Mathematical and Computational Applications

MDPI

Article

Differential Evolution under Fixed Point Arithmetic and FP16 Numbers

Luis Gerardo de la Fraga

Computer Science Department, Center for Research and Advanced Studies of the National Polytechnic Institute (CINVESTAV), Ciudad de Mexico 07360, Mexico; fraga@cs.cinvestav.mx; Tel.: +52-55-57473755

Abstract: In this work, the differential evolution algorithm behavior under a fixed point arithmetic is analyzed also using half-precision floating point (FP) numbers of 16 bits, and these last numbers are known as FP16. In this paper, it is considered that it is important to analyze differential evolution (DE) in these circumstances with the goal of reducing its consumption power, storage size of the variables, and improve its speed behavior. All these aspects become important if one needs to design a dedicated hardware, as an embedded DE within a circuit chip, that performs optimization. With these conditions DE is tested using three common multimodal benchmark functions: Rosenbrock, Rastrigin, and Ackley, in 10 dimensions. Results are obtained in software by simulating all numbers using C programming language.

Keywords: differential evolution; fixed point arithmetic; FP16; pseudo random number generator

Citation: de la Fraga, L.G. Differential Evolution under Fixed Point Arithmetic and FP16 Numbers. *Math. Comput. Appl.* **2021**, *26*, 13. https://doi.org/10.3390/mca26010013

Academic Editor: Leonardo Trujillo

Received: 19 December 2020
Accepted: 2 February 2021
Published: 4 February 2021

1. Introduction

The use of different number types in machine learning applications has been analyzed extensively in previous years, more specifically in deep learning neural networks [1,2]. These kinds of neural networks use the convolution as the basic function and have thousands of parameters and must be trained first; that is, the network must be optimized by modifying all the parameters to obtain a local minimum of the goal function. The optimization step is called training and it could take hours in modern hardware of general purpose graphics processor units (GPGPUs). A special type of number, Brain Floating Point (bfloat16), which is a half-precision FP format of 16 bits with the same range of the usual single precision FP numbers (float in C programming language, of 32 bits length), has been proposed for training deep learning neural networks [2]. Other FP numbers of 16 bit length are the so-called FP16 numbers, these are an IEEE standard [1,2] for half-precision FP numbers and can be used on ARM processors.

The goal of using different, shorter numbers in machine learning applications is to improve the speed, and as a consequence reduce the power consumption as it would take less time to train a deep learning network, and also reduce the storage memory or disk size for the variables. In [1] it is mentioned that half precision is also attractive for accelerating general purpose scientific computing, such as weather forecasting, climate modeling, and solution of linear systems of equations. The supercomputer Summit (it was in the Top 500 list https://www.top500.org (accessed on 3 February 2021)), has a peak performance of 148.6 petaflops in the LINPACK benchmark, a benchmark that employs only double precision. For a genetics application that uses half precision, the same machine has a peak performance of 2.36 exaflops [1].

In this work it is proposed to analyze the well known heuristic for single objective optimization, the differential evolution (DE) algorithm, under FP16 numbers, and also under fixed point arithmetic that uses integer numbers of different lengths. This analysis is important if we think of embedded optimization algorithms within a chip [3], which performs a dedicated task. One constraint in these kinds of applications must be that the power consumption is as low as possible. Also it is important if one designs a dedicated

algorithm in hardware, just as in FPGAs (Field Programmable Gate Arrays), to accelerate its behavior. Also, another possible application is to execute a fast and small DE inside each core in a GPGPU. These three application scenarios justify the analysis of the DE performed in this work.

The rest of this article is organized as: in Section 2 a very brief description of fixed point arithmetic and FP numbers is made. In Section 3 the DE algorithm is analyzed for which parts could be improved by using other different number types. In Section 4 some experiments and their results are described. Finally, in Section 6 some conclusions are presented.

2. Fixed Point Arithmetic and Floating Point Numbers

The notation $a.b$ will be used here to represent a set of integer numbers that uses a bits in the integer part, and b bits in the fractional part. Each number is of size $a + b + 1$ bits (plus the sign bit).

For a number $x \in a.b$, the range of numbers that can be represented is:

$$-2^a \leq x \leq 2^a - 2^{-b} \tag{1}$$

Summing up two numbers $a.b$ results in a number $(a + 1).b$ [4]. The multiplication of two numbers $a.b$ results in a number $(2a + 1).2b$ [4]. It is possible to verify these results by applying the respective operation to two extreme numbers in (1).

