Variable Precision 16-Bit Floating-Point Vector Unit for Embedded Processors

Alberto Nannarelli
Department of Applied Mathematics and Computer Science
Technical University, Denmark
alna@dtu.dk

***Abstract*—The increasing demand of computation at the edge and the tight power budgets push designers to migrate double and single-precision calculations to formats of reduced precision and dynamic range for applications that can tolerate some inaccuracy.**

In this context, we introduce a variable format for reduced precision floating-point with storage limited to 16 bits. This format is suitable for applications in signal processing, machine learning and other applications in embedded systems. We present the hardware implementations for multiplication and division units that can sustain a throughput of one result per clock cycle designed for vector processing. We also show some examples of applications that can benefit from the proposed format.

***Index Terms*—Floating-point, variable precision, customizable bias.**

I. Introduction

The miniaturization of electronic devices and the flourishing of the Internet-of-Things (IoT) bring new challenges in computation. On one hand, devices can send data to some server (the cloud) for the data processing. On the other hand, the data processing can be done on the device itself so that the results are available without requiring communication to the cloud, or only the results are sent to the server by reducing the communication bandwidth.

In this latter case, sometimes referred as "edge computing", it is crucial to optimize the computation resources to trade off several aspects, such as, performance, computation accuracy, and energy consumption. The optimization can be done by offloading the main processor and executing the heavy calculations in special purpose processors or in general purpose accelerators.

A large number of applications in IoT, signal processing and Machine Learning (ML) do not require very high numerical accuracy, and the trend is to migrate from the double or single-precision formats (*binary64* and *binary32* in the IEEE 754 Standard [1]), to formats of reduced precision and storage, for example, the Google's Brain-FP [2] and the IBM's DLFloat16 [3] formats used in ML. These floating-point formats require 16 bits of storage, equivalent to the IEEE *binary16* format, with different precisions and dynamic ranges.

However, with a limited storage of 16 bits there is a conflict between precision and dynamic range. If we increase the precision (more bits used to store the significand), we have to decrease the dynamic range (less bits used for the exponent).

In this work, we present a 16-bit floating-point unit, operating on vectors, in which precision and dynamic range can be adjusted depending on the application. The first element of the vector specifies the format in which the numbers in the vector are stored.

Moreover, for formats of reduced dynamic range, such as *binary16*, we can change the exponent bias to skew the dynamic range from symmetric to 2^0, as in IEEE 754, to asymmetric toward either positive or negative exponents.

Among the variable precision 16-bit FP unit, we focus on division, that for reduced precision can be implemented by non-iterative algorithms and pipelined to produce a throughput of one result per clock cycle.

The contributions of this paper are:

- The description of the variable precision 16-bit floating-point format, and the design of the hardware to handle it.
- The design of a 16-bit FP non-iterative division unit.
- Some examples of the benefit of the proposed format when running applications on embedded processors.

II. Variable Precision Floating-Point

The floating-point representation of a real number x is

$$x = (-1)^{S_x} \cdot M_x \cdot b^{E_x} \qquad x \in \mathcal{R}$$

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 standard [1] has significand normalized $1.0 \leq M_x < 2.0$ and biased exponent: *bias*$= 2^{e-1} - 1$.

The dynamic range is the ratio between the largest and the smallest (non-zero and positive) FP number [4]

$$DR_{FP} = (2^m - 1) \cdot 2^{2^e - 1} \ ,$$

and the precision is given by the weight of the bit in the last position 2^{-f}, where f is the number of fractional bits $f = m - 1$.

For example, for *binary16* – $m = 11$, $f = 10$, $e = 5$ (Figure 1) – the dynamic range, including *subnormals*, is

$$DR_{b16} = (2^{11} - 1)2^{2^5 - 1} \approx 4.4 \times 10^{12}$$

and the precision is 2^{-10}.

In contrast, for 16-bit fixed point $x \in [-1.0, 1.0)$ the dynamic range is

$$DR_{fxp16} = 2^{16} - 1 = 65,536 \approx 6.5 \times 10^4$$

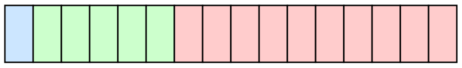
Fig. 1. IEEE 754 binary16 (half-precision) format [1].

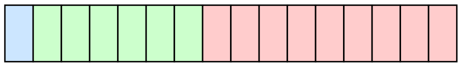
Fig. 2. IBM's DLFloat16 format [3].

