An efficient midpoint-radius representation format to deal with symmetric fuzzy numbers

Manuel Marin, David Defour, Federico Milano

An efficient midpoint-radius representation format to deal with symmetric fuzzy numbers

Manuel Marin, *Student Member, IEEE*, David Defour, and Federico Milano, *Senior Member, IEEE*

Abstract

This paper proposes a novel representation for symmetric fuzzy numbers that uses the midpoint-radius approach instead of the conventional lower-upper representation. A theoretical analysis based on the α-cut concept shows that the proposed format requires half the amount of operations and memory than the traditional one. Also, a novel technique involving radius increments is introduced, to mitigate floating-point rounding errors when using the proposed representation. We describe the implementation of all these features into a fuzzy arithmetic library, specifically tuned to run on Graphic Processing Units (GPU). The results of a series of tests using compute-bound and memory-bound benchmarks, show that the proposed format provides a performance gain of two to twenty over the traditional one. Finally, several implementation issues regarding GPU are discussed in light of these results.

Index Terms

Uncertainty, fuzzy systems, floating-point arithmetic, performance analysis, computer architecture.

I. Introduction

MANAGEMENT of uncertainty has appeared as a necessity in the design of expert systems, where the information in the knowledge base is ambiguous and imprecise. The information about system quantities is usually presented in the form of ambiguous sentences, such as, “up to 1, but maybe no greater than 0.7,” or, “between 0.5 and 1.5, but closer to the second,” etc. This particular kind

M. Marin is with Université de Perpignan Via Domitia, DALI and Université Montpellier 2, LIRMM, France, and also with the School of Electrical, Electronic and Communications Engineering of the University College Dublin, Dublin, Ireland (e-mail: manuel.marin@univ-perp.fr).

D. Defour is with Université de Perpignan Via Domitia, DALI and Université Montpellier 2, LIRMM, France (e-mail: david.defour@univ-perp.fr).

F. Milano is with the School of Electrical, Electronic and Communications Engineering of the University College Dublin, Dublin, Ireland (e-mail: federico.milano@ucd.ie).

of uncertainty is best modelled using fuzzy numbers, as they allow to deal with such type of quantifiers [1]. The fuzzy approach has been successfully applied in finance [2], health [3], transportation [4], control [5], supply chain management [6], load flow [7], [8], and others, to configure expert systems in order to run under uncertainty scenarios. Fuzzy arithmetic is often approached through the α-cut concept, which turns it into an extension of the well-known interval arithmetic [9]. However, practical implementations of this approach can be computationally expensive, as each fuzzy operation requires at least two interval operations to complete. Depending on the number of uncertainty levels in the model, the computational burden can increase dramatically and so the calculation time.

GPUs have driven substantial acceleration to numerous regular applications thanks to several mechanism. The first and most powerful one is Thread Level Parallelism (TLP) exploited by the numerous compute unit available. Instruction Level Parallelism (ILP) has also proven to be very useful to hide instruction latency, achieving better performance at lower occupancy [10]. Another source of acceleration, but less frequently used, come from the hardware accelerated compute unit dedicated to the evaluation of interpolation, special functions or fused operations such as FMA. By going deeper into these subtleties, one can notice that latest CUDA GPUs allow to statically select rounding attribute for every floating-point operation. This characteristic simplifies interval arithmetic operations [11], as it suppresses the overhead associated with the changes of rounding mode.

Applications relying on fuzzy logic have already been successfully ported on GPU [12], [13]. However, to our knowledge there were no prior work considering fuzzy arithmetic and the underlying representation of data dedicated to a GPU execution. In this article we investigate how GPU can improve performance of fuzzy arithmetic. To achieve this goal, we propose to replace the traditional lower-upper encoding, which requires to manage a different lower and upper bound for each α-cut, by a midpoint-radius encoding, whenever possible. This modification of data representation greatly reduces the required bandwidth as well as the number of instructions to perform basic operation. These algorithmic modifications combined with hardware specificity of Nvidia's GPU greatly reduce the cost of fuzzy arithmetic, making fuzzy arithmetic more affordable in term of execution time.

The main contributions brought by this article are as follows:

- **Representation format.** We propose two new representation formats for fuzzy numbers with symmetric membership function, based on midpoint-radius intervals. The first one reduces the number of operations and memory bandwidth of fuzzy computations compared to the traditional lower-upper

approach. The second one is an extension of the former and is designed to improve accuracy. The proposed formats are based on original theorems, i.e., Theorems 1, 2 and 4, for which we provide the proofs.

- **Fuzzy arithmetic library.** We describe the implementation of a fuzzy arithmetic library which allows to handle fuzzy numbers in the proposed representation formats as well as in the traditional format. This library is highly efficient and versatile, as it combines specific features of the CUDA and C++ programming languages, such as rounding attributes and operator overloading.
- **Insight on GPU-optimized fuzzy arithmetic.** We discuss, both from a theoretical and an empirical point of view, the benefits brought by GPUs to different types of fuzzy computations. We explain how TLP, ILP, static rounding mode and memory usage impact performance at a fine granularity of basic operations on fuzzy formats.

The remainder of the article is divided as follows: in Sections II and III, we expose the theory underlying our fuzzy arithmetic library. In Section IV, we present the implementation issues and discuss about trade-off related to ILP, TLP, bandwidth and latency. Then, in Section V, we evaluate the performance of our library of fuzzy arithmetic. Section VI draws conclusions and outlines future work.

II. Outlines on Interval Arithmetic and Fuzzy Numbers

The α-cut approach to fuzzy arithmetic is based on representing a fuzzy number as a collection of intervals, which in turn represent different levels of uncertainty in the model or α-*levels*. These intervals are called the α-cuts. Fuzzy arithmetic operations, then, are performed as sets of interval operations, which ultimately requires to deal with basic interval algebra [9].

In this section, we provide basic concepts of interval arithmetic and fuzzy arithmetic based on the α-cut concept. The outlines given below are needed to introduce the discussion on different alternatives to implement fuzzy arithmetic that we present in section III.

A. Interval arithmetic

Interval arithmetic accounts for single level uncertainty of numerical modelling. It deals with intervals which are defined as convex sets of real numbers. When expressing a variable as an interval, each element in the range is considered equally possible.

Intervals can be represented in three ways, either by giving the lower and upper bounds, the midpoint along with the radius of the interval, the lower bound along with the diameter. Most of the existing

implementations of interval arithmetic such as Boost [14], MPFI [15] and GAOL [16], are based on the lower-upper representation encoding. In this article, we consider both lower-upper and midpoint-radius representations, as defined below.

Definition 1 (Lower-upper representation)**.** Let $l, u \in \mathbb{R}$, $l \leq u$. Let $\mathcal{I}$ be an interval, defined by:

$$\mathcal{I} = \{x \in \mathbb{R}, l \leq x \leq u\}. \tag{1}$$

We note $\mathcal{I} = [l, u]$, and call this the lower-upper representation, with l being the lower bound and u the upper bound.

Definition 2 (Midpoint-radius representation)**.** Let $m, \rho \in \mathbb{R}$, $\rho \geq 0$. Let $|\cdot|$ represent the absolute value operation. Let $\mathcal{I}$ be an interval, defined by:

$$\mathcal{I} = \{x \in \mathbb{R}, |x - m| \leq \rho\}. \tag{2}$$

We note $\mathcal{I} = \langle m, \rho \rangle$, and call this the midpoint-radius representation, with m being the midpoint and ρ the radius.

With these basic concepts we can proceed to define interval operations. With this aim, we assume the basic property of *inclusion isotonocity*, as follows.

Definition 3 (Inclusion isotonicity)**.** Let $\circ \in \{+, -, \cdot, /\}$, $\mathcal{I}_1$ and $\mathcal{I}_2$ be intervals. If

$$x_1 \circ x_2 \subseteq \mathcal{I}_1 \circ \mathcal{I}_2, \ \forall\, x_1 \in \mathcal{I}_1, \ \forall\, x_2 \in \mathcal{I}_2, \tag{3}$$

then $\circ$ is said to be inclusion isotone.

Inclusion isotonicity is needed to ensure that no possible values are "left behind" when performing interval operations. In order to respect this property, interval arithmetic implementations require to deal with floating-point rounding errors.

The IEEE-754 Standard for floating-point computation provides all the necessary elements to such task. In particular, this standard establishes that the following three rounding attributes must be available: rounding upwards (towards infinity), rounding downwards (towards minus infinity), and rounding to nearest [17]. The IEEE-754 Standard also allows to compute the relative rounding error and the smallest representable (unnormalized) floating point number, the latter being associated with the underflow error [18], [19]. These

elements and the corresponding notation are summarized in Table I.

TABLE I
ROUNDING ATTRIBUTES AND ERROR TERMS IN THE IEEE-754 STANDARD.

Symbol	Meaning
$\triangle(\cdot)$	Rounding upwards.†
$\triangledown(\cdot)$	Rounding downwards.†
$\square(\cdot)$	Rounding to nearest.†
ϵ	Relative rounding error (machine epsilon).
η	Smallest representable (unnormalized) floating-point positive number.

†The rounding attribute applies on all the operations included within the parentheses.

In the lower-upper encoding, isotonicity implies rounding downwards when computing the lower bounds, and upwards when computing the upper bounds [9]. The interval operations for the lower-upper encoding are defined below.

