ bits when $\alpha_{Ma} = 1$ and $\alpha_{Mb} = 1$, thus degrading the performances of the adder.

In order to optimize the MAA unit, we analyze Equation (11) considering all the possible combinations for α_{Ma} and α_{Mb}. Figure 6 shows the alignments of the signals reporting the exact MAA for reference (Figure 6a), and the alignments with segmentation in the case $n_m = 8$ and $m = 6$ (Figure 6b–d).

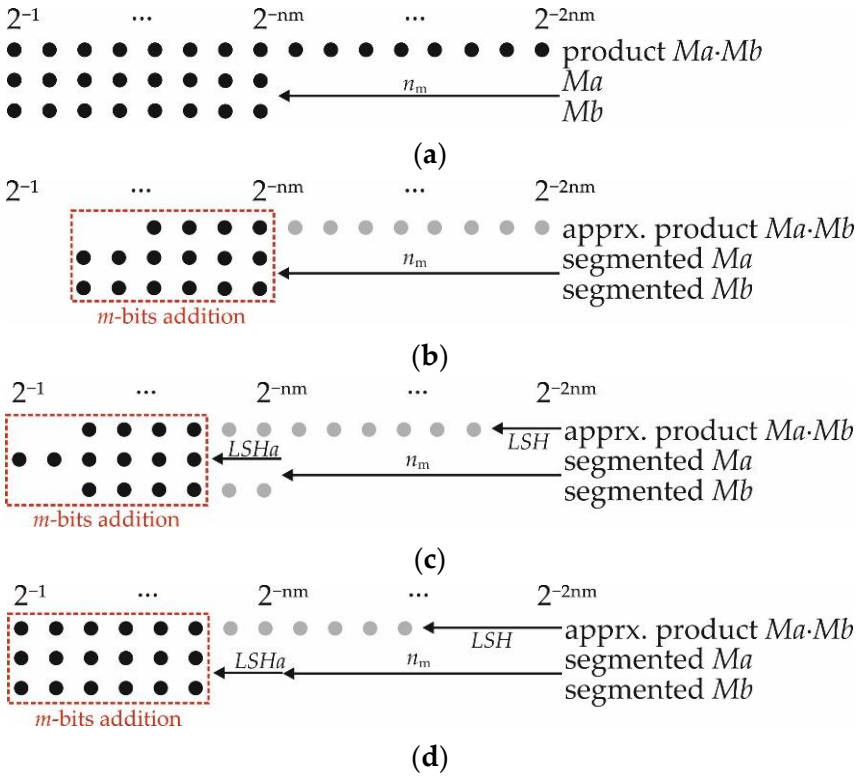


Figure 6. Signals alignment for the mantissa product in (**a**) the exact case, and the approximate static segmented cases with (**b**) $\alpha_{Ma} = 0$, $\alpha_{Mb} = 0$, (**c**) $\alpha_{Ma} = 1$, $\alpha_{Mb} = 0$, and (**d**) $\alpha_{Ma} = 1$, $\alpha_{Mb} = 1$. In this example, we consider $n_m = 8$ bits and $m = 6$ bit.

If $\alpha_{Ma} = 0$ and $\alpha_{Mb} = 0$, the shifts *LSH*, *LSHa* and *LSHb* are zero, and Equation (11) becomes:

$$P'_{apprx} = \left[(Ma_{ssm} \cdot Mb_{ssm}) \cdot 2^{-nm} + Ma_{ssm} + Mb_{ssm}\right] \cdot 2^{-nm} \tag{12}$$

As shown in Figure 6b, the product is on $2 \cdot m$ bits, whereas the shifted mantissas imply the usage of an adder on $(n_m + m)$ bits. To employ an m-bit adder, we truncate the product $Ma_{ssm} \cdot Mb_{ssm} \cdot 2^{-nm}$ discarding the gray LSBs in the figure. Therefore, we approximate Equation (12) as follows:

$$P'_{apprx} = \left[\left\lfloor Ma_{ssm} \cdot Mb_{ssm} \cdot 2^{-nm} \right\rfloor + Ma_{ssm} + Mb_{ssm}\right] \cdot 2^{-nm} \tag{13}$$

where $\lfloor \cdot \rfloor$ is used to represent the floor operator.

In order to implement the calculations in (13) with a fused PPM and an unique CPA, we rearrange (13) as follows:

$$P'_{apprx} = [Ma_{ssm,mult} \cdot Mb_{ssm,mult} + Ma_{ssm,add} + Mb_{ssm,add}] \cdot 2^{-nm} \tag{14}$$

where:

$$\begin{aligned} Ma_{ssm,mult} &= \left\lfloor Ma_{ssm} \cdot 2^{-\lceil nm/2 \rceil} \right\rfloor \\ Mb_{ssm,mult} &= \left\lfloor Mb_{ssm} \cdot 2^{-\lfloor nm/2 \rfloor} \right\rfloor \\ Ma_{ssm,add} &= Ma_{ssm} \\ Mb_{ssm,add} &= Mb_{ssm} \end{aligned} \tag{15}$$

and $\lceil \cdot \rceil$ is the ceiling operator.

For the sake of simplicity, let us consider the case in which n_m is even. The final floor operation on $Ma_{ssm,mult}$ and $Mb_{ssm,mult}$ of (15) results in truncating $n_m/2$ bits from m bit operands and, therefore, the computation of (14) requires a $(m - n_m/2) \times (m - n_m/2)$ multiplier.

As observable from the formula, we can first compute $Ma_{ssm,mult}$, $\mathrm{Mb}_{ssm,mult}$, that are truncated versions of Ma_{ssm}, Mb_{ssm}, and then execute the multiply-and-add operation. In this way, we remove the shift between the multiplier and the adder and design the MAA unit with a fused PPM and a unique CPA.

When $\alpha_{Ma} = 1$ and $\alpha_{Mb} = 0$, we have $LSH = LSHa = n_\mathrm{m} - m$ (refer also to (9),(10)). Therefore, the (11) becomes

$$P'_{apprx} = \left[(Ma_{ssm} \cdot Mb_{ssm}) \cdot 2^{-nm} + Ma_{ssm} + Mb_{ssm} \cdot 2^{-LSH} \right] \cdot 2^{-nm+LSH} \tag{16}$$

Applying the same reasoning, we need to discard the gray LSBs of $Ma_{ssm} \cdot Mb_{ssm} \cdot 2^{-nm}$ and of $Mb_{ssm} \cdot 2^{-LSH}$ (see Figure 6c) in order to involve again an m-bit addition. Furthermore, we need also to remove the shift at the output of the multiplier.

The above considerations lead to the following approximation for (16):

$$P'_{apprx} = [Ma_{ssm,mult} \cdot Mb_{ssm,mult} + Ma_{ssm,add} + Mb_{ssm,add}] \cdot 2^{-nm+LSH} \tag{17}$$

with $Ma_{ssm,mult}$, $Mb_{ssm,mult}$, and $Ma_{ssm,add}$ defined as in (15), and $Mb_{ssm,add}$ that is

$$Mb_{ssm,add} = \left\lfloor Mb_{ssm} \cdot 2^{-LSH} \right\rfloor \tag{18}$$

Here, the addend Mb_{ssm} is also truncated along with the multiplier inputs. Since the expression of $Ma_{ssm,mult}$ and $Mb_{ssm,mult}$ remains the same, also in this case the computation of (17) requires a $(m - n_m/2) \times (m - n_m/2)$ multiplier.

A similar reasoning applies also for the case $\alpha_{Ma} = 1$ and $\alpha_{Mb} = 0$, with $Ma_{ssm,mult}$, $Mb_{ssm,mult,}$ and $Mb_{ssm,add}$ defined as in (15) and $Ma_{ssm,add} = \left\lfloor Ma_{ssm} \cdot 2^{-LSH} \right\rfloor$.

Finally, when $\alpha_{Ma} = 1$ and $\alpha_{Mb} = 1$, we have $LSH = 2 \cdot (n_\mathrm{m} - m)$ and $LSHa = LSHb = n_\mathrm{m} - m$. Therefore, the (11) becomes

$$P'_{apprx} = \left[(Ma_{ssm} \cdot Mb_{ssm}) \cdot 2^{-nm+LSHa} + Ma_{ssm} + Mb_{ssm} \right] \cdot 2^{-nm+LSHa} \tag{19}$$

As shown in Figure 6d, we need to truncate $(Ma_{ssm} \cdot Mb_{ssm}) \cdot 2^{-nm+LSHa}$ for employing an m-bit adder, and to shift the inputs of the multiplier for employing a unique CPA. Therefore, the (19) is approximated as follows

$$P'_{apprx} = [Ma_{ssm,mult} \cdot Mb_{ssm,mult} + Ma_{ssm,add} + Mb_{ssm,add}] \cdot 2^{-nm+LSHa} \tag{20}$$

With

$$\begin{aligned} Ma_{ssm,mult} &= \left\lfloor Ma_{ssm} \cdot 2^{\lceil (-nm+LSHa)/2 \rceil} \right\rfloor \\ Mb_{ssm,mult} &= \left\lfloor Mb_{ssm} \cdot 2^{\lfloor (-nm+LSHa)/2 \rfloor} \right\rfloor \end{aligned} \tag{21}$$

and $Ma_{ssm,add}$, $Mb_{ssm,add}$ defined as in (15). If for the sake of simplicity we consider the case in which n_m and m are even, the final floor operation on $Ma_{ssm,mult}$ and $Mb_{ssm,mult}$ of (21) results in truncating $(n_m - LSHa)/2 = m/2$ bits from m bit operands, therefore the

computation of (20) requires a $(m/2) \times (m/2)$ multiplier. Since $m - n_m/2 < m/2$, overall, the computation of P'_{apprx} requires a $(m/2) \times (m/2)$ multiplier.