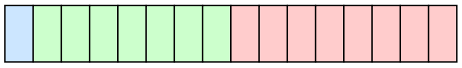
Fig. 3. TFP16 $e = 7$, $m = 9$ format.

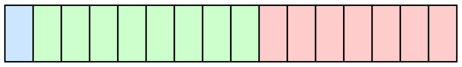
Fig. 4. Google's Brain-FP [2].

and the precision is 2^{-15}.

The starting point of our variable precision 16-bit floating-point representation is the IEEE *binary16* format in Figure 1. However, our Tunable 16-bit Floating-Point (TFP16) format, derived from [5], includes other three formats as in Figure 2 through Figure 4.

Since *subnormal* support is quite expensive, and we address error tolerant applications, we opted to flush-to-zero TFP16 numbers with exponent less than $-(E_{max} - 1)$.

TFP16 supports several rounding modes including the IEEE 754 default mode *roundTiesToEven*: round-to-the-nearest-even (on a tie). However, in this work, we implement the simpler *round-to-odd* (RTO) mode [6].

A. Customizable Bias

Especially for formats with $e = 5$ and $e = 6$ the range of the representable numbers is limited, and overflows, or underflows, may occur in some applications.

If the distribution of the exponents of FP numbers in a given application is skewed toward negative or positive exponents, we can extend the range of representable numbers by changing the bias. The bias is set to $2^{e-1} - 1$ in the *binary16* format.

For example, if in a given application implemented in *binary16* (bias=15), the range of the partial results is $[2^{-2}, 2^{17}]$, numbers with exponents 2^{16} or 2^{17} will result in an overflow. In this case, the range of biased exponent from 1 to 12 (corresponding to exponent 2^{-3} are not used. Therefore, we can extend the range for the specific application by setting the bias to 12, so that the largest numbers (exponent 2^{17}) will have biased exponent $17 + 12 = 29 < 31 \rightarrow \pm\infty$, and the smallest numbers will have biased exponent $-2 + 12 = 10 \gg 0$.

B. Format and Bias Specification

To be able to adapt the format to the specific application, we need a way to specify the format, and the custom bias. Since we opted for vector FP units, one solution providing little overhead is to add an element (FP number) to the vector as

TABLE I
ENCODING OF "ELEMENT 0" TO SPECIFY FORMATS AND BIAS.

e m	ENCODING s	e_{MAX}	e	bias	bias value
5 11	0	1 1 1 1 1 1 1 1	0 1	0 1 1 1 1	$0\ 1111\|_2 = 15$
	1	1 1 1 1 1 1 1 1	0 1	b b b b b	$b\ bbbb\|_2$
6 10	0	1 1 1 1 1 1 1 1	1 0	1 1 1 1 1	$01\ 1111\|_2 = 31$
	1	1 1 1 1 1 1 1 1	1 0	b b b b b	$bb\ bbb0\|_2$
7 9	0	1 1 1 1 1 1 1 1	1 1	1 1 1 1 1	$011\ 1111\|_2 = 63$
	1	1 1 1 1 1 1 1 1	1 1	b b b b b	$bbb\ bb00\|_2$
8 8	0	1 1 1 1 1 1 1 1	0 0	1 1 1 1 1	$0111\ 1111\|_2 = 127$
	1	1 1 1 1 1 1 1 1	0 0	b b b b b	$bbbb\ b000\|_2$

"element 0". In this "element 0", we store the format (number of exponent bits e), and the bias. To make sure that the "element 0" is not confused with a 16-bit FP value, we encode it as a *"not-a-number (NaN)"* with bits in the significand field indicating e and the bias. The encoding of "element 0" is explained in Table I.

By the encoding in Table I, the value is always recognized as a NaN, independently of the format. The format, specified by the two least-significant bits (LSBs) of the exponent size, is read from the two bits in position 6 and 5 (the LSB is in 0). If the sign bit (MSB) is 0, the bias is the default one: $2^{e-1} - 1$, otherwise the 5 LSBs specify the custom bias. The granularity of the custom bias is 2^{e-5}. For example, granularity 1 for $e = 5$, 2 for $e = 6$, etc..

For $e = 8$ the custom bias bits cannot be all zero: $-\infty$. However, because the exponent range for $e = 8$ is as large as *binary32*, probably it is never necessary to skew the bias for this format.

If a NaN is detected in the body of the vector, i.e., an operand, the result of the operation is set to NaN by copying the input NaN.