Definition 4 (Lower-upper interval operations)**.** Let $\mathcal{I}_1 = [l_1, u_1]$ and $\mathcal{I}_2 = [l_2, u_2]$. Let $\triangledown(\cdot)$ and $\triangle(\cdot)$ be respectively the rounding attributes towards minus infinity and plus infinity, applying on all the operations within the parentheses. Then

$$\mathcal{I}_1 + \mathcal{I}_2 = [\triangledown(l_1 + l_2), \triangle(u_1 + u_2)], \tag{4a}$$

$$\mathcal{I}_1 - \mathcal{I}_2 = [\triangledown(l_1 - u_2), \triangle(u_1 - l_2)], \tag{4b}$$

$$\mathcal{I}_1 \cdot \mathcal{I}_2 = [\triangledown(\min(\mathcal{S})), \triangle(\max(\mathcal{S}))], \tag{4c}$$

$$\text{where } \mathcal{S} = \{l_1 l_2, l_1 u_2, u_1 l_2, u_1 u_2\},$$

$$\frac{1}{\mathcal{I}_2} = \left[\triangledown\left(\frac{1}{u_2}\right), \triangle\left(\frac{1}{l_2}\right)\right], \tag{4d}$$

$$\text{where } 0 \notin [l_2, u_2].$$

For the midpoint-radius encoding, in addition to appropriate rounding, isotonocity also requires adding to the radius the error due to midpoint rounding [18]. The interval operations for the midpoint-radius encoding are defined below.

Definition 5 (Midpoint-radius interval operations)**.** Let $\mathcal{I}_1 = \langle m_1, \rho_1 \rangle$ and $\mathcal{I}_2 = \langle m_2, \rho_2 \rangle$. Let $\square(\cdot)$ and $\triangle(\cdot)$ be respectively the rounding attributes to nearest and towards plus infinity, applying on all the operations within the parentheses. Let ϵ be the relative rounding error and η be the smallest representable

(unnormalized) floating point number. Then

$$\mathcal{I}_1 \pm \mathcal{I}_2 = \langle \square(m_1 \pm m_2), \triangle(\epsilon_m + \rho_1 + \rho_2) \rangle, \tag{5a}$$
$$\text{where } \epsilon_m = \frac{1}{2}\epsilon |\square(m_1 \pm m_2)|,$$
$$\mathcal{I}_1 \cdot \mathcal{I}_2 = \langle \square(m_1 \cdot m_2), \triangle(\epsilon_m + (|m_1| + \rho_1)\rho_2 + |m_2|\rho_1) \rangle, \tag{5b}$$
$$\text{where } \epsilon_m = \eta + \frac{1}{2}\epsilon |\square(m_1 \cdot m_2)|,$$
$$\frac{1}{\mathcal{I}_2} = \left\langle \square\left(\frac{1}{m_2}\right), \triangle\left(|\epsilon_m| + \frac{-\rho_2}{|m_2|(\rho_2 - |m_2|)}\right) \right\rangle, \tag{5c}$$
$$\text{where } \epsilon_m = \eta + \frac{1}{2}\epsilon \left|\square\left(\frac{1}{m_2}\right)\right|, \text{ and } |m_2| > \rho_2.$$

Equation (5c) is based on the definition provided by Neumaier in [20] for real intervals. With respect to the original definition, we introduce appropriate rounding in order to account for the floating-point case. A similar procedure is presented in [18].

The impact of rounding on performance depends on the computing architecture. For example, for CPUs, the rounding attribute is implemented as a processor state, which implies flushing the entire pipeline every time it changes during a program execution.

B. Fuzzy arithmetic

While interval arithmetic accounts for one single level of uncertainty, fuzzy arithmetic enables modelling several levels of uncertainty as a whole. Fuzzy arithmetic deals with fuzzy numbers, defined in a similar way as intervals with the addition that each element in the set has associated a degree of membership. Membership degrees correspond to levels of uncertainty or α-levels. Elements with higher degrees are considered by a greater number of α-levels, as possible values for the modelled variable to assume.

The degree of membership is implemented as a membership function, which represents a possibility distribution. Not any real function can be a membership function; it has to satisfy certain assumptions, as stated in the following definition.

Definition 6 (Membership function)**.** Let $\mu(\cdot) : \mathbb{R} \to [0, 1]$ be a continuous function. If $\exists\, l, m, u \in \mathbb{R}, l \leq$

$m \leq u$, such that:

$$\mu(m) = 1, \tag{6a}$$

$$\mu(x) < \mu(y), \ \forall\, x, y \in [l, m], \ x < y, \tag{6b}$$

$$\mu(x) > \mu(y), \ \forall\, x, y \in [m, u], \ x < y, \tag{6c}$$

$$\mu(x) = 0, \ \forall\, x \notin [l, u] \tag{6d}$$

then $\mu(\cdot)$ is called a membership function.

In summary, a membership function has to show strict monotonicity around a central point. The formal definition of fuzzy number is as follows.

Definition 7 (Fuzzy number)**.** Let $\mu(\cdot)$ be a membership function, and $m \in \mathbb{R}, \mu(m) = 1$. Then

$$\mathcal{A} = \{(x, \mu(x)), x \in \mathbb{R}\} \tag{7}$$

is called a fuzzy number, with kernel m and membership function $\mu(\cdot)$.

According to Definition 7, a fuzzy number is a set of ordered pairs, composed of a real number and a degree associated to it, the latter given by means of a membership function. Figure 1 shows an example of fuzzy number, with truncated normal possibility distribution.

Now we introduce the α-cut approach to fuzzy arithmetic, which is a direct way of performing a fuzzy operation through considering only a finite number of α-levels of uncertainty. Let us say this number is N. To each α-level $i \in \{1, \ldots, N\}$, we associate a measure of uncertainty, $\alpha_i \in [0, 1]$, such that $\alpha_1 > \ldots > \alpha_N$. Then we take the fuzzy operands and split them into sets of intervals for each α-level. These intervals, called the α-cuts, contain all the values considered as possible above a certain measure. The definition of α-cut is as follows.

Definition 8 (α-cut)**.** Let $\mathcal{A}$ be a fuzzy number, $\mu_{\mathcal{A}}(\cdot)$ its membership function. Let $N \in \mathbb{N}$, $i \in \{1, \ldots, N\}$, and $\alpha_i \in [0, 1]$. Then

$$\mathcal{A}_i = \{x \in \mathbb{R}, \mu_{\mathcal{A}}(x) \geq \alpha_i\}, \tag{8}$$

is called an α-cut (of level i) of the fuzzy number $\mathcal{A}$.

The conditions that the membership function has to satisfy allow the α-cuts to become well-defined

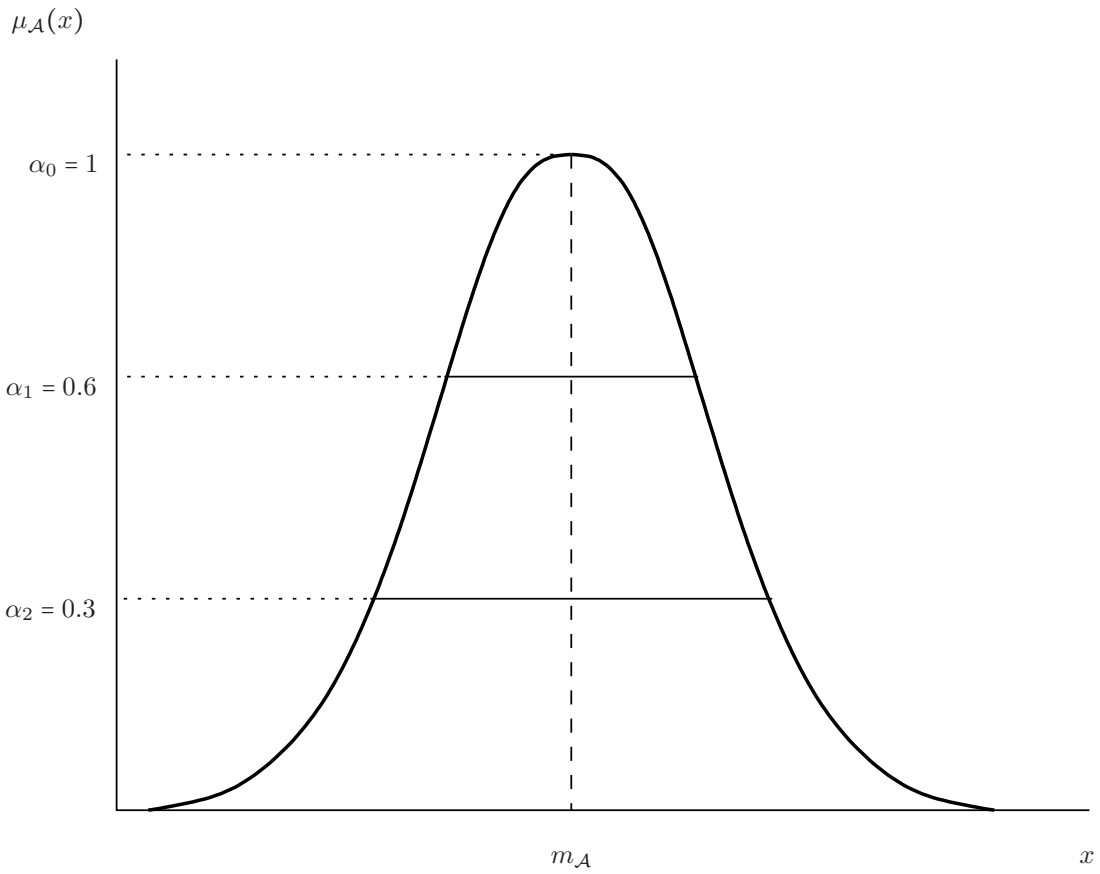

Fig. 1. Fuzzy number, membership function, kernel and α-cuts.

intervals, (see Fig. 1). Once we have the α-cuts for each operand, we may apply the α-cut concept defined below.