Figure 7a shows the circuit that implements the static segmented multiply-and-add unit (SSMAA), whereas Table 1 collects the segments of Ma_{ssm}, Mb_{ssm} obtained with the segment-and-truncate approach. The multiplexers on *Ma* and *Mb* perform the segmentation of Table 1 at the input of both the multiplier and the adder. Then, a further multiplexer applies the final shift on P'_{ssm} in order to express the result P'_{apprx} on $(2{\cdot}n_{\mathrm{m}} + 2) = 48$ bits.

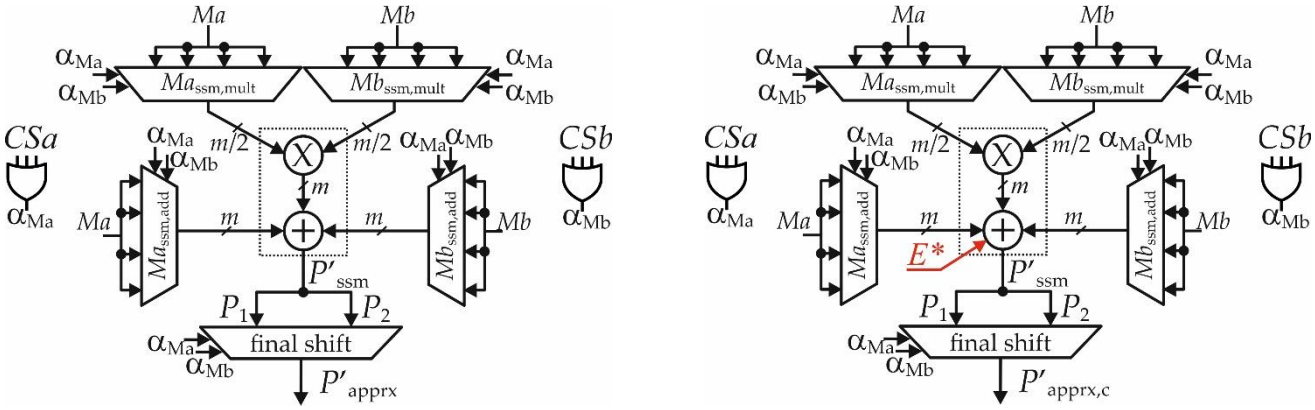


Figure 7. Block diagram of the SSMAA (**a**) without the correction term and (**b**) with the correction term (highlighted in red).

Table 1. Segmentation scheme applied to *Ma*, *Mb* at the inputs of both the multiplier and the adder.

α_{Ma}, α_{Mb}	$Ma_{ssm,mult}$	$Mb_{ssm,mult}$	$Ma_{ssm,add}$	$Mb_{ssm,add}$	Final Shift
00	$Ma[m-1:\lceil n_{\mathrm{m}}/2 \rceil]$	$Mb[m-1:\lfloor n_{\mathrm{m}}/2 \rfloor]$	$Ma[m{-}1{:}0]$	$Mb[m{-}1{:}0]$	$n_{\mathrm{m}}{-}n_{\mathrm{q}}$
01	$Ma[m-1:\lceil n_{\mathrm{m}}/2 \rceil]$	$Mb[n_{\mathrm{m}}-1:n_{\mathrm{m}}-m+\lfloor n_{\mathrm{m}}/2 \rfloor]$	$Ma[m{-}1{:}n_{\mathrm{m}}{-}m]$	$Mb[n_{\mathrm{m}}{-}1{:}n_{\mathrm{m}}{-}m]$	$2n_{\mathrm{m}}{-}m{-}n_{\mathrm{q}}$
10	$Ma[n_{\mathrm{m}}-1:n_{\mathrm{m}}-m+\lceil n_{\mathrm{m}}/2 \rceil]$	$Mb[m-1:\lfloor n_{\mathrm{m}}/2 \rfloor]$	$Ma[n_{\mathrm{m}}{-}1{:}n_{\mathrm{m}}{-}m]$	$Mb[m{-}1{:}n_{\mathrm{m}}{-}m]$	$2n_{\mathrm{m}}{-}m{-}n_{\mathrm{q}}$
11	$Ma[n_{\mathrm{m}}-1:n_{\mathrm{m}}-m+\lceil m/2 \rceil]$	$Mb[n_{\mathrm{m}}-1:n_{\mathrm{m}}-m+\lfloor m/2 \rfloor]$	$Ma[n_{\mathrm{m}}{-}1{:}n_{\mathrm{m}}{-}m]$	$Mb[n_{\mathrm{m}}{-}1{:}n_{\mathrm{m}}{-}m]$	$2n_{\mathrm{m}}{-}m{-}n_{\mathrm{q}}$

It is worth mentioning that the P'_{apprx} is subsequently quantized in the normalization process. It follows that the rounding allows us to reduce the size of the final multiplexer, since the result can be expressed on $(2{\cdot}n_{\mathrm{m}} + 2 - n_{\mathrm{q}})$ bits, n_{q} being the number of discarded LSBs ($n_{\mathrm{q}} = 22$ for the single precision FPM).

2.5. Error Analysis and Correction

The approximation errors that affect the proposed SSMAA are due to the discarding of the least significant parts of *Ma* and *Mb* (when the segmentation selects the upper segments), and due to the truncations used to employ the *m*-bit adder and a unique CPA. It follows that the largest error arises when $\alpha_{Ma} = 1$, $\alpha_{Mb} = 1$.

Estimating the error $E = P' - P'_{apprx}$ can help in improving the accuracy of the SSMAA unit. The idea is to compute the SSMAA result as $P'_{\mathrm{apprx,c}} = P'_{apprx} + E^*$, where E^* is a term which approximates *E*.

To study the error, by considering Table 1, let us write the inputs of the SSMAA as:

$$\begin{aligned} Ma_{ssm,mult} = UPa_{mult} = \sum_{k=nm-m'}^{nm-1} ma_k 2^k \quad Mb_{ssm,mult} = UPb_{mult} = \sum_{k=nm-m'}^{nm-1} mb_k 2^k \\ Ma_{ssm,add} = UPa_{add} = \sum_{k=nm-m}^{nm-1} ma_k 2^k \quad Mb_{ssm,add} = UPb_{add} = \sum_{k=nm-m}^{nm-1} mb_k 2^k \end{aligned} \tag{22}$$

where UPa_{mult}, UPb_{mult}, UPa_{add}, and UPb_{add} are the quantities selected for the SSMAA, and $m' = m - m/2 = m/2$. In this discussion we consider an even value of *m* for the sake of simplicity in order to have the same truncation for both $Ma_{ssm,mult}$ and $Mb_{ssm,mult}$.

Additionally, the quantities pruned for the segmentation are:

$$\begin{aligned} \varepsilon_{Ma,mult} &= \sum_{k=0}^{nm-m'-1} ma_k 2^k \quad \varepsilon_{Mb,mult} = \sum_{k=0}^{nm-m'-1} mb_k 2^k \\ \varepsilon_{Ma,add} &= \sum_{k=0}^{nm-m-1} ma_k 2^k \quad \varepsilon_{Mb,add} = \sum_{k=0}^{nm-m-1} mb_k 2^k \end{aligned} \tag{23}$$

By writing *Ma* and *Mb* as follows for the product:

$$\begin{aligned} Ma &= (UPa_{mult} + \varepsilon_{Ma,mult}) \cdot 2^{-nm} \\ Mb &= (UPb_{mult} + \varepsilon_{Mb,mult}) \cdot 2^{-nm} \end{aligned} \tag{24}$$

and as follows for the addition:

$$\begin{aligned} Ma &= (UPa_{add} + \varepsilon_{Ma,add}) \cdot 2^{-nm} \\ Mb &= (UPb_{add} + \varepsilon_{Mb,add}) \cdot 2^{-nm} \end{aligned} \tag{25}$$

we obtain the exact result of MAA:

$$\begin{aligned} P' = (UPa_{mult} UPb_{mult} + UPa_{mult} \varepsilon_{Mb,mult} + UPb_{mult} \varepsilon_{Ma,mult} + \varepsilon_{Ma,mult} \varepsilon_{Mb,mult}) \cdot 2^{-2nm} + \\ (UPa_{add} + \varepsilon_{Ma,add}) \cdot 2^{-nm} + (UPb_{add} + \varepsilon_{Mb,add}) \cdot 2^{-nm} \end{aligned} \tag{26}$$

Therefore, being P'_{apprx} given by:

$$P'_{apprx} = UPa_{mult} UPb_{mult} \cdot 2^{-2nm} + UPa_{add} \cdot 2^{-nm} + UPb_{add} \cdot 2^{-nm} \tag{27}$$

the error *E* is:

$$\begin{aligned} E \quad &= (UPa_{mult} \varepsilon_{Mb,mult} + UPb_{mult} \varepsilon_{Ma,mult} + \varepsilon_{Ma,mult} \varepsilon_{Mb,mult}) \cdot 2^{-2nm} + \\ &+ \varepsilon_{Mba,add} \cdot 2^{-nm} + \varepsilon_{Mb,add} \cdot 2^{-nm} \end{aligned} \tag{28}$$

In order to simplify the discussion, let us focus on the most significant term *E**:

$$E^* = (UPa_{mult} \varepsilon_{Mb,mult} + UPb_{mult} \varepsilon_{Ma,mult}) \cdot 2^{-2nm} \tag{29}$$