The bias can be set differently for each single operation. However, it is probably more practical to set the same format and bias for the whole kernel of computation. The choice can be made based on the profiling of the kernel, for example.

C. TFP16 Packing and Unpacking

The TFP16 units must be able to operate on the four formats in Figure 1–Figure 4. Therefore, it is necessary to unpack and pack the different fields of the operands from/to the 16-bit storage format.

The TFP16 units must accommodate the longest words in the datapath: 8-bit for exponents and 11-bit for significands. The exponent (integer) is aligned with LSB to the right, while the significand (fractional) is aligned with MSB to the left (integer bit). Figure 5 shows how the 16-bit words are unpacked and repacked according to e. The alignment is done by shifts implemented by 4:1 multiplexers. In parallel, with the unpacking, the exponent bits are OR'ed to detect if the exponent is zero and set the integer bit of the significand accordingly.

In the packing, after the operation, the significand is flushed to zero if an overflow or underflow condition is detected.

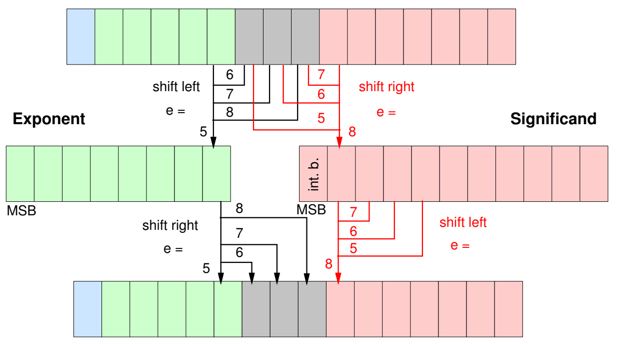

Fig. 5. TFP16 unpacking and re-packing in 16-bit storage format. Only LSB of exponent and MSB of fraction shown.

TABLE II
STANDARD BIAS SELECTED BY TWO BITS OF e.

e	bias		e	bias	
5	15	0000 1111	7	63	0011 1111
6	31	0001 1111	8	127	0111 1111

D. TFP16 Bias Handling

The different values of bias can be easily selected by a simple circuit generating the standard bias (symmetric as in IEEE 754) for each exponent length, as in Table II.

In addition, when the sign bit of the "element 0" is "1", the custom bias is set as specified in Table I.

Overflows are detected when the e-bit exponent is larger or equal than the maximum exponent for that format. For example if the exponent of the result is 0110 0000, it is an overflow for $e = 5$ and $e = 6$, but not for $e = 7$ and $e = 8$. Similarly, underflows are flushed to zero when the e-bit exponent is less or equal than zero.

III. TFP16 Multiplication

The unit for TFP16 multiplication is sketched in Figure 6. In the figure, unpacking and packing of TFP16 numbers are omitted to keep the drawing simple.

For the significand path, we need a 11×11-bit multiplier producing a 22-bit product P. For $m < 11$, the LSBs of the significands are set to zero during the unpacking, resulting in zeros in the corresponding partial products array.

The 10 LSBs of P are used to compute the sticky bit. The multiplexer connected to the 12 MSBs of P is used to normalize the result when P≥ 2.0, i.e., the MSB $\mathrm{P}_{(21)}$ is "1". We rename $\mathrm{P}_{(21)}$ as OVF (overflow) in Figure 6.

Next, we perform rounding-to-odd, as follows:

1) The sticky bit T_1 computed by OR'ing the 10 LSBs of P is OR'ed to bit $\mathrm{P}_{(10)}$ if OVF=1.

$$T_2 = T_1 + (P_{(10)} \cdot \mathrm{OVF})$$

Otherwise, $\mathrm{P}_{(10)}$ is the bit in the last position of the 11-bit normalized significand;

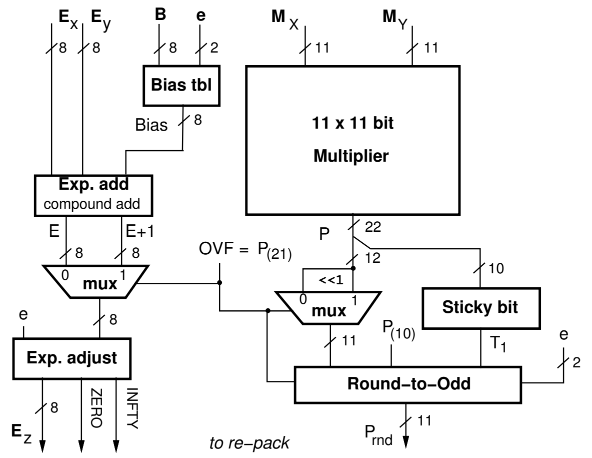

Fig. 6. TFP16 multiplier. Unpacking, re-packing and sign computation not shown.