Definition 9 (α-cut concept)**.** Let $\circ \in \{+, -, \cdot, /\}$, and $\mathcal{A}, \mathcal{B}, \mathcal{C}$, be fuzzy numbers such that $\mathcal{C} = \mathcal{A} \circ \mathcal{B}$. Let $\mathcal{A}_i, \mathcal{B}_i, \mathcal{C}_i$, be the α-cuts of level i of $\mathcal{A}, \mathcal{B}, \mathcal{C}$, respectively. Then

$$\mathcal{C}_i = \mathcal{A}_i \circ \mathcal{B}_i. \tag{9}$$

The above definition suggests that to compute a fuzzy operation, we shall proceed by computing the α-cuts of the result at any given level, via the corresponding interval operation between α-cuts of the operands.

III. Proposed Representation for Fuzzy Numbers

Fuzzy arithmetic implementations are available in various programming languages [21], [22]. These implementations are based on the lower-upper interval representation format. In this section, we introduce two novel approaches to encode fuzzy numbers based on the midpoint-radius representation. Table II

presents the general notation used in the remainder of this paper.

TABLE II
FUZZY NUMBER NOTATION.

Symbol	Meaning
$\mathcal{A}$	Fuzzy number.
$m_{\mathcal{A}}$	Kernel of fuzzy number $\mathcal{A}$.
$\mathcal{A}_i$	α-cut of level i of fuzzy number $\mathcal{A}$.
$l_{\mathcal{A},i}$	Lower bound of α-cut $\mathcal{A}_i$.
$u_{\mathcal{A},i}$	Upper bound of α-cut $\mathcal{A}_i$.
$m_{\mathcal{A},i}$	Midpoint of α-cut $\mathcal{A}_i$.
$\rho_{\mathcal{A},i}$	Radius of α-cut $\mathcal{A}_i$.
$\delta_{\mathcal{A},i}$	Radius increment of α-cut $\mathcal{A}_i$.

A. *Midpoint-radius encoding for symmetric fuzzy numbers*

The shape of the membership function is a decisive aspect in fuzzy modelling. Depending on the application, this function may be symmetric with respect to a vertical axis [23], [24]. We prove that in the case of symmetric possibility distribution, all the α-cuts are centered on the kernel leading to relevant properties which we exploit for the proposed representation format. Symmetric fuzzy numbers are defined as follows.

Definition 10 (Symmetric fuzzy number)**.** Let $\mathcal{A}$ be a fuzzy number, $\mu_{\mathcal{A}}(\cdot)$ its membership function and $m_{\mathcal{A}}$ its kernel. If $\mu_{\mathcal{A}}(\cdot)$ is symmetric around $m_{\mathcal{A}}$, i.e.,

$$\mu_{\mathcal{A}}(m_{\mathcal{A}} - x) = \mu_{\mathcal{A}}(m_{\mathcal{A}} + x), \ \forall\, x \in \mathbb{R}, \tag{10}$$

then $\mathcal{A}$ is called a symmetric fuzzy number.

Although symmetry is immaterial for the lower-upper interval representation, it is certainly relevant for the midpoint-radius one, as it is stated by the following Theorem.

Theorem 1. *Let $\mathcal{A}$ be a symmetric fuzzy number and $m_{\mathcal{A}} \in \mathbb{R}$ its kernel. Let $N \in \mathbb{N}$, $i \in \{1, \ldots, N\}$, and $\mathcal{A}_i = \langle m_{\mathcal{A},i}, \rho_{\mathcal{A},i} \rangle$, an α-cut. Then,*

$$\mathcal{A} \textit{ is symmetric} \iff m_{\mathcal{A},i} = m_{\mathcal{A}}, \ \forall\, i \in \{1, \ldots, N\}. \tag{11}$$

Proof: See Appendix A.

The above result can be graphically seen in Fig. 1, where the fuzzy number happens to be symmetric. Note how all the α-cuts are centered on the kernel. The property of symmetry is also preserved by fuzzy basic arithmetic operations, as stated by the following Theorem.

Theorem 2. *Let $\mathcal{A}$ and $\mathcal{B}$ be fuzzy numbers and $\circ \in \{+, -, \cdot, /\}$. If $\mathcal{A}$ and $\mathcal{B}$ are symmetric, then $\mathcal{C} = \mathcal{A} \circ \mathcal{B}$ is also symmetric.*

Proof: See Appendix B.

Now we apply Theorems 1 and 2 to propose an efficient encoding of symmetric fuzzy numbers, which is an original contribution of this paper. The proposed encoding consists of two components: the kernel (common midpoint for all α-cuts), and the set of radii (for each α-cut). Fuzzy operations are decomposed subsequently into two parts: (i) the computation of the kernel, and (ii) the computation of the individual radius of each α-cut. Algorithm 1 illustrates the above in the case of fuzzy multiplication. As we discuss in Section V, this encoding and algorithm allows us to save both bandwidth and execution time when performing fuzzy operations, provided that fuzzy numbers are symmetric.

Algorithm 1 Symmetric fuzzy multiplication in the midpoint-radius encoding. (See Table II for notation details.)

Input: Symmetric fuzzy operands $\mathcal{A}$ and $\mathcal{B}$.
Output: Symmetric fuzzy result $\mathcal{C} = \mathcal{A} \cdot \mathcal{B}$.

1: kernel: $m_{\mathcal{C}} = \square(m_{\mathcal{A}} \cdot m_{\mathcal{B}})$
2: **for** i in $1, \ldots, N$ **do**
3: radius: $\rho_{\mathcal{C},i} = \triangle(\eta + \frac{1}{2}\epsilon|m_{\mathcal{C}}| + (|m_{\mathcal{A}}| + \rho_{\mathcal{A},i})\rho_{\mathcal{B},i} + +|m_{\mathcal{B}}|\rho_{\mathcal{A},i})$
4: **end for**

B. Midpoint-increment encoding

Another interesting property of fuzzy numbers is that any α-cut is contained on all the α-cuts of a lower level; this is an immediate consequence of the membership function being monotone around the kernel.

Proposition 1. *Let $\mathcal{A}$ be a fuzzy number, $N \in \mathbb{N}$, $i, j \in \{1, \ldots, N\}$, and $\mathcal{A}_i, \mathcal{A}_j$, α-cuts. Then*

$$\alpha_i > \alpha_j \implies \mathcal{A}_i \subset \mathcal{A}_j. \tag{12}$$

Proof: See Appendix C.

We apply the above result to propose an alternative encoding for symmetric fuzzy numbers. This encoding also consists of two elements: the kernel, which is the common midpoint of every α-cut, and a set of radius increments, which correspond to the difference between the radii of two consecutive α-cuts. The formal definition and notation is as follows.

Definition 11 (Radius increments)**.** Let $\mathcal{A}$ be a fuzzy number, $N \in \mathbb{N}$ and $i \in \{1, \ldots, N\}$. Let $\{\alpha_1, \ldots, \alpha_N\} \in [0, 1]$, such that $\alpha_1 > \ldots > \alpha_N$. Let $\mathcal{A}_1, \ldots, \mathcal{A}_N$, be α-cuts and $\rho_{\mathcal{A},1}, \ldots, \rho_{\mathcal{A},N}$, their respective radii. We define radius increments, $\delta_{\mathcal{A},1}, \ldots, \delta_{\mathcal{A},N}$, as:

$$\delta_{\mathcal{A},i} = \begin{cases} \rho_{\mathcal{A},i} - \rho_{\mathcal{A},i-1} & \text{if } 2 \leq i \leq N, \\ \rho_{\mathcal{A},1} & \text{if } i = 1. \end{cases} \tag{13}$$

From the above definition, it is transparent that any radius of level i can be computed as a sum of increments:

$$\rho_{\mathcal{A},i} = \sum_{k=1}^{i} \delta_{\mathcal{A},k}. \tag{14}$$

The purpose of this alternative encoding is to increase accuracy in fuzzy computations. As the increments are by definition smaller than the radii, the rounding errors associated with increments computation are also smaller. We develop this concept in detail in Section III-C. First, let us introduce the algorithms to perform fuzzy operations in the midpoint-increment encoding.

Algorithms 2, 3 and 4 show how to compute the addition, subtraction, multiplication and inversion. Note that the number of floating-point operations needed by this representation format is not increased compared to the midpoint-radius representation. In fact, addition and subtraction require one fewer operation per α-cut than in the midpoint-radius encoding; multiplication and inversion require the same amount of operations per α-cut, provided that the α-cuts are treated sequentially (so certain computations associated to a given α-level can be re-used in the following level).

Algorithm 2 Symmetric fuzzy addition and subtraction in the midpoint-increment encoding. (See Table II for notation details.)

Input: Symmetric fuzzy operands $\mathcal{A}$ and $\mathcal{B}$.
Output: Symmetric fuzzy result $\mathcal{C} = \mathcal{A} \pm \mathcal{B}$.

1: kernel: $m_{\mathcal{C}} = \square(m_{\mathcal{A}} \pm m_{\mathcal{B}})$
2: first inc.: $\delta_{\mathcal{C},1} = \triangle(\frac{1}{2}\epsilon|m_{\mathcal{C}}| + \delta_{\mathcal{A},1} + \delta_{\mathcal{A},1})$
3: **for** i in $2, \ldots, N$ **do**
4: inc.: $\delta_{\mathcal{C},i} = \triangle(\delta_{\mathcal{A},i} + \delta_{\mathcal{B},i})$
5: **end for**

Algorithm 3 Symmetric fuzzy multiplication in the midpoint-increment encoding. (See Table II for notation details.)

Input: Symmetric fuzzy operands $\mathcal{A}$ and $\mathcal{B}$.
Output: Symmetric fuzzy result $\mathcal{C} = \mathcal{A} \cdot \mathcal{B}$.