Following the approach of [41], we approximate $\varepsilon_{Ma,mult}$ with:

$$\varepsilon_{Ma,mult} \approx (2ma_{nm-m'-1} + 1) \cdot 2^{nm-m'-2} \tag{30}$$

and write UPa_{mult} as:

$$UPa_{mult} = 2^{nm-m'} \sum_{k=0}^{m'-1} ma_{k+nm-m'} 2^k \tag{31}$$

A similar expression holds for $\varepsilon_{\mathrm{Mb,mult}}$ and for UPb_{mult}.
Then, substituting (30) and (31) in (29), the error becomes:

$$E^* = \left[2^{2nm-2m'-2} \sum_{k=0}^{m'-1} e_{k+nm-m'} 2^k \right] \cdot 2^{-2nm} \tag{32}$$

with:

$$\begin{aligned} e_{k+nm-m'} \quad &= 2 \cdot (ma_{k+nm-m'} mb_{nm-m'-1} + mb_{k+nm-m'} ma_{nm-m'-1}) + \\ &+ ma_{k+nm-m'} + mb_{k+nm-m'} \end{aligned} \tag{33}$$

Approximating $e_{k+nm\text{-}m'}$ with:

$$e^*_{k+nm-m'} = 4 \cdot (ma_{k+nm-m'} mb_{nm-m'-1} \; OR \; mb_{k+nm-m'} ma_{nm-m'-1}) \tag{34}$$

for further simplification, we obtain:

$$E^* \approx \left[2^{2nm-2m'-2} \sum_{k=0}^{m'-1} {e^*}_{k+nm-m'} 2^k \right] \cdot 2^{-2nm} \tag{35}$$

In the case of m odd, we consider the following approximate expression for ${e^*}_{k+nm-m'}$:

$${e^*}_{k+nm-m'} = 4 \cdot (ma_{k+nm-\lceil m' \rceil} mb_{nm-\lfloor m' \rfloor -1} \; OR \; mb_{k+nm-\lfloor m' \rfloor} ma_{nm-\lceil m' \rceil -1}) \tag{36}$$

Approximating the summation in (35) with two or three terms allows us to sufficiently reduce the approximation error in the SSMAA.

Figure 7b depicts the scheme of the proposed corrected SSMAA (cSSMAA in the following). The correction term E^*, highlighted in red, can be directly fused in the PPM, thus implying a minimum impact on the hardware performances.

3. Results

3.1. Error Metrics Analysis

We exploit some of the common error metrics to verify the performances of the proposed SSFPM. Let us define the exact and the approximate result of the floating-point multiplier as C and C_{apprx}, respectively. The approximation error is given by: $E_{FMP} = C - C'_{apprx}$ while the error distance is $ED = |E_{FMP}|$. The normalized mean error and the normalized mean error distance are $NM = mean(E_{FMP})/C_{max}$ and $NMED = mean(ED)/C_{max}$, respectively, where $mean(\cdot)$ is the average operator and $C_{max} = 2^{128}$ is the maximum value of C. The mean relative error distance is $MRED = mean(ED/C)$, whereas the normalized maximum error distance is defined as $NmaxED = ED_{max}/C_{max}$, ED_{max} being the maximum value of ED.

We compute the error metrics by simulating the SSFPM with 10^7 couples of random inputs laying in the whole range of representation (that is about $[-2^{128}, 2^{128}]$).

Figure 8 represents the behavior of *NMED* and *MRED* as function of the parameter m, used to define the accuracy of the segmentation. In the corrected version of the SSFPM, two terms are used to approximate the (35).

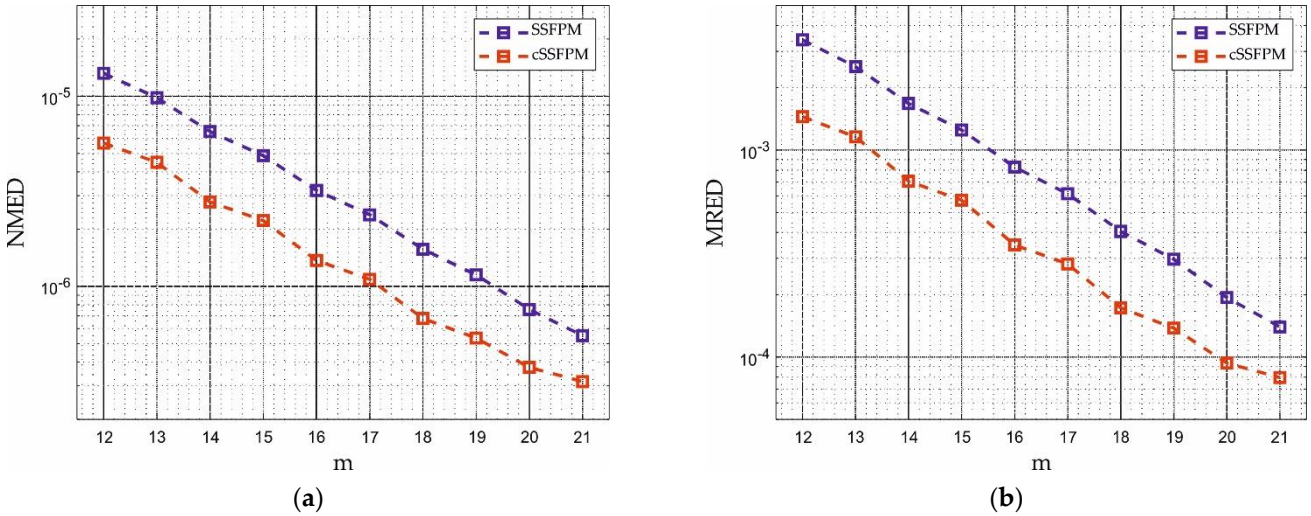


Figure 8. Behavior of (**a**) *NMED* and (**b**) *MRED* as function of m for the SSFPM and the cSSFPM.

As expected, increasing m allows us to improve both *NMED* and *MRED*, which lowers from 1.32×10^{-5} to 3.16×10^{-7} and from 3.41×10^{-3} to 7.96×10^{-5}, respectively. The figure also highlights the beneficial effects of the correction technique, since *NMED* and *MRED* are about halved in the corrected case.

Table 2 collects the error metrics of the proposed SSFPM and cSSFPM in the cases m = 12, 14, 16, and 18. For the sake of comparison, we show also the accuracy of the

approximate FPMs obtained by exploiting the techniques of [22,42,43] for the computation of the product $Ma \cdot Mb$ in (5), and by implementing the proposal of [17].

Table 2. Error metrics of the approximate FPMs.

Approximate FPM	*NM*	*NMED*	*MRED*	*NmaxED*
TOSAM [42], $h = 2$	8.13×10^{-9}	7.79×10^{-6}	1.99×10^{-3}	1.48×10^{-2}
TOSAM [42], $h = 3$	-1.93×10^{-8}	3.91×10^{-6}	1.00×10^{-3}	7.27×10^{-3}
DRUM [43], $k = 4$	-3.99×10^{-8}	2.08×10^{-5}	5.39×10^{-3}	2.85×10^{-2}
DRUM [43], $k = 6$	-7.30×10^{-9}	5.20×10^{-6}	1.35×10^{-3}	7.31×10^{-3}
AFMB [17], $t = 14$	3.65×10^{-7}	9.01×10^{-5}	2.50×10^{-2}	4.72×10^{-2}
AFMB [17], $t = 16$	3.28×10^{-7}	9.10×10^{-5}	2.53×10^{-2}	5.30×10^{-2}
AFMB [17], $t = 18$	-2.81×10^{-6}	1.02×10^{-3}	2.91×10^{-1}	4.85×10^{-1}
DATE17 [22], $L = 2$	2.63×10^{-10}	5.30×10^{-8}	1.35×10^{-5}	1.82×10^{-4}
DATE17 [22], $L = 4$	-7.94×10^{-8}	1.03×10^{-5}	2.68×10^{-3}	2.03×10^{-2}
DATE17 [22], $L = 6$	-1.93×10^{-7}	5.28×10^{-5}	1.34×10^{-2}	8.70×10^{-2}
SSFPM $m = 12$	4.06×10^{-8}	1.32×10^{-5}	3.41×10^{-3}	7.55×10^{-3}
cSSFPM $m = 12$	3.46×10^{-9}	5.67×10^{-6}	1.45×10^{-3}	4.62×10^{-3}
SSFPM $m = 14$	1.69×10^{-8}	6.52×10^{-6}	1.68×10^{-3}	3.80×10^{-3}
cSSFPM $m = 14$	5.23×10^{-9}	2.78×10^{-6}	7.08×10^{-4}	2.28×10^{-3}
SSFPM $m = 16$	1.06×10^{-8}	3.20×10^{-6}	8.28×10^{-4}	1.85×10^{-3}
cSSFPM $m = 16$	8.91×10^{-9}	1.37×10^{-6}	3.48×10^{-4}	1.12×10^{-3}
SSFPM $m = 18$	6.98×10^{-9}	1.57×10^{-6}	4.05×10^{-4}	9.03×10^{-4}
cSSFPM $m = 18$	2.42×10^{-9}	6.79×10^{-7}	1.73×10^{-4}	5.37×10^{-4}

In [42] (referred as TOSAM in the table), the authors devise the usage of a dynamic segmentation to perform multiplication with good precision and optimized power and area. Here, the multiplication is revisited as multiply-and-add operation, with the multiplicands truncated on h bits after the leading one. The addends are also truncated since $h + 4$ LSBs are discarded. The work [43] (referred as DRUM in the table) explores the dynamic segmentation selecting a segment of k bits (comprising the leading one bit and a correction bit at the least significant position) for the multiplication. Then, a barrel shifter allows us to extend the result on the desired number of bits. The HDL description of DRUM multiplier [43] is available in [46]. The technique of [22] (referred as DATE17 in the table) organizes the PPM in groups of L rows. Then, the rows of each group are compressed by means of L-input OR gates. In [17] (referred as AFMB in the table), a modified version of the Mitchell algorithm is employed to compute the product of (3), with the input signals that are truncated on t bits. Here, since the leading one bit always corresponds to the implicit bit, no leading one detectors and barrel shifters are used, with beneficial improvements on power and area.

