Software-Defined Number Formats for High-Speed Belief Propagation

Amir Sabbagh Molahosseini, JunKyu Lee, and Hans Vandierendonck

Abstract—This paper presents the design and implementation of Software-Defined Floating-Point (SDF) number formats for high-speed implementation of the Belief Propagation (BP) algorithm. SDF formats are designed specifically to meet the numeric needs of the computation and are more compact representations of the data. They reduce memory footprint and memory bandwidth requirements without sacrificing accuracy, given that BP for loopy graphs inherently involves algorithmic errors. This paper designs several SDF formats for sum-product BP applications by careful analysis of the computation. Our theoretical analysis leads to the design of 16-bit (half-precision) and 8-bit (mini-precision) widths. We moreover present highly efficient software implementation of the proposed SDF formats which is centered around conversion to hardware-supported single-precision arithmetic hardware. Our solution demonstrates negligible conversion overhead on commercially available CPUs. For Ising grids with sizes from 100×100 to 500×500, the 16- and 8-bit SDF formats along with our conversion module produce equivalent accuracy to double-precision floating-point format but with 2.86× speedups on average on an Intel Xeon processor. Particularly, increasing the grid size results in higher speed-up. For example, the proposed half-precision format with 3-bit exponent and 13-bit mantissa achieved the minimum and maximum speedups of 1.30× and 1.39× over single-precision, and 2.55× and 3.40× over double-precision, by increasing grid size from 100×100 to 500×500.

Index Terms—Reduced-Precision Floating-Point Number Format, Belief Propagation, Transprecise Computing, Software-Defined Number Format.

1 Introduction

Belief Propagation (BP) [1] is a probabilistic inference algorithm on graphs that has been widely used in communications [2], [3], machine learning [4], [5], cooperative localization [6] and mobile robots navigation [7]. The exact inference for large-scale graphs is computationally intractable in practice, requiring efficient implementation of BP for such large-scale applications [8], [9]. Two main paths have been considered for improving the efficiency of BP implementations, namely, message scheduling and parallel processing. For example, asynchronous message scheduling techniques such as residual BP (RBP) [10] and parallel implementation on GPUs [11] have been widely used. However, these methods are still limited for real-time applications on resource-constrained devices [9] as well as large graphs, since BP requires considerable data movements from and to memory.

The number of bits required for hardware floating-point (FP) formats (e.g. 64 and 32 bits for double- and single-precision, respectively [12]) has a direct effect on the capable amount of data transfer for BP applications, given memory bandwidth budget. Since BP is an iterative approximate algorithm for loopy graphs, high-precision arithmetic such as single- or double-precision is often not needed. We notice that (i) no research has been done on the use of reduced-precision FP formats for BP so far and (ii) the computational speed for BP mainly depends on the memory traffic since BP is a memory-bound kernel [13]. Therefore, identifying the minimal bit-width number format that maintains sufficient application accuracy is the key consideration to accelerate BP.

Reduced-precision FP formats such as BFloat [14], DLFloat [15], Flexpoint [16], TFP [17], and Flytes [18] were mostly proposed for deep learning applications to achieve high speed and energy efficiency for training and inference tasks. However, there are limited works on the use of customized FP formats for graph processing algorithms. In this regard, [19] explored customized half-precision FP formats for the PageRank [19]. The motivation of [19] was to reduce memory traffic, given a prescribed accuracy. [20] proposed a customized precision mantissa segmentation approach for PageRank to accelerate access to memory by using smaller number of bits for data. Moreover, [21] explored the feasibility of using reduced-precision graph processing on GPU tensor cores.

This paper, for the first time, seeks FP formats customized to the sum-product BP algorithm to accelerate its applications by improving the data traffic efficiency. Our proposed Software-Defined Floating-Point (SDF) number formats designed in 8- and 16-bit frames are used for *memory storage of BP normalized messages* to improve data communication efficiency given a memory bandwidth as they are fetched for the message updates. To maximize the benefits of reduced precision number formats, we implement highly efficient conversion modules to ease conversions between single-precision and SDF number formats on off-

This work has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 101031148.

- *A.S. Molahosseini and H. Vandierendonck are with the School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, UK.*
  E-mail: {A.SabbaghMolahosseini, H.Vandierendonck }@qub.ac.uk
- *J. Lee is with the Institute for Analytics and Data Science, University of Essex, UK. The work has been done at University of Essex and Queen's University of Belfast.*
  E-mail: j.lee@essex.ac.uk

the-shelf processors. In our case study used for Ising grids, both the proposed half- and the mini-precision formats achieve equivalent accuracy to the double-precision format. This work shows that even 8-bit is sufficient for practical sum-product BP applications, since most BP applications are approximate, producing higher algorithmic error than rounding error. In other words, in BP for loopy graphs, small convergence thresholds and high-precision arithmetic do not help BP increase its accuracy, as shown in Section 4.1. The rounding error for 2D sum-product BP over Ising grids is limited since the maximum edge degree in Ising grids is four while in other graphs it could be millions.

Therefore, the main contributions of this paper are fourfold:

- Analysis of required bit width for exponent and mantissa for sum-product BP applications based on the application properties that can produce equivalent accuracy to double-precision format.
- Application condition of our SDF formats that can produce equivalent accuracy to double-precision format (i.e., higher algorithmic error of approximate BP algorithm than rounding error of our SDF formats).
- Effective implementation to support SDF formats on off-the-shelf processors,
- Experimental evaluation of SDF formats on the accuracy and speed-up on off-the-shelf processors, compared to single- and double-precision.

We discuss a brief introduction to BP and FP arithmetic in Section 2, the proposed SDF formats in Section 3, the experimental evaluation of SDF formats on performance and accuracy in Section 4, and conclude the paper in Section 5.

2 Background

2.1 Belief Propagation

Graph G is a pair of (V, L) where V and L are called node and link (edge) sets, respectively. $|V|$ indicates the total number of nodes, and Γ_i means the set of neighbors of node i. Moreover, $\Gamma_{i\setminus j}$ indicates neighbors set of node i excluding the node j. Probabilistic graphical models [1] use a graph-based representation for encoding joint probability distribution over a set of variables, where nodes are random variables and edges are probabilistic relations between nodes. This probabilistic model can be used in a range of applications for inference tasks to reach a conclusion about a phenomenon based on its model. They include two major classes, Bayesian networks (directed graphs) and Markov random fields (undirected graphs). A factor graph is often used as an alternative representation of Markov random fields to represent probability distributions with factors of nodes (i.e., a set of functions of the node variables) [22]. For example, a factor function $\phi_i(x_i)$ maps a random variable x_i to a positive real number [23]. Therefore, Markov random field based factor graphs can represent joint probability distribution over all nodes, i.e. $P(x_1, x_2, \ldots, x_n)$. A practical application of the inference over factor graphs often requires to compute the marginal distribution of each node's random variable, i.e. $P(x_i)$.

BP is an inference algorithm over a factor graph, that can calculate these marginals through message passing. BP iteratively exchanges messages between edges and nodes until convergence, and then marginals, called beliefs, can be computed. The BP is communication-intensive, and its performance mainly depends on the message communication. The Loopy Belief Propagation (LBP) is a synchronous method which updates every edge in every iteration resulting in high latency and slow convergence. On the other hand, asynchronous methods such as Residual Belief Propagation (RBP) focus on updates of the less-stable graph regions to speed up convergence [11], [24]. Algorithm 1 and Table 1 present RBP with complete implementation details including normalization. In Algorithm 1, A_i indicates a set of finite alphabet, and its members depend on the application. It should be noted that BP messages are normalized (i.e. $\sum_{x_i} m_{k\to i}(x_i) = 1$) for numerical stability and preventing overflow/underflow after each message update [25], [26]. Similarly, marginals should be normalized by $1/Z$ coefficient where Z is the total sum of total intermediate marginals of x_i. The coefficient Z is usually called the partition function. More details about Belief Propagation and its various message scheduling methods can be found in [1], [27].

It should be noted that in practice, Algorithm 1 will be implemented using floating-point arithmetic in computing systems since messages, node and edge factors are fractional numbers. Generally, the order of operations is important due to rounding errors in floating-point arithmetic. However, in the context of BP for loopy graphs, the order is arbitrary since it is of negligible significance due to the substantial algorithmic error in comparison to rounding errors.

2.2 Floating-Point Number Format

The IEEE 754-2019 standard FP system is defined by a radix b (2 or 10), an exponent range $[e_{min}, e_{max}]$ and a precision p. In this system, a number representation is defined by a sign bit s, an exponent e and a significand m where $0 \leq m < b$. The corresponding value is equal to [40]:

$$X = (-1)^s \times b^e \times m \tag{1}$$

The significand is a number represented by a digit string of the form $d_0 \cdot d_1 d_2 \ldots d_{p-1}$ where d_i is an integer digit $0 \leq d_i < b$, and p is the number of digits in the significand ([40]). The limits on the exponent and significand determine the dynamic range and precision of the format. The IEEE 754-2019 standard includes double-precision (64-bit), single-precision (32-bit) and half-precision (16-bit) formats. The double- and single-precision formats have been used widely in computing systems [28], [29]. The IEEE 754 standard requires correct rounding for the four floating-point arithmetic operations such as addition, subtraction, multiplication, and division. In other words, the floating-point arithmetic result should be identical to the one obtained from the final rounding after the exact calculation [12].