1: kernel: $m_{\mathcal{C}} = \square(m_{\mathcal{A}} \cdot m_{\mathcal{B}})$
2: accum. in $\mathcal{A}$: $t_{\mathcal{A},1} = \triangle(|m_{\mathcal{A}}| + \delta_{\mathcal{A},i})$
3: accum. in $\mathcal{B}$: $t_{\mathcal{B},1} = \triangle(|m_{\mathcal{B}}|)$
4: first inc.: $\delta_{\mathcal{C},1} = \triangle(\eta + \frac{1}{2}\epsilon|m_{\mathcal{C}}| + t_{\mathcal{A},1}\delta_{\mathcal{B},1} + t_{\mathcal{B},1}\delta_{\mathcal{A},1})$
5: **for** i in $2, \ldots, N$ **do**
6: accum. in $\mathcal{A}$: $t_{\mathcal{A},i} = \triangle(t_{\mathcal{A},i-1} + \delta_{\mathcal{A},i})$
7: accum. in $\mathcal{B}$: $t_{\mathcal{B},i} = \triangle(t_{\mathcal{B},i-1} + \delta_{\mathcal{B},i})$
8: inc.: $\delta_{\mathcal{C},i} = \triangle(\eta + \frac{1}{2}\epsilon|m_{\mathcal{C}}| + t_{\mathcal{A},i}\delta_{\mathcal{B},i} + t_{\mathcal{B},i}\delta_{\mathcal{A},i})$
9: **end for**

Algorithm 4 Symmetric fuzzy inversion in the midpoint-increment encoding. (See Table II for notation details.)

Input: Symmetric fuzzy operand $\mathcal{B}$.
Output: Symmetric fuzzy result $\mathcal{C} = \dfrac{1}{\mathcal{B}}$.

1: kernel: $m_{\mathcal{C}} = \square\left(\dfrac{1}{m_{\mathcal{B}}}\right)$
2: accum.: $t_{\mathcal{B},1} = |m_{\mathcal{B}}|$
3: first inc.: $\delta_{\mathcal{C},1} = \triangle\left(\eta + \frac{1}{2}\epsilon|m_{\mathcal{C}}| + \dfrac{-\delta_{\mathcal{B},1}}{t_{\mathcal{B},1}(\delta_{\mathcal{B},1} - t_{\mathcal{B},1})}\right)$
4: **for** i in $2, \ldots, N$ **do**
5: accum.: $t_{\mathcal{B},i} = \triangle(t_{\mathcal{B},i-1} - \delta_{\mathcal{B},i-1})$
6: inc.: $\delta_{\mathcal{C},i} = \triangle\left(\eta + \frac{1}{2}\epsilon|m_{\mathcal{C}}| + \dfrac{-\delta_{\mathcal{B},i}}{t_{\mathcal{B},i}(\delta_{\mathcal{B},i} - t_{\mathcal{B},i})}\right)$
7: **end for**

In the algorithms above, the kernel's rounding error is added only to the first increment, so one operation per α-cut is saved. In the case of the multiplication and inversion, it is required to increase i sequentially. We introduce accumulators $t_{(\cdot)}$. In the multiplication, these accumulators store the maximum (absolute) value of two α-cuts, one in each operand, one level apart from each other. Only one operation is needed to update the accumulator at each α-level. Interestingly, these accumulators allow for a quite transparent calculation of the increment; the same amount of operations are needed as in computing the full radius in the midpoint-radius representation.

C. Error analysis

In this section we compare the accuracy of the methods to compute the radius of α-cuts in both proposed representation formats. We can see, from the different algorithms we have presented, that the main operation involved in this kind of computation is the rounded floating-point addition. The following

result, which concerns the error associated to floating-point summation algorithms, is useful for our further analysis.

Theorem 3 (Bound for the absolute error in summation algorithms [19])**.** *Let* $x_1, \ldots, x_n \in \mathbb{R}$ *and* $S_n = \sum_{i=1}^{n} x_i$. *Let* $\hat{S}_n$ *be an approximation to* S_n *computed by a summation algorithm, which performs exactly* $n-1$ *floating-point additions. Let* $\hat{T}_1, \ldots, \hat{T}_{n-1}$, *be the* $n-1$ *partial sums computed by such algorithm. Then,*

$$E_n := |S_n - \hat{S}_n| \leq \epsilon \sum_{i=1}^{n-1} |\hat{T}_i|, \tag{15}$$

where ϵ *is the relative rounding error.*

Proof: See [19].

Hence, the absolute error introduced by a floating-point summation algorithm is no greater than the epsilon machine, multiplied by the sum of magnitudes of all the intermediate sums.

Theorem 3 can be used to prove that the midpoint-increment representation improves the accuracy of basic fuzzy computations, compared to the original midpoint-radius representation. The result that we present below is a direct consequence of the increments being smaller in magnitude than the radii.

Theorem 4. *Let* $\circ \in \{+, -, \cdot, /\}$ *and* $\mathcal{A}, \mathcal{B}, \mathcal{C}$, *be fuzzy numbers such that* $\mathcal{C} = \mathcal{A} \circ \mathcal{B}$. *Let* $\mathcal{C}_i$ *be the* α*-cut of level* i *of* $\mathcal{C}$. *Let* $\rho_{\mathcal{C},i}$, *be the exact radius of* $\mathcal{C}_i$. *Let us call* $E_{rad}(\cdot)$, *the absolute error of a calculation performed using the midpoint-radius representation, and* $E_{inc}(\cdot)$, *the absolute error of a calculation performed using the midpoint-increment alternative. Then*

$$E_{inc}(\rho_{\mathcal{C},i}) \leq E_{rad}(\rho_{\mathcal{C},i}), \quad \forall\, i \geq 2. \tag{16}$$

Proof: See Appendix D.

Theorem 4 allows concluding that the gain in accuracy is driven ultimately by the fact that the radius at any α-level is always greater than the previous one. This, in turn, is a consequence of the monotony of the membership function. In the midpoint-increment encoding, we compute the smallest radius first, and then the others by simply adding the increments. In other words, we break large values into smaller ones, prior to perform fuzzy computations. As these computations are composed mainly of floating-point additions, this reduces rounding error in absolute value.

It can be shown that the result holds for any summation algorithm chosen to compute the increments and the radii, as long as it is the same for both. However, we omit this proof for sake of brevity. The

accuracy gain can also be quantified. It depends on the combination of several factors, e.g., the summation algorithm used (i.e., the order of the operations), the ratio of increment to radius at different α-levels, the "precision" of the intervals (i.e., the ratio of midpoint to radius) and the relative rounding error ϵ.

IV. Implementation Aspects

The proposed fuzzy arithmetic library is written in CUDA C and C++, and callable from either CPU and GPU code, effectively adapting itself to the underlying architecture [25].[1]

The implementation consists of a series of wrappers. At the user-end, we have the two fuzzy classes, one for each type of fuzzy encoding. We decided to offer the two representation formats, lower-upper for the general case, and midpoint-radius for when the user is dealing with symmetric fuzzy numbers. The lower-upper fuzzy class relies on a lower-upper interval class, which serves the purpose of holding the α-cuts. The lower-upper interval class, in turn, relies on a rounded arithmetic class for operating between the α-cuts, according to the rules of interval arithmetic. The midpoint-radius fuzzy class is linked directly to the rounded-arithmetic class, as the α-cuts are managed as a whole and not individually, to make use of the result from Theorem 2 in the previous section. Ultimately, the rounded arithmetic class is a wrapper of C and CUDA compiler *intrinsics* directly associated to specific machine instructions.

The fuzzy template classes are parametrized by the number of α-cuts N and the data type T. In the lower-upper encoding, N and T specify the size and type of the array of intervals which corresponds to the different α-cuts. In the midpoint-radius encoding, N and T are the size and type of the array of radii, and T is also the type of the scalar midpoint common to all the α-cuts.

The basic arithmetic operators are overloaded to work on both fuzzy classes. The loop that sweeps over the set of α-cuts (see Algorithms 1, 2, 3 and 4) is a sequential one; we decided not to spread it among different threads in GPU code, since most applications in real life do not involve in their data model more than three to four degrees of uncertainty [2]–[8], which is way too low to conveniently exploit thread level parallelism (TLP) on GPUs. However, TLP may be exploited in vector operations involving fuzzy numbers, through kernels that assigns each element of a fuzzy array to a different thread.

Performance of basic operations over complex data type such as fuzzy numbers are impacted by numerous factors. Following subsections discuss and compare the two considered representation formats, namely lower-upper and midpoint-radius, regarding number of instructions, memory requirements and instruction level parallelism (ILP).

[1]The source code is available at https://code.google.com/p/fuzzy-gpu/.

A. Number of instructions

Table III shows the number of instructions, such as basic operations with different rounding, minimum, maximum and absolute value, required in the addition, multiplication and inversion of different data types, including fuzzy numbers. In fuzzy data types, N represents the number of α-cuts. Comparing the two main fuzzy approaches, i.e., lower-upper and midpoint-radius, the following are relevant remarks. The lower-upper fuzzy requires only 3 operations less than midpoint-radius in the addition, but when it comes to the multiplication, it requires 2.8 times more operations per α-cut. In the case of a full division, which corresponds to one inversion plus one multiplication, the lower-upper fuzzy requires $16N$ operations, whereas the midpoint-radius requires $11 + 9N$, i.e., lower-upper requires 1.77 more operations per α-cut. Therefore, just by comparing the number of instructions, we can anticipate that the midpoint-radius representation shall bring a speed-up over lower-upper.

TABLE III
NUMBER OF INSTRUCTIONS PER OPERATION, FOR DIFFERENT DATA TYPES.