As shown in the table, the proposed SSFPM and cSSFPM are competitive with the state-of-the-art. The implementations cSSFPM perform better than [22] with $L = 4$, $L = 6$ and slightly overcome [42,43], exhibiting *NMED* and *MRED* up to 1.37×10^{-6} and 3.48×10^{-4} ($m = 16$), and 6.79×10^{-7} and 1.73×10^{-4} ($m = 18$). In the case $m = 18$, also SSFPM performs better than [42,43], with *NMED* = 1.57×10^{-6} and *MRED* = 4.05×10^{-4}. The FPM [22] with $L = 2$ shows best accuracy with *NMED* = 5.30×10^{-8} and *MRED* = 1.35×10^{-5}. On the other hand, the implementations [17] exhibits worst performances, with *NMED* around $10^{-4}/10^{-3}$ and *MRED* that ranges between 2.5×10^{-2} and 2.9×10^{-1}.

It is also worth noting that the correction technique allows us to reduce the *NmaxED* since it lowers the maximum error of the segmented multiplier.

3.2. Electrical Performances

We synthesize the proposed segmented floating-point multipliers and the state-of-the-art with a physical flow in TSMC 28 nm CMOS technology using Cadence Genus, with

clock period T_{clk} of 500 ps and using standard threshold voltage cell library. Furthermore, the exact FPM described in Section 2.2. is implemented for reference. In the implementation of FPMs, pipeline levels are often inserted, to shorten the critical. In our case, we employ a single pipeline level between the arithmetic stage and the normalization logic in all investigated FPMs.

The power and area are obtained from post-synthesis analyses, with power consumption computed by simulating the synthesized netlist at 1 GHz. To this aim, SDF and TCF format files are employed to annotate the path delays and the toggle activity of the signals. In addition, we also study the minimum clock period employable for each FPMs, corresponding to the minimum clock period that ensures positive slack.

As shown in Table 3, the proposed SSFPM and cSSFPM are also competitive from a hardware point of view, exhibiting remarkable results with area and power reductions up to −82.3% and −85.5% with SSFPM and $m = 12$. The corrected circuits exhibit only a slight worsening of the performances with respect to the uncorrected ones, with degradations of area and power in the range of 1–3%.

Table 3. Hardware performances of the proposed SSFPM and cSSFPM, and the state-of-the-art.

FPM	Min T_{clk} [ps]	Area [μm^2]	[Power μW@1 GHz]
Exact	374	1753.8	2874.3
TOSAM [42], $h = 2$	430 (15.0%)	922.7 (−47.4%)	1290.1 (−55.1%)
TOSAM [42], $h = 3$	468 (25.1%)	1226.6 (−30.1%)	1710.6 (−40.5%)
DRUM [43], $k = 4$	447 (19.5%)	873.9 (−50.2%)	1411.5 (−50.9%)
DRUM [43], $k = 6$	503 (34.5%)	1320.2 (−24.7%)	2361.0 (−17.9%)
AFMB [17], $t = 14$	144 (−61.5%)	112.4 (−93.6%)	179.7 (−93.7%)
AFMB [17], $t = 16$	134 (−64.2%)	99.4 (−94.3%)	157.9 (−94.5%)
AFMB [17], $t = 18$	110 (−70.6%)	85.2 (−95.1%)	124.5 (−95.7%)
DATE17 [22], $L = 2$	333 (−11.0%)	1136.3 (−35.2%)	1508.5 (−47.5%)
DATE17 [22], $L = 4$	299 (−20.1%)	815.2 (−53.5%)	1199.1 (−58.3%)
DATE17 [22], $L = 6$	261 (−30.2%)	707.0 (−59.7%)	1005.2 (−65.0%)
SSFPM $m = 12$	289 (−22.7%)	310.5 (−82.3%)	417.1 (−85.5%)
cSSFPM $m = 12$	286 (−23.5%)	364.8 (−79.2%)	454.8 (−84.2%)
SSFPM $m = 14$	323 (−13.6%)	392.7 (−77.6%)	504.1 (−82.5%)
cSSFPM $m = 14$	314 (−16.0%)	437.9 (−75.0%)	532.4 (−81.5%)
SSFPM $m = 16$	333 (−11.0%)	484.9 (−72.4%)	594.5 (−79.3%)
cSSFPM $m = 16$	337 (−9.9%)	477.5 (−72.8%)	603.6 (−79.0%)
SSFPM $m = 18$	355 (−5.1%)	524.7 (−70.1%)	683.6 (−76.2%)
cSSFPM $m = 18$	360 (−3.7%)	611.4 (−65.2%)	787.9 (−72.6%)

The minimum T_{clk} improves with respect to the exact implementation, with the multiplier $m = 12$ that exhibits the best speed.

The FPMs with [22,42,43] show poorer performances, with area improvement limited to 47.4%, 50.2%, and 60%, respectively, and power saving up to 55.1%, 50.9% and 65%. In particular [22], $L = 2$, which offers best accuracy, exhibits limited improvements, with area and power reductions of 35.2% and 47.5%. The minimum T_{clk} worsens with [42,43], increasing up to 25% and 35%, respectively. On the contrary, the minimum T_{clk} improves up to 30% in the case of [22]. The work [17] shows best hardware performances with area reduction around −94% and power saving up to −95.7%. Moreover, the minimum T_{clk} exhibits best improvement (up to 70% with $t = 18$). These performances are due to the realization of the multiplication by means of a truncated adder in the logarithmic number system. However, by looking to the data of Table 2, we can conclude that these electrical features are achieved at the price of an accuracy loss.

3.3. Image Processing Applications

3.3.1. Image Filtering

We verify the validity of our proposal exploring the performances of the segmented FPMs in an image filtering application. The image filtering performs the following operation on the input image I

$$I_{filtered}(i,j) = \sum_{p=-d}^{d} \sum_{q=-d}^{d} I(i+p, j+q) \cdot h(i+d+1, j+d+1) \tag{37}$$

where h is the kernel matrix of the filter, with size $(2d+1) \times (2d+1)$.

In this example, we consider a smoothing application with gaussian kernel of size 5×5 and standard deviation 2, whose coefficients are floating-point numbers with values normalized to 1. The Matlab command *fspecial*('*gaussian*', 5, 2) allows us to obtain the filter mask. Then, the products in (37) are realized by means of our approximate FPMs and the state-of-the-art.

As a second example, we analyze the performances in edge detection. In this case, the gradient G of the original image is computed to highlight its edges. To this aim, the Sobel kernel h_{Sobel} and its transpose ${h^{\mathrm{T}}}_{Sobel}$ are used to compute the x and the y component of G (named G_{x} and G_{y} respectively). Then, the gradient G is computed as follows:

$$G = \sqrt{G_x^2 + G_y^2} \tag{38}$$

In our trial, we use the following Sobel kernel:

$$h_{Sobel} = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} \tag{39}$$

and used investigated FPMs to implement the multiplications in (38), (39). Table 4 collects the results in terms of structural similarity index measure (SSIM) and peak signal-to-noise-ratio (PSNR, in dB), obtained by filtering three images (Lena, Cameraman, and Lady). For each example, we average the SSIM and the PSNR obtained by processing the test images, as well as we indicate the overall average SSIM and PSNR as synthetic parameter in the last column of the table.

As shown in Table 4, the segmented FPMs achieve SSIM very close to 1 and PSNR up to 70 dB with gaussian filtering and produce the exact result with $m > 12$ in the edge detection. The correction technique allows us to improve the PSNR, with a maximum increment of 5.4 dB in the case $m = 14$, smoothing application.

Moreover, the implementations with [22,42,43] $L = 4, 6$ achieve good results, with SSIM very close to 1 and a PSNR values up to 63.1 dB in the gaussian case. Similarly, the PSNR overcomes 60 dB with [42,43] $k = 6$, and [22] $L = 4$ in the edge detection. The design [22] $L = 2$ offers best results on average, whereas the implementation [17] shows worst performances due to the stronger approximation, with a PSNR up to 37.6 dB on average (case $t = 14$).

Table 4. Accuracy of the proposed segmented FPMs and of the state-of-the-art in image filtering application.

Approximate FPM	Gaussian Filter		Edge Detector		Average	
	SSIM	PSNR [dB]	SSIM	PSNR [dB]	SSIM	PSNR [dB]
TOSAM [42], $h = 2$	1.000	61.7	1.000	62.4	1.000	62.1
TOSAM [42], $h = 3$	1.000	63.1	1.000	64.4	1.000	63.7
DRUM [43], $k = 4$	0.999	54.7	1.000	55.4	0.999	55.1
DRUM [43], $k = 6$	0.999	56.7	1.000	61.2	0.999	58.9
AFMB [17], $t = 14$	0.996	41.5	0.988	33.7	0.993	37.6
AFMB [17], $t = 16$	0.996	40.9	0.969	32.7	0.986	36.8
AFMB [17], $t = 18$	0.998	39.8	0.810	27.8	0.929	33.8
DATE17 [22], $L = 2$	1.000	91.3	1.000	∞	1.000	∞
DATE17 [22], $L = 4$	0.999	56.4	1.000	63.9	0.999	60.1
DATE17 [22], $L = 6$	0.998	47.3	0.999	50.4	0.999	48.9
SSFPM $m = 12$	0.999	55.5	1.000	71.9	0.999	63.7
cSSFPM $m = 12$	1.000	60.4	1.000	76.4	1.000	68.4
SSFPM $m = 14$	0.999	59.0	1.000	∞	1.000	∞
cSSFPM $m = 14$	1.000	64.4	1.000	∞	1.000	∞
SSFPM $m = 16$	1.000	62.6	1.000	∞	1.000	∞
cSSFPM $m = 16$	1.000	67.0	1.000	∞	1.000	∞
SSFPM $m = 18$	1.000	66.1	1.000	∞	1.000	∞
cSSFPM $m = 18$	1.000	70.0	1.000	∞	1.000	∞

Figure 9 shows the results of the edge detection using our SSFPMs with $m = 12, 18$ and the implementations from [17]. We report the negative of G for better highlighting the computed edges. The images obtained with the proposed multipliers are practically unchanged with respect to the exact one (as also expected from the high values of SSIM and PSNR). The results from [17] depend on the truncation t, with a sensible degradation of the detection in the background with $t = 18$ (again confirmed by the lower values of SSIM and PSNR).