3 Software-Defined Number Format for BP

This section presents our approach for designing customized 16- and 8-bit floating-point formats for BP messages. We seek the minimum bit-width of the exponent

Algorithm 1: Residual Belief Propagation [10], [11]

```
   Input: Graph: $G = (V, L), |V| = n$
   Random Variables: $x_1, x_2, \ldots, x_n$ $(x_i \in A_i)$
   Node Factors: $\phi_i : A_i \to R^+ | i \in V$
   Edge Factors: $\psi_{i,j} : A_i \times A_j \to R^+ | (i,j) \in L$
   Convergence threshold: $\epsilon$
   Output: Marginal Distribution: $P(x_i)$
1  Begin
2  for $(i,j) \in L$ do
3      for $x_j \in A_j$ do
4          $m_{i\to j}(x_j) = \frac{1}{|A_j|}$
5  rpq=priority_queue(L, default_residual) while $top\_residual > \epsilon$ do
6      $(i,j) = rpq.top()$
7      for $x_j \in A_j$ do
8          $m^{temp}_{i\to j}(x_j) = \sum_{x_i \in A_i} \psi_{i,j}(x_i,x_j)\phi_i(x_i)\prod_{k\in\Gamma_{i\setminus j}} m_{k\to i}(x_i)$
9      $sum = \sum_{x_j \in A_j} m^{temp}_{i\to j}(x_j)$
10     for $x_j \in A_j$ do
11         $m_{i\to j}(x_j) = \frac{m^{temp}_{i\to j}(x_j)}{sum}$
12     for $(j,h) \in \{outgoing\ edges\ from\ node\ j\}$ do
13         $sum = 0$
14         for $x_h \in A_h$ do
15             $m^{temp}_{j\to h}(x_h) =$
16             $\sum_{x_j \in A_j} \psi_{j,h}(x_j,x_h)\phi_j(x_j)$
17             $\prod_{k\in\Gamma_{j\setminus h}} m_{k\to j}(x_j)$
18             $sum = sum + m^{temp}_{j\to h}(x_h)$
19         for $x_h \in A_h$ do
20             $m^{temp}_{j\to h}(x_h) = \frac{m^{temp}_{j\to h}(x_h)}{sum}$
21         $sum = 0$
22         for $x_h \in A_h$ do
23             $sum = sum + (m^{temp}_{j\to h}(x_h) - m_{j\to h}(x_h))$
24         $rpq(j,h).residual = sum$
25     $top\_residual = rpq.top().residual$
26 for $i \in V$ do
27     $sum = 0$
28     for $x_i \in A_i$ do
29         $P(x_i) = \phi_i(x_i)\prod_{k\in\Gamma_i} m_{k\to i}(x_i)$
           $sum = sum + P(x_i)$
30     for $x_i \in A_i$ do
31         $P(x_i) = \frac{P(x_i)}{sum}$
32 End
```

TABLE 1
Comments on Algorithm 1 (RBP details)

Lines	Description
2-4	Initializing all messages to the uniform distribution (assigning the same probability to each message according to the size of its random variable category)
5	Creating a priority_queue to store **message's residual** with the same default residual value for all elements. The residuals will be updated during the while loop, and will be used for selecting which *message* to be updated next
7	Selecting the message with top residual from priority queue to update
8-9	Updating $m_{i\to j}$ by considering incoming messages to node i
10-12	Normalizing $m_{i\to j}(x_j)$ for all $x_j \in A_j$
13-20	Computing normalized messages which are affected by $m_{i\to j}$ (i.e. outgoing messages from node j) to achieve their updated residuals
21-24	Computing residuals of affected messages and updating them in priority queue
25	The top residual will be used in while condition to check the convergence by comparing it with ϵ
26-31	Computing normalized marginals for all nodes

that allows representing the entire dynamic range of the algorithm.

3.1 Application Specifications

We consider Ising grids since they are standard datasets for evaluating BP algorithms [10], [11]. Ising grids are used in statistical physics for modelling the interaction between atoms in a physical system [1]. Considering n graph nodes $x_1, x_2, \ldots, x_n$, a binary random variable $x_i \in \{0,1\}$ defines the atom's spin which is associated with each atom [1]. Then, each node can take in isolation a node potential (i.e. node factor) that works as a label for that node. Besides, edge potentials (i.e. edge factors) are employed to model the interaction between neighboring nodes. The messages can be calculated based on node and edge factors. The goal is to calculate nodes' marginals (i.e. $P(x_i)$) that can be achieved based on final converged messages. There are many types of node and edge factors that can be used to model Ising grids. Here, we consider the parameters presented in [10], [11]. In this model, each random variable takes a binary value $x_i \in \{0,1\}$, and node factors are randomly sampled as follows:

$$\phi_i(x_i) = \begin{cases} p_i & x_i = 0 \\ 1 - p_i & x_i = 1, \end{cases} \quad (2)$$

where $p_i \in (0,1]$. Besides, the edge factors between nodes i and j can be computed as follows:

$$\psi_{i,j}(x_i,x_j) = \begin{cases} e^{\lambda_{i,j}c} & x_i = x_j \\ e^{-\lambda_{i,j}c} & x_i \neq x_j, \end{cases} \quad (3)$$

where $\lambda_{i,j} \in [-0.5, 0.5]$, and is randomly generated for each edge factor (i.e., for each combination of nodes i and j, a random $\lambda_{i,j}$ will be used to produce the corresponding edge factor). Moreover, $c \in \{2,3\}$ determines the difficulty of the inference task and is the same for all edge factors in (3). These ranges have typically been considered for the evaluation of Ising grids [10], [11]. The $\lambda_{i,j}c$'s are called the energy functions since the probability of a physical state in statistical physics depends inversely on the energy [1]. Therefore, randomly selected node and edge factors according to (2) and (3) form the dataset. In other words, for a $n \times n$ model, we have n^2 nodes, and for each node, there are two node factors (one for $x_i = 0$ and the other for $x_i = 1$). For the internal nodes, each has four edges, and for each edge, there are four possible edge factors since $x_i, x_j \in \{0, 1\}$ but with only two possible values as given by (3).

3.2 Analysis of the Exponent Range

The BP requires special data structures for handling node factors, edge factors, messages and marginals [1]. The node and edge factors will be extracted from the dataset, and they are constant while the program is running. The marginals will be computed once at the final stage after all messages are successfully converged. Therefore, we consider the message communications only to derive the required bit-width for exponent and mantissa, since they are the main cause of high memory traffic. Updated BP messages in each iteration are stored in memory and re-fetched from memory at the next iteration for calculating new messages.

3.2.1 Theoretical Study

This work focuses on sum-product RBP over pairwise Markov random fields based on Ising grid datasets with the same assumptions on initial values as used in [11]. All of the arithmetic operations considered in this section are rounded floating-point operations. In this regard, the intermediate messages $m_{i\to j}^{temp}$ in Algorithm 1 are computed as follows [11]:

$$m_{i\to j}^{temp}(x_j) = \sum_{x_i \in A_i} \psi_{i,j}(x_i, x_j)\phi_i(x_i) \prod_{k \in \Gamma_{i\setminus j}} m_{k\to i}(x_i). \quad (4)$$

Then, the computed intermediate messages are normalized. The normalization will be done after each message update to have $\sum_{x_i} m_{k\to i}(x_i) = 1$, which results in improving numerical stability and preventing overflow/underflow in the next iterations [25], [26]. The normalized messages' values should be transferred between BP iterations until convergence. The main BP formula that calculates the normalized messages for binary pairwise models, i.e. $x_j \in \{0, 1\}$, is as follows [11] (its calculation is indicated on lines 10-12 in Algorithm 1):

$$m_{i\to j}(x_j) = \frac{m_{i\to j}^{temp}(x_j)}{m_{i\to j}^{temp}(x_j) + m_{i\to j}^{temp}(1 - x_j)}. \quad (5)$$

Then, the max norm based on normalized messages and the previous iteration messages can be computed to check the convergence with convergence threshold ϵ [10], [11]. Finally, after program convergence, the final marginals (a.k.a. beliefs) can be calculated as follows [11]:

$$P(x_i) \propto \phi_i(x_i) \prod_{k \in \Gamma_i} m_{k\to i}(x_i). \quad (6)$$

It should be noted that similar to messages, the normalization can also be applied on marginals as shown in last loops of Algorithm 1. Now, the aim is to determine the lower and upper bounds of the exponent range for the normalized message, $m_{i\to j}(x_j)$, in (5) as follows. First, we assume that the operations in (5) are computed exactly (i.e. ignoring rounding errors), and then we study the effect of rounding errors on it. In this regard, (5) and (4) can be rewritten as follows:

$$m_{i\to j}(x_j) = \frac{1}{1 + \frac{m_{i\to j}^{temp}(1-x_j)}{m_{i\to j}^{temp}(x_j)}}, \quad (7)$$

$$m_{i\to j}^{temp}(x_j) = \sum_{x_i \in A_i} \psi_{i,j}(x_i, x_j) F_{i,j}(x_i), \quad (8)$$

where

$$F_{i,j}(x_i) = \phi_i(x_i) \prod_{k \in \Gamma_{i\setminus j}} m_{k\to i}(x_i). \quad (9)$$

Now, we can get bounds on $m_{i\to j}^{temp}(x_j)$ and $m_{i\to j}^{temp}(1 - x_j)$ based on (8) as follows:

$$m_{i\to j}^{temp}(\hat{x}) = \psi_{i,j}(0, \hat{x})F_{i,j}(0) + \psi_{i,j}(1, \hat{x})F_{i,j}(1) \quad (10)$$

where $\hat{x} = x_j$ or $1 - x_j$. From (3) we know that:

$$e^{-\lambda_{i,j}c} \le \psi_{i,j}(x_i, x_j) \le e^{\lambda_{i,j}c}. \quad (11)$$

Therefore, we can get bounds on (10) considering the minimum and maximum values of $\lambda_{i,j}$ as follows:

$$e^{-0.5c}(F(0) + F(1)) \le m_{i\to j}^{temp}(\hat{x}) \le e^{0.5c}(F(0) + F(1)). \quad (12)$$

(12) implies:

$$e^{-c} \le \frac{m_{i\to j}^{temp}(1 - x_j)}{m_{i\to j}^{temp}(x_j)} \le e^c \quad (13)$$

therefore,

$$\frac{1}{1 + e^c} \le \frac{1}{1 + \frac{m_{i\to j}^{temp}(1-x_j)}{m_{i\to j}^{temp}(x_j)}} \le \frac{1}{1 + e^{-c}}. \quad (14)$$