2) Depending on e, we OR the bits up to the last position to form the correct sticky bit:

$$\begin{aligned} e=5 &: T = T_2 \\ e=6 &: T = P_{(11)} + T_2 \\ e=7 &: T = P_{(12)} + P_{(11)} + T_2 \\ e=8 &: T = P_{(13)} + P_{(12)} + P_{(11)} + T_2 \end{aligned}$$

3) The bit in the last position L for the m-bit significand is OR'ed to T:

$$P_{rnd(L)} = P_{(L)} + T$$

and the rounding-to-odd is completed.

For the exponent path, we need to set the bias B to the selected exponent or to the custom one (*Bias tbl* in Figure 6). Then, we need to add the two exponents and subtract the bias

$$E = E_x + E_y - B \quad .$$

We perform the three operand addition by a 3:2 carry-save adder, with a complemented input to handle subtraction, followed by a compound adder producing E and $E{+}1$. We select one of the two values according to OVF. If P≥ 2.0, the product is shifted one position to the right and the exponent $E+1$ is selected. The selected exponent is checked for overflow or underflow and the two condition are used in the re-packing block.

The sign of the product in obtained by XOR'ing the signs of the two operands.

IV. TFP16 Division

Division can be implemented by two classes of iterative algorithms: digit-recurrence and iterative approximation of the reciprocal followed by multiplication [4].

The digit-recurrence approach requires simpler hardware, but the convergence is linear: a fixed number of bits of the quotient are produced per iteration. The number of bits computed per iteration depends on the radix [4].

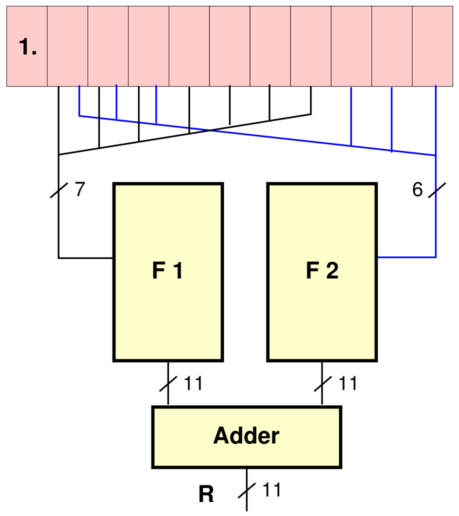

Fig. 7. Implementation of bipartite table for $R = 1/d$.

In contrast, multiplicative methods, such as reciprocal approximation by the Newton-Raphson method, converge quadratically, but require multipliers.

For *binary64*, double-precision, digit-recurrence algorithms give the best trade-off between latency and power dissipation. Recent work in digit-recurrence dividers shows several implementations with the radix ranging from 4 (2 bits for iteration) to 64 (6 bits per iteration) [7], [8].

For TFP16 precisions, $m \in [8, 11]$ bits, digit-recurrence algorithms require more than two iterations. For example, the radix-8 divider of [8] requires $\lceil 11/3 \rceil = 4$ iterations for $m = 11$ significands, while the radix-64 divider of [7] requires $\lceil 11/6 \rceil = 2$ iterations. In both examples, extra cycles are necessary for initialization and rounding.

However, for a vector unit, it is desirable to have one result produced per clock cycle in a fully pipelined unit. By giving up accuracy, we can implement a divider for TFP16 precision, by approximating a 11-bit reciprocal in one cycle, followed by a multiplication in a second cycle

$$\begin{array}{rcrcl} & & q & = & x/d \\ \hline \text{cycle 1} & : & R & = & 1/d \\ \text{cycle 2} & : & q & = & x \cdot R \end{array}$$

Since we use rounding-to-odd, we do not need to compute the remainder.

A. Reciprocal Approximation

The simplest approximation of a function of limited precision is by using a look-up table. In the case of the reciprocal $R = 1/d$, we tried to implement the approximation for $m = 8$ significands with a single table by synthesis in a standard cell library. The table, synthesized as multi-level logic, resulted in an area of 215 NAND-2 equivalent gates. However, for $m = 11$ (largest precision for TFP16) the area resulted to be 600 NAND-2.