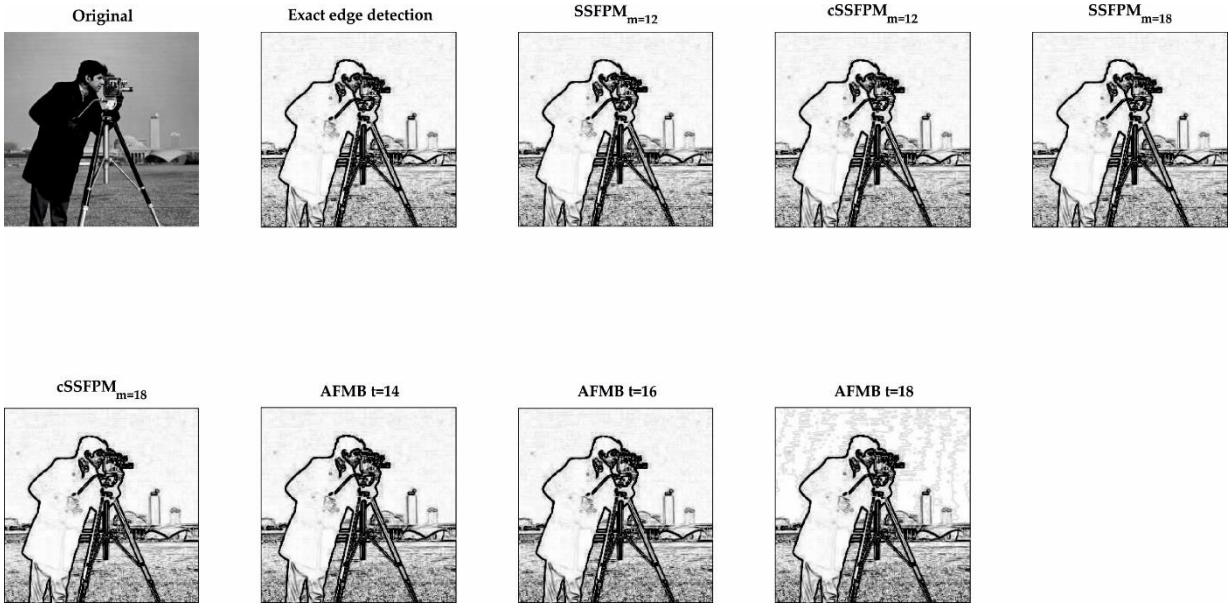


Figure 9. Results of the edge detection for the Cameraman image.

3.3.2. JPEG Compression

The JPEG compression leverages the limit of human senses to reduce the bit-volume of images. The compression algorithm exploits the discrete cosine transformation (DCT), applied to disjoined blocks of size 8×8 pixels, and performs a quantization to the transformed image with a variable resolution. The lower frequency components, more visible to human eyes, are approximated with a finer quantization step, whereas the high frequency component, less appreciable to human eyes, are approximated with a rougher quantization step.

In addition, a quality factor Q, defined in the range [0, 100], allows us to further modify the quantization accuracy and, as consequence, the compression, with $Q = 0$ that implies hardest compression and $Q = 100$ that implies lightest compression. Then, the quantized transformed image is reported in the original domain by means of the inverse discrete cosine transformation (iDCT). In the algorithm, the DCT and the iDCT require the multiplication between real numbers and are suitable to verify the validity of our proposal in a concrete scenario.

For the performances assessment, we approximate the DCT and the iDCT by using the proposed segmented FPMs and the state-of-the-art, considering the cases $Q = 40$, $Q = 70$, and $Q = 100$.

Table 5 collects the results, again expressed in terms of SSIM and PSNR, obtained by compressing three grayscale images: Lena, Cameraman, and Peppers (SSIM and PSNR are computed relative to image compression performed with exact multiplier). In this case, we also report the mean SSIM and the mean PSNR for each Q, obtained by processing the three images, and indicate the overall average SSIM and PSNR in the last column of the table. Figure 10 reports Peppers compressed images in case of segmented multipliers, $Q = 40$.

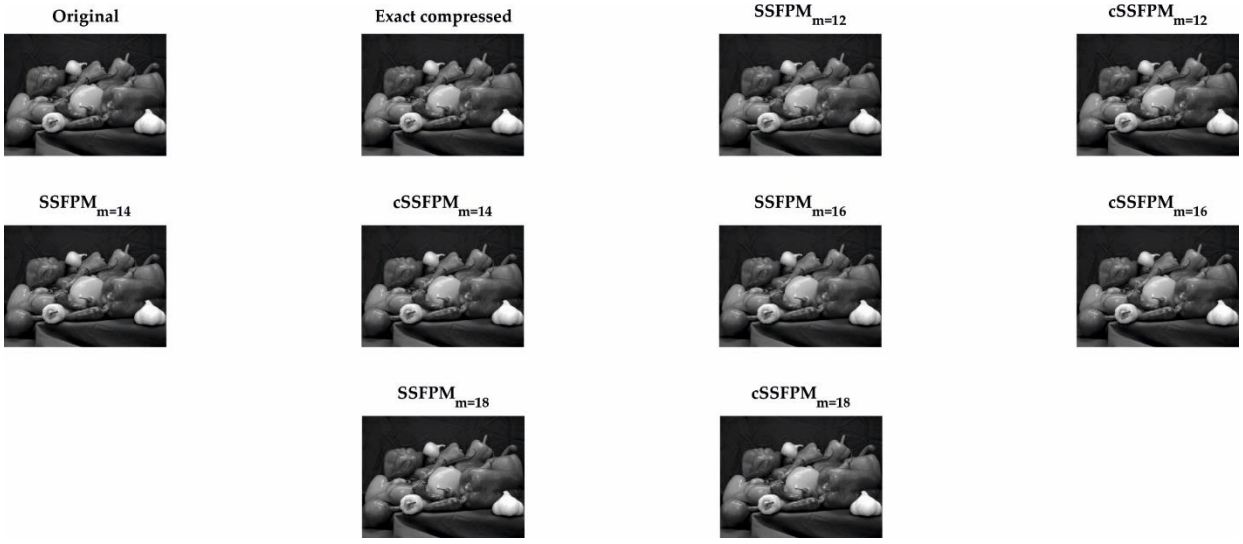


Figure 10. JPEG compressed images with multiplications realized by means of proposed SSFPMs. The quality factor is $Q = 40$.

As observable, our segmented multipliers ensure again a SSIM very close to 1 in all the cases, and a PSNR that ranges between 47 dB and 63 dB on average. Increasing m allows us to improve the quality of results, while the correction technique leads to a remarkable PSNR increment especially for small values of m (up to +8 dB in the case $m = 12$, $Q = 100$). In addition, performances are almost constant with respect to the quality factor Q. The designs with [22,42,43] $L = 4, 6$ exhibit lower accuracy, with the average PSNR limited to 53 dB, whereas [22] $L = 2$ allows best compression with PSNR of 71.4 dB on average.

Table 5. Performances of the proposed SSFPM, cSSFPM and the state-of-the-art in the JPEG compression.

Approximate FPM	Q = 40		Q = 70		Q = 100		Average	
	SSIM	PSNR [dB]	SSIM	PSNR [dB]	SSIM	PSNR [dB]	SSIM	PSNR [dB]
TOSAM [42], h = 2	0.997	48.9	0.997	49.5	0.999	53.9	0.998	50.8
TOSAM [42], h = 3	0.998	50.8	0.998	51.5	0.999	56.1	0.999	52.8
DRUM [43], k = 4	0.992	44.8	0.993	45.6	0.997	49.2	0.994	46.5
DRUM [43], k = 6	0.998	50.0	0.998	50.9	0.999	56.1	0.998	52.3
AFMB [17], t = 14	0.973	34.5	0.980	35.0	0.986	35.7	0.980	35.1
AFMB [17], t = 16	0.972	34.4	0.979	34.9	0.985	35.6	0.979	34.9
AFMB [17], t = 18	0.960	32.7	0.968	33.0	0.974	33.6	0.967	33.1
DATE17 [22], L = 2	1.000	63.9	1.000	75.5	1.000	74.7	1.000	71.4
DATE17 [22], L = 4	0.993	43.9	0.995	44.8	0.998	47.6	0.995	45.4
DATE17 [22], L = 6	0.970	34.2	0.975	34.6	0.982	35.5	0.976	34.7
SSFPM m = 12	0.995	45.6	0.997	46.9	0.999	49.7	0.997	47.4
cSSFPM m = 12	0.999	52.9	0.999	53.0	1.000	57.8	0.999	54.6
SSFPM m = 14	0.999	51.7	0.999	52.4	1.000	56.9	0.999	53.7
cSSFPM m = 14	0.999	55.4	0.999	56.7	1.000	60.2	1.000	57.4
SSFPM m = 16	0.999	54.9	0.999	56.3	1.000	60.1	1.000	57.1
cSSFPM m = 16	1.000	58.1	1.000	59.3	1.000	62.5	1.000	60.0
SSFPM m = 18	1.000	58.3	1.000	59.4	1.000	62.6	1.000	60.1
cSSFPM m = 18	1.000	62.0	1.000	62.9	1.000	64.8	1.000	63.2

3.3.3. Tone Mapping of HDR Images

As a last example, we employ the investigated multipliers for a tone mapping application on HDR images. An HDR image exploits floating-point pixels to represent a high dynamic range of luminance. A mapping operation is needed to properly adapt the high dynamic range of luminance to a lower range of values, whenever requested by the application. The algorithm devised in [44] exploits the following formula to perform tone mapping:

$$L_{mapped}(i,j) = \frac{L(i,j)}{1 + L(i,j)} \tag{40}$$

The $L(i,j)$ is a pixel of the luminance matrix of the image, obtained by executing the following steps:

$$\begin{aligned} L_{tmp}(i,j) &= 0.27 \cdot R(i,j) + 0.67 \cdot G(i,j) + 0.06 \cdot B(i,j) \\ L_m &= \exp\left(\tfrac{1}{N}\textstyle\sum \log(L_{tmp}(i,j))\right) \\ L(i,j) &= \tfrac{\beta}{L_m} \cdot L_{tmp}(i,j) \end{aligned} \tag{41}$$

where R, G, B are the three channels of the input HDR image, N is the number of pixels in a channel, L_{m} is the geometric mean of L_{tmp}, and β is a value in the range [0, 1].