It can be seen that the values of $F(0)$ and $F(1)$ do not affect the deduction of (13) from (12), as these terms are eliminated in the division since their sum is positive. Therefore, for the purpose of analyzing rounding errors, $F(0)$ and $F(1)$ can be considered as exact values. Moreover, $\psi_{i,j}(x_i, x_j)$ can be affected by rounding errors. Even though $\lambda_{i,j}c$ in (3) may be affected by a rounding error for $c = 3$, $\pm c/2$ is a valid bound for the computed value of $\pm\lambda_{i,j}c$ since ± 0.5 is exactly representable and the division by 2 is exact. Additionally, the computation of the exponential function in (3) introduces rounding errors that can be expressed as $(1+\epsilon)^v$ where $v \ge 1$. Therefore, when considering rounding errors based on Higham error analysis [32], there will be an error factor

- $(1 + \epsilon)^{(2+v)}$ for both $m_{i\to j}^{temp}(x_j)$ and $m_{i\to j}^{temp}(1 - x_j)$,
- $(1 + \epsilon)^{(3+v)}$ for $m_{i\to j}^{temp}(x_j) + m_{i\to j}^{temp}(1 - x_j)$,
- $(1 + \epsilon)^{(6+2v)}$ for the normalized message given by (5),

where ϵ depends on the variable and is bounded by the arithmetic machine epsilon [32]. For $c = 2$ and 3, the values of $\frac{1}{1+e^c}$ and $\frac{1}{1+e^{-c}}$ are far enough from powers of 2 so that the factor $(1+\epsilon)^6$ will not change the exponent. It should be noted that if p in (2) is tiny, the $\psi_{i,j}(0,\hat{x})F_{i,j}(0)$ term in (10) may underflow. But in this case, this term is so small compared to $\psi_{i,j}(1,\hat{x})F_{i,j}(1)$ that it can be regarded as part of a $(1+\epsilon)$ error factor. Thus, the $(1+\epsilon)^6$ is still valid.

It is clear that for $c=2$, $\frac{1}{1+e^c} \approx 0.119$ and $\frac{1}{1+e^{-c}} \approx 0.88$, and for $c=3$, $\frac{1}{1+e^c} \approx 0.047$ and $\frac{1}{1+e^{-c}} \approx 0.95$. Therefore, according to (14), we have:

$$\begin{cases} 0.119 \le m_{i\to j}(x_j) \le 0.89 & c=2 \\ 0.047 \le m_{i\to j}(x_j) \le 0.96 & c=3 \end{cases}. \tag{15}$$

It should be noted that due to the small ϵ value, the $(1+\epsilon)^{(6+2v)}$ factor is included in the outward rounding considered in (15). This implies:

$$E(m_{i\to j}(x_j)) \in \begin{cases} \{-4,-3,-2,-1\} & c=2 \\ \{-5,-4,-3,-2,-1\} & c=3 \end{cases}, \tag{16}$$

where $E(expr)$ is the unbiased exponent of the expression $expr$, computed in floating-point arithmetic. Therefore, it can be seen that the exponents of normalized BP messages are independent of the graph sizes but *dependent on the parameters that determine the edge factors*.

3.2.2 Experimental Evaluation

We experimentally analyzed the BP messages based on some Ising grids datasets in order to check the matching between theoretical findings and experimental results. We considered Ising grids with sizes from 10×10 to 500×500 with node and edge factors as indicated in (2) and (3) where $\lambda \in [-0.5, 0.5]$ and $c \in \{2,3\}$. The exponent range of normalized messages during all algorithm's iterations is as follows:

$$E^{Experimental}(m_{i\to j}(x_j)) \in \begin{cases} \{-4,-3,-2,-1\} & c=2 \\ \{-5,-4,-3,-2,-1\} & c=3 \end{cases}. \tag{17}$$

It can be seen that the experimental study of exponents confirms the validity of theoretical findings. In other words, the experimental evaluation exponents' bounds exactly match those from the theoretical study given in (16).

3.3 Accuracy Analysis of SDF Format

We analyze the accuracy without SDF format quantization first and the accuracy with the quantization later. Our analysis is focused on one-step update for normalized messages, given the messages updated in the previous step.

3.3.1 Without SDF Quantization

This section analyzes the rounding error propagation for the message update processes, (i.e. (4) and (5)). (4) can be represented as dot-product operations as follows:

$$m^{temp}_{i\to j}(x_j) = \sum_{x_i\in A_i} \psi_{i,j}(x_i,x_j)F_{i,j}(x_i). \tag{18}$$

The computation for $F_{i,j}(x_i)$ follows (9). For the computation, the number of neighboring nodes is n_{ne} and the number of multiplications is $(n_{ne}-1)$ (i.e., multiplications between a factor function $\phi_i(x_i)$ and the messages from neighboring nodes except the node j). The rounding error in calculating $F_{i,j}(x_i)$ is as follows [32].

$$\tilde{F}_{i,j}(x_i) \approx F_{i,j}(x_i)(1+\bar{n}_{ne}\epsilon_F), \tag{19}$$

where $|\epsilon_F| \le \epsilon_{ar}$; ϵ_{ar} is an arithmetic machine epsilon value used for computing $F_{i,j}(x_i)$ [12], and $\bar{n}_{ne} = n_{ne}-1$. $\tilde{F}(\cdot)$ in (19) represents the perturbed $F(\cdot)$ due to finite precision arithmetic. We will use the tilde notations for the variables containing rounding errors due to finite precision arithmetic from this point forward. The error analysis in (19) considers the rounding error effects from each message update procedure, given the previous messages. The error in the computed quantity $\tilde{F}_{i,j}(x_i)$ can be represented as:

$$\|F_{i,j}(x_i) - \tilde{F}_{i,j}(x_i)\| \lesssim F_{i,j}(x_i)\bar{n}_{ne}\epsilon_{ar}, \tag{20}$$

The computed $\tilde{m}^{temp}_{i\to j}(x_j)$ can be represented as follows [41]:

$$\begin{aligned} \tilde{m}^{temp}_{i\to j}(x_j) = (...((&\tilde{\psi}_{i,j}(x_i=s_0,x_j)\tilde{F}_{i,j}(x_i)_{(x_i=s_0)}(1+\epsilon_{m(0)}) \\ &+\tilde{\psi}_{i,j}(x_i=s_1,x_j)\tilde{F}_{i,j}(x_i)_{(x_i=s_1)} \\ &(1+\epsilon_{m(1)}))(1+\epsilon_{a(0)})) + ...)(1+\epsilon_{a(n_{st}-2)}) \\ \approx &\sum_{x_i\in A_i} \tilde{\psi}_{i,j}(x_i,x_j)\tilde{F}_{i,j}(x_i)(1+\theta), \end{aligned} \tag{21}$$

where n_{st} represents the number of states (i.e., possible values) in every x_i, s_k represents the k^{th} state of x_i, $\tilde{F}_{i,j}(x_i)_{(x_i=s_k)}$ represents $\tilde{F}_{i,j}(x_i)$ when $x_i = s_k$, $\tilde{\psi}_{i,j}(x_i,x_j) = \psi_{i,j}(x_i,x_j)(1+\bar{\epsilon}_i)$, where $\bar{\epsilon}_i$ is rounding error in computing $\psi_{i,j}(x_i,x_j)$ and $|\theta| \lesssim n_{st}\epsilon_{ar}/(1-n_{st}\epsilon_{ar}) \approx n_{st}\epsilon_{ar}$. For example, in the Ising model stated in Section 3.1, the possible states for x_i are: $x_i \in \{0,1\}$ (i.e., $s_0=0$ and $s_1=1$). Please note that if the sum is done via a balanced binary tree, the error bound would be smaller.

Using (19) and ignoring O(ϵ^2) or higher order impact, the numeric error bound in the computed quantity $\tilde{m}^{temp}_{i\to j}(x_j)$ can be found as:

$$\|m^{temp}_{i\to j}(x_j) - \tilde{m}^{temp}_{i\to j}(x_j)\| \lesssim m^{temp}_{i\to j}(x_j) \times (n_{st}+\bar{n}_{ne})\epsilon_{ar}. \tag{22}$$

In (22), the rounding error, $(n_{st}+\bar{n}_{ne})\epsilon_{ar}$, is the approximation of the maximum rounding error; the rounding error is often smaller than this bound in practice. According to Algorithm 1, normalized messages are obtained by dividing the messages computed based on (21) by sum in Algorithm 1. Given the computed intermediate messages, the sum generates rounding error from the $(n_{st}-1)$ summations :

$$\begin{aligned} \tilde{sum} &\approx sum \times (1+(n_{st}+\tilde{n}_{ne})\epsilon_F)(1+(n_{st}-1)\epsilon_F) \\ &\approx sum \times (1+(2n_{st}+\tilde{n}_{ne}-1)\epsilon_F). \end{aligned} \tag{23}$$

The maximum error of each normalized message occurs when the $\tilde{sum}$ reaches the minimum caused by rounding error while the intermediate messages reaches the maximum, resulting in(24).

$$\begin{aligned} &\|m_{i\to j}(x_j) - \tilde{m}_{i\to j}(x_j)\| \lesssim \\ m_{i\to j}(x_j)(&\frac{1+(n_{st}+\tilde{n}_{ne})\epsilon_{ar}}{1-(2n_{st}+\tilde{n}_{ne}-1)\epsilon_{ar}})(1+\epsilon_{ar}) - m_{i\to j}(x_j). \end{aligned} \tag{24}$$

3.3.2 With SDF Quantization

In SDF, the computed data are quantized when they are written to the memory, and two of the quantized data are fetched for an arithmetic operation with the arithmetic supported in hardware. Therefore, the SDF quantization does not affect the high-level description of the algorithm. SDF quantization is carried out after the messages computed in (22) are normalized. Unlike SDF quantization, computing intermediate messages in (22) utilizes quantizations based on the precision supported by the hardware.

