Tunable Floating-Point Adder

Nannarelli, Alberto

Published in:
IEEE Transactions on Computers

Link to article, DOI:
10.1109/TC.2019.2906907

Publication date:
2019

Document Version
Peer reviewed version

Citation (APA):
Tunable Floating-Point Adder

Alberto Nannarelli, Senior Member, IEEE

Abstract—In this work, we address the design of an adder in Tunable Floating-Point (TFP) precision. TFP is a variable precision format in which a given precision can be chosen for a single operation by selecting a specific number of bits for significand and exponent in the floating-point representation. By tuning the precision of an algorithm to the minimum precision achieving an acceptable target error, we can make the computation more power efficient. In previous work, we introduced a unit for TFP multiplication. Here, we focus on floating-point addition.

Index Terms—Floating-point, addition, IEEE rounding, energy efficiency.

1 INTRODUCTION

Deep Learning (DL) is currently one of the hottest application fields for arithmetic processors. DL algorithms execute a huge number of operations, predominantly multiplications and additions, and resort to dedicated hardware, GPUs [1] or FPGAs [2], to accelerate the execution.

Due to the size of the datasets in DL, the processing time and the energy necessary is very large [3]. To increase the power efficiency, the computation is migrated from double-precision (binary64 in IEEE 754-2008 standard [4]) to single (binary32) and half (binary16) precision. In addition to the IEEE formats, several other formats have been introduced, such as the Google’s Brain-FP format [5], or the Intel’s “Flexpoint” format [6].

Although floating-point operations are more complex to implement in hardware than fixed-point, the advantage is that the precision scaling is handled by the unit and it is transparent to the programmer (no operand shifting or re-scaling).

In [7], we introduced the Tunable Floating-Point (TFP), a format in which the precision of operands and results can be chosen for a single operation by selecting a specific number of bits for the significand and the exponent in the floating-point representation. By tuning the precision of a given algorithm to the minimum precision achieving an acceptable target error, we can make the computation more power efficient.

In [7], we described the implementation of a TFP multiplier. In this work, we present a Tunable Floating-Point adder (TFP-add) to complete, along with the multiplier, a TFP unit which can be used as part of on-chip accelerators. The TFP unit can handle significand precision from 24 to 4 bits, and exponent from 8 to 5 bits. The maximum precision \( m = 24, e = 8 \) is that of binary32 (single-precision), and the tunable range includes binary16 \( m = 11, e = 5 \) and Google’s Brain-FP format \( m = 8, e = 8 \).

The contributions of this paper are: 1) The revisitation of the “double-path” adder [8] implementing the baseline binary32 adder. 2) The design of the TFP adder providing correct rounding when reducing the precision of the significands – the main contribution. 3) An example showing the benefits of the Tunable Floating-Point for a simple neural network used to approximate a generic function and the error-energy trade-offs.

2 TUNABLE FLOATING-POINT

The floating-point representation of a real number \( x \) is

\[
x = (-1)^{S_x} \cdot M_x \cdot b^{E_x} \quad x \in \mathbb{R}
\]

where \( S_x \) is the sign, \( M_x \) is the significand or mantissa (represented by \( m \) bits), \( b \) is the base (\( b = 2 \) in the following), and \( E_x \) is the exponent (represented by \( e \) bits). The representation in the IEEE 745-2008 standard [4] has significand normalized \( 1.0 \leq M_x < 2.0 \) and biased exponent: \( \text{bias} = 2^{e-1} - 1 \).

We introduced the Tunable Floating-Point (TFP) representation in [7]. In TFP, we only consider dynamic ranges from and below the binary32 representation. We support significand’s bit-width \( m \) from 24 to 4 and exponent’s bit-width \( e \) from 8 to 5.

We assume that the TFP representation is normalized to have the conversions compatible with the IEEE 754-2008 standard. Therefore, the implicit (integer) bit is not stored. Subnormals support is quite expensive and, therefore, we opted to flush-to-zero TFP numbers when the exponent is less than the minimum representable in \( e \) bits. TFP supports three types of rounding:

- **RTZ** Round-to-zero (truncation);
- **RTN** Round-to-the-nearest where half \( \text{ulp} \) (unit in last position) is always added before the rounding;
- **RTNE** Round-to-the-nearest-even (on a tie) which is the default mode \( \text{roundToEven} \) in IEEE 754.