Applying (40) allows us to properly scale the luminance since large values of $L(i,j)$ are normalized to 1, whereas small values of $L(i,j)$ are practically unmodified. Then, the channels of the original image are weighted for $L(i,j)$ as follows:

$$\begin{aligned} R_{out}(i,j) &= [L(i,j) \cdot R(i,j)]/L_m \\ G_{out}(i,j) &= [L(i,j) \cdot G(i,j)]/L_m \\ B_{out}(i,j) &= [L(i,j) \cdot B(i,j)]/L_m \end{aligned} \tag{42}$$

and quantized to integer values in the range [0, 255].

As in the previous demonstrations, the approximate tone mapping is obtained using the approximate FPMs in the multiplications in (41) and (42). In our trials, we pose β = 0.5.

Table 6 collects the results of the comparison between the exact the approximate algorithm, again expressed in terms of SSIM and PSNR, with the processed HDR images

that are Oxford Church, Office and Bottles_small. Figure 11 depicts the result for the Bottles_small image.

Table 6. Results for tone mapping of HDR images.

Approximate FPM	Bottle_Small		Oxford Church		Office		Average	
	SSIM	PSNR [dB]	SSIM	PSNR [dB]	SSIM	PSNR [dB]	SSIM	PSNR [dB]
TOSAM [42], $h = 2$	1.000	51.4	0.999	50.4	0.999	48.4	0.999	50.1
TOSAM [42], $h = 3$	1.000	53.5	0.999	55.4	1.000	55.4	1.000	54.8
DRUM [43], $k = 4$	0.999	43.9	0.996	43.5	0.998	42.1	0.998	43.1
DRUM [43], $k = 6$	1.000	55.3	0.999	55.5	1.000	54.3	1.000	55.0
AFMB [17], $t = 14$	1.000	33.6	0.962	26.2	0.988	35.3	0.981	31.7
AFMB [17], $t = 16$	1.000	34.4	0.962	26.3	0.988	35.4	0.982	32.1
AFMB [17], $t = 18$	1.000	33.3	0.949	26.5	0.976	30.2	0.972	30.0
DATE17 [22], $L = 2$	1.000	73.6	1.000	72.7	1.000	72.7	1.000	73.0
DATE17 [22], $L = 4$	1.000	45.1	0.998	45.9	0.999	45.5	0.999	45.5
DATE17 [22], $L = 6$	0.995	33.0	0.981	33.0	0.987	31.9	0.988	32.6
SSFPM $m = 12$	1.000	47.5	0.999	45.2	0.999	46.0	0.999	46.2
cSSFPM $m = 12$	1.000	56.1	0.999	54.2	0.999	51.3	1.000	53.9
SSFPM $m = 14$	1.000	52.4	0.999	51.4	0.999	52.0	0.999	52.0
cSSFPM $m = 14$	1.000	58.1	1.000	60.7	1.000	56.9	1.000	58.6
SSFPM $m = 16$	1.000	54.8	1.000	57.2	1.000	54.6	1.000	55.6
cSSFPM $m = 16$	1.000	60.6	1.000	60.5	1.000	60.1	1.000	60.4
SSFPM $m = 18$	1.000	59.3	1.000	60.2	1.000	57.8	1.000	59.1
cSSFPM $m = 18$	1.000	67.1	1.000	62.5	1.000	62.9	1.000	64.2

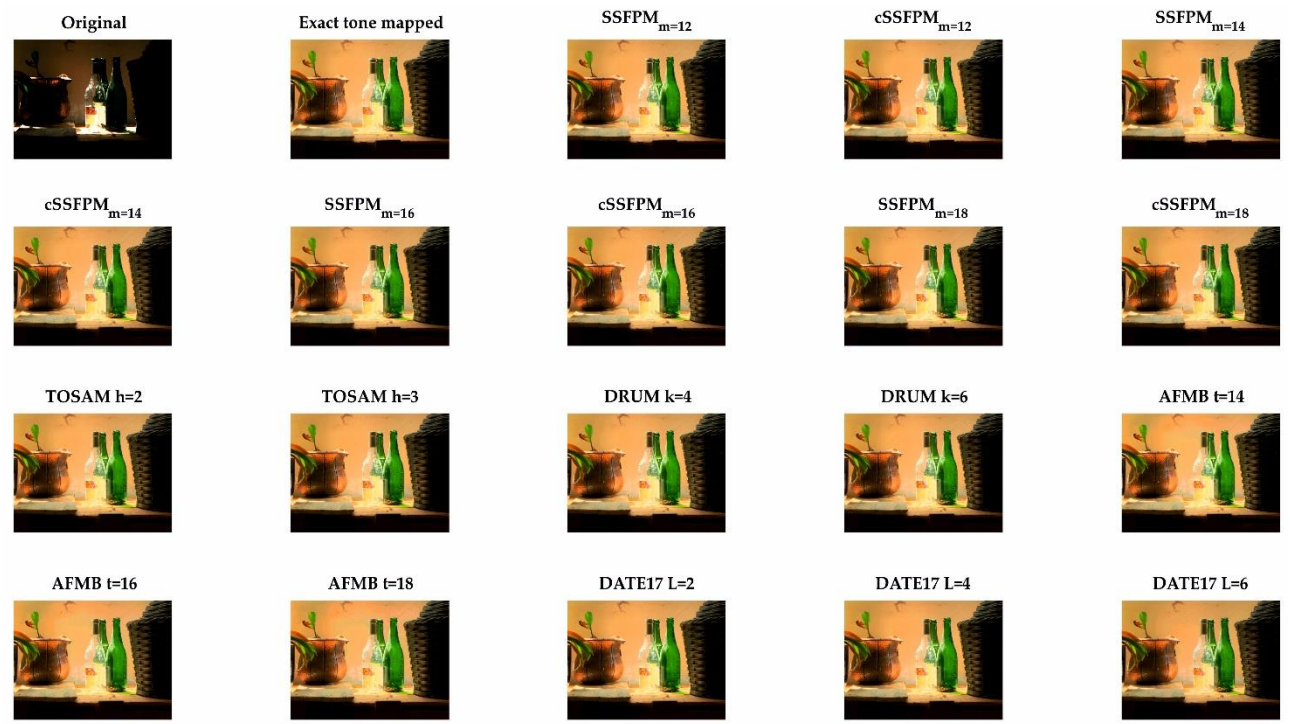


Figure 11. Example of tone mapping with the proposed segmented multipliers and the state-of-the-art.

In this case, the SSIM achieved with the segmented multipliers is also very close to 1, and the PSNR ranges between 46 dB and 64 dB on average. The results are comparable with [42,43], whereas the implementation [17,22] $L = 6$ exhibits lower performances (with PSNR up to 32.6 dB on average).

4. Discussion

The static segmentation applied to the MAA operation of (5) allows us to reduce power, delay, and area of the FPM while preserving remarkable accuracy performances. This is mainly due to the reduction in the input bit-width in the MAA unit. In addition, the proposed shift-and-truncate technique allows us to realize a fused PPM for the MAA unit with a unique CPA, with beneficial effects on the hardware performances.

At the same time, the SSFPM exhibits very good accuracy since (i) the approximation is applied only to the mantissa computation and (ii) the employed approximation, based on static segmentation of the fixed-point multiplier needed in mantissa computation, provides a small relative error. Indeed, the SSM approach introduces an error only when large values are represented, whereas small values are not approximated. This leads to good error performances that are suitable for the implementation of a floating-point multiplier.

The multipliers that exploit [42,43] benefit from the dynamic segmentation to approximate the product $Ma \cdot Mb$ in (5). These solutions achieve satisfactory error performances, as also demonstrated in the image processing applications. The error metrics are comparable with the SSFPM, as well as the SSIM and PSNR values demonstrate the high capabilities of these solutions. On the other hand, shifters are required between the multiplier and the adder of the MAA, thus implying the usage of two CPAs. This leads to a worsening of the hardware performances with respect to the SSFPM, as demonstrated by the lower power and area savings, limited to 55% and 50%, respectively. Furthermore, the minimum T_{clk} is larger due to the shifters placed between the multiplier and the adder of the MAA unit.

The FPMs with the approximation of [22] exhibit performances that strongly depend on L, with the accuracy that worsens as L increases. Power and area reductions are limited in the case $L = 2$ (35.2% and 47.5%, respectively), whereas an improvement is registered with $L = 4, 6$ (up to 59.7% and 65.0%, respectively) at the cost of precision.

The proposal of [17] exhibits best hardware performances, with area and power saving that overcome 90% in both cases. These improvements are due to the usage of an adder in place of a multiplier for the realization of the mantissa product. In addition, leading one detectors and barrel shifters are not used in this case since the position of the implicit bit is always known. In any way, approximating the product with a logarithmic sum leads to a larger error, due to logarithm approximation. In addition, the adder is also truncated for further optimization, with a consequent accuracy loss.