An alternative to the single table is the bipartite table. By resorting to Taylor's series the approximation of the function is obtained by the sum of two parts implemented by tables of reduced size [4]. In our case, we approximated the reciprocal by bipartite tables of 2^7 and 2^6 entries. The reciprocal approximation, depicted in Figure 7 resulted in an area of 470 NAND-2 gates, including the adder. Table F2 output is 4 bits wide and it is sign-extended to 11 bits.

We report in Table III the errors obtained from exhaustive simulations for approximating $1/d$ for the four TFP16 formats.

TABLE III
AVERAGE AND MAXIMUM ERROR FOR $R = 1/d$ APPROXIMATION WITH BIPARTITE TABLE.

		ϵ_{ave}		ϵ_{max}	
m	e	value	weight	value	weight
11	5	0.000413	-12	0.001943	-10
10	6	0.000496	-11	0.002049	-9
9	7	0.000760	-11	0.002496	-9
8	8	0.001391	-10	0.004259	-8

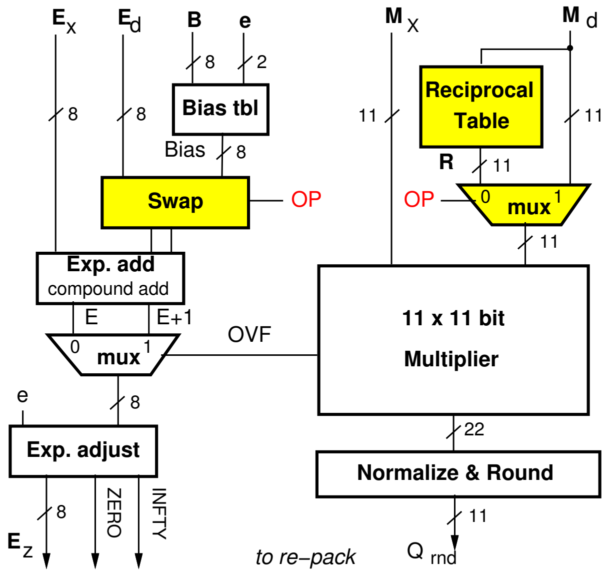

Fig. 8. TFP16 divider/multiplier, depending on OP.

B. Divider Implementation

By combining the reciprocal table and the TFP16 multiplier of Sec. III we obtain the TFP16 division unit. Since multiplication is far more frequent than division, we decided to implement a unit to compute both division and multiplication.

The architecture of the combined division/multiplication unit is sketched in Figure 8. In the figure, we highlighted the extra blocks necessary to implement division with respect to the TFP16 multiplier of Figure 6.

In the significand path (Figure 8 at right), we need to select R for division (OP=0) or M_d for multiplication. In the exponent path (Figure 8 at left), for division the exponents are subtracted, and the bias is added:

$$E = E_x - E_d + B \qquad \text{OP=0,}$$

while for multiplication the exponents are added and the bias subtracted:

$$E = E_x + E_d - B \qquad \text{OP=1}.$$

Therefore, we need to swap E_d and B depending on the operation.

The remaining parts of Figure 8, normalization and rounding, and exponent update and adjustment do not change. The combined unit is completed by TFP16 unpacking and packing hardware and a XOR gate to compute the sign.

C. Division and Square Root

The architecture in Figure 8 could easily be adapted to compute square root by adding a table to approximate $1/\sqrt{x}$. The square root is then computed by

$$s = x \cdot \frac{1}{\sqrt{x}} = \sqrt{x} \; .$$

Modifications are required in the exponent handling. We leave this improvement for future work.

V. Hardware Implementation

The TFP16 combined divider and multiplier of Figure 8, completed with unpacking and packing units is implemented in the STM 45 nm low-power library of standard cells.

In Figure 9, we show the critical path of the unit. The critical path is through the following blocks:

$$\text{Unpack} \rightarrow \text{R Table} \rightarrow \text{Mux} \rightarrow \text{Mult.} \rightarrow$$
$$\rightarrow \text{Mux (norm.)} \rightarrow \text{Round} \rightarrow \text{Repack} \quad \simeq 2 \; ns$$

The target is to have the TFP16 units clocked at 1 GHz, corresponding to a clock period $T_C = 1.0 \; ns$, or to about 15 FO4 delay in the library. To meet the timing constraint, the unit is pipelined in two stages:

1) unpacking, reciprocal table and exponent computation (E and E+1);
2) multiplication, normalization and rounding, exponent update, and packing.