Associated with the quantization error ϵ_{mq}, the numeric error bound in the computed quantity $\tilde{m}_{i\to j}(x_j)$ can be represented as:

$$\begin{aligned} &\|m_{i\to j}(x_j) - \tilde{m}_{i\to j}(x_j)\| \lesssim \\ &m_{i\to j}(x_j)\Big(\frac{1+(n_{st}+\tilde{n}_{ne})\epsilon_{ar}}{1-(2n_{st}+\tilde{n}_{ne}-1)\epsilon_{ar}}\Big)(1+\epsilon_{ar})(1+\epsilon_{mq}) \\ &\qquad -m_{i\to j}(x_j) \\ &\approx m_{i\to j}(x_j)\frac{1+(n_{st}+\tilde{n}_{ne}+1)\epsilon_{ar}+\epsilon_{mq}}{1-(2n_{st}+\tilde{n}_{ne}-1)\epsilon_{ar}} - m_{i\to j}(x_j). \end{aligned} \tag{25}$$

Based on (25), employing SDF formats would not lose accuracy, compared to (22) if arithmetic machine epsilon multiplied by $(n_{st}+n_{ne}+1)$ is relatively higher than the quantization machine epsilon. Please note that the equivalent accuracy condition excludes the effects of undermodelling noise on the accuracy, which will be discussed in the following section. For example, if the effects of quantization errors on the accuracy are negligible, compared to the undermodelling noise, the equivalent accuracy with SDF quantization can be achieved even if the SDF quantization error is noticeably higher than rounding errors.

3.4 Guidelines for SDF Design

Based on our analyses of required bits of exponents and the accuracy depending on the quantization level, we design SDF formats for our case studies based on the guidelines as follows.

3.4.1 Use as Many Bits as Possible for Mantissa

(25) implies that it is desirable to provide as many bits as possible for mantissa for practical applications (i.e., Ising grids, stereo matching, etc.), since the quantization precision is the main factor to determine the numeric error. Based on this guideline, we seek the least sufficient bit-width for the exponent for our SDF formats, and the rest of bits are used for the mantissa part, given that the total number of bits is fixed (e.g. 8 or 16) as the values should be stored in memory in an efficient way. The details will be given in Section 3.5.1.

3.4.2 Balance between Algorithmic and Rounding/Quantization Error

BP is an approximate algorithm for loopy graphs. This implies that using exact arithmetic in BP still results in the error in the computed solution. We name this as undermodelling error e_u, which is not avoidable for an approximate algorithm even with exact arithmetic. Since BP is composed of linear operators, linearity properties such as additivity, $f(\mathbf{a}+\mathbf{b}) = f(\mathbf{a}) + f(\mathbf{b})$, and homogeneity of degree 1, $f(c\times\mathbf{a}) = c\times f(\mathbf{a})$, should hold. The exact arithmetic BP algorithm with the input $\mathbf{x}$, $f_{BP}(\mathbf{x})$, produces the computed beliefs, $\mathbf{y}$. The computed beliefs contain the under-modelling error, $\mathbf{e}_u$. I.e., $\mathbf{y} = \mathbf{y}^* + \mathbf{e}_u$, where $\mathbf{y}^*$ is the desirable beliefs. Utilizing the linearity properties and the rounding error decomposition technique of [33] can represent the computed beliefs, $\tilde{\mathbf{y}}$, using finite precision arithmetic BP, $\tilde{f}_{BP}(\mathbf{x})$, as:

$$\tilde{\mathbf{y}} = \mathbf{y}^* + \mathbf{e}_u + \mathbf{e}_{qr}, \tag{26}$$

where $\mathbf{e}_{qr}$ is error caused by rounding error after arithmetic operations with SDF quantization.

For higher algorithmic error case, (26) implies that the mantissa width can be adjusted according to the algorithmic error $\mathbf{e}_u$. For example, the mantissa bit width can be reduced without compromising the accuracy if $\mathbf{e}_u$ is relatively high, since $\mathbf{e}_u$ is independently additive to $\mathbf{e}_{qr}$ [33]. The $\mathbf{e}_u$ in BP highly depends on the strength of potentials [34] - lower potential, lower $\mathbf{e}_u$. Therefore, the required mantissa bit width is in proportion to reciprocal of potential strength to keep equivalent accuracy to the one with a higher precision format. In our experiments, Tables 4 and 5 demonstrate that a weaker potential ($c = 2$) requires a larger convergence threshold, compared to a stronger potential ($c = 3$). The higher algorithmic error case is highly probable for Ising grids since n_{ne} is relatively small, resulting in small error using quantization data based on (25).

For higher rounding error case, we recommend users to increase the mantissa bit width (i.e., quantization error) to reduce ϵ_{mq} in (25).

In our case studies using $n_{st} = 2$, $\epsilon_{ar} = 2^{-24}$ (i.e., single precision), and ϵ_{mq}= $2^{-k}, k \in 14, 13, 12$, the effect of rounding error to the accuracy is negligible while the quantization error is dominant factor to determine the accuracy. Our case studies demonstrate that the accuracy is not compromised in practice since the quantization error is lower than the algorithmic error for our case studies thanks to a small n_{ne}.

3.5 SDF Format Design and Implementation

This section describes two important aspects of software-defined number formats. The first one is the structure of the SDF number format including its total size as well as each field size. The second is the implementation method.

3.5.1 Number Format

An IEEE 754 standard binary FP format includes a sign bit, biased exponent and normalized mantissa. However, the BP messages are unsigned probabilities, and therefore sign bit is unnecessary for representing them. Therefore, the sign bit can be removed from the messages' coding format. This results in one additional bit for the mantissa to have higher precision. On the other hand, both theoretical (i.e. (16)) and experimental studies (i.e. (17)) of BP normalized messages' exponents showed that exponents are restricted in a limited range which depends on edge factors, and are independent from dataset size.

The theoretical and experimental studies (i.e. (16) and (17)) both suggest that maximally 3 bits are sufficient to represent the exponents of normalized BP messages for $c = 3$, and 2 bits for $c = 2$. Then, after determining the exponent's field size, the remaining bits can be dedicated to the mantissa. To have efficient read and/or write, a multiple of bytes should be considered for the SDF formats. Therefore, we select 16- and 8-bit frames for the proposed FP number formats as shown in Fig. 1 where $EXP \in \{2, 3, 4\}$ and $M \in \{8, 16\}$. It should be noted that the SDF format with a 4-bit exponent (Half(4,12)) is proposed to assess the impact of one fewer mantissa bit in half-precision formats on accuracy and performance. However, in reality, 3 exponent bits are sufficient to cover all the required exponents without any compromise in dynamic range, as indicated by the theoretical and experimental analyses (i.e. (16) and (17)). Due to this, Half (3,13) and Mini (3,5) are suggested. Finally, the Mini (2,6) and Half (2,14) formats are suggested only for the case of $c = 2$, where two bits in the exponent field are adequate according to (16) and (17).

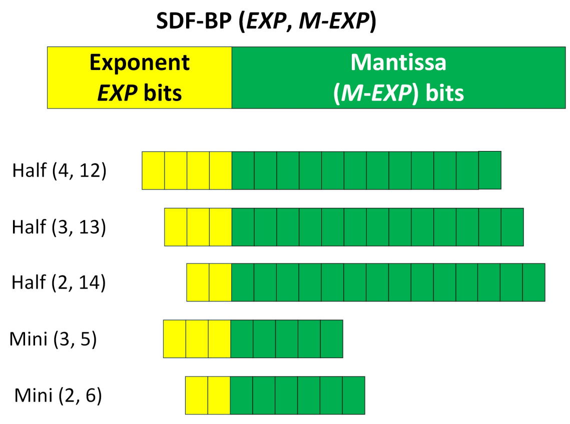

Fig. 1. The Proposed Software-Defined FP formats for BP.

3.5.2 *Implementation*

There are two main options for implementing customized number formats. The first one is to design new circuits and systems to support arithmetic using the new number format. This method includes high time-to-market and changing several layers of computing stack. It is not practical to design new hardware for every application-specific number format. The second method is implementing the new number formats using general-purpose commercial processors such as CPUs and GPUs using emulation. This method relies on fast time-to-market and easy implementation since only the application/system-software level of the computing stack needs to be changed. However, emulation of FP arithmetic using integer instructions can lead to high latency implementations due to exceptional cases handling including under/overflow. In other words, the overhead due to inefficient conversion to/from the conventional number format to the customized number format can offset the speed gain that can be achieved by less data communication.