The microprocessors offer the sum and multiplication of two integer numbers and the result is stored in a number of the same size as the operands. In a hardware design for a given application, one must use a big enough number to store the sum of two $a.b$ numbers, and the result to multiply two $a.b$ numbers must be returned to a $a.b$ number. The easiest way to perform this is by truncating the result: the resulted $2a.2b$ is shifted b bits to the right, again the number must be big enough to store the resulted $a.b$ number. In a PC, if one uses 32 bit integer numbers, the first bit is the sign bit, and then one could multiply up two $\sqrt{2^{31}} = 2^{31/2}$ values to keep the result within the used 31 bits. In any application, normally one does not take care if the used numbers can keep the result of the operations applied to them, and one trusts that the numbers are big enough to store the results.

The operations sum and multiplication of two integer numbers are the fastest because each operation is built in the hardware and both take a single clock step.

The sum and multiplication of two FP numbers is totally different. An FP number is composed as $s \cdot 2^e$, where s is the significant and e the exponent. If p bits are used for the significant, it is an integer that could take values from 0 to $2^p - 1$. The exponent e is an integer number too. The sum of two FP numbers is carried on first by expressing both numbers with the same exponent, then summing up both significants. The greater exponent of both numbers is used to express them with the same exponent. The result must be rounded to express the same number of bits used in the significants. Also, the result could be normalized, which means that the exponent will have a single binary precision number.

The multiplication takes more steps because two numbers $s_1 \cdot 2^{e_1}$, and $s_2 \cdot 2^{e_2}$ are multiplied as $s = s_1 \cdot s_2$ and the exponents are summed ($e = e_1 + e_2$), and also both results are rounded and the final result is normalized.

In the IEEE 754 standard [5], an FP number has a sign bit, i, and the represented number is equal to $(-1)^i \cdot s \cdot 2^e$, where $e_{\min} \leq p + e - 1 \leq e_{\max}$. The values used in common FP numbers are shown in Table 1.

Table 1. Characteristics of floating point (FP) numbers in the IEEE 754 standard.

Precision	Exponent	Significant	$e_{\min}$	$e_{\max}$	Smallest	Biggest
Half	5	10	-14	$+15$	6.10×10^{-5}	6.55×10^{4}
Single	8	23	-126	$+127$	1.17549×10^{-38}	3.40282×10^{38}
Double	11	52	-1022	$+1023$	2.22507×10^{-308}	1.79769×10^{308}

Floating point operations take more than a clock cycle within a microprocessor.

The IEEE 754 standard [5] gives much more aspects that are necessary to work with FP numbers, such as rounding methods, Not a Number (NaN), infinities, and how to handle exceptions. In [6] all these details about FPs are explained.

3. DE Analysis

DE is a heuristic used for global optimization under continuous spaces. DE solves problems as:

$$\begin{aligned} &\text{minimize: } f(\mathbf{x}), \\ &\text{subject to: } \mathbf{g}(\mathbf{x}) \geq 0, \text{and} \\ &\qquad\qquad \mathbf{h}(\mathbf{x}) = 0, \\ &\qquad\qquad \mathbf{x} \in S \subset \mathbb{R}^n. \end{aligned} \tag{2}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is the function to optimize; $\mathbf{x} \in \mathbb{R}^n$, that is, the problem has n variables; and also we could have $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^{m_1}$, m_1 inequality constraints; and $\mathbf{h} : \mathbb{R}^n \to \mathbb{R}^{m_2}$, m_2 equality constraints. The solution to the problem $\mathbf{x}$ is in a subset S of the whole search space $\mathbb{R}^n$ and where the constraints are satisfied, this space S is called the *feasible space*.

Also, the *search space* contains the feasible space and is defined by the *box constraints*:

$$x_i \in [l_i, u_i], \text{ for } i = \{1, 2, \ldots, n\}. \tag{3}$$

This is, each variable x_i is searched in the interval defined by the lower bound value l_i, and the upper bound value u_i, for $i = \{1, 2, \ldots, n\}$.

Constraints can be incorporated into the problem (2) by modifying the objective function as:

$$f_1(\mathbf{x}) = f(\mathbf{x}) + \alpha \sum_{i=1}^{m_1} \min[0, g_i(\mathbf{x})]^2 + \beta \sum_{i=1}^{m_2} h_i^2(\mathbf{x}) \tag{4}$$

Now the f_1 will be optimized instead of f in (2). α and β in (4) represent the penalty coefficients that weigh the relative importance of each kind of constraint.