The position of the pipeline registers is marked by horizontal blue lines in Figure 9.

The synthesized circuit meets the timing constraints of $T_C = 1.0 \; ns$ with an area of about 4,000 NAND-2 equiv. gates. The overhead to support TFP16 is about 330 NAND-2 (8% of total area): unpacking 210, *Bias tbl* 40, and repacking 80 NAND-2.

The power dissipation for the different TFP16 formats is reported in Table IV. The power estimation is based on test vectors with the significand of the divisor M_d covering all the combinations for normalized operands: i.e., 2^{10} combinations for $m = 11$ (*binary16*). Random values are used for the significand of the dividend M_x and for both the exponents.

In Table IV, along with average power dissipation measured at 1 GHz, we report also the energy necessary to complete a division

$$E_{div} = P_{ave} \times \text{n. cycles} \times T_C \qquad [J],$$

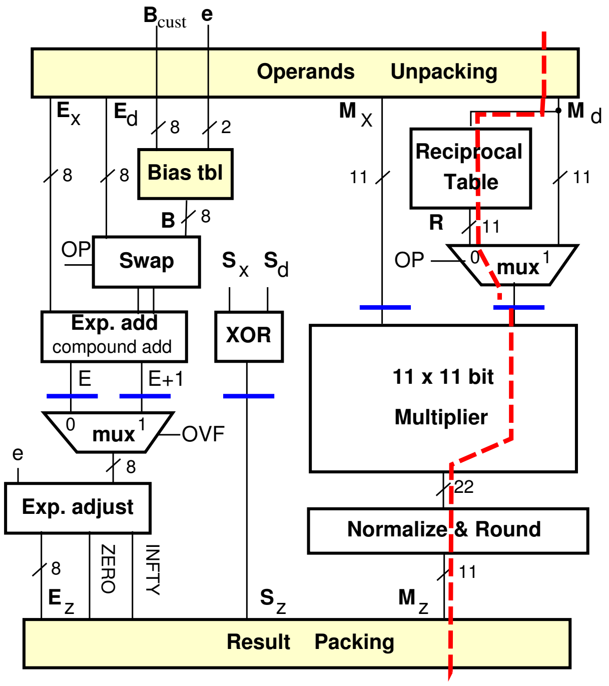

Fig. 9. Critical path for TFP16 divider/multiplier. The position of the pipeline registers is marked by the horizontal blue segments.

TABLE IV
Average power dissipation and energy per operation for TFP16 div/mult and radix-8 divider.

OP	m	P_{ave} [mW]	cycles	E_{div} [pJ]	ratio
div	11	3.16	2	6.33	1.00
div	10	3.12	2	6.23	0.99
div	9	3.04	2	6.08	0.96
div	8	2.91	2	5.81	0.92
mul	11	2.68	2	5.35	0.85
radix-8	m	P_{ave} [mW]	cycles	E_{div} [pJ]	ratio
binary16	11	8.50	7	59.50	9.41
BFP-16	8	8.47	6	50.81	8.03
binary64	53	9.09	24	218.26	34.50

P_{ave} measured at 1 GHz.

and the E_{div} ratios with respect to $m = 11$.

For division in the TFP16 unit (OP=div), the power dissipation (and energy) is reduced very little when the precision scales. We also report the power dissipation when the TFP16 combined unit is used for multiplication (OP=div) for significant precision $m = 11$ only. In this case, P_{ave} and E_{div} are reduced by about 15% because in the reciprocal table we set the input to 1.0 when OP=mul.

In the bottom part of Table IV, we report the power and energy consumed by division implemented in a digit-recurrence unit. We opted out the low latency unit of [7] because the high level of speculation results in a large unit, and the reduced number of iterations for TFP16 do not offset the larger power dissipation per cycle. We chose to compare the TFP16 divider to the radix-8 unit of [8] modified to handle formats smaller than *binary64*. This decision can be justified when the reduced precision processor/accelerator is not equipped with a division unit, and the regular FP-unit is used instead.