To fix these problems, we developed a fast emulation method for implementing our software-defined number formats on off-the-shelf processors. The key elements of the proposed technique are as follows:

- SDF format is used for storage and communication of normalized messages whilst single-precision format is selected for performing arithmetic operations on messages on general-purpose processors. This FP-to-FP conversion prevents inefficient emulation of FP arithmetic using integer instructions.
- SDF format is not applied to *all* parts of the BP algorithm to substitute double(single)-precision formats. Table 2 shows the required data structures to implement BP, and where the proposed formats are used. The communication-intensive structure is for normalized message storage since it is updated per iteration and then retrieved in the next iterations. Due to this, the SDF formats are only applied to the normalized messages to minimize the number of conversions while reducing the normalized message size to reduce data communication between memory hierarchy levels.
- SDF uses a novel truncation-based exponent's coding which has resulted in extremely lightweight conversion mechanisms between SDF formats and single(double)-precision formats. Table 3 shows how exponent values which are identified by (16) and (17) can be translated from double(single)-precision to SDF formats that showed in Fig. 1. It can be seen that for the conversions to/from SDF formats with 3 or 4 bits exponents only truncation is needed. However, subtraction (addition) of single-precision exponent by 3 is required to achieve equivalent exponents for SDF formats with 2-bit exponent. Fig. 2(a) shows details of conversion between one of the proposed 16-bit SDF formats (i.e. Half (4,12)) and the IEEE single-precision format. It should be noted that the conversion mechanism for Half(3,13) is the same as Half(4,12) but with truncating only 3 bits of single-precision exponent for forward conversion, and then embedding 01111 to the top bits for backward conversion. Furthermore, Fig. 2(b) elaborates exponent conversion from 8- to 2-bit, and vice versa as required for the Mini(2,6) and Half(2,14) formats. Moreover, the conversion mechanism for the Mini(3,5) format is similar to Half(3,13) with the exception of 5-bit mantissa truncation rather than 13.
- SDF leverages exponent coding of Table 3, to derive high-speed software-based conversion routines to enable read(write) from(to) normalized messages as shown in *Algorithm 2*. The *Encode* routine converts single-precision data to the SDF format, and vice versa for the *Decode* routine. It should be noted that truncation can also be used for the encoding of single-precision exponents to SDF formats with 2-bit exponents. However, during decoding, "if-else" instructions are needed to put 0 in the 3rd exponent position for the case of 11, and 1 for other cases. The use of if-else instructions decreases the efficiency of the processor pipeline, rather than a simple integer addition instruction. Therefore, to minimize the conversion overhead, we used minus 3 instead of truncation for encoding, and plus 3 instead of "if-else" for the decoding of exponents of the SDF formats with 2-bit exponents.

3.5.3 *Rounding and Special Cases*

Fig. 2 describes the data format conversion to write the

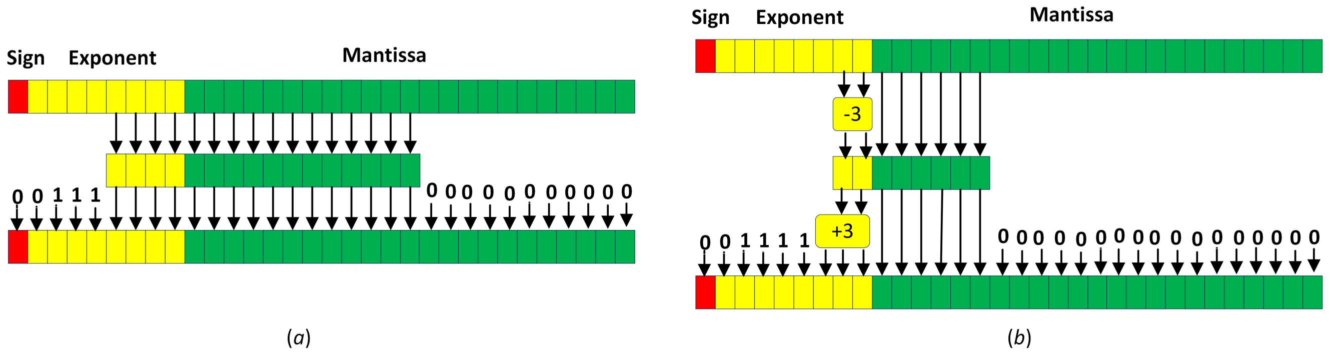

Fig. 2. The proposed conversion mechanisms from/to (a) Half(4,12) (b) Mini (2,6) to IEEE single precision format and vice versa.

TABLE 2
Floating-Point Data Structures needed for different implementations of BP with various precisions

Component	Double Precision	Single Precision	Half Precision	Mini Precision
Marginals	Double	Single	Single	Single
Normalized Messages (Edges)	Double	Single	**16-bit SDF**	**8-bit SDF**
Temporary Messages (Workspace)	Double	Single	Single	Single
Edge Factors	Double	Single	Single	Single
Node Factors	Double	Single	Single	Single

arithmetic results into storage (i.e., conversion from IEEE single precision to SDF) and to perform arithmetic operations using SDF operands (i.e., conversion from SDF to IEEE single precision). Rounding towards zero is considered for the conversion from single precision to an SDF. We take this rounding mode, since it only truncates the least significant bits of the mantissa as shown in Fig. 2, realizing fast conversion.

Since the SDFs are converted back to IEEE single precision format for performing IEEE single arithmetic, special cases regarding the results from the arithmetic operations follow IEEE 754 standard guidelines [40]. None of the special cases, including Not a Number (NaN), infinity, denormals and zero, occur for our use cases using normalized BP messages. For example, the theoretical and experimental study on normalized messages' exponents (i.e., (39) and (40)) confirm that the exponents of the result from the message passing range from 1013 to 1023 in double-precision biased form (as shown in Table 3), which belong to none of special cases of NaN, infinity, zero, and denormalized numbers (i.e., the exponent value is neither 2047 nor zero) [29]. Therefore, we do not consider special cases for our data conversion to an SDF.

3.6 Generalization

The target of the SDF approach is to use numerical analysis at the algorithm level to determine the real number range that is needed for the most frequently used data array, and then develop a suitable compact number format for it to reduce the data communication and memory traffic. Although this work only reports the approach for the BP use case, it is also possible to use this design method for other applications, especially iterative algorithms where a large amount of data transfer between iterations is needed.

Fig. 3 shows a guideline for using the SDF design approach in different applications. The first step is to analyze the target algorithm to determine the big critical arrays (BCAs). Then, theoretical and experimental studies on numbers stored in BCAs should be done to find the required range of numbers. Moreover, some specific behaviours and correlations of exponent and mantissa numbers can be found at this stage to further simplify and shorten the number format. Finally, conversion routines should be developed with the minimum overhead followed by accuracy and performance analysis.

4 Evaluation

This section evaluates the accuracy and performance aspects of the proposed SDF formats against the conventional IEEE double- and single-precision formats which exist in off-the-shelf commercial processors.

4.1 Accuracy Analysis

The target of BP inference algorithm is calculating node marginals based on final converged messages as shown in lines 28-33 of *Algorithm* 5. Therefore, accuracy comparison of various BP algorithms is done using quality of marginals evaluation. The BP is an approximate algorithm when applied on graphs with loops such as Ising grids. Therefore, its accuracy evaluation should be performed against exact methods such as variable elimination (VE) [1].

Exact inference is computationally infeasible for large grid sizes, and only tractable for small grids. Due to this, previous BP implementations [10], [11], [30] only reported exact marginals for Ising grids from 7×7 up to 13×13. However, we also achieved the exact marginals for grid sizes up to 17×17 with both $c = 2$ and $c = 3$ based on VE algorithm which is developed in [31]. Fig. 4 shows the runtime for both exact (VE) and approximate programs (BP with double-precision messages) for grids with c=2 based on a machine with an Intel Xeon Gold 6126 CPU. It can be seen that the runtime of the exact method significantly increases with the grid size, while the approximate method runtime remains minimal.

TABLE 3
Exponent Conversion between different FP formats

Exponents	**Double-Precision (64-bit)**		**Single-Precision (32-bit)**		**The Proposed Exponent Coding**					
unbiased	**Decimal**	**Binary (11 bits)**	**Decimal**	**Binary (8 bits)**	**Decimal**	**Binary (4 bits)**	**Decimal**	**Binary (3 bits)**	**Decimal**	**Binary (2 bits)**
0	1023	01111111 **111**	127	01111 **111**	15	**1111**	**7**	**111**	-	-
-1	1022	01111111 **110**	126	01111 **110**	14	**1110**	6	**110**	3	**11**
-2	1021	01111111 **101**	125	01111 **101**	13	**1101**	5	**101**	2	**10**
-3	1020	01111111 **100**	124	01111 **100**	12	**1100**	4	**100**	1	**01**
-4	1019	01111111 **011**	123	01111 **011**	11	**1011**	3	**011**	0	**00**
-5	1018	01111111 **010**	122	01111 **010**	10	**1010**	2	**010**	-	-
-6	1017	01111111 **001**	121	01111 **001**	9	**1001**	**1**	**001**	-	-
-7	1016	01111111 **000**	120	01111 **000**	8	**1000**	**0**	**000**	-	-
-8	1015	01111110 **111**	119	01110 **111**	7	**0111**	-	-	-	-
-9	1014	01111110 **110**	118	01110 **110**	6	**0110**	-	-	-	-
-10	1013	01111110 **101**	117	01110 **101**	5	**0101**	-	-	-	-

Then, similar to previous works [10], [30], mean square error (MSE) metric is used to calculate the error of approximate marginals based on BP against exact marginals as follows [30]:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}\sum_{x_i \in A_i} |P_{BP}(x_i) - P_{Exact}(x_i)|^2 \qquad (27)$$

where $P_{BP}(x_i)$ is approximate marginal that is achieved from BP using different FP formats including double, single, half and mini. Besides, $P_{Exact}(x_i)$ denotes exact marginals that are computed using the VE algorithm ([1], [31]). Moreover, N is the total number of nodes.

Tables 4 and 5 present MSE results for BP implementation based on configurations described in Table 2. It should be noted we calculated MSE results for various convergence thresholds (ϵ) to find what is the optimum value of ϵ to achieve the best accuracy. Note that smaller ϵ leads to higher iterations and consequently higher runtime. It can be seen that the best MSE for grids with $c = 2$ is achieved when $\epsilon = 0.1$, and in this point the proposed SDF formats have almost the same accuracy as double- or single-precision. Moreover, for grids with $c = 3$, the optimum point is $\epsilon = 0.01$ for 10×10, and only the Half (3,13) can reach this point. However, for 15×15 grid, a very small improvement in MSE can be achieved when $\epsilon = 0.001$. Finally, for 17×17 grid with $c = 3$, the optimum threshold is $\epsilon = 0.1$ since all of the formats except Mini(3,5) achieved their best MSE.