For the rounding, we use the terminology of [8] summarized for reference in Fig. 1.

We also developed a TFP simulator [7] consisting in a library of \( C \) functions and implementing TFP operations. Each operation is implemented with a standard FP algorithm by limiting the computation of the significand bits to \( m \) and applying the specified rounding mode. The simulator
provides the error double-precision vs. TFP in key points of the algorithm and generates test patterns for the hardware verification.

3 Baseline FP Adder

The algorithm for floating-point addition can be summarized in the following steps [8]:

1) Subtract exponents $D = E_X - E_Y$.
2) Align significands $M_X$, $M_Y$ by shifting right $D$ positions the significand with the smallest exponent.
3) Add, or subtract, the significands and determine the sign of the result.
4) Normalize and round the result.

Among the different implementations of the FP addition algorithm, we select as the baseline FP adder a “double-path” architecture derived from the schemes of [8] and [9]. The architecture is sketched for the significand part in Fig. 2 for binary32. The significands $M_X$, $M_Y$ and $M_Z$ include the integer bit (“1” if the biased exponent is larger than zero). The exponent difference ($D$) is performed at the beginning of the FP operation to determine the alignment of the two operands and to set which path will produce the correct result.

The close path is taken when the effective operation is subtraction ($EOP=1$) and the exponent difference is one ($D=1$) or zero ($D=0$). Therefore, the select signal $CLS$ in Fig. 2 is set as:

$$CLS = EOP \text{ AND } ((D=1) \text{ OR } (D=0))$$

The far path is taken for all additions and subtractions when $D > 1$ ($CLS=0$).

The main advantage of the double-path implementation is that only one variable shifter is in the critical path so that the FP-addition latency is reduced. The unit can be pipelined in two stages as suggested in [8]. In the following, we explain how this is done for the baseline unit. The architectures in the close and far paths are shown in Fig. 3 and Fig. 4. With respect to the implementation of [8], the unit to swap the operands is not shared but moved in the far path to reduce the latency in the close path.

3.1 Close Path

The close path is taken when the effective operation is subtraction ($EOP=1$) and the exponent difference is one ($D=1$) or zero ($D=0$). The architecture of the close path is shown in Fig. 3. We explain next how it works.

Since in the close path the exponent difference is one or zero, operands alignment can be done by shifting one position to the right the significand of the number with the smallest exponent when $D=1$. This alignment is done by the two 2:1 multiplexers “MUX” in Fig. 3. The control signals are set as:

$$\text{SHIFT } X = \text{sign}(D) \text{ AND } (D=1)$$
$$\text{SHIFT } Y = \text{sign}(D) \text{ AND } (D=1)$$

Determining the sign of the result before the actual subtraction is expensive because it requires to compare the two significands when the exponent is the same ($D=0$) to decide which operand to invert.

An alternative is to complement a fixed operand, for example $M_Y$, and then complement the subtraction result if negative. Since two’s complementation requires a full precision addition, we opted for a one’s complement adder, or carry-around adder [8], followed by conditional bit complementation, implemented by an array of XOR gates. These are the blocks indicated as “carry-around adder” and “bit invert” in Fig. 3. The complementation of the bits of $M_Y$ is marked by the circle at the output of the MUX.

Next, a leading-one-detector (LOD) detects the position of the leading one for subtractions producing leading zeros. The position (SHAMT) is then used in the variable left-shifter to normalize the result.
In the close path, rounding is required when \( D = 1 \) and the result is normalized. In contrast, variable shifting is required when the result is not normalized. Since rounding and shifting are mutually exclusive, they are implemented in parallel paths in Fig. 3. The selection is done by the signal

\[
\text{RND} = \text{MSB}(S) \quad \text{AND} \quad (D = 1)
\]

For the rounding, a simple incrementer is used. The result is rounded up when the last (L) and the guard (G) bits are both “1”: \( LG = 1 \).

The blue horizontal line in Fig. 3 shows the optimal placement of the pipeline registers in the close path.