Data type	**Number of instructions**		
	Addition	**Multiplication**	**Inversion**
Scalar	1	1	1
Lower-upper interval	2	14	2
Midpoint-radius interval	5	11	9
Lower-upper fuzzy	$2N$	$14N$	$2N$
Midpoint-radius fuzzy	$3 + 2N$	$6 + 5N$	$5 + 5N$
Midpoint-increment fuzzy	$4 + N$	$6 + 5N$	$5 + 5N$

N: number of α-cuts.

B. Memory usage

Table IV shows the memory space required to store different numbers in data types. The units have been normalized to the size of one scalar. In the fuzzy case, once again, N represents the number of α-cuts. The lower-upper fuzzy requires double the space than the midpoint-radius and midpoint-increment representation. As the bandwidth requirement of both the latter is half the one of the former, an application that needs to access memory at a high rate will definitely benefit from that property.

We should mention that the memory footprint of all these representation formats can be further reduced by lowering the type used to store either one of the bounds, the radius or the increment, in the case where

the α-cuts are not too wide. The impact is solely on the width of the α-cuts which can be considered and not on the dynamic of numbers. For example, the lower bound can be stored in double precision and the upper bound in single precision; or, similarly, the midpoint in double precision and the radius in single precision. This again is an advantage for the proposed formats, as the lower bound is an array of scalars, whereas the midpoint is only one scalar.

TABLE IV
MEMORY REQUIREMENTS OF DIFFERENT DATA TYPES.

Data type	Memory usage
Scalar	1
Lower-upper interval	2
Midpoint-radius interval	2
Lower-upper fuzzy	$2N$
Midpoint-radius fuzzy	$1 + N$
Midpoint-increment fuzzy	$1 + N$

N: number of α-cuts.

C. Instruction level parallelism

Ideal ILP, defined as the ratio of the number of instructions to the number of levels in the dependency tree [26], is a good measure of how a given sequence of instructions could be handled on today's but also future architectures. The higher the ideal ILP is, the higher the amount of instructions that can be pipelined during the execution of a given application. However, a bigger ideal ILP requires a larger amount of hardware resources.

Table V shows the ideal ILP in the addition, multiplication and inversion of different data types. The number of α-cuts in lower-upper and midpoint-radius fuzzy data types does not affect the size of the dependency tree, as each α-cut is processed independently from all others. The same is valid for the addition in the midpoint-increment representation. However it does not hold in the multiplication and inversion, where we face dependencies between computations belonging to different α-cuts. Accordingly, the size of the dependency tree depends in this case on N, the number of α-levels.

In general, lower-upper fuzzy exhibits more ideal ILP than midpoint-radius and midpoint-increment in either the addition, the multiplication and the inversion. However, ILP is exploited differently depending

on the GPU generation as well as the precision in use (single or double). For example, GPUs with CUDA capability 3.0 can schedule up to 2 independent instructions for a given warp scheduler. The impact of ILP on example applications is also studied in the next section.

TABLE V
ILP PER ARITHMETICAL OPERATION, FOR DIFFERENT DATA TYPES.

Data type	**ILP**		
	Addition	**Multiplication**	**Inversion**
Scalar	1	1	1
Lower-upper interval	2	$\frac{14}{3}$	2
Midpoint-radius interval	$\frac{5}{4}$	$\frac{11}{5}$	$\frac{9}{2}$
Lower-upper fuzzy	$2N$	$\frac{11}{5}N$	$2N$
Midpoint-radius fuzzy	$\frac{3}{4}+\frac{1}{2}N$	$\frac{6}{5}+N$	$1+\frac{4}{5}N$
Midpoint-increment fuzzy	$1+\frac{1}{4}N$	$\frac{6+5N}{3+N}$	$\frac{5+5N}{3+N}$

N: number of α-cuts.

V. TESTS AND RESULTS

In this section we evaluate the performance of the proposed fuzzy library on GPU. According to the CUDA Programming Guide, GPU performance is a matter of balance between computing intensity and memory usage [27]. With this regard, we consider two types of applications: compute-bound application, where arithmetic instructions are dominant over memory accesses; and memory-bound application, where the opposite is true. By measuring the performance of the library in this two extreme cases, we obtain a general idea of the behaviour to expect for any application in between.

The computing environment used in the proposed case study is defined in Table VI. To generate different scenarios, we varied some key parameters as shown in Table VII.

A. Compute-bound test: AXPY kernel

Figure 2 shows a kernel that computes the n-th element of the series $x_{k+1} = ax_k + b$, where all the values are fuzzy numbers. The threads read the value of a from an input array, perform the computation through an iterative loop, and then write the result to an output array. As we increment the number of

TABLE VI
COMPUTING ENVIRONMENT USED IN THE TESTS.

Hardware	**Compute capability**	**Number of cores**
Xeon X560 CPU	N/A	12 (1 used)
GeForce GTX 480 GPU	2.0	480
GeForce GTX 680 GPU	3.0	1,536
Software	**Version**	
GCC	4.8.2	
CUDA	6.0	

TABLE VII
PARAMETERS AND THEIR VALUES, USED IN GENERATING DIFFERENT TESTS SCENARIOS.

Parameter	**Values**
Number of α-cuts	1 to 24
Fuzzy encoding	Lower-upper
	Midpoint-radius
Precision	Single
	Double

iterations, n, the number of floating point instructions grows, however, the number of accesses to global memory remains constant, one for loading the starting value and one for storing the result. In this way, we can arbitrarily increment the ratio of floating-point instructions to global memory accesses.

We measured the execution time of our kernel, as well as other performance indicators, under a series

```
#include "fuzzy_lib.h"

template<class T, int N>
__global__ void axpy(int n,
fuzzy<T, N> * input,
fuzzy<T, N> b,
fuzzy<T, N> * output)
{
  int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
  fuzzy<T, N> a, c = 0;
  a = input[thread_id];
  for (int i = 0; i < n; i++)
  c = a * c + b;
  output[thread_id] = c;
}
```

Fig. 2. AXPY kernel, compute-bound test.

of scenarios generated by varying parameters, according to Table VII. The kernel arguments were tuned for different runs in order to achieve maximum occupancy of the architecture. The results are plotted in Fig. 3. Figure 3(a) shows different performances achieved by different configurations on different CPU and GPU architectures. Performance is represented on a logarithmic scale as the number of iteration performed per second depending on the number of α-cut. The Xeon CPU is running a single threaded version of the kernel using (i) our library, and (ii) the java library in [22]; however, note that the latter does not provide certified interval arithmetic with correct rounding attributes. We observe differences and performance gains coming from heterogeneous sources. First, there is a pure architectural gain of about three orders of magnitude on passing from CPU to GPU to execute the code. Second, within the GPU, there is an algorithmic gain on choosing the midpoint-radius over the lower-upper encoding, although this remains in the same order of magnitude. Finally, there is a gain of one order of magnitude by considering single instead of double precision format in fuzzy calculations.

The most interesting of these gains is the one driven by the algorithm, as it puts on relief the differences between lower-upper and midpoint-radius fuzzy implementations. Figure 3(b) shows the speed-up achieved by midpoint-radius over lower-upper using different precision on two GPU architectures. The shapes we observe are the results of differences in the number of operations, memory size and stronger data dependencies in the case of the midpoint-radius. The difference in the number of operations accounts for the main trend and the other factors account for the irregularities.

The main trend is observed between 1 and 8 to 9 α-cuts. In this range, the curve follows a growing pattern which is consistent with the theoretical analysis of the number of operations per different type of fuzzy operations, presented in section IV. According to Table III, the ratio between the number of operations per cycle of the AXPY loop required by each fuzzy encoding is:

$$r_c(N) = \frac{16N}{7 + 7N}, \tag{17}$$

where N is the number of α-cuts. Note that equation (17) conveniently describes the shape of the speed-up curve between 1 and 8 to 9 α-cuts, as can be seen in Fig. 3b. Moreover, such behaviour does not depend on the architecture neither on the precision. This result validates the theoretical analysis and illustrates the performance of the library when architecture constraints are not binding. From 9 to 10 α-cuts, the effect of register spilling and local memory accesses with high latency becomes preponderant. It is worth noting that CUDA devices of compute capability 2.0 and 3.0 may allocate up to 63 registers of 32-bits