Note that as long as the messages have converged with the same convergence threshold, the marginals are equivalent to each other regardless of different number formats as can be seen from MSEs in Tables 4 and 5. Therefore, we assume that the accuracy of computed beliefs using each number format is equivalent as long as the messages have successfully converged under the same convergence threshold. A higher convergence threshold results in fewer iterations to converge, and consequently higher speed-up, but at the expense of reduced accuracy. In other words, the least significant bits of double- or single-precision messages based on a high threshold do not essentially include accurate message digits. Moreover, BP for Ising grids is approximate either with double- or single-precision formats. Therefore, rounding messages towards zero does not necessarily reduce their precision. For example, the minimum experimental error of BP relative to exact results, as shown in Table 4, is 2.31556×10^{-4}. This error magnitude is significantly larger than the precision limits of both double-precision ($2^{-53} \approx 1.11 \times 10^{-16}$) and single-precision ($2^{-24} \approx 5.96 \times 10^{-8}$). This suggests that the predominant source of error in BP for loopy graphs is the algorithmic error, rather than the precision of the numerical representation.

4.2 Performance Evaluation

We used the BP message scheduling library of [31], and then applied various data formats on it to derive RBP implementations with different precisions. In this regard, a server machine which includes an Intel(R) Xeon(R) E5-2683-v4 CPU, and GCC v10.2.1, is used to analyze the runtime of BP implementations with different message precisions[1]. The experimental results in terms of runtime (seconds) and speedup (against single precision) are presented in Tables 6 and 7. Note that the runtime results in Table 6 are based on the average of 5 runs, and speedups are calculated as follows:

$$\text{Speedup} = \left(\frac{\text{Runtime}_{\text{Double/Single}} - \text{Runtime}_{\text{SDF}}}{\text{Runtime}_{\text{Double/Single}}}\right) \times 100 \qquad (28)$$

It can be seen from Tables 6 and 7 that SDF formats have significantly improved latency. The best speedups are achieved by Mini (2,6), which is a byte-sized format. Specifically, for the largest grid (500×500), the proposed 8- and 16-bit formats achieved speedups of 73.77% and 70.59% over double-precision, respectively. Compared to the single-precision format, the Mini format achieved a maximum speedup of 40.35% for a grid size of 400×400, while for 500×500, the speedup decreased to 31.41%. In general, the speedup depends on the overall size of the working set relative to the cache size. Therefore, increasing the number of graph nodes results in more data transfer between the last level cache and main memory, leading to a higher total runtime for formats with larger sizes. However, when the working set is very large, the working set size for reduced-precision formats may still be substantially larger than the cache size, implying limited performance gains. This effect has also been observed and described in [35].

1. https://github.com/DIPSA-QUB/LowPrecisionBP

Algorithm 2: The Proposed RBP implementation with Software-Defined FP format for messages

```
    Input: Graph: $G = (V, L), |V| = n$
    Random Variables: $x_1, x_2, \ldots, x_n \; (x_i \in A_i)$
    Node Factors: $\phi_i : A_i \to R^+ | i \in V$
    Edge Factors: $\psi_{i,j} : A_i \times A_j \to R^+ | (i,j) \in L$
    Convergence threshold: $\epsilon$
    Output: Marginal Distribution: $P(x_i)$
1   Begin
2   for $(i,j) \in L$ do
3       for $x_j \in A_j$ do
4           $m_{i \to j}(x_j) = Encode(\frac{1}{|A_j|})$
5   rpq=priority_queue(L, default_residual)
6   while $top\_residual > \epsilon$ do
7       $(i,j) = rpq.top()$
8       for $x_j \in A_j$ do
9           $m^{temp}_{i \to j}(x_j) = \sum_{x_i \in A_i} \psi_{i,j}(x_i, x_j)\phi_i(x_i)$
10          $\prod_{k \in \Gamma_{i \setminus j}} Decode(m_{k \to i}(x_i))$
11      $sum = \sum_{x_j \in A_j} m^{temp}_{i \to j}(x_j)$
12      for $x_j \in A_j$ do
13          $m_{i \to j}(x_j) = Encode(\frac{m^{temp}_{i \to j}(x_j)}{sum})$
14      for $(j,h) \in \{outgoing\ edges\ from\ node\ j\}$ do
15          $sum = 0$
16          for $x_h \in A_h$ do
17              $m^{temp}_{j \to h}(x_h) =$
18              $\sum_{x_j \in A_j} \psi_{j,h}(x_j, x_h)\phi_j(x_j) \prod_{k \in \Gamma_{j \setminus h}} Decode(m_{k \to j}(x_j))$
19              $sum = sum + m^{temp}_{j \to h}(x_h)$
20          for $x_h \in A_h$ do
21              $m^{temp}_{j \to h}(x_h) = \frac{m^{temp}_{j \to h}(x_h)}{sum}$
22          $sum = 0$
23          for $x_h \in A_h$ do
24              $sum = sum +$
25              $(m^{temp}_{j \to h}(x_h) - Decode(m_{j \to h}(x_h)))$
26          $rpq(j,h).residual = sum$
27      $top\_residual = rpq.top().residual$
28  for $i \in V$ do
29      $sum = 0$
30      for $x_i \in A_i$ do
31          $P(x_i) = \phi_i(x_i) \prod_{k \in \Gamma_i} Decode(m_{k \to i}(x_i))$
            $sum = sum + P(x_i)$
32      for $x_i \in A_i$ do
33          $P(x_i) = \frac{P(x_i)}{sum}$
34  End
```

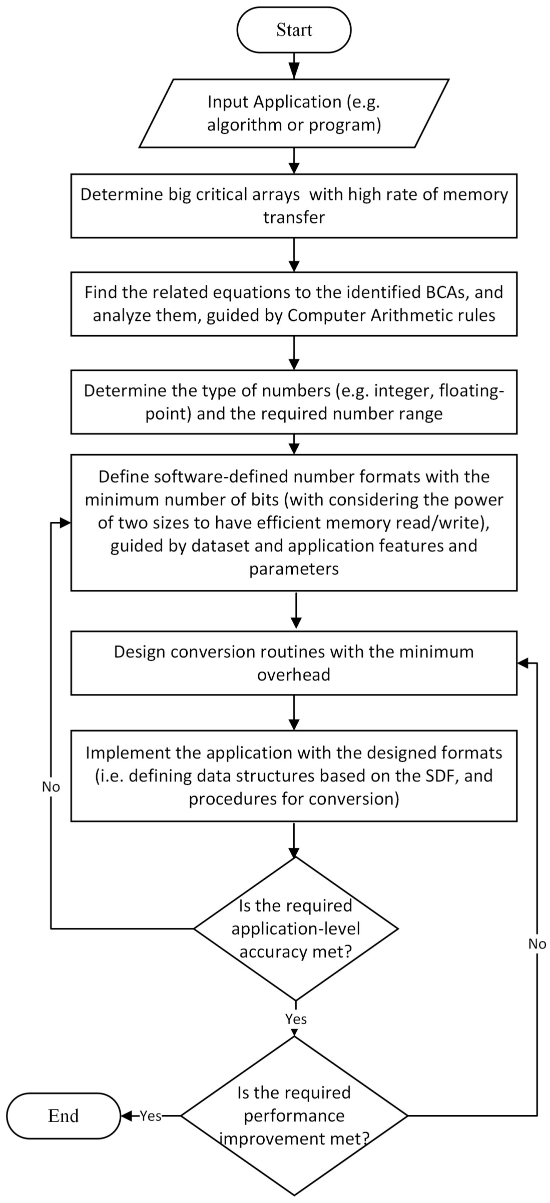
Start
Input Application (e.g. algorithm or program)
Determine big critical arrays with high rate of memory transfer
Find the related equations to the identified BCAs, and analyze them, guided by Computer Arithmetic rules
Determine the type of numbers (e.g. integer, floating-point) and the required number range
Define software-defined number formats with the minimum number of bits (with considering the power of two sizes to have efficient memory read/write), guided by dataset and application features and parameters
Design conversion routines with the minimum overhead
Implement the application with the designed formats (i.e. defining data structures based on the SDF, and procedures for conversion)
Is the required application-level accuracy met?
No
Yes
Is the required performance improvement met?
No
Yes
End

Fig. 3. A general method to use the SDF approach for various data-intensive applications.

5 Related Works

Prior research on reduced-precision floating-point for memory-traffic and data transfer reduction can be categorized in two main categories: *i*) low-precision floating-point formats, and *ii*) general floating-point compression algorithms. The main difference between the proposed software-defined formats with all of these methods is designing formats based on deep theoretical and numerical analysis of the applications. In other words, in the proposed approach, based on algorithm-level numerical analysis, the designer exactly knows that in which situations which formats are accurate and can cover the required number range.