### 3.2 FAR Path

All additions and subtractions with exponent difference larger than one are executed in the FAR path (Fig. 4).

Due to many positions shifts that may be necessary to align operands and to the fact that the shifter is expensive, one barrel shifter is used. To shift the significand with the smallest exponent in one shifter, it is necessary to swap the operands. This is done by the “swap” and “right-shift” blocks in Fig. 4. The swap is activated when \( E_Y > E_X \): \( \text{sign}(D) = 1 \).

The “right-shift” block does preserve the shifted out bits (22 least-significant bits, or LSBs) which are used to compute the sticky bit \( T \).

The most-significant part, 26 bits including guard (G) and round (R) bits, is conditionally inverted in case of subtraction (EOP=1).

The rounding is combined with the addition to reduce the latency. The sum of \( F_X \) and \( F_Y \) is performed in a compound adder depicted by the two carry-propagate adders, or CPAs, followed by the MUX in Fig. 4. Block CPA0 in Fig. 4 computes \( F_X + F_Y = S0 \), CPA1 computes \( F_X + F_Y + 1 = S1 \). The selection between the two sums is done according to the rounding.

To explain the rounding, we separate between the case of effective operation addition (EOP=0) and effective operation subtraction (EOP=1).

#### 3.2.1 FAR Path Rounding: Addition

When the effective operation is addition, we obtain for the sum \( F_X + F_Y \) a rounding pattern similar to the one in Fig. 1.

For the round-to-the-nearest-even mode \( \text{roundTiesToEven} \), we round up (select S1) if

\[
\text{MUXC}(0) : \quad \text{MUXC}(1) : \quad \begin{align*}
\text{MUXC}(0) & : \quad G(R \lor T \lor L) = 1 \\
\text{MUXC}(1) & : \quad L(L^* \lor G \lor R \lor T) = 1.
\end{align*}
\]

where we use “\( \lor \)” as a compact way for the logic or to avoid confusion with the “+” used for addition. However, if the sum result is larger than 2.0, we have an overflow (OVF=1) and the result is not normalized. In this case, we need to shift the sum one position to the right\(^5 \) (block “shift-1 L or R” in Fig. 4) and perform the rounding as follows. By considering \( L^* \), the bit to the left of bit L, we round up if

\[
\text{MUXC}(1) : \quad L(L^* \lor G \lor R \lor T) = 1.
\]

That is, the guard bit is \( L \) and the last bit is \( L^* \).

1. The sticky bit \( T \) is needed to detect a tie. \( T \) is computed by ORing the trailing bits of the result beyond bit \( R \) (Fig. 1).
2. The exponent need to be incremented by one when OVF=1.

### Fig. 4. FAR path (binary32).

Expressions (1) and (2) are evaluated in block “rounding control 1” in Fig. 4 from LGR (and \( L^* \)) extracted from \( F_X + F_Y \) and the sticky bit T. The output is the 2-bit MUXC signal.

By indicating with \( S_i \) the bit of weight \( 2^i \) of the sum, the overflow detection is done by

\[
\text{OVF} = (S_1 \cdot S_0) \lor (S_1 \cdot \text{MUXC}(1) \cdot \text{MUXC}(0)).
\]

where the second term is used for correction when the round up bit causes overflow.

Based on the overflow, either MUXC(0) or MUXC(1) is selected to determine the result between \( S0 \) and \( S1 \)

\[
\text{MUXR} = (\text{OVF} \cdot \text{MUXC}(1)) \lor \overline{\text{OVF}} \cdot \text{MUXC}(0).
\]

#### 3.2.2 FAR Path Rounding: Subtraction

When the effective operation is subtraction (EOP=1), the significand with the smallest exponent \( Y \) may be shifted and its 26 most-significant bits (MSBs) are one’s complemented to form \( F_Y \). To perform two’s complement, a “1” must be added in the least-significant position of \( Y \) (before the shifting). However, the complement’s “1” is propagated to the 26 MSBs \( (F_Y) \) only if the shifted out bits of \( Y \) are all zero. Consequently, we can use the sticky bit \( T \) to determine if we need to add the complement’s “1” to \( -F_Y \), which is rounded up by the sign bit.

