FuzzyGPU: a fuzzy arithmetic library for GPU

David Defour, Manuel Marin

FuzzyGPU: a fuzzy arithmetic library for GPU

David Defour and Manuel Marin
Univ. Perpignan Via Domitia, DALI F-66860, Perpignan, France
Univ. Montpellier II, LIRMM, UMR 5506, F-34095, Montpellier, France
CNRS, LIRMM, UMR 5506, F-34095, Montpellier, France

***Abstract*—Data are traditionally represented using native format such as integer or floating-point numbers in various flavor. However, some applications rely on more complex representation format. This is the case when uncertainty needs to be apprehended. Fuzzy arithmetic is one of the major tools to address this problem, but the execution time of basic operations such as addition or multiplication makes its usage prohibitive. In this article, thanks to a new representation format and modern GPU characteristics we show that it is possible to greatly reduce the execution time of those operations. These techniques have been implemented in fuzzyGPU, a freely distributed library of common operations over fuzzy number.