5.1 Low-Precision FP Formats

There are a variety of low-precision FP formats designed in either general or specific forms. Some formats such as IEEE

TABLE 4
Accuracy analysis based on MSE for approximate versus exact marginals for Ising grids with $c = 2$

Grid	**Data**	Convergence threshold (ϵ)							
Size	**Type**	0.5	0.4	0.3	0.2	**0.1**	0.01	0.001	0.0001
10×10	**Double**	0.00822912	0.00426939	0.00179014	0.000421159	**0.000231556**	0.000239182	0.000239257	0.000239256
	Single	0.00822912	0.0042694	0.00179013	0.000421158	**0.000231556**	0.000239182	0.000239257	
	Half(2,14)	0.00822872	0.00426917	0.00179	0.000421086	**0.000231575**	0.000239332		
	Half(3,13)	0.00822833	0.00426881	0.00178991	0.000420984	**0.00023151**	0.000239229		
	Half(4,12)	0.00822805	0.00426834	0.00178974	0.000421302	**0.000231721**			
	Mini(2,6)	0.0128525	0.00619731	0.00238194	0.000416036	**0.000219834**			
	Mini(3,5)	0.0117685	0.00558941	0.00174523	**0.000462102**				
15×15	**Double**	0.0181336	0.00748778	0.00243625	0.00150177	**0.000265028**	0.000295168	0.000295135	0.000295131
	Single	0.0181336	0.00748778	0.00243625	0.00150177	**0.000265028**	0.000295173	0.000295135	
	Half(2,14)	0.0181335	0.0074877	0.00243623	0.00150187	**0.000265108**	0.000295284		
	Half(3,13)	0.0181336	0.00748772	0.00243631	0.000294865	**0.000265235**	0.000294865		
	Half(4,12)	0.0157979	0.00748723	0.00240956	0.00150127	**0.00026476**			
	Mini(2,6)	0.0166339	0.00751996	0.0036612	0.00105801	**0.000295038**			
	Mini(3,5)	0.0140129	0.00687104	0.00324215	**0.000595435**				
17×17	**Double**	0.00939097	0.00560023	0.00220594	0.000807271	**0.000544534**	0.000645984	0.000645742	0.000645735
	Single	0.00939097	0.00560023	0.00220593	0.00080727	**0.000544532**	0.000645984	0.000645736	
	Half(2,14)	0.00939087	0.00560026	0.00220599	0.000807355	**0.000544615**	0.000645714		
	Half(3,13)	0.00939073	0.00560015	0.00220611	0.000807237	**0.00054384**	0.000646047		
	Half(4,12)	0.00939003	0.0056003	0.00220587	0.00080554	**0.000558197**			
	Mini(2,6)	0.00978629	0.0051275	0.00227594	0.000920818	**0.000718533**			
	Mini(3,5)	0.0088269	0.00468414	0.00276699	**0.00113778**				

TABLE 5
Accuracy analysis based on MSE for approximate versus exact marginals for Ising grids with $c = 3$

Grid	**Data**	Convergence threshold (ϵ)							
Size	**Type**	0.5	0.4	0.3	0.2	0.1	**0.01**	**0.001**	0.0001
10×10	**Double**	0.0126671	0.0056398	0.00354458	0.00340195	0.00325852	**0.00313105**	0.00313319	0.00313318
	Single	0.0126671	0.0056398	0.00354458	0.00340195	0.00325852	**0.00313105**	0.00313318	
	Half(3,13)	0.0126675	0.00563994	0.00354479	0.00340203	0.00327099	**0.00313403**		
	Half(4,12)	0.0126663	0.00564001	0.00354501	0.0034026	**0.00326245**			
	Mini(3,5)	0.0134648	0.00609877	0.00411701	**0.00321347**				
15×15	**Double**	0.0157558	0.00941784	0.00571443	0.00509169	0.00431023	0.00417454	**0.00417396**	0.00417396
	Single	0.0157558	0.00941784	0.00571443	0.00509168	0.00431023	0.00417454	**0.00417398**	
	Half(3,13)	0.0157562	0.00941739	0.00571466	0.00509201	0.0043785	**0.0041751**		
	Half(4,12)	0.0157553	0.00941721	0.0057145	0.00507989	**0.00441603**			
	Mini(3,5)	0.0159247	0.00944025	0.00575789	**0.00480953**				
17×17	**Double**	0.030694	0.00913192	0.00586818	0.00375661	**0.0019857**	0.00221424	0.00221327	0.00221326
	Single	0.030694	0.00913191	0.00586818	0.00375661	**0.0019857**	0.00221424	0.00221326	
	Half(3,13)	0.0306952	0.00913157	0.00586833	0.00375648	**0.00198536**	0.00221191		
	Half(4,12)	0.0405906	0.0231538	0.0053776	0.00279462	**0.00213574**			
	Mini(3,5)	0.0400407	0.0121727	0.00580563	**0.00279949**				

TABLE 6
Runtime comparison of different BP implementations for Ising grids with $c = 2$ and $\epsilon = 0.1$

	Runtime (seconds)					
Grid Size	**Double 64-bit**	**Single 32-bit**	**Half (2,14) 16-bit**	**Half (3,13) 16-bit**	**Half (4,12) 16-bit**	**Mini (2,6) 8-bit**
500×500	4.08×10^4	1.56×10^4	1.20×10^4	1.20×10^4	1.20×10^4	1.07×10^4
400×400	1.31×10^4	6.84×10^3	4.93×10^3	4.93×10^3	4.93×10^3	4.08×10^3
300×300	4.03×10^3	2.19×10^3	1.57×10^3	1.57×10^3	1.57×10^3	1.40×10^3
200×200	8.25×10^2	4.20×10^2	3.14×10^2	3.14×10^2	3.14×10^2	2.65×10^2
100×100	5.50×10	2.81×10	2.16×10	2.16×10	2.16×10	1.85×10

TABLE 7
Speedup comparison of different BP implementations for Ising grids with $c = 2$ and $\epsilon = 0.1$

	Speed Up (%) over Double				**Speed Up (%) over Single**			
Grid Size	**Half (2,14)**	**Half (3,13)**	**Half (4,12)**	**Mini (2,6)**	**Half (2,14)**	**Half (3,13)**	**Half (4,12)**	**Mini (2,6)**
500×500	70.59	70.59	70.59	73.77	23.08	23.08	23.08	31.41
400×400	62.37	62.37	62.37	68.85	27.92	27.92	27.92	40.35
300×300	61.04	61.04	61.04	65.26	28.31	28.31	28.31	36.07
200×200	61.94	61.94	61.94	67.88	25.24	25.24	25.24	36.90
100×100	60.73	60.73	60.73	66.36	23.13	23.13	23.13	34.16

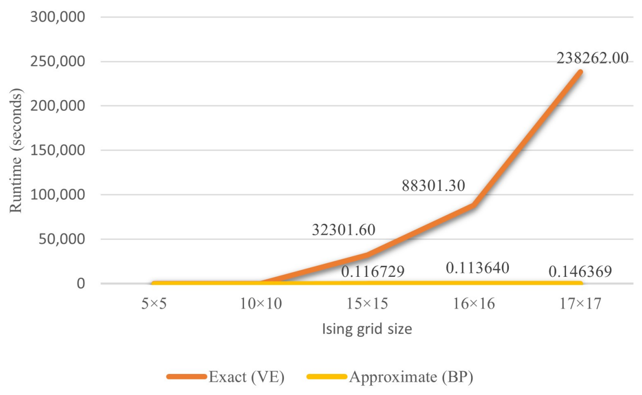

Fig. 4. Runtime of exact and approximate inference for Ising grids based on VE and BP, respectively.

half-precision and BFloat have hardware support mostly on GPUs for performing computation, while others have been designed with the aim of data size and communication reduction. In this regard, CPMS was an important work that decouples data format from arithmetic format [20]. However, the CPMS-based formats are based on a 16-bit exponent field to enable computations using double-precision, which makes them inefficient for applications like BP where a few bits for exponents are enough. [19] solved this problem and introduced two 16-bit FP formats with 3- and 8-bit exponent fields customized for PageRank application through experimentally analyzing FP values exponents in each iteration. But the large number of random memory accesses needed in PageRank prevents [19] from achieving significant speed-up. Furthermore, the number formats in [19] are only suitable for the PageRank algorithm, and cannot be used for BP. On the other hand, [35] developed customized narrow floating-point formats for irregular graph processing workloads such as accelerated PageRank and single-source shortest path algorithms. However, the number formats in [35] have a width of 16 bits, which makes them less memory-efficient than 8-bit formats. Apart from these application-specific formats, [18] proposed two customized FP formats smaller than single-precision formats with an 8-bit exponent to simplify the conversion to single-precision for performing arithmetic. However, in this work, we showed that it is possible to have exponents with less than 8-bit size together with having very simple conversion routines to single-precision.

5.2 FP Compression Algorithms

FP compression methods aim to reduce the storage and communication requirements by exploring dependencies between consecutive data in large arrays of FP numbers. Lossy FP compression methods [36] provide significant size reduction at the cost of accuracy loss, while, in Lossless FP compression, the original FP data can be recovered without accuracy loss [37]. However, compression and decompression overheads increase latency and reduce the effective use of bandwidth since predictor, difference calculator, and residual coder are needed for decoding and encoding [38].
As an example, the double-precision FP compression algorithm introduced in [39] uses a hash-based predictor to guess the current FP record based on the previous record, and therefore, hash tables larger than the cache size results in significant performance degradation. In contrast, in the proposed SDF approach, the conversion procedures are very simple. FP compression algorithms experimentally work on the dataset and try to find similarities on whole data while SDF first numerically analyses the algorithm and dataset parameters to determine the optimized format with the best possible size. Moreover, in an iterative algorithm like RBP where in each iteration only a few numbers need to be updated (i.e. messages), FP compression algorithms cannot be useful since they operate on the whole FP data array.

6 Conclusion

This article proposes SDF format design, for the first time, for sum-product BP applications. We incorporate BP features as a first-class SDF design constraint together with the highly efficient software implementation on top of the off-the-shelf processors. We show that message-passing algorithms such as BP can leverage reduced-precision number formats to achieve speedup via memory traffic reduction even on commercially available CPUs. We theoretically analyze exponents of messages and the accuracy depending on quantization levels of SDF formats to derive guidelines for designing SDF formats. Our experimental results show that an SDF format along with an efficient data conversion module can achieve up to $1.61\times$ speedup in our case studies without losing accuracy. The proposed SDF approach can be extended to other applications particularly GPU-based implementations to improve computational speed via data communication optimization.

Acknowledgments

The authors would like to thank the reviewers for their constructive and insightful comments, which have significantly enhanced the quality of the paper. We are particularly thankful to the reviewers for their contribution to improving the theoretical study of the work.