As part of a joint assessment between hardware performances and accuracy, we show the power and the area saving versus the *NMED* and *MRED* of each FPM in Figure 12. The black dotted line indicates the Pareto Front. For $NMED < 10^{-5}$ the proposed SSFPMs overcomes [22,42,43] with $L = 4, 6$, exhibiting a power saving greater than 70% and an area improvement larger than 60%. The cSSFPMs also define the Pareto front in that region of the graph. Similar observations also apply to the case $MRED < 10^{-2}$. The figures show again that [17] performs better from a hardware point of view, but at the cost of a loss of accuracy ($NMED > 8 \times 10^{-5}$ and $MRED > 2 \times 10^{-2}$). Similarly, [22] $L = 2$ has the best accuracy, but at the cost of a degradation of power and area.

Therefore, we can conclude that our proposal offers the best trade-off between hardware improvement and accuracy of results.

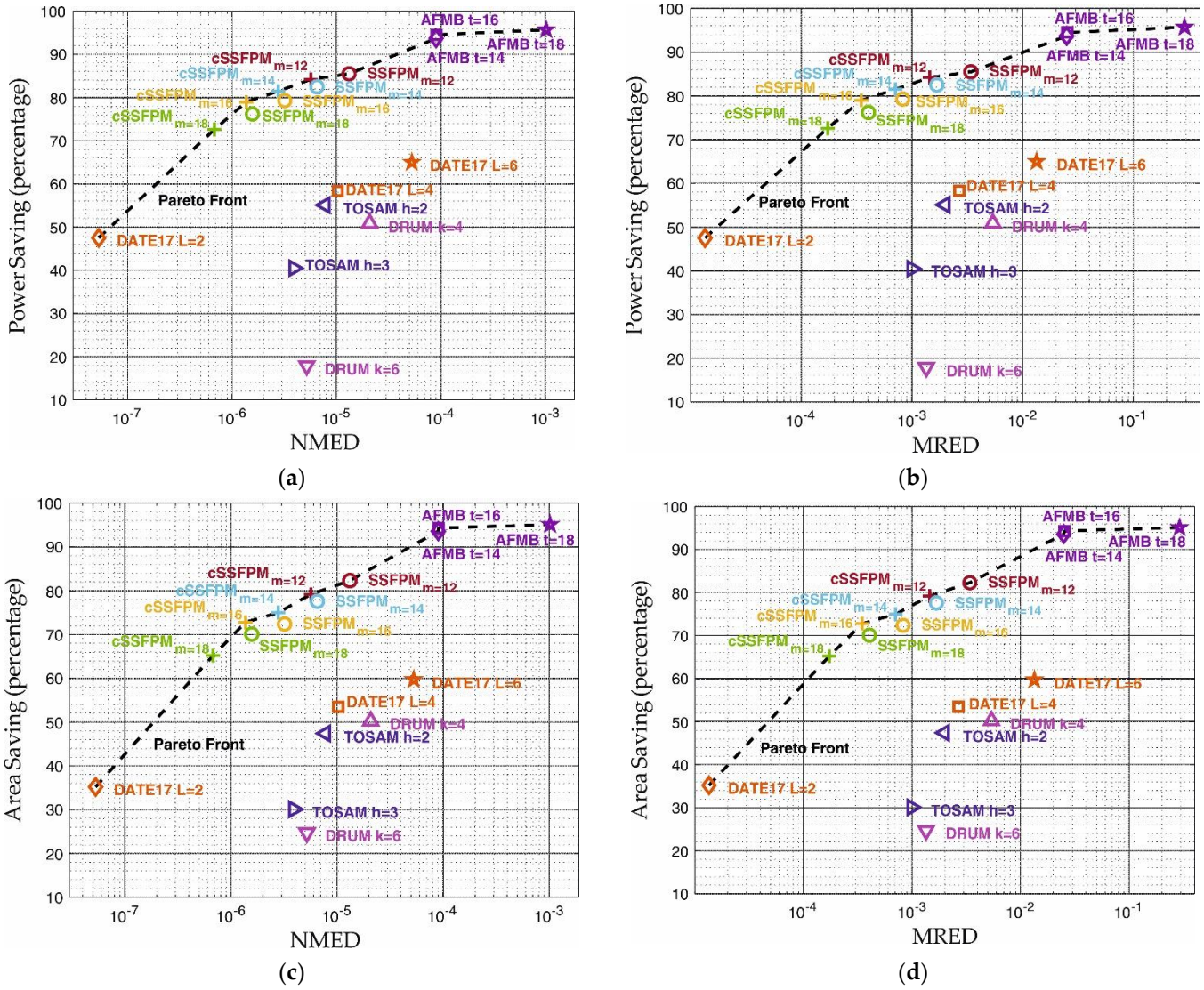


Figure 12. Power saving versus (**a**) *NMED* and (**b**) *MRED*, and area saving versus (**c**) *NMED* and (**d**) *MRED*.

5. Conclusions

In this paper we propose a novel low-power approximate floating-point multiplier, based on static segmentation. To optimize hardware performances, the mantissa product is first revisited as a multiply-and-add operation. In this way the implicit bit is excluded from the computation to reduce the complexity of the multiplier, and additive logic is introduced to recover the exact result.

Then, a segmentation scheme is applied to the mantissas, to reduce the size of the multiplier. The proposed technique leverages a segment-and-truncate approach to eliminate the left-shift operation at the output of the multiplier. In this way, we can realize the mantissa multiplier by means of a fused partial product matrix and a unique carry-propagate adder, with beneficial effects on the hardware performances. In addition, a correction term is introduced, to reduce the approximation error due to the segmentation. The accuracy of the circuit can be accurately tailored at the design time, by acting on a single parameter, m.

Analysis of error metrics show that proposed floating-point multiplier is competitive with the state-of-the-art for values of m in the range 12–18 (in the considered case of single-precision floating-point format). Syntheses in 28 nm CMOS reveal a remarkable reduction

in the power consumption and area, with best results achieved with $m = 12$ (up to 85% of power saving and up to 82% of area reduction compared to exact floating-point multiplier).

Implementations of several image processing algorithms (JPEG compression, image filtering, tone mapping of HDR images) show the effectiveness of the proposed architecture in real applications.

By a joint analysis of electrical performances and error metrics, the proposed approximate floating-point multiplier overcomes the state-of-the-art, exhibiting the best trade-off between hardware improvements and quality of results.

Author Contributions: Conceptualization, G.D.M., G.S. and A.G.M.S.; methodology, G.D.M. and G.S.; software, G.D.M. and G.S.; validation, G.D.M., G.S. and A.G.M.S.; formal analysis, G.D.M., G.S. and A.G.M.S.; investigation, G.D.M., G.S. and A.G.M.S.; data curation, G.D.M. and G.S.; writing—original draft preparation, G.D.M., G.S., A.G.M.S. and D.D.C.; writing—review and editing, A.G.M.S., D.D.C. and N.P.; visualization, G.D.M. and G.S.; supervision, A.G.M.S., D.D.C. and N.P.; project administration, A.G.M.S. and D.D.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The Verilog code is available on GitHub at https://github.com/GenDiMeo/SSFPM (accessed on 18 September 2022).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Montanari, D.; Castellano, G.; Kargaran, E.; Pini, G.; Tijani, S.; De Caro, D.; Strollo, A.G.M.; Manstretta, D.; Castello, R. An FDD Wireless Diversity Receiver with Transmitter Leakage Cancellation in Transmit and Receive Bands. *IEEE J. Solid-State Circuits* **2018**, *53*, 1945–1959. [CrossRef]
2. Kurzo, Y.; Kristensen, A.T.; Burg, A.; Balatsoukas-Stimming, A. Hardware Implementation of Neural Self-Interference Cancellation. *IEEE J. Emerg. Sel. Top. Circuits Syst.* **2020**, *10*, 204–216. [CrossRef]
3. Mookherjee, S.; DeBrunner, L.; DeBrunner, V. A low power radix-2 FFT accelerator for FPGA. In Proceedings of the 2015 49th Asilomar Conference on Signals, Systems and Computers, Pacific Grove, CA, USA, 8–11 November 2015; pp. 447–451. [CrossRef]
4. Wei, X.; Du, G.-M.; Wang, X.; Cao, H.; Hu, S.; Zhang, D.; Li, Z. FPGA Implementation of Hardware Accelerator for Real-time Video Image Edge Detection. In Proceedings of the 2021 IEEE 15th International Conference on Anti-counterfeiting, Security, and Identification (ASID), Xiamen, China, 29–31 October 2021; pp. 16–20. [CrossRef]
5. Xie, G.; Wang, C. An All-Digital PLL for Video Pixel Clock Regeneration Applications. In Proceedings of the 2009 WRI World Congress on Computer Science and Information Engineering, Los Angeles, CA, USA, 31 March–2 April 2009; pp. 392–396. [CrossRef]
6. De Caro, D.; Tessitore, F.; Vai, G.; Imperato, N.; Petra, N.; Napoli, E.; Parrella, C.; Strollo, A.G.M. A 3.3 GHz Spread-Spectrum Clock Generator Supporting Discontinuous Frequency Modulations in 28 nm CMOS. *IEEE J. Solid-State Circuits* **2015**, *50*, 2074–2089. [CrossRef]
7. De Caro, D.; Di Meo, G.; Napoli, E.; Petra, N.; Strollo, A.G.M. A 1.45 GHz All-Digital Spread Spectrum Clock Generator in 65nm CMOS for Synchronization-Free SoC Applications. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2020**, *67*, 3839–3852. [CrossRef]
8. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*; Microprocessor Standards Committee, IEEE Standard for Floating-Point Arithmetic. IEEE Computer Society: New York, NY, USA, 2019; pp. 1–84. [CrossRef]
9. Tong, J.; Nagle, D.; Rutenbar, R. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2000**, *8*, 273–286. [CrossRef]
10. Han, J.; Orshansky, M. Approximate computing: An emerging paradigm for energy-efficient design. In Proceedings of the 2013 18th IEEE European Test Symposium (ETS), Avignon, France, 27–30 May 2013; pp. 1–6. [CrossRef]
11. Chippa, V.K.; Chakradhar, S.T.; Roy, K.; Raghunathan, A. Analysis and characterization of inherent application resilience for approximate computing. In Proceedings of the 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, USA, 29 May–7 June 2013; pp. 1–9. [CrossRef]
12. Chen, C.; Qian, W.; Imani, M.; Yin, X.; Zhuo, C. PAM: A Piecewise-Linearly-Approximated Floating-Point Multiplier with Unbiasedness and Configurability. *IEEE Trans. Comput.* **2021**, *71*, 2473–2486. [CrossRef]
13. Camus, V.; Schlachter, J.; Enz, C.; Gautschi, M.; Gurkaynak, F.K. Approximate 32-bit floating-point unit design with 53% power-area product reduction. In Proceedings of the ESSCIRC Conference 2016: 42nd European Solid-State Circuits Conference, Lausanne, Switzerland, 12–15 September 2016; pp. 465–468. [CrossRef]
14. Imani, M.; Peroni, D.; Rosing, T. CFPU: Configurable floating point multiplier for energy-efficient computing. In Proceedings of the 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, USA, 18–22 June 2017; pp. 1–6. [CrossRef]