Therefore, the condition to add a “1” to the LSB of \( -F_Y \) is

\[
\text{ADDI} = \text{EOP} \quad \text{AND} \quad T.
\]

Consequently, for EOP=1, ADDI is the complement of T. Since \( F_X \) is a normalized 24-bit significand represented in 26
bits, and its G and R bits are zero, the ADD1 bit is appended in the LSB of $F_X$.

The result of subtraction can be not normalized if $F_X \simeq 1$, for example, $1.0 - 0.125 = 0.875 < 1.0$. In this case, the result must be shifted one position to the left, and the exponent decremented by one. This condition is signaled by SHIL=1. Therefore, also for subtraction we need to consider two rounding positions, as in Fig. 5.

1) The result is normalized (SHIL=0). Comparing expression (1) with the bit positions in Fig. 5 (top), if $G = 1$ either there are non-zero bits shifted out (T=1), or we have to add ADD1 (T=0). Therefore, we have to check for G only. However, there is an extra case when round up is needed: L=1 (odd LSB), G=0, R=1 and T=0. In this case, $R + T = 2$ produces G=1 and the tie condition. Consequently, the result must be rounded up. Summarizing, when SHIL=0 the round up condition is

$$\text{MUXC}(0) : \quad G \lor \left( L \land R \land T \right) = 1 . \quad (4)$$

2) The result is not normalized (SHIL=1). In this case, the rounding position is in R as in Fig. 5 (bottom) and the last bit is G. The condition for the round up is

$$\text{MUXC}(1) : \quad (G \lor R) \lor RT = 1 . \quad (5)$$

Also expressions (4) and (5) are evaluated in block “rounding control 1” in Fig. 4 and multiplexed with (1) and (2) (selection by EOP) in the 2-bit MUXC signal. The shifting condition is determined by

$$\text{SH1L} = (S_{T_0} \land S_{1-1}) \lor (S_{0_0} \land S_{0-1} \land \text{MUXC}(1) \land \text{MUXC}(0))$$

Combining overflow and left-shifting, mutually exclusive conditions, the select bit MUXR is

$$\text{SHRL} = \text{OVF} \lor \text{SH1L}$$

$$\text{MUXR} = (\text{SHRL} \land \text{MUXC}(1)) \lor (\text{SHRL} \land \text{MUXC}(0))$$

In Fig. 4 the rounding control is split in two blocks because the OVFR and SHIL signals are available at a later time.

### 3.3 The Common Path

The double-path adder is completed by functional units which are common to both the the CLOSE and FAR paths. Beside the exponent handling (selection and update) and the sign computation, we need extra hardware to check for subnormals at the input and in the result. These are the functional units.

- A unit to detect if the exponent of the operands ($E_X$ and $E_Y$) is zero and set the integer bit in $M_{X/Y}$ accordingly.
- A unit to check is the sum $M_Z$ is a subnormal and in case it is, sets $M_Z = 0$ and $E_Z = 0$. The unit also checks for $\pm \infty$ and sets $M_Z, E_Z$, and $S_Z$ accordingly.

### 4 TFP ADDER

We now consider how to augment the binary32 FP adder of Section 3 to support TFP for significant ranges from $m=24$ (binary32) to $m=4$. The precision of the significand $m$ can be changed on a cycle basis by setting $m$ in a control register.

The TFP adder can also be used to convert a standard format (binary32, binary16) to TFP, or vice-versa, by setting the output precision $m$ and adding “0” to the number to be converted.

The main challenge to implement the adder in TFP is that the rounding must be done in a variable position. This is different from the case of the fused multiply-add where the operands are pre-shifted, depending of the exponent, and the rounding is done in a fixed position [8].

#### 4.1 FAR Path

One way to round in a variable position is to shift both operands by $2^{-m}$ bits and perform the rounding in fixed position in the least-significant bits of the result. However, the significand of the smallest number in the FAR path needs to be further shifted for alignment. The two shift-right operations can be combined by right-shifting the significand of the smallest number by $(24-m)+D$ position. However, while $m$ is known at the beginning of the operation, $D$ (exponent difference) computation is in the critical path, and adding $D + m$ to form the control of the shifter increases the delay of the critical path in the first stage of the adder.