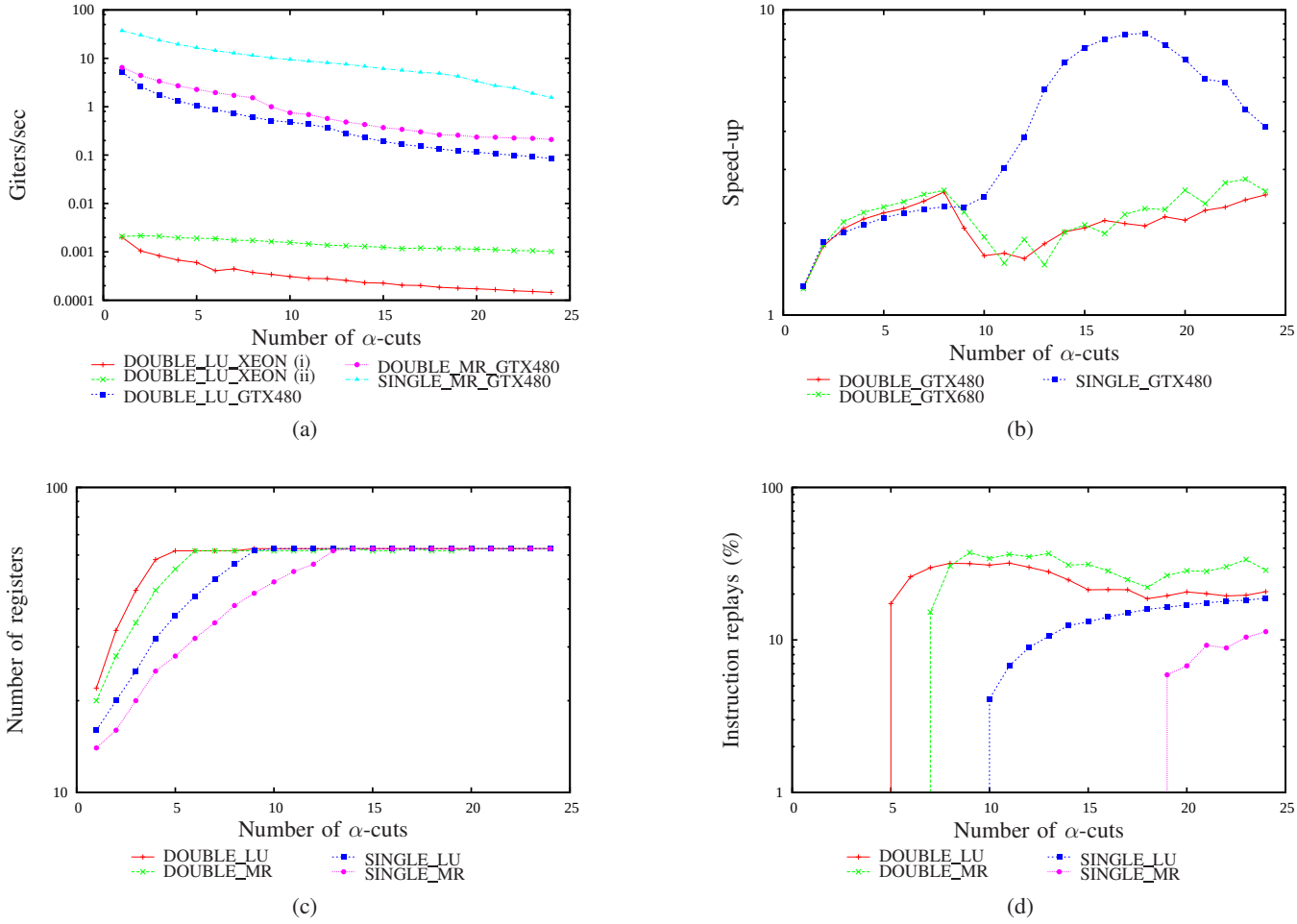

Fig. 3. Results of the compute-bound test: (a) Performance comparison of different representation formats and scenarios. (b) Speed-up achieved by midpoint-radius over lower-upper, algorithmic gain. (c) Registers used by either representation format. (d) Instruction replays due to local memory accesses, in either representation format.

per thread on a kernel execution. Double precision values are stored in two consecutive registers. When this limit is binding, the kernel uses local memory to allocate extra data. Figure 3(c) shows the number of registers used by the AXPY kernel per implementation. Figure 3(d) shows the percentage of replayed instructions due to local memory accesses.

When using the lower-upper encoding and single precision values, the 63 registers limit is reached at about 9 α-cuts. Local memory transactions start at 10 α-cuts. Figure 3(b) shows an increase of the speed-up in single precision for 10 α-cuts, as the midpoint-radius implementation is not suffering from register spilling yet. The size of the midpoint-radius fuzzy being about the half of the lower-upper, spilling only starts at double the α-cuts, i.e., 18. For more than 18 α-cuts, the speed-up curve in single precision moves back to the ideal shape that can be explained purely by the ratio of number of operations.

In double precision, the scenario is slightly more complex. Both lower-upper and midpoint-radius fuzzy

start spilling registers relatively early, between 6 and 8 α-cuts. Figure 3(b) shows that there is a slight drop in the speed-up in double precision, at about 9 α-cuts. The performance of the midpoint-radius fuzzy implementation decreases in this zone. We believe this is due to spilling the midpoint of one of the fuzzy numbers involved in the calculation. Note that midpoint-radius fuzzy arithmetic proceeds by computing the midpoint first and then uses the result to calculate all the radii. If one midpoint is spilled to off-chip local memory, there might not be enough independent arithmetic instructions to hide the latency of accessing that value. This effect cannot be appreciated in single precision as register spilling does not have a relevant impact despite the high number of α-cuts considered. This proves that performance is mainly driven by the amount of memory required to store data. Temporal dependency among data has a limited impact.

As stated in Section IV, real-world applications typically do not involve more than 3 to 4 α-cuts. In this range, register spilling is not an issue. However, if a larger number of α-cuts is necessary, we may consider the following. CUDA allows to change the amount of memory allocated to different GPU features, such as L1 cache and shared memory [27]. Typically, 64 KB of memory are to be distributed among these two functions. In our experiment, we observe an interesting trade-off. On one hand, increasing L1 memory cache allows to increase the number of α-cuts for which register spilling is not an issue. On the other hand, increasing shared memory allows to mitigate the negative effect of register spilling whenever it appears.

B. Memory-bound test: sort by keys

Figure 4 shows a THRUST [28] program that sorts a vector of fuzzy numbers on the device. This example shows how easily the fuzzy type is integrated to other GPU libraries, as all the arithmetic operators are overloaded.

The vector is sorted by keys, which are integers of 32 bits. The sorting algorithm used by THRUST in this case is radix sort. The THRUST kernels read the fuzzy array from global memory, sort it on the device, and then write the sorted array back to global memory. The sorting process performs one step per key bit, i.e., 32 steps in this case. At each step, each element in the fuzzy array is read and copied into a new position. This is a typical case of memory-bound application.

GPU sorting time is measured when varying different parameters, according to Table VII. Results for the GTX 480 are plotted in Fig. 5. Figure 5(a) shows performance of different fuzzy encodings and precisions, in terms of millions of sorted elements per second. Figure 5(b) presents the same information

```
#include "fuzzy_lib.h"
#include <thrust/sort.h>

int main(){
  thrust::device_vector<fuzzy<double, 4> > d_a(M);
  thrust::device_vector<unsigned int> k(M);
  ...
  thrust::sort_by_key(k.begin(), k.end(), d_a.begin());
}
```

Fig. 4. THRUST's sort by keys, memory-bound test.

in terms of speed-up of midpoint-radius over lower-upper. The time spent in transferring data between host and device is not considered in the experiment.

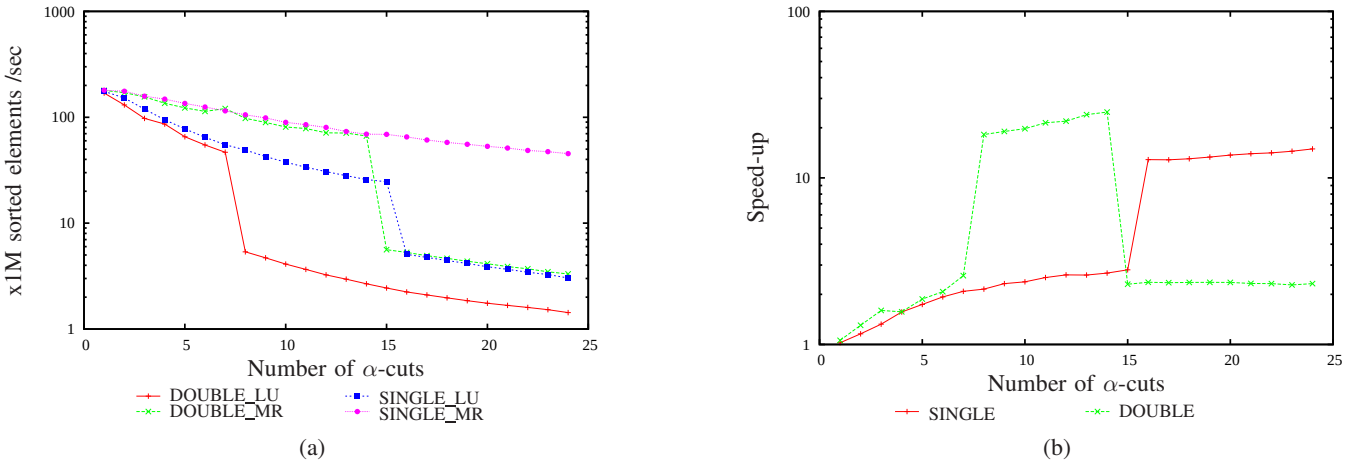

Fig. 5. Results of the memory-bound test: (a) Performance comparison of different representation formats and scenarios. (b) Speed-up achieved by midpoint-radius over lower-upper, algorithmic gain.

Figure 5(a) shows that for the same number of α-cuts, midpoint-radius encoding allows to sort twice the amount of fuzzy numbers than lower-upper encoding, whether we are in single or double precision. For this application, the size of the data array being sorted becomes the main factor driving performance. Midpoint-radius requires half the size of lower-upper representation, thanks to a shared midpoint between all the α-cuts. The memory ratio of midpoint-radius to lower-upper representation can be obtained from Table IV:

$$r_m(N) = \frac{2N}{1+N}. \tag{18}$$

Observe that the speed-up curves in Fig. 5(b) follow approximately (18), except between 8 and 14 α-cuts in double precision and for more than 15 α-cuts in single precision, where sudden performance drops are causing the speed-up to fluctuate. THRUST's sort uses shared memory to speed-up the sorting process. When the values being treated by threads within one Streaming Multiprocessor (SM) do not longer fit

in shared memory, it starts using global memory with a direct impact over performance. As we stated above, lower-upper fuzzy needs twice the amount of memory than midpoint-radius fuzzy. As a result of this, it saturates shared memory at half the number of α-cuts. When both architectures saturate shared memory, the speed-up curve goes back to the normal behaviour, explained by the ratio of memory sizes. Note that single precision midpoint-radius does not saturate shared memory in this experiment and keeps a high performance all along the considered range.