15. Imani, M.; Garcia, R.; Gupta, S.; Rosing, T. RMAC: Runtime Configurable Floating Point Multiplier for Approximate Computing. In Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED' 18), Seattle, WA, USA, 23–25 July 2018; Association for Computing Machinery: New York, NY, USA, 2018; pp. 1–6. [CrossRef]
16. Zhang, H.; Ko, S.-B. Variable-Precision Approximate Floating-Point Multiplier for Efficient Deep Learning Computation. *IEEE Trans. Circuits Syst. II Express Briefs* **2022**, *69*, 2503–2507. [CrossRef]
17. Saadat, H.; Bokhari, H.; Parameswaran, S. Minimally Biased Multipliers for Approximate Integer and Floating-Point Multiplication. *IEEE Trans. Comput. Des. Integr. Circuits Syst.* **2018**, *37*, 2623–2635. [CrossRef]
18. Petra, N.; De Caro, D.; Garofalo, V.; Napoli, E.; Strollo, A.G.M. Design of Fixed-Width Multipliers With Linear Compensation Function. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2010**, *58*, 947–960. [CrossRef]
19. De Caro, D.; Petra, N.; Strollo, A.G.M.; Tessitore, F.; Napoli, E. Fixed-Width Multipliers and Multipliers-Accumulators With Min-Max Approximation Error. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2013**, *60*, 2375–2388. [CrossRef]
20. Zervakis, G.; Tsoumanis, K.; Xydis, S.; Soudris, D.; Pekmestzi, K. Design-Efficient Approximate Multiplication Circuits Through Partial Product Perforation. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2016**, *24*, 3105–3117. [CrossRef]
21. Leon, V.; Zervakis, G.; Xydis, S.; Soudris, D.; Pekmestzi, K. Walking through the Energy-Error Pareto Frontier of Approximate Multipliers. *IEEE Micro* **2018**, *38*, 40–49. [CrossRef]
22. Qiqieh, I.; Shafik, R.; Tarawneh, G.; Sokolov, D.; Yakovlev, A. Energy-efficient approximate multiplier design using bit significance-driven logic compression. In Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), Lausanne, Switzerland, 27–31 March 2017; Volume 2017, pp. 7–12. [CrossRef]
23. Esposito, D.; Strollo, A.G.M.; Alioto, M. Low-power approximate MAC unit. In Proceedings of the 2017 13th Conference on Ph.D. Research in Microelectronics and Electronics (PRIME), Taormina, Italy, 12–15 June 2017; pp. 81–84. [CrossRef]
24. Esposito, D.; Strollo, A.G.M.; Napoli, E.; De Caro, D.; Petra, N. Approximate Multipliers Based on New Approximate Compressors. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2018**, *65*, 4169–4182. [CrossRef]
25. Yang, Z.; Han, J.; Lombardi, F. Approximate compressors for error-resilient multiplier design. In Proceedings of the 2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS), Amherst, MA, USA, 12–14 October 2015; pp. 183–186. [CrossRef]
26. Akbari, O.; Kamal, M.; Afzali-Kusha, A.; Pedram, M. Dual-Quality 4:2 Compressors for Utilizing in Dynamic Accuracy Configurable Multipliers. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2017**, *25*, 1352–1361. [CrossRef]
27. Ahmadinejad, M.; Moaiyeri, M.H.; Sabetzadeh, F. Energy and area efficient imprecise compressors for approximate multiplication at nanoscale. *AEU-Int. J. Electron. Commun.* **2019**, *110*, 152859. [CrossRef]
28. Strollo, A.G.M.; Napoli, E.; De Caro, D.; Petra, N.; Di Meo, G. Comparison and Extension of Approximate 4-2 Compressors for Low-Power Approximate Multipliers. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2020**, *67*, 3021–3034. [CrossRef]
29. Strollo, A.G.M.; De Caro, D.; Napoli, E.; Petra, N.; Di Meo, G. Low-Power Approximate Multiplier with Error Recovery using a New Approximate 4-2 Compressor. In Proceedings of the 2020 IEEE International Symposium on Circuits and Systems (ISCAS), Seville, Spain, 12–14 October 2020; pp. 1–4. [CrossRef]
30. Kulkarni, P.; Gupta, P.; Ercegovac, M. Trading Accuracy for Power with an Underdesigned Multiplier Architecture. In Proceedings of the 2011 24th Internatioal Conference on VLSI Design, Chennai, India, 2–7 January 2011; pp. 346–351. [CrossRef]
31. Gillani, G.A.; Hanif, M.A.; Verstoep, B.; Gerez, S.H.; Shafique, M.; Kokkeler, A.B.J. MACISH: Designing Approximate MAC Accelerators with Internal-Self-Healing. *IEEE Access* **2019**, *7*, 77142–77160. [CrossRef]
32. Ansari, M.S.; Jiang, H.; Cockburn, B.F.; Han, J. Low-Power Approximate Multipliers Using Encoded Partial Products and Approximate Compressors. *IEEE J. Emerg. Sel. Top. Circuits Syst.* **2018**, *8*, 404–416. [CrossRef]
33. Waris, H.; Wang, C.; Liu, W.; Han, J.; Lombardi, F. Hybrid Partial Product-Based High-Performance Approximate Recursive Multipliers. *IEEE Trans. Emerg. Top. Comput.* **2020**, *10*, 507–513. [CrossRef]
34. Nunziata, I.; Zacharelos, E.; Saggese, G.; Strollo, A.M.G.; Napoli, E. Approximate Recursive Multipliers Using Carry Truncation and Error Compensation. In Proceedings of the 2022 17th Conference on Ph.D. Research in Microelectronics and Electronics (PRIME), Villasimius, Italy, 12–15 June 2022; pp. 137–140. [CrossRef]
35. Zacharelos, E.; Nunziata, I.; Saggese, G.; Strollo, A.G.; Napoli, E. Approximate Recursive Multipliers Using Low Power Building Blocks. *IEEE Trans. Emerg. Top. Comput.* **2022**, *10*, 1315–1330. [CrossRef]
36. Kim, M.S.; Del Barrio, A.A.; Hermida, R.; Bagherzadeh, N. Low-power implementation of Mitchell's approximate logarithmic multiplication for convolutional neural networks. In Proceedings of the 2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC), Jeju, Korea, 22–25 January 2018; pp. 617–622. [CrossRef]
37. Liu, W.; Xu, J.; Wang, D.; Wang, C.; Montuschi, P.; Lombardi, F. Design and Evaluation of Approximate Logarithmic Multipliers for Low Power Error-Tolerant Applications. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2018**, *65*, 2856–2868. [CrossRef]
38. Kim, M.S.; Del Barrio, A.A.; Oliveira, L.T.; Hermida, R.; Bagherzadeh, N. Efficient Mitchell's Approximate Log Multipliers for Convolutional Neural Networks. *IEEE Trans. Comput.* **2018**, *68*, 660–675. [CrossRef]
39. Ansari, M.S.; Cockburn, B.F.; Han, J. An Improved Logarithmic Multiplier for Energy-Efficient Neural Computing. *IEEE Trans. Comput.* **2021**, *70*, 614–625. [CrossRef]
40. Narayanamoorthy, S.; Moghaddam, H.A.; Liu, Z.; Park, T.; Kim, N.S. Energy-Efficient Approximate Multiplication for Digital Signal Processing and Classification Applications. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2014**, *23*, 1180–1184. [CrossRef]

41. Strollo, A.G.M.; Napoli, E.; De Caro, D.; Petra, N.; Saggese, G.; Di Meo, G. Approximate Multipliers Using Static Segmentation: Error Analysis and Improvements. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2022**, *69*, 2449–2462. [CrossRef]
42. Vahdat, S.; Kamal, M.; Afzali-Kusha, A.; Pedram, M. TOSAM: An Energy-Efficient Truncation- and Rounding-Based Scalable Approximate Multiplier. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2019**, *27*, 1161–1173. [CrossRef]
43. Hashemi, S.; Bahar, R.I.; Reda, S. DRUM: A Dynamic Range Unbiased Multiplier for approximate applications. In Proceedings of the 2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Austin, TX, USA, 2–6 November 2015; pp. 418–425. [CrossRef]
44. Kinoshita, Y.; Shiota, S.; Kiya, H. Fast inverse tone mapping with Reinhard's global operator. In Proceedings of the 2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), New Orleans, LA, USA, 5–9 March 2017; pp. 1972–1976. [CrossRef]
45. GitHub. Available online: https://github.com/astrollo/SSM (accessed on 16 February 2020).
46. GitHub. Available online: https://github.com/scale-lab/DRUM (accessed on 18 April 2020).