One alternative way is to use a reference bit-vector to hold the rounding position. We refer to this bit-vector as “Rounding Word”, or RW. The RW is a bit-vector in which all bits are zero except the one in the rounding position. A simple decoder is used to generate RW depending on $m$.

Fig. 6 shows two example scenarios for $m=24$ (binary32) and $m=21$. In the figure, we assume the result $S$ be normalized and the rounding position (bit of weight $2^{-24}$ for $m=24$) is marked by “•” in RW.

The RW is used to extract one bit from $S$ by a simple AND-OR network. Fig. 7a illustrates how the extraction of bit G is done for a 4-bit example. Clearly, for a full RW extraction, 22 bits for $m \in [4, 24]$, 22 AND gates and a tree of OR gates are required.
Fig. 7. AND-OR networks (4-bit examples): a) to extract bit G. b) to append bit C after bit L.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2906907, IEEE Transactions on Computers

Fig. 8. Example of sticky bit calculation for a network of Fig. 7.a to Y. We can use a 24-bit vector with m MSBs set to “1” and the remaining bits set to zero. This bit-vector, called MASK, is generated by another decoder depending on m. By shifting MASK to the right by m+2 positions3 we obtain the pattern displayed in the third row of Fig. 8. As a final step, by applying the AND-OR network of Fig. 7.a to Y and MASK we obtain the sticky bit T. The bottom row in Fig. 8 displays the bits used to compute T (input to the OR tree).

The sticky bit calculation requires an additional shifter and a 48-bit AND-OR network. However, since MASK depends only on the precision m, the shifting as no impact on the latency and little impact on the power dissipation if m is not changed frequently.

Fig. 9 shows where this second shifter is placed in the augmented FAR path to support TFP. The AND-OR network is labeled “extract bit” in the figure. The extraction of bits L*, L, G, and R is done by combining the AND-OR networks for X and Y and followed by the computation of the L*, L, G, R bits of the result (block “extract LGR” in Fig. 9).

So far, we have described how to obtain the relevant bits for the rounding. Next, how to perform the actual rounding for the FAR path.

The architecture is similar to the one of Fig. 4 with a few modifications. The round up bit (in CPA1) and ADD1 bit must be placed in a variable position. This is done in two steps (refer to Fig. 9):

1) Both X and Y are masked (logic AND) with MASK to preserve only the m-MSBs:

\[ F_X = X \text{AND MASK} , \quad F_Y = Y \text{AND MASK} \]

2) The two extra positions are necessary to exclude the G and R bits.

This operation removes G, R and trailing bits in Y and has no effect on X, unless the significand has higher precision (conversion from longer formats).

2) The round up bit (U=1) and ADD1 (renamed C) are appended after the L bit of F_X and F_Y, as shown in Fig. 10.

Bit C, or ADD1, is computed in block “rounding control 1” in a different way from Section 3. The objective is to inject a carry in position L in CPA1:

\[ C = \overline{\text{EOP}} \vee \text{EOP}(G \vee \overline{R}) \]

For addition (EOP=0), C is always set to “1”. For subtraction (EOP=1), C can become the LSBS when the result needs to be left-shifted. To append the round up bit (U) in the correct position we simply OR the bits of F_Y to RW (rounding word). To append C, we need first to mark RW with the value carried by C (AND gates), then, the resulting bit-vector is ORed with the bits of F_Y. The two level network to append C is sketched in Fig. 7.b (for 4 bits), and the actual 26-bit network is labeled as “append bit” in Fig. 9.
4.2 CLOSE Path

As for the CLOSE path, the modifications are limited to the extraction of bits L and G, and appending the round up bit in the second stage of the CLOSE path. The result needs to be rounded up if LG=U=1, and U is ANDed with RW to form a bit-vector with a “1” in the rounding position, or an all-zero bit-vector. The rounding in the CLOSE path is illustrated in Fig. 11 (blocks “extract LG” and “AND”).

4.3 Rounding in Variable Position: Summary

Table 1 summarizes how the rounding is done once the right bits L, G, R, and T are extracted (Fig. 6).

![Table 1](image)

<table>
<thead>
<tr>
<th>EOP=0</th>
<th>EOP=1</th>
<th>CLOSE path leading 0’s</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVF</td>
<td>G(R\lor V\lor L)=1</td>
<td>SHRL</td>
</tr>
<tr>
<td>OVF</td>
<td>L(G\lor V\lor G\lor V\lor T)=1</td>
<td>SHRL</td>
</tr>
</tbody>
</table>

**Fig. 10.** Position where bits U and C are appended to F_X and F_Y.

**Fig. 11.** Additions in the second stage of the CLOSE path in TFP-add.

4.4 The Common Path

The TFP adder is completed by the exponent and sign handling blocks. The top-level view of the TFP adder is shown in Fig. 12. A decoder provides the rounding word (RW) and the mask (MASK) to implement the TFP operation to the CLOSE and FAR paths. The figure does not include the sign computation and the units to determine the integer bit of M_X and M_Y.

The exponent is set to the one of the number with the largest exponent and later adjusted depending on the necessary normalization of the result. The unit “condition detect” (“cond. det.”) in Fig. 12, determines if the result is a subnormal or plus/minus infinity. In this case, M_Z is flushed to zero and the exponent E_Z is set accordingly.

Subnormals are flushed to zero. If any of the inputs is a subnormal, its significand is flushed to zero while the exponent difference is computed. If the result of the addition is a subnormal, M_Z and E_Z are flushed by the multiplexers at the bottom of Fig. 12.

![Fig. 12](image)

Table 2

<table>
<thead>
<tr>
<th>binary32</th>
<th>TFP</th>
<th>difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>max. delay [ps]</td>
<td>950</td>
<td>980</td>
</tr>
<tr>
<td>AREA [mW] at 1 GHz</td>
<td>9.45</td>
<td>10.76</td>
</tr>
<tr>
<td>Total power [mW]</td>
<td>6,200</td>
<td>8,200</td>
</tr>
</tbody>
</table>

(1)[NAND2 equiv.]. (2)[mW] at 1 GHz

**TABLE 2**

Post-synthesis comparison between binary32 and TFP adders (RTNE).

5 HARDWARE IMPLEMENTATION

For the implementation of the TFP adder we opted for a 45 nm CMOS library of standard cells by using commercial synthesis and place-and-route tools (Synopsys). The FO4 delay^4 for this low power library is 64 ps and the area of the NAND-2 gate is 1.06 \mu m\(^2\).

We set as the target clock period a delay of 15 FO4 which is reasonable for a pipelined FP-unit. Since 15 FO4 \approx 1.0 ns, the target throughput is 1 GFLOPS for a single adder.

The architecture of the TFP adder (TFP-add) of Fig. 12, supporting the IEEE 754 roundTiesToEven (RTNE) rounding mode, is pipelined in two stages, with pipeline registers placed as indicated in Fig. 9, Fig. 11 and Fig. 12 by the blue horizontal lines.

For comparison purposes, we also implemented the baseline binary32 double-path adder (B32-add), supporting RTNE, of Fig. 2, Fig. 3 and Fig. 4.

A post-place-and-route comparison of the implemented units is reported in Table 2. Both units meet the timing constraint of \( T_C = 1.0 \text{ ns} \).

For both units, the delay of the critical path is considered the same, since the difference of a few pico-seconds is less than 1 FO4 delay. Table 3 reports the timing paths in the four parts (two stages and the CLOSE and FAR paths) of the TFP-add. All timing paths are close to 1 ns because the synthesizer uses the available slack to minimize the area and the power dissipation. The critical path originates in the second stage of the FAR path.

The area overhead for the TFP-add is about 32\% and it is mostly due to the parts labeled (1)-(4) (zones) in Fig. 9 and Fig. 11. The area and power dissipation overheads are

4. A 1 FO4 delay is the delay of an inverter of minimum size with a load of four minimum sized inverters.
TABLE 3
Timing paths in TFP-add (critical path in boldface).

<table>
<thead>
<tr>
<th>Zone</th>
<th>AREA (1)</th>
<th>Power (2)</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zone 1</td>
<td>480</td>
<td>7.7</td>
<td>91</td>
</tr>
<tr>
<td>Zone 2</td>
<td>340</td>
<td>5.5</td>
<td>216</td>
</tr>
<tr>
<td>Zone 3</td>
<td>390</td>
<td>6.3</td>
<td>687</td>
</tr>
<tr>
<td>Zone 4</td>
<td>360</td>
<td>5.8</td>
<td>143</td>
</tr>
<tr>
<td>decoder</td>
<td>112</td>
<td>1.8</td>
<td>1</td>
</tr>
<tr>
<td>regs. RW+MASK</td>
<td>288</td>
<td>4.6</td>
<td>2</td>
</tr>
<tr>
<td>MUX MZ</td>
<td>80</td>
<td>1.3</td>
<td>98</td>
</tr>
<tr>
<td>Sum</td>
<td>2050</td>
<td>33.1</td>
<td>1200</td>
</tr>
</tbody>
</table>

(1)[NAND2 equiv.], (2)[μW] at 1 GHz

TABLE 4
Overheads in TFP-add with respect to the binary32 adder.

TABLE 5
Average power dissipation for matrix multiplication for cases 1–4.

Fig. 13. Trends of average power dissipation in adders for matrix multiplication.

6 TFP AND DEEP LEARNING

Neural Networks (NNs) typically contain a very large number of parameters (weights $w_i$) and are usually trained iteratively over vast amounts of data. After training, the optimal weights are determined and the NN is ready to classify new input data (inference).

FIG. 14 shows the architecture for the two-hidden-layers NN used as example to illustrate the properties of TFP. The NN is used to interpolate a function $y = f(x)$ approximating the distribution of some sample points in the XY-plane. More detail on the example is given in [10].

We apply TFP to the operations in the NN of Fig. 14 for both training and inference, and we evaluate the trade-off precision vs. error and power dissipation. Differently from [10], both the TFP multiplier (TFP-mul) and the TFP-add implement round-to-the-nearest unbiased rounding (RTNE).

Table 6 reports the trade-off for training the NN with a “cosine-like” distribution for several TFP precisions. The table lists the approximation relative errors ($\epsilon_{\text{ave}}, \epsilon_{\text{max}}$) of the interpolated function with respect to the training points,
the average power dissipation (for addition, multiplication and total), and the ratio to the binary32 total dissipated power of the scaled precision runs.

The ratios in Table 6 show that the power dissipation drops linearly as \( m \) is scaled to about 50% for \( m=5, e=5 \). Therefore, finely tuning the precision, as in TFP, may significantly improve the performance of the training with respect to standard-precision formats.

Moreover, TFP can be advantageous also in the inference that usually requires lower precision than the training. A NN with two hidden layers has three different weights’ levels: \( w^0_i, w^1_{ji}, \) and \( w^2_{ji} \) (Fig. 14). In TFP, the precision can be adjusted on the level.

In the next experiment, we estimate the performance of several combinations of the precision \( m=\{24, 16, 8\} \) \( (e=8) \) for the classification of points within a given distance from the interpolated curve/function (more detail is given in [10]). Due to the approximation error (value of weights), out of 1,000 randomly distributed points in the XY-plane, 55 result miss-classified (about 5%). Table 7 reports the quantization error of the reduced precision weights as \( N_{\text{quant}}^{\text{MC}} \): number of miss-classified points due to quantization. In the table, the first three columns indicate the precision in the levels of the NN and the last columns the corresponding power dissipation. The best trade off is probably for \( (m^0,m^1,m^2)=(16,8,8) \) with \( N_{\text{quant}}^{\text{MC}}=4 \) points and power savings of about 30%. Instead, the largest power reduction (40%, ratio = 0.61) is reached for \( (m^0,m^1,m^2)=(16,8,8) \) with \( N_{\text{quant}}^{\text{MC}}=11 \) points.

### 7 Conclusions

In [7], we presented the Tunable Floating-Point representation to reduce precision and dynamic range of FP numbers for applications tolerating some error. In [7], we also presented architectures to implement TFP multiplication.

### REFERENCES