References

[1] D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques*, The MIT Press, Cambridge, Massachusetts, 2009 .
[2] S. H. Lee, M. Kim, H. Shin and I. Lee, ”Belief Propagation for Energy Efficiency Maximization in Wireless Heterogeneous Networks,” *IEEE Transactions on Wireless Communications*, vol. 20, no. 1, pp. 56-68, Jan. 2021.
[3] H. Ji, Y. Shen, W. Song, Z. Zhang, X. You and C. Zhang, ”Hardware Implementation for Belief Propagation Flip Decoding of Polar Codes,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 3, pp. 1330-1341, March 2021.
[4] P. Knobelreiter, C. Sormann, A. Shekhovtsov, F. Fraundorfer, and T. Pock, ”Belief Propagation Reloaded: Learning BP-Layers for Labeling Problems,” In Proc. of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 7900-7909, 2020.
[5] B.J. Frey and D. Dueck, ”Clustering by Passing Messages Between Data Points,” *Science*, vol. 315, no. 5814, pp. 972-976, Feb. 2007.
[6] A.F. García-Fernández, L. Svensson, and S. Särkkä, ”Cooperative Localization Using Posterior Linearization Belief Propagation,” *IEEE Transactions on Vehicular Technology*, vol. 67, np. 1, pp. 832-836, 2018.
[7] L. Nardi, and C. Stachniss, ”Long-Term Robot Navigation in Indoor Environments Estimating Patterns in Traversability Changes,” In Proc. of IEEE International Conference on Robotics and Automation (ICRA), pp. 300-306. Paris, France, 2020.

[8] K.Y. Hsieh, C.H. Lai, S.H. Lai, J.K. Lee, ”Parallelization of Belief Propagation on Cell Processors for Stereo Vision,” *ACM Transactions on Embedded Computing Systems*, vol. 11, no. 1, 2012.

[9] M. Trotter, T. Wood, and H.H. Huang, ”Rumor Has It: Optimizing the Belief Propagation Algorithm for Parallel Processing,” In Proc. of ACM International Conference on Parallel Processing, pp. 1-10. New York, NY, USA, 2020.

[10] G. Elidan, I. McGraw, and D. Koller, ”Residual belief Propagation: informed scheduling for asynchronous message passing,” In Proc. of the ACM Conference on Uncertainty in Artificial Intelligence (UAI’06), pp. 165–173. AUAI Press, Arlington, Virginia, USA, 2006.

[11] M.V.D. Merwe, V. Joseph and G. Gopalakrishnan, ”Message Scheduling for Performant, Many-Core Belief Propagation,” In Proc. of IEEE High Performance Extreme Computing Conference (HPEC), pp. 1-7. Waltham, MA, USA, 2019.

[12] D. Goldberg, ”What Every Computer Scientist Should Know about Floating-Point Arithmetic,” *ACM Computing Surveys*, vol. 23, no. 1, 1991.

[13] S. Williams, A. Waterman, and D. Patterson, ”Roofline: an insightful visual performance model for multicore architectures,” *Communications of the ACM*, vol. 52, no. 4, pp. 65-76, 2009.

[14] BFloat16 – Hardware Numerics Definition, White Paper, Intel, Nov. 2018.

[15] A. Agrawal, S. Mueller, B.M. Fleischer, X. Sun, N. Wang, J. Choi, and K. Gopalakrishnan, ”Dlfloat: A 16-b floating point format designed for deep learning training and inference,” In Proc. of IEEE Symposium on Computer Arithmetic (ARITH), pp. 92-95. Kyoto, Japan, 2019.

[16] U. Köster, T. Webb, X. Wang, M. Nassar, A.K. Bansal, W. Constable, O. Elibol, S. Gray, S. Hall, L. Hornof, A. Khosrowshahi, “Flexpoint: an adaptive numerical format for efficient training of deep neural networks,” In Proc. of the 31st International Conference on Neural Information Processing Systems, Red Hook, NY, USA, pp. 1740–1750, 2017.

[17] A. Nannarelli, “Variable Precision 16-Bit Floating-Point Vector Unit for Embedded Processors,” In Proc. of IEEE Symposium on Computer Arithmetic (ARITH), 2020.

[18] A. Anderson, S. Muralidharan and D. Gregg, ”Efficient Multibyte Floating Point Data Formats Using Vectorization,” *IEEE Transactions on Computers*, vol. 66, no. 12, pp. 2081-2096, 1 Dec. 2017.

[19] A.S. Molahosseini and H. Vandierendonck, ”Half-Precision Floating-Point Formats for PageRank: Opportunities and Challenges,” In Proc. of IEEE High Performance Extreme Computing Conference (HPEC), pp. 1-7. Waltham, MA, USA 2020.

[20] T. Grützmacher, T. Cojean, G. Flegar, H. Anzt, E.S. Quintana-Ortí, ”Acceleration of PageRank with Customized Precision Based on Mantissa Segmentation,” *ACM Transactions on Parallel Computing*, vol. 7, no. 1, pp. 1-19, 2020.

[21] J.S. Firoz, A. Li, J. Li, K. Barker, ”On the Feasibility of Using Reduced-Precision Tensor Core Operations for Graph Analytics,” In Proc. of IEEE High Performance Extreme Computing Conference (HPEC), pp. 1-7. Waltham, MA, USA 2020.

[22] M. Fatemi, ”Bayesian Inference for Automotive Applications,” PhD Thesis, Chalmers University of Technology, Göteborg, Sweden, 2016.

[23] A.J. Bean, ”Message Passing Algorithms: Methods and Applications,” PhD Thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, USA, 2015.

[24] U. Kang, D.H.P. Chau, C. Faloutsos, ”Inference of Beliefs on Billion-Scale Graphs,” ACM SIGKDD Conference on Knowledge Discovery and Data Mining July 25-28, Washington, DC, USA, 2010.

[25] J. Coughlanm, ”A tutorial introduction to belief propagation,” The Smith-Kettlewell Eye Research Institute, San Francisco, CA, USA, 2009.

[26] V. Martin, J.M. Lasgouttes, C. Furtlehner, ”The role of normalization in the belief propagation algorithm,” arXiv preprint arXiv:1101.4170, 2011.

[27] C. Knoll, ”Understanding the Behavior of Belief Propagation,” PhD Thesis, Graz University of Technology, Austria, 2019 https://arxiv.org/abs/2209.05464.

[28] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, 2nd edition Oxford University Press, New York, 2010.

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

[30] C. Knoll, M. Rath, S. Tschiatschek, and F. Pernkopf, ”Message scheduling methods for belief propagation,” In Proc. Joint European Conference on Machine Learning and Knowledge Discovery in Databases, pp. 295-310, Springer, 2015.

[31] M.V.D. Merwe, BP Message Scheduling Library, http://github.com/mvandermerwe/BP-GPU-Message-Sheduling. Last accessed 11 February 2022.

[32] N.J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd edition, Society for Industrial and Applied Mathematics, 2002.

[33] J. Lee, D.S. Nikolopoulos, and H. Vandierendonck, ”Mixed-Precision Kernel Recursive Least Squares,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 3, pp. 1284-1298, March 2022.

[34] A. Ihler, J. Fisher, and A. Wilsky, ”Message Errors in Belief Propagation,” Advances in Neural Information Processing Systems 17 (NIPS 2004).

[35] H. Vandierendonck, ”Software-defined floating-point number formats and their application to graph processing,” In Proc. of the 36th ACM International Conference on Supercomputing (ICS), Jun. 28-30, pp. 1-17, New York, NY, USA 2022.

[36] F. Cappello, S. Di, S. Li, X. Liang, A. M. Gok, D. Tao, C.H. Yoon, X. C. Wu, Y. Alexeev, F. T. Chong, ”Use cases of lossy compression for floating-point data in scientific data sets,” *The International Journal of High Performance Computing Applications*, vol. 33, no. .6, pp. 1201-1220, 2019.

[37] N. Fout and K. Ma, ”An Adaptive Prediction-Based Approach to Lossless Compression of Floating-Point Volume Data,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 12, pp. 2295-2304, Dec. 2012.

[38] P. Ratanaworabhan, J.Ke and M. Burtscher, ”Fast lossless compression of scientific floating-point data,” In. Proc. Data Compression Conference (DCC’06), 2006, pp. 133-142.

[39] M. Burtscher and P. Ratanaworabhan, ”FPC: A High-Speed Compressor for Double-Precision Floating-Point Data,” *IEEE Transactions on Computers*, vol. 58, no. 1, pp. 18-31, Jan. 2009.

[40] ”IEEE Standard for Floating-Point Arithmetic,” in IEEE Std 754-2019 (Revision of IEEE 754-2008), pp.1-84, 22 July 2019.

[41] N. Higham, ”Accuracy and Stability of Numerical Algorithms,” Society for Industrial and Applied Mathematics, 2002.

Amir Sabbagh Molahosseini received the B.Sc. degree from Shahid Bahonar University of Kerman, Iran in 2005, M.Sc. and Ph.D. degrees (Hons.) in computer engineering from the Science and Research Branch of Azad University, Tehran, Iran, in 2007 and 2010, respectively. He is currently a Marie Skłodowska-Curie Fellow with the Centre for Intelligent Sustainable Computing (CISC), School of Electronics, Electrical Engineering and Computer Science, Queen’s University Belfast, UK. His research interests are Computer Arithmetic and Transprecise Computing.

JunKyu Lee received the Ph.D. degree in computer engineering from The University of Tennessee, Knoxville, TN, USA, in 2012. He was a Postdoctoral Research Associate with the Joint Institute for Computational Sciences, The University of Tennessee, Oak Ridge National Laboratory, Oak Ridge, TN, USA, a Research Associate with the School of Electrical and Information Engineering, The University of Sydney, NSW, Australia, and a Research Fellow with the Institute of Electronics, Communications and Information Technology, Queen’s University Belfast, U.K. He is currently a Research Fellow with the Institute for Analytics and Data Science, University of Essex, U.K. His research interests include linear algebra and machine learning.

Hans Vandierendonck received the Ph.D. and M.Sc. degrees from Ghent University, Ghent, Belgium, in 2004 and 1999, respectively. He is currently a Professor of high-performance and data-intensive computing with the School of Electronics, Electrical Engineering and Computer Science, and a Fellow of the Institute on Electronics, Communications and Information Technology, Queen's University Belfast, U.K. His research aims to build efficient and scalable computing systems for data-intensive applications.