One important point about DE is that the heuristic needs to only evaluate the problem to solve. Classical mathematical optimization methods use the first and perhaps also the second derivative of the given problem. These derivatives are easy to obtain if one has in hand the mathematical expression to the given problem. It is possible to approximate the derivatives numerically but with a very high computational cost [7].

According to the test in the CEC 2005 conference [8], DE is the second best heuristic to solve real parameter optimization problems, when the number of parameters is around 10.

The DE pseudocode is shown in Algorithm 1.

DE works with a population that is composed of a set of individuals, or vectors, of real numbers. All vectors are initialized with random numbers with a uniform distribution within the search bounds of each parameter (line 1 in Algorithm 1). For a certain number of iterations (line 4) the population is modified and this modified population could replace the original individuals. The core of DE is in the loop on lines 8–13: a new individual is generated from three different individuals chosen randomly; each value of the new vector (it represents a new individual) is calculated from the first father, plus the difference of the other two fathers multiplied by F, the difference constant; the new vector value is calculated if a random real number (between zero and one) is less than R, the DE's recombination constant. To prevent the case when the new individual could be equal to the current father i, at least one vector's component (a variable value) is forced to be calculated from their random fathers values: it is in line 9 of the pseudocode, when $j = j_{\text{rand}}$, and j_{rand} is an integer random number between 1 and n. In lines 10–12 it is checked if each combined variable value is within the search space. Then the new individual is evaluated, and if it is better than the father (in lines 11–12), then the child replaces its father. The stop condition used here is: if the number of iterations is greater than a maximum number of iterations or when the difference in the objective function values of the worst and best individuals

is less than v. This stop condition is called *diff* criterion in [9], and is recommended for a global optimization task.

Algorithm 1 Differential evolution algorithm (rand/1/bin version)

Require: The search space and the value v for the stop condition. The values for population size, μ; maximum number of generations, g; difference and recombination constants, F and R, respectively.

Ensure: A solution of the minimization problem

initialize $(P = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_\mu\})$
evaluate (P)
$k = 0$
repeat
  for $i = 1$ to μ **do**
    Let r_1, r_2 and r_3 be three random integers in $[1, \mu]$, such that $r_1 \neq r_2 \neq r_3$
    Let j_{rand} be a random integer in $[1, n]$
    for $j = 1$ to n **do**
      $$x'_j = \begin{cases} x_{r_3,j} + F(x_{r_1,j} - x_{r_2,j}) & \text{if } U(0,1) < R \text{ or } j = j_{\text{rand}} \\ x_{i,j} & \text{otherwise} \end{cases}$$
      if $x'_j < l_i$ or $x'_j > u_i$ **then** ▷ Check bounds
        $x'_j = U(0,1)(u_i - l_i) + l_i$
      end if
    end for
    if $f(\mathbf{x}') < f(\mathbf{x}_i)$ **then**
      $\mathbf{x}_i = \mathbf{x}'$
    end if
  end for
  $\min = f(\mathbf{x}_1)$, $\max = f(\mathbf{x}_1)$
  for $i = 2$ to μ **do**
    if $f(\mathbf{x}_i) < \min$ **then**
      $\min = f(\mathbf{x}_i)$
    end if
    if $f(\mathbf{x}_i) > \max$ **then**
      $\max = f(\mathbf{x}_i)$
    end if
  end for
  $k \leftarrow k + 1$
until $(\max - \min) < v$ or $k > g$

A general form to set the parameter values for DE is: if d is the number of variables, the population size is set to $10d$, $F \in [0.5, 1.0]$, and $R \in [0.8, 1.0]$ [9].

The DE in Algorithm 1 can be improved by using a random integer number generator as the one described in [10], which does not use divisions or FP numbers. This idea could improve the algorithm in line 6 (to generate three numbers in the interval $[1, \mu]$, and in line 7 where another random integer number is generated in the interval $[1, n]$. Also, the values for F and R are within the interval $[0.5, 1.0]$, and usually no more than one or two decimal values are used for these constants, thus these values are not affected by using half precision numbers (see Table 1). Even more, a totally integer arithmetic could be used in the comparison $U(0,1) < R$) (in line 9 in Algorithm 1), if it is used instead $\text{rand}(1, 2^{31}) < I$, with $I = \lfloor 2^{31} \cdot R \rfloor$.