The radix-8 unit requires one initialization cycle and two termination cycles. Therefore, its latency in clock cycles is:

$$\text{n. cycles} = 1 + \left\lceil \frac{m}{3} \right\rceil + 2 \ .$$

By synthesizing the radix-8 divider in the same low-power library, we meet the $T_C = 1.0\ ns$ timing constraint with an area of about 8700 NAND-2 – it must handle *binary64*. We run the power estimation on the radix-8 divider for $m = 11$, $m = 8$ and *binary64*. The results in Table IV (bottom part) show that the TFP16 divider is much more power efficient for low precision divisions. Furthermore, the TFP16 divider can sustain a throughput of one result per clock cycle.

VI. Examples of Applications

In this section, we provide some examples of applications that can benefit from the TFP16 format. The examples are run on a bit accurate simulator, implementing all the features of the TFP16 format of Sec. II.

A. Chi Square

The Large Hadron Collider, at CERN (Switzerland), is built to accelerate sub-atomic particles and allow state-of-the-art research in particle physics. One of the major experiments is ATLAS [9], a large detector to measure the trajectories of particles, called "tracks", formed after the collision of two particles.

Each collision generates tracks of interest, but also noise (i.e., particles not to be tracked). The detector collects a huge amount of data, about 40 TB/s, and the noise needs to be filtered out.

One of the filtering algorithms is the so called *"Chi-square"* algorithm, a measure of the distance between the observed track and the expected track. If the observed track is close enough to the expected, the corresponding data is stored, otherwise the data is discarded.

The χ^2 (Chi-square) is calculated as:

$$\chi^2 = \sum_{i=1}^{6} \left(\sum_{j=1}^{11} S_{ij} x_j + h_i \right)^2 \qquad (1)$$

where S_{ij} and h_i are pre-calculated constants for the detector sector (several of them) and x_j are the local hit coordinates [10].

The χ^2 calculations are currently done in *binary32* (FPGA), but it is desirable to move the computation to a reduced format to save hardware.

By looking at the distribution of the exponents (biased) for the *binary32* representation of S_{ij} and h_i in Figure 10, we notice that the values are between 97 (2^{-30} unbiased) and

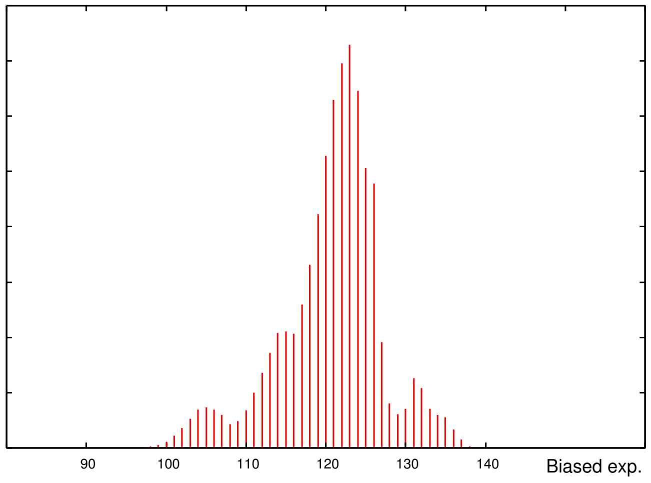

Fig. 10. Distribution of biased exponents for S_{ij} and h_i of 2,000 detector sectors. E=127 corresponds to 2^0.

138 (2^{11} unbiased). With this exponent range (difference is 39), we would be able to run (1) in TFP16 $m = 10$ $e = 6$.

However, since the minimum exponent allowed for $e = 6$ is 2^{-32}, about one third of the performed operations result in underflow with a rather large error. By setting the bias to 42 (the minimum unbiased exponent detected is 2^{-42}), we eliminate the underflows and reduce the approximation error with respect to the *binary32* computation. For exponents $e = 7$ and $e = 8$, we do not get overflows, but the precision is reduced with respect to the case for $e = 6$ and $B = 42$.

B. Gaussian Elimination

Gaussian elimination is an algorithm used in several linear algebra kernels. It consists of two steps:

1) A first pass on the rows of matrix A to transform it in an upper triangular matrix;
2) A substitution step in which the roots `x[i]` are computed from the last row and backwards.

The pseudo-code to implement the Gaussian elimination is illustrated in Algorithm 1.

Both steps require division that for single or double-precision can be a bottleneck because of the several iterations/cycles needed for it.

In TFP16, division has a latency of two clock cycles and a throughput of one result per clock cycle similar to addition and multiplication.

We run simulations with fractional inputs in $(-1.0,\ 1.0)$ and dynamic range of 24 bits. As an example, we use a order 10 system (10 unknowns) and compared the error in the roots. Table V shows the average error obtained for *binary32* and the four TFP16 formats. Rounding-to-odd is applied in all cases.