VI. Conclusion

Fuzzy arithmetic is an efficient tool to handle uncertainty in numerical modelling. The α-cut concept, which allows to express fuzzy operations in terms of simpler interval algebra, is currently accepted as the implementation standard, with the lower-upper interval representation as the native format of choice. However, this approach suffers from performance, memory and floating-point error issues, especially when several levels of uncertainty are considered. In this article, we propose a novel approach to represent fuzzy numbers having a symmetric membership function, that allows to deal with such issues. The proposed midpoint-radius format addresses the issues of performance and memory usage. The proposed midpoint-increment format addresses the issue of floating-point rounding errors.

The paper also describes the implementation of fuzzy arithmetic using the proposed formats in CUDA and C++, and discusses the performance impact of TLP, ILP and memory usage for recent NVIDIA GPUs. We show through a theoretical study and experimental results considering compute-bound and memory-bound applications, that a gain of 2 to 20 can be obtained by using the midpoint-radius encoding instead of the conventional lower-upper format.

Future work will consider the extension of the library in order to include more operations, such as the square root and the *c-mean*, the latter associated to fuzzy clustering. Also, an efficient way of handling asymmetry (e.g., in the case of the square root operation) will be investigated through the use of symmetric envelopes.

Appendix A
Proof of Theorem 1

We prove the reciprocal, i.e.,

$$\mathcal{A} \text{ is non-symmetric} \iff \exists\, j \in \{1, \ldots, N\},\ m_{\mathcal{A},j} \neq m_{\mathcal{A}}. \tag{19}$$

i) ($\Longrightarrow$)

If $\mathcal{A}$ is non-symmetric, then there is $x_0 \in \mathbb{R}$ such that:

$$\mu_{\mathcal{A}}(m_{\mathcal{A}} - x_0) \neq \mu_{\mathcal{A}}(m_{\mathcal{A}} + x_0). \tag{20}$$

Let $j \in \{1, \ldots, N\}$ and $\mathcal{A}_j = [l_{\mathcal{A},j}, u_{\mathcal{A},j}]$ be an α-cut, such that:

$$l_{\mathcal{A},j} = m_{\mathcal{A}} - x_0. \tag{21}$$

Let $m_{\mathcal{A},j}$ be the midpoint of $\mathcal{A}_j$ and assume:

$$m_{\mathcal{A},j} = m_{\mathcal{A}}. \tag{22}$$

Combining equations (21) and (22), we have the following for the upper bound of $\mathcal{A}_j$:

$$u_{\mathcal{A},j} = 2m_{\mathcal{A},j} - l_{\mathcal{A},j} = 2m_{\mathcal{A}} - l_{\mathcal{A},j} = m_{\mathcal{A}} + x_0. \tag{23}$$

By definition of α-cut, in equation (8):

$$\mu_{\mathcal{A}}(l_{\mathcal{A},j}) = \mu_{\mathcal{A}}(u_{\mathcal{A},j}). \tag{24}$$

And replacing $l_{\mathcal{A},j}$ and $u_{\mathcal{A},j}$ from equations (21) and (23), respectively, in (24):

$$\mu_{\mathcal{A}}(m_{\mathcal{A}} - x_0) = \mu_{\mathcal{A}}(m_{\mathcal{A}} + x_0), \tag{25}$$

which clearly contradicts equation (20). Hence, the assumption made in (22) cannot hold.

ii) ($\Longleftarrow$)

Again, let $\mathcal{A}_j = [l_{\mathcal{A},j}, u_{\mathcal{A},j}]$, and also $x_0 \in \mathbb{R}$ such that:

$$l_{\mathcal{A},j} = m_{\mathcal{A}} - x_0. \tag{26}$$

By definition of α-cut, in (8):

$$\mu_{\mathcal{A}}(l_{\mathcal{A},j}) = \mu_{\mathcal{A}}(u_{\mathcal{A},j}). \tag{27}$$

Combining the assumption, $m_{\mathcal{A},j} \neq m_{\mathcal{A}}$, and (26), we have, for $u_{\mathcal{A},j}$:

$$u_{\mathcal{A},j} = 2m_{\mathcal{A},j} - l_{\mathcal{A},j} \neq 2m_{\mathcal{A}} - l_{\mathcal{A},j} = m_{\mathcal{A}} + x_0. \tag{28}$$

But by construction, both numbers $u_{\mathcal{A},j}$ and $m_{\mathcal{A}} + x_0$ are greater than $m_{\mathcal{A}}$. Thus, we can apply the assumption in (6c) regarding monotonocity of the membership function, obtaining:

$$\mu_{\mathcal{A}}(u_{\mathcal{A},j}) \neq \mu_{\mathcal{A}}(m_{\mathcal{A}} + x_0). \tag{29}$$

And replacing equations (26) and (27) in (29):

$$\mu_{\mathcal{A}}(m_{\mathcal{A}} - x_0) \neq \mu_{\mathcal{A}}(m_{\mathcal{A}} + x_0). \tag{30}$$

Hence $\mathcal{A}$ is non-symmetric.

■

APPENDIX B
PROOF OF THEOREM 2

Let i be a level of uncertainty. Let the kernels, α-cuts, midpoints and radii associated to $\mathcal{A}, \mathcal{B}, \mathcal{C}$, be noted in accordance with Table II. According to equation (9):

$$\mathcal{C}_i = \mathcal{A}_i \circ \mathcal{B}_i. \tag{31}$$

As $\mathcal{A}$ and $\mathcal{B}$ are symmetric, by equation (11) from Theorem 1:

$$m_{\mathcal{A},i} = m_{\mathcal{A}}, \quad m_{\mathcal{B},i} = m_{\mathcal{B}}, \quad \forall\, i. \tag{32}$$

The kernel of $\mathcal{C}$ can be trivially obtained as the midpoint of the α-cut of level 1, i.e.:

$$m_{\mathcal{C}} = m_{\mathcal{C},1}. \tag{33}$$

If $\circ \in \{+, -, \cdot\}$, then we have from equations (5), (31), (32) and (33):

$$\begin{aligned} m_{\mathcal{C},i} &= \square(m_{\mathcal{A},i} \circ m_{\mathcal{B},i}) \\ &= \square(m_{\mathcal{A},1} \circ m_{\mathcal{B},1}) \\ &= m_{\mathcal{C},1} \\ &= m_{\mathcal{C}}, \quad \forall\, i. \end{aligned} \tag{34}$$

Hence $\mathcal{C}$ is symmetric. If $\circ$ is the inversion, the analogous calculation yields the result. ■

APPENDIX C
PROOF OF PROPOSITION 1

Let us recall the following result which characterizes inclusion of intervals [20]:

$$\langle m_1, \rho_1 \rangle \subset \langle m_2, \rho_2 \rangle \iff |m_2 - m_1| < \rho_2 - \rho_1. \tag{35}$$

Let $\mathcal{A}_i = \langle m_{\mathcal{A},i}, \rho_{\mathcal{A},i} \rangle$ and $\mathcal{A}_j = \langle m_{\mathcal{A},j}, \rho_{\mathcal{A},j} \rangle$. Also, let $\mu_{\mathcal{A}}(\cdot)$ be $\mathcal{A}$'s membership function and $m_{\mathcal{A}}$ its kernel. By definition of α-cut, in (8):

$$\mu_{\mathcal{A}}(m_{\mathcal{A},i} - \rho_{\mathcal{A},i}) = \mu_{\mathcal{A}}(m_{\mathcal{A},i} + \rho_{\mathcal{A},i}) = \alpha_i, \tag{36a}$$

$$\mu_{\mathcal{A}}(m_{\mathcal{A},j} - \rho_{\mathcal{A},j}) = \mu_{\mathcal{A}}(m_{\mathcal{A},j} + \rho_{\mathcal{A},j}) = \alpha_j, \tag{36b}$$

Note that the kernel $m_{\mathcal{A}}$ can be interpreted as the α-cut of level 1, so we have the following order relations:

$$m_{\mathcal{A},i} - \rho_{\mathcal{A},i} \leq m_{\mathcal{A}} \leq m_{\mathcal{A},i} + \rho_{\mathcal{A},i}, \tag{37a}$$

$$m_{\mathcal{A},j} - \rho_{\mathcal{A},j} \leq m_{\mathcal{A}} \leq m_{\mathcal{A},j} + \rho_{\mathcal{A},j}. \tag{37b}$$

Now we invoke the assumptions in (6b) and (6c) from Definition 6, regarding strict monotonocity of the membership function, and also the assumption $\alpha_i > \alpha_j$. Then from equations (36) and (37):

$$m_{\mathcal{A},i} - \rho_{\mathcal{A},i} > m_{\mathcal{A},j} - \rho_{\mathcal{A},j}, \tag{38a}$$

$$m_{\mathcal{A},i} + \rho_{\mathcal{A},i} < m_{\mathcal{A},j} + \rho_{\mathcal{A},j}. \tag{38b}$$

Combining both:

$$-(\rho_{\mathcal{A},j} - \rho_{\mathcal{A},i}) < m_{\mathcal{A},j} - m_{\mathcal{A},i} < \rho_{\mathcal{A},j} - \rho_{\mathcal{A},i}. \tag{39}$$

And more synthetically:

$$|m_{\mathcal{A},j} - m_{\mathcal{A},i}| < \rho_{\mathcal{A},j} - \rho_{\mathcal{A},i}. \tag{40}$$

■

APPENDIX D
PROOF OF THEOREM 4

We prove the result for the addition and subtraction. The other two cases (multiplication and inversion) can be deduced using a similar procedure. The notation used below is based on that shown in Table II.

We proceed by induction.

i) (Assume $i = 2$.)