Two implementations of DE were used in this work: one with fixed point arithmetic, and another one using FP16 numbers. The implementation with fixed point arithmetic uses integer (of 32 bits) numbers for all the variables. The implementation using FP16 numbers uses half precision floats (FP16, 16 bits) for all the variables. In this paper a computer of 64 bits architecture was used, then the multiplication of two integers was stored in a long type variable of 64 bits, shifted and truncated to a integer of 32 bits. The core part of DE (lines 8–13 in Algorithm 1) calculates the selected and mutated vector $\mathbf{x}'$ as:

$$x'_j = \begin{cases} x_{r_3,j} + F(x_{r_1,j} - x_{r_2,j}) & \text{if } U(0,1) < R \text{ or } j = j_{\text{rand}}, \\ x_{i,j} & \text{otherwise,} \end{cases} \tag{5}$$

for $j = \{1, 2, \ldots, n\}$, this is for each variable of the given problem. Thus, one subtraction $(x_{r_1,j} - x_{r_2,j})$ followed of one multiplication (by constant F) and one summation (with $x_{r_3,j}$) are needed to calculated the new vector $\mathbf{x}'$. The greatest value for F could be 1, if all the search space is equal for all variables, the result in (5) could be the double of the current x'_j value.

Then, the maximum possible values in the search space could be the double of the bound values of the search space. Another problem is to find the maximum possible value in the function space. Also, it is not clear how many bits are necessary in the fractional part for the fixed point arithmetic. These items are solved in the following section.

4. Experiments with Three Multimodal Functions in 10 Dimensions

Three very well known benchmark functions were used: shifted version of Rosenbrock, Rastrigin, and Ackley functions in 10 dimensions. All these functions are multimodal, which justify solving them using the DE heuristic. The used Rosenbrock function is defined as:

$$f_1(\mathbf{x}) = 0.39 + \frac{1}{10} \sum_{i=1}^{n-1} \left\{ \left[(x_i + 1)^2 - (x_{i+1} + 1) \right]^2 + \frac{x_i^2}{100} \right\}, \tag{6}$$

its minimum value is 0.39 with $\mathbf{x} = [0, 0, \ldots, 0]$.

The Rastrigin function is defined as:

$$f_2(\mathbf{x}) = -33 + \sum_{i=1}^{n} \left[\frac{x_i^2}{10} - \cos(2\pi\, x_i) + 1 \right], \tag{7}$$

its minimum value is -33 for $\mathbf{x} = [0, 0, \ldots, 0]$.

The Ackley function is defined as:

$$f_3(\mathbf{x}) = \frac{1}{20} \left\{ e - \exp\left[\frac{1}{n} \sum_{i=1}^{n} \cos(2\pi\, x_i) \right] \right\} - 6 - \exp\left[-\frac{1}{5} \sqrt{\frac{1}{n} \sum_{i=1}^{n} x_i^2} \right], \tag{8}$$

its minimum value is -7 with $\mathbf{x}$ also equal to $\mathbf{x} = [0, 0, \ldots, 0]$. These three functions are scaled with respect to the three ones defined in [11] in order to keep their amplitudes within the range of half precision FP numbers (see Table 1). A summary of these three functions is described in Table 2.

All functions were programmed in single precision FP (float in C) arithmetic.

Table 2. The three test functions used in this work. The search space was set to $[-10, 10]$, thus the shown values are the extreme possible values that the functions could take, also the minimum value is shown at the optimum solution $\mathbf{x} = [0, \ldots, 0]$, and the evaluation at $\mathbf{x} = [1, \ldots, 1]$ is shown in the last column.

Function	$\mathbf{x} = [10, \ldots, 10]$	$\mathbf{x} = [-10, \ldots, -10]$	$\mathbf{x} = [0, \ldots, 0]$	$\mathbf{x} = [1, \ldots, 1]$
Rosenbrock	10891.29	7291.29	0.39	4.00
Rastrigin	67.00	67.00	−33.00	−32.00
Ackley	−6.14	−6.14	−7.00	−6.82

The number of bits used for the integer and fractional parts for the simulations in fixed point arithmetic is shown in Table 3. The number of bits in the integer part is set according to Table 2 because the maximum number in the third column in Table 3 must be greater than the maximum extreme value shown in Table 2.

Table 3. Calculation of the number of bits in the integer part for the simulations using fixed point arithmetic. Numbers shown here must be greater than the corresponding ones in Table 2 to permit the optimization operations for differential evolution (DE).

Functions	Bits Integer Part	Max. Value	Bits Fractional Part
Rosenbrock	14	$2^{14} = 16384$	1–17
Rastrigin	7	$2^{7} = 128$	1–24
Ackley	5	$2^{5} = 32$	1–26

The resulted statistics for the simulations using 100 runs per bit in the fractional part and FP16 arithmetic are shown in Tables 4–6, for the Rosenbrock, Rastrigin, and Ackley functions, respectively. In those tables the statistics for the number of generations and the obtained function values are shown. The used number of bits in the integer part are shown in Table 3. These number of bits in the integer part were calculated from data in Table 2, for example, for the Rosenbrock function in Table 2 the maximum obtained value function is 10891.29, thus the number of bits for the integer part must be greater than this number, therefore 14 bits were selected because $2^{14} = 16,384 > 10,891.29$. The corresponding variable values for the minimum for each function for the FP16 simulations are shown in Table 7. The obtained mean value for the FP16 simulation for the Rosenbrock function is 0.391538 (see at the end of sixth column in Table 4). The equivalent mean for the fixed point arithmetic is 0.391079 at 11 bits in the fractional part; the associated variable values at this simulation with 11 bits is also shown in Table 7. The same procedure was repeated for the results for the Rastrigin and Ackley functions and are also shown in Table 7.

Table 4. Statistics of the 100 runs per bits used in the fractional part for the fixed point arithmetic and for the Rosenbrock function (14 bits were used for the integer part). Results for 100 runs for the FP16 are also shown. g represents the number of generations.

Bits	$\bar{g}$	$\sigma(g)$	$\min(g)$	$\max(g)$	$\bar{f}_1$	$\sigma(f_1)$	$\min(f_1)$	$\max(f_1)$
1	400.0	0.0	400	400	0.005	0.05	0.000	0.500
2	400.0	0.0	400	400	0.250	0.00	0.250	0.250
3	400.0	0.0	400	400	0.375	0.00	0.375	0.375
4	400.0	0.0	400	400	0.375	0.00	0.375	0.375
5	400.0	0.0	400	400	0.375	0.00	0.375	0.375
6	400.0	0.0	400	400	0.390625	0.00000	0.390625	0.390625
7	400.0	0.0	400	400	0.393360	0.00375	0.390625	0.398438
8	400.0	0.0	400	400	0.394765	0.00145	0.390625	0.398438
9	400.0	0.0	400	400	0.394472	0.00137	0.390625	0.396484
10	400.0	0.0	400	400	0.393555	0.00153	0.390625	0.396484
11	400.0	0.0	400	400	0.391079	0.00098	0.390137	0.395020
12	400.0	0.0	400	400	0.390174	0.00025	0.389893	0.391602
13	400.0	0.0	400	400	0.390064	0.00040	0.390015	0.394043
14	370.02	25.73	298	400	0.390012	2.19×10^{-5}	0.389954	0.390076
15	337.72	25.02	280	400	0.390029	2.17×10^{-5}	0.389984	0.390106
16	333.27	25.73	278	400	0.390040	3.07×10^{-5}	0.389999	0.390167
17	330.18	23.06	259	396	0.390046	2.79×10^{-5}	0.390007	0.390152
FP16	400.0	0.0	400	400	0.391538	0.00040	0.390869	0.392578

Table 5. Simulation results for Rastrigin function. Statistics of the 100 runs per bits used in the fractional part for the fixed point arithmetic (7 bits were used for the integer part). Results for 100 runs for the FP16 are also shown. g is the number of generations.

Bits	$\bar{g}$	$\sigma(g)$	$\min(g)$	$\max(g)$	$\bar{f}_2$	$\sigma(f_2)$	$\min(f_2)$	$\max(f_2)$
1	200.0	0.0	200	200	−33.0	0.0	−33.0	−33.0
2	200.0	0.0	200	200	−33.0	0.0	−33.0	−33.0
3	200.0	0.0	200	200	−33.0	0.0	−33.0	−33.0
4	200.0	0.0	200	200	−33.0	0.0	−33.0	−33.0
5	200.0	0.0	200	200	−33.0	0.0	−33.0	−33.0
6	200.0	0.0	200	200	−32.9917	0.00784	−33.0000	−32.9844
7	200.0	0.0	200	200	−32.9923	0.00136	−33.0000	−32.9844
8	200.0	0.0	200	200	−32.9960	0.00055	−32.9961	−32.9922
9	200.0	0.0	200	200	−32.9979	0.00101	−32.9980	−32.9883
10	200.0	0.0	200	200	−32.9988	0.00113	−32.9990	−32.9883
11	200.0	0.0	200	200	−32.9993	0.00077	−32.9995	−32.9936
12	200.0	0.0	200	200	−32.9996	0.00033	−32.9998	−32.9976
13	200.0	0.0	200	200	−32.9997	0.00031	−32.9999	−32.9971
14	200.0	0.0	200	200	−32.9997	0.00042	−32.9999	−32.9964
15	200.0	0.0	200	200	−32.9998	0.00029	−33.0000	−32.9981
16	200.0	0.0	200	200	−32.9997	0.00090	−33.0000	−32.9915
17	200.0	0.0	200	200	−32.9997	0.00080	−33.0000	−32.9938
18	200.0	0.0	200	200	−32.9997	0.00066	−33.0000	−32.9938
19	200.0	0.0	200	200	−32.9998	0.00066	−33.0000	−32.9938
20	200.0	0.0	200	200	−32.9997	0.00070	−33.0000	−32.9938
21	200.0	0.0	200	200	−32.9997	0.00070	−33.0000	−32.9938
22	200.0	0.0	200	200	−32.9997	0.00071	−33.0000	−32.9938
23	200.0	0.0	200	200	−32.9997	0.00071	−33.0000	−32.9938
24	200.0	0.0	200	200	−32.9997	0.00079	−33.0000	−32.9928
FP16	200.0	0.0	200	200	−32.9997	0.00313	−33.0000	−32.9688

Table 6. Simulation results for Ackley function. Statistics of the 100 runs per bits used in the fractional part for the fixed point arithmetic (5 bits were used for the integer part). Results for 100 runs for the FP16 are also shown. g is the number of generations.

Bits	$\bar{g}$	$\sigma(g)$	$\min(g)$	$\max(g)$	$\bar{f}_3$	$\sigma(f_3)$	$\min(f_3)$	$\max(f_3)$
1	200.0	0.0	200	200	−7.00000	0.00000	−7.00000	−7.0000
2	200.0	0.0	200	200	−6.95500	0.14381	−7.00000	−6.5000
3	200.0	0.0	200	200	−6.82625	0.06128	−6.87500	−6.75000
4	200.0	0.0	200	200	−6.93750	0.00000	−6.93750	−6.93750
5	200.0	0.0	200	200	−6.96875	0.00000	−6.96875	−6.96875
6	200.0	0.0	200	200	−6.98406	0.00312	−6.98438	−6.95313
7	200.0	0.0	200	200	−6.99219	$5.08 \times 10-7$	−6.99219	−6.99219
8	200.0	0.0	200	200	−6.99609	$8.47 \times 10-8$	−6.99609	−6.99609
9	200.0	0.0	200	200	−6.99748	0.00566	−6.99805	−6.94141
10	200.0	0.0	200	200	−6.99900	$9.76 \times 10-5$	−6.99902	−6.99805
11	200.0	0.0	200	200	−6.99950	$9.63 \times 10-5$	−6.99951	−6.99902
12	200.0	0.0	200	200	−6.99850	0.00811	−6.99976	−6.94214
13	200.0	0.0	200	200	−6.99990	$5.93 \times 10-5$	−6.99988	−6.99939
14	108.05	12.28	82	161	−6.99990	$3.79 \times 10-5$	−6.99994	−6.99963
15	85.25	3.85	76	94	−6.99990	$1.99 \times 10-4$	−6.99997	−6.99796
16	83.58	3.37	76	91	−6.99993	$2.91 \times 10-5$	−6.99997	−6.99973
17	82.23	3.26	75	93	−6.99992	$4.84 \times 10-5$	−6.99996	−6.99961
18	82.06	3.62	75	93	−6.99992	$9.82 \times 10-5$	−6.99997	−6.99899
19	81.78	3.16	75	88	−6.99990	$1.61 \times 10-4$	−6.99997	−6.99878
20	81.53	3.14	72	88	−6.99991	$1.16 \times 10-4$	−6.99996	−6.99887
21	81.31	3.31	75	91	−6.99991	$1.00 \times 10-4$	−6.99998	−6.99907
22	81.68	3.58	73	90	−6.99991	$8.37 \times 10-5$	−6.99997	−6.99929
23	81.59	3.41	74	90	−6.99991	$8.64 \times 10-5$	−6.99996	−6.99927
24	81.49	3.20	73	89	−6.99991	$1.10 \times 10-4$	−6.99996	−6.99904
25	81.80	3.23	73	93	−6.99991	$1.10 \times 10-4$	−6.99997	−6.99904
26	81.69	3.26	73	93	−6.99991	$1.10 \times 10-4$	−6.99997	−6.99904
FP16	66.5	15.54	49	106	−6.99711	0.00172	−7.00000	−6.99609

Table 7. Variables values for the minimum function value for FP16 simulation, and the integer arithmetic simulation. The shown numbers 11, 12, and 11 correspond to the used bits in the fractional part for integer arithmetic, which also correspond to the same mean of FP16 results for each function in Tables 4–6.

	Rosenbrock		Rastrigin		Ackley	
Bits	**FP16**	**11**	**FP16**	**12**	**FP16**	**11**
$\min(f)$	0.39087	0.39111	−33.0000	−32.9998	−7.0000	−6.9995
x_1	−0.003113	−0.000488	−0.000257	−0.000244	0.011742	−0.000488
x_2	−0.014801	−0.006836	−0.004440	−0.000244	−0.008049	−0.000488
x_3	−0.020218	−0.027832	0.017883	−0.001709	−0.009605	0.003418
x_4	−0.048187	−0.071289	0.003246	−0.000244	−0.002329	0.000000
x_5	−0.077759	−0.101562	0.001313	0.001221	−0.001261	0.000977
x_6	−0.147705	−0.153809	−0.002254	0.000000	0.000976	−0.001953
x_7	−0.288574	−0.318359	0.000988	0.000977	0.006023	−0.000977
x_8	−0.471924	−0.544922	−0.014542	−0.001709	0.005493	0.000000
x_9	−0.720215	−0.806641	−0.008965	0.000488	0.004948	0.001953
x_{10}	−0.934082	−0.953613	0.000543	0.000488	−0.001174	−0.001465

5. Discussion

With the simulation results shown in Tables 4–7 it is confirmed that the heuristic DE can be executed in fixed point arithmetic or half precision FP numbers.

As one can see in Tables 4–6 not all the fractional numbers of bits are necessary with a given application. From Table 7 same results for FP16 numbers can be obtained with numbers 14.11, 7.12, and 5.11 for the scaled Rosenbrok, Rastrigin, and Acklen functions.

About the precision obtained in the solution using FP16 or integer arithmetic. The defined machine epsilon value is that such when $\epsilon \neq 1 + \epsilon$. In most of the modern

microprocessors (that use two's complement arithmetic) this machine epsilon value for each data type is shown in Table 8.

Table 8. Machine epsilon values for the different floating point numbers, for a general integer number of n bits in the fractional part, and also for the integer arithmetic of results shown in Table 7.

Data Type	Machine Epsilon Value	Precision Bits
double	$2.220446 \cdot 10^{-16} \approx 2^{-52}$	53
float	$1.192093 \cdot 10^{-7} \approx 2^{-23}$	24
FP16	$9.765625 \cdot 10^{-4} \approx 2^{-10}$	11
n bits fractional part	2^{-n}	n
11 bits	$4.882813 \cdot 10^{-4} = 2^{-11}$	11
12 bits	$2.441406 \cdot 10^{-4} = 2^{-12}$	12

The *precision bits* is one bit more than the positive exponent of epsilon in floating point types and equal to the number of bits used in the fractional part in integer arithmetic.

Roughly, one cannot expect a result in an optimization problem beyond the precision of the machine epsilon. Thus, using FP16 numbers will give precision in the result at most 9.765625×10^{-4}. Or using an integer number $a.b$, the result will have at most a precision of 2^{-b}. This means also that using FP16 numbers the heuristic, DE in this case, will finish early compared to using single or double precision floating point numbers. In the experiment in this work the DE's stop condition was set equal to 10^{-4}. It is expected that using a smaller stop condition the heuristic will finish in more generations but then is necessary to change to other number types.

One possible application of using FP16 numbers of integer arithmetic could be to obtain first a low precision result within the precision given by the used type numbers (see Table 8). If a bigger precision is required, then a traditional mathematical algorithm, such as the Newton method, could be used. The starting solution for the Newton method will be the previous obtained low resolution solution.

Of course if FP16 numbers of integer arithmetic are used, the application should work at the precision results given by those type numbers. Finally, this behavior must be analyzed in advance for a given application.

For all the simulations the DE's stop condition was set equal to 0.0001. This number in 3.28 notation is equal to 0x000068db (it is a hexadecimal number of 32 bits), and this number can be written by convenience with the binary point as 0x0.00068db. The 13 bits after the binary point are all zeros, thus the stop condition is equal to zero for less than 13 bits used in the fractional part, as one can confirm in Tables 4 and 6 where the simulations show the maximum number of iterations and the stop condition is not taken into account for lesser than and equal to 13 bits.

For the use of fixed point arithmetic in DE, it is critical to know in advance the range of values for the function to optimize. Here the extremes values of the search space were used to know those quantities. In a practical task, it could be tried with the extremes and perhaps other points, on a very coarse grid, to evaluate the function to optimize. The same procedure should be applied to use FP16 numbers.

DE core (in Algorithm 1) uses one difference and one multiplication, thus there is not a numerical problem to be used with fixed point arithmetic or FP16 numbers.

A naive implementation of fixed point arithmetic with a word length of 32 bits is not required, in general. As one can see in Table 4, the same results using 14–17 bits in the fractional part for the Rosenbrock function are obtained. The same applies from results in Table 5 for the Rastringin function for 11–24 bits, and in Table 6 for the Ackley function from 13 to 26 bits in the fractional part.

A future work will be the design in the hardware of DE, which should include the random number generator that can be optimized to use directly the generated bits without FP divisions, as is suggested in [10]. This idea of this design also could be used in software

within each core of a GPGPU. Also an interesting idea is to incorporate a random number generator based in chaos [12], which is easy to implement.

6. Conclusions

The DE optimization heuristic was analyzed under its implementation with fixed point arithmetic and half precision floating point arithmetic. Results were shown in software simulation with three multimodal functions: Rosenbrock, Rastrigin, and Ackley in 10 dimensions. To apply these arithmetic representations, it is necessary first to know how to scale the function values to be inside the ranges of FP16 numbers. It is suggested to use the extreme search values to have an idea of those range function values. If this point is solved, DE can be perfectly used in these arithmetics.

Still is possible to optimize the DE algorithm in the pseudo random number generator, without using FP arithmetic. This analysis is required if DE will be embedded in hardware inside a circuit chip or in massive parallel versions in GPGPUs.

Funding: This research received no external funding.

Acknowledgments: The author would like to thank the anonymous reviewers for their valuable comments which have helped to improve the quality of this article.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Higham, N.; Pranesh, S. Simulating Low Precision Floating-Point Arithmetic. *SIAM J. Sci. Comput.* **2019**, *41*, C585–C602. [CrossRef]
2. Kalamkar, D.D.; Mudigere, D.; Mellempudi, N.; Das, D.; Banerjee, K.; Avancha, S.; Vooturi, D.T.; Jammalamadaka, N.; Huang, J.; Yuen, H.; et al. A Study of BFLOAT16 for Deep Learning Training. *arXiv* **2019**, arXiv:1905.12322.
3. Lee, S.; Shi, C.; Wang, J.; Sanabria, A.; Osman, H.; Hu, J.; Sánchez-Sinencio, E. A Built-In Self-Test and In Situ Analog Circuit Optimization Platform. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2018**, *65*, 3445–3458. [CrossRef]
4. Tlelo-Cuautle, E.; Rangel-Magdaleno, J.; de la Fraga, L. *Engineering Applications of FPGAs*; Springer: Berlin/Heidelberg, Germany, 2016. [CrossRef]
5. IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*; IEEE Computer Society: Washington, DC, USA, 2008. [CrossRef]
6. Goldberg, D. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Comput. Surv.* **1991**, *23*, 5–48. [CrossRef]
7. Guerra-Gómez, I.; Tlelo-Cuautle, E.; De la Fraga, L. Richardson extrapolation-based sensitivity analysis in the multi-objective optimization of analog circuits. *Appl. Math. Comput.* **2013**, *222*, 167–176. [CrossRef]
8. Hansen, N. Comparisons Results among the Accepted Papers to the Special Session on Real-Parameter Optimization at CEC-05. 2006. Available online: http://www.ntu.edu.sg/home/epnsugan/index_files/CEC-05/compareresults.pdf (accessed on 3 February 2021).
9. Zielinski, K.; Laur, R. Stopping criteria for differential evolution in Constrained Single-Objective Optimization. In *Advances in Differential Evolution*; Springer: Berlin/Heidelberg, Germany, 2008.
10. Lemire, D. Fast Random Integer Generation in an Interval. *ACM Trans. Model. Comput. Simul.* **2019**, *29*. [CrossRef]
11. Zhang, X.; Kang, Q.; Cheng, J.; Wang, X. A novel hybrid algorithm based on Biogeography-Based Optimization and Grey Wolf Optimizer. *Appl. Soft Comput.* **2018**, *67*, 197–214. [CrossRef]
12. de la Fraga, L.; Torres-Pérez, E.; Tlelo-Cuautle, E.; Mancillas-López, C. Hardware Implementation of pseudo-random number generators based on chaotic maps. *Nonlinear Dyn.* **2017**, *90*, 1661–1670. [CrossRef]