The simulations show that for reduced precision of the operands, the error is still acceptable for error tolerant applications. Moreover, all TFP16 operations can sustain a throughput of one result per clock cycle.

Algorithm 1 Gauss elimination.

```
/* rows transformation to upper triangular matrix */
for i=1 to n do
  for j=i+1 to n+1 do
    t=a[j][i]/a[i][i];
    for k=i to n+1 do
      a[j][k]=a[j][k]-t*a[i][k];
    end for
  end for
end for

/* backward substitutions */
x[n]=a[n][n+1]/a[n][n];
for i=n-1 down to 1 do
  s=0;
  for k=n down to i+1 do
    s=s+a[i][k]*x[k];
  end for
  x[i]=(a[i][n+1]-s)/a[i][i];
end for
```

TABLE V
AVERAGE AND MAXIMUM ERRORS FOR GAUSSIAN ELIMINATION (ORDER 10) FOR *binary32* AND TFP16 FORMATS.

		ϵ_{ave}		ϵ_{max}	
m	e	value	weight	value	weight
binary32		0.000001	-21	0.000002	-20
11	5	0.005531	-8	0.007938	-7
10	6	0.008565	-7	0.018774	-6
9	7	0.017579	-6	0.062387	-5
8	8	0.051686	-5	0.093637	-4

VII. CONCLUSIONS AND FUTURE WORK

The increasing demand of computation at the edge and the reduced power budgets push designers to migrate calculation from the traditional double and single-precision to formats of reduced precision and dynamic range for applications that can tolerate some inaccuracy.

Although the hardware for floating-point is more complicated than the one for fixed-point arithmetic, floating-point makes the development of applications easier because operations such as scaling and operand alignment, are taken care by the hardware.

In this context, we introduced TFP16, a variable format for reduced precision floating-point with storage limited to 16 bits. We illustrated the trade-offs between dynamic range and precision for the four TFP16 formats, and showed that the dynamic range of the FP representation can be extended by skewing the bias.

We presented TFP16 units for multiplication and division that can sustain a throughput of one result per clock cycle and have a latency of two clock cycles.

A TFP16 adder can easily be derived by the variable precision unit of [11] by scaling the significand datapath from $m = 24$ to $m = 11$ and by including the blocks to handle packing/unpacking and the bias. The adder design will be addressed in future work, as the inclusion of a table to compute square-root in the TFP16 division unit.

The TFP16 units are designed to handle vector processing, and the architecture of the whole vector unit will also be part of future work.

REFERENCES

[1] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, Jul. 2019.

[2] N. P. Jouppi *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 1–12.

[3] A. Agrawal, S. M. Mueller, B. M. Fleischer, J. Choi, N. Wang, X. Sun, and K. Gopalakrishnan, "DLFloat: A 16-b Floating Point Format Designed for Deep Learning Training and Inference," in *26th IEEE Symposium on Computer Arithmetic*, Jun. 2019, pp. 92–95.

[4] M. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann Publishers, 2004.

[5] A. Nannarelli, "Tunable Floating-Point for Energy Efficient Accelerators," in *25th IEEE Symposium on Computer Arithmetic*, Jun. 2018, pp. 29–36.

[6] S. Boldo and G. Melquiond, "Emulation of FMA and Correctly Rounded Sums: Proved Algorithms Using Rounding to Odd," *IEEE Transactions on Computers*, vol. 57, no. 4, pp. 462–471, Apr. 2008.

[7] J. D. Bruguera, "Low Latency Floating-Point Division and Square Root Unit," *IEEE Transactions on Computers*, vol. 69, no. 2, pp. 274–287, Feb. 2020.

[8] A. Nannarelli, "Performance/Power Space Exploration for Binary64 Division Units," *IEEE Transactions on Computers*, vol. 65, no. 5, pp. 1671–1677, May 2016.

[9] ATLAS Experiment, CERN. "Atlas Web Page". [Online]. Available: https://atlas.cern

[10] A. Marantis, "The ATLAS Fast TracKer - Architecture, Status and High-Level Data Quality Monitoring Framework," in *CERN, ATL-DAQ-PROC*, no. 41, 2018, pp. 1–8.

[11] A. Nannarelli, "Tunable Floating-Point Adder," *IEEE Transactions on Computers*, vol. 68, no. 10, pp. 1553–1560, Oct. 2019.