In the midpoint-increment representation, according to equation (14), $\rho_{\mathcal{C},2}$ can be computed as:

$$\rho_{\mathcal{C},2} = \delta_{\mathcal{C},1} + \delta_{\mathcal{C},2}. \tag{41}$$

Thus, the error in computing $\rho_{\mathcal{C},2}$ is the sum of the errors in computing $\delta_{\mathcal{C},1}$ and $\delta_{\mathcal{C},2}$, i.e,:

$$E_{inc}(\rho_{\mathcal{C},2}) = E_{inc}(\delta_{\mathcal{C},1}) + E_{inc}(\delta_{\mathcal{C},2}). \tag{42}$$

By applying Theorem 3 to bound these two error terms, we have:

$$\delta_{\mathcal{C},1} = \triangle(\frac{1}{2}\epsilon|m_C| + \delta_{\mathcal{A},1} + \delta_{\mathcal{B},1}). \tag{43}$$

We compute the above with a summation algorithm, that performs the following two steps:

$$\begin{aligned} \hat{T}_1 &= \triangle(\frac{1}{2}\epsilon|m_C| + \delta_{\mathcal{A},1}), \\ \hat{T}_2 &= \triangle(\hat{T}_1 + \delta_{\mathcal{B},1}). \end{aligned} \tag{44}$$

Then, from equation (15):

$$\begin{aligned} E_{inc}(\delta_{\mathcal{C},1}) &\leq \epsilon(|\hat{T}_1| + |\hat{T}_2|) \\ &= \epsilon(2\delta_{\mathcal{A},1} + \delta_{\mathcal{B},1}) + \epsilon^2|m_C|. \end{aligned} \tag{45}$$

We do the same for $\delta_{\mathcal{C},2}$, in this case obtaining:

$$E_{inc}(\delta_{\mathcal{C},2}) \leq \epsilon(\delta_{\mathcal{A},2} + \delta_{\mathcal{B},2}). \tag{46}$$

Replacing equations (45) and (46) in (42), and using equation (14) again:

$$E_{inc}(\rho_{\mathcal{C},2}) \leq \epsilon(\delta_{\mathcal{A},1} + \rho_{\mathcal{A},2} + \rho_{\mathcal{B},2}) + \epsilon^2|m_C|. \tag{47}$$

Following an analogous procedure, we obtain a bound on the error in computing $\rho_{\mathcal{C},2}$ using the midpoint-radius representation:

$$E_{rad}(\rho_{\mathcal{C},2}) \leq \epsilon(\delta_{\mathcal{A},1} + \delta_{\mathcal{A},2} + \rho_{\mathcal{A},2} + \rho_{\mathcal{B},2}) + \epsilon^2|m_C|. \tag{48}$$

And as $\delta_{\mathcal{A},2} > 0$, we have our result:

$$E_{inc}(\rho_{\mathcal{C},2}) \leq E_{rad}(\rho_{\mathcal{C},2}). \tag{49}$$

ii) (Assume $E_{inc}(\rho_{\mathcal{C},i}) \leq E_{rad}(\rho_{\mathcal{C},i}),\ \forall\, i \geq 2$.)

In the midpoint-increment representation, $\rho_{\mathcal{C},i+1}$ can be computed as:

$$\rho_{\mathcal{C},i+1} = \rho_{\mathcal{C},i} + \delta_{\mathcal{C},i+1}. \tag{50}$$

As in the previous part, we decompose the error in two terms:

$$E_{inc}(\rho_{\mathcal{C},i+1}) = E_{inc}(\rho_{\mathcal{C},i}) + E_{inc}(\delta_{\mathcal{C},i+1}). \tag{51}$$

Now applying the induction assumption:

$$E_{inc}(\rho_{\mathcal{C},i+1}) \leq E_{rad}(\rho_{\mathcal{C},i}) + E_{inc}(\delta_{\mathcal{C},i+1}). \tag{52}$$

Using Theorem 3:

$$E_{inc}(\rho_{\mathcal{C},i+1}) \leq \epsilon(\rho_{\mathcal{A},i} + \rho_{\mathcal{A},i+1} + \rho_{\mathcal{B},i+1}) + \epsilon^2|m_C|. \tag{53}$$

Similarly:

$$E_{rad}(\rho_{\mathcal{C},i+1}) \leq \epsilon(2\rho_{\mathcal{A},i+1} + \rho_{\mathcal{B},i+1}) + \epsilon^2|m_C|. \tag{54}$$

And, since $\rho_{\mathcal{A},i} < \rho_{\mathcal{A},i+1}$:

$$E_{inc}(\rho_{\mathcal{C},i+1}) \leq E_{rad}(\rho_{\mathcal{C},i+1}). \tag{55}$$

■

REFERENCES

[1] W. Siler and J. J. Buckley, *Fuzzy expert systems and fuzzy reasoning*. John Wiley & Sons, 2005.

[2] Y. Yoshida, M. Yasuda, J.-i. Nakagami, and M. Kurano, "A new evaluation of mean value for fuzzy numbers and its application to american put option under uncertainty," *Fuzzy Sets and Systems*, vol. 157, no. 19, pp. 2614–2626, 2006.

[3] J.-C. Buisson and A. Garel, "Balancing meals using fuzzy arithmetic and heuristic search algorithms," *IEEE Transactions on Fuzzy Systems*, vol. 11, no. 1, pp. 68–78, Feb 2003.

[4] S. Bonvicini, P. Leonelli, and G. Spadoni, "Risk analysis of hazardous materials transportation: evaluating uncertainty by means of fuzzy logic," *Journal of Hazardous Materials*, vol. 62, no. 1, pp. 59–74, 1998.

[5] J. Bondia, A. Sala, J. Pico, and M. Sainz, "Controller design under fuzzy pole-placement specifications: An interval arithmetic approach," *IEEE Transactions on Fuzzy Systems*, vol. 14, no. 6, pp. 822–836, Dec 2006.

[6] C.-T. Chen, C.-T. Lin, and S.-F. Huang, "A fuzzy approach for supplier evaluation and selection in supply chain management," *International journal of production economics*, vol. 102, no. 2, pp. 289–301, 2006.

[7] V. Miranda and J. Saraiva, "Fuzzy modelling of power system optimal load flow," in *Power Industry Computer Application Conference, 1991. Conference Proceedings*. IEEE, 1991, pp. 386–392.

[8] M. Cortes-Carmona, R. Palma-Behnke, and G. Jimenez-Estevez, "Fuzzy arithmetic for the DC load flow," *IEEE Transactions on Power Systems*, vol. 25, no. 1, pp. 206–214, Feb 2010.

[9] D. Dubois and H. Prade, "Operations on fuzzy numbers," *International Journal of systems science*, vol. 9, no. 6, pp. 613–626, 1978.

[10] V. Volkov, "Better performance at lower occupancy," in *Proceedings of the GPU Technology Conference, GTC*, vol. 10, 2010.

[11] S. Collange, M. Daumas, and D. Defour, "Interval arithmetic in CUDA," *GPU Computing Gems*, vol. 2, p. 99, 2012.

[12] D. T. Anderson, R. H. Luke, and J. M. Keller, "Speedup of fuzzy clustering through stream processing on graphics processing units," *IEEE Transactions on Fuzzy Systems*, vol. 16, no. 4, pp. 1101–1106, 2008.

[13] M. Martínez-Zarzuela, F. J. D. Pernas, J. F. D. Higuera, and M. A. Rodríguez, "Fuzzy ART neural network parallel computing on the GPU," in *Computational and Ambient Intelligence*. Springer, 2007, pp. 463–470.

[14] H. Brönnimann, G. Melquiond, and S. Pion, "The design of the Boost interval arithmetic library," *Theoretical Computer Science*, vol. 351, no. 1, pp. 111–118, 2006.

[15] N. Revol and F. Rouillier, "The MPFI library," 2001.

[16] F. Goualard, "Gaol 3.1.1: Not just another interval arithmetic library," *Laboratoire d'Informatique de Nantes-Atlantique*, vol. 4, 2006.

[17] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, pp. 1–70, Aug 2008.

[18] S. M. Rump, "Fast and parallel interval arithmetic," *BIT Numerical Mathematics*, vol. 39, no. 3, pp. 534–554, 1999.

[19] N. J. Higham, *Accuracy and stability of numerical algorithms*. Siam, 2002.

[20] A. Neumaier, "A distributive interval arithmetic," *Freiburger Intervall-Berichte*, vol. 82, no. 10, pp. 31–38, 1982.

[21] A. Anile, S. Deodato, and G. Privitera, "Implementing fuzzy arithmetic," *Fuzzy Sets and Systems*, vol. 72, no. 2, pp. 239–250, 1995.

[22] N. Kolarovi, "Fuzzy numbers and basic fuzzy arithmetics (+, -, *, /, 1/x) implementation written in Java." 2013.

[23] C.-H. Chang and Y.-C. Wu, "The genetic algorithm based tuning method for symmetric membership functions of fuzzy logic control systems," in *International IEEE/IAS Conference on Industrial Automation and Control: Emerging Technologies, 1995.* IEEE, 1995, pp. 421–428.

[24] J. Mendel and H. Wu, "Type-2 fuzzistics for symmetric interval type-2 fuzzy sets: Part 1, forward problems," *IEEE Transactions on Fuzzy Systems*, vol. 14, no. 6, pp. 781–792, Dec 2006.

[25] D. Defour and M. Marin, "FuzzyGPU: A Fuzzy Arithmetic Library for GPU," in *2014 22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, Feb 2014, pp. 624–631.

[26] B. Goossens and D. Parello, "Limits of instruction-level parallelism capture," *Procedia Computer Science*, vol. 18, pp. 1664–1673, 2013.

[27] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide*, June 2011.

[28] J. Hoberock and N. Bell, "Thrust: A parallel template library," 2010, version 1.7.0. [Online]. Available: http://thrust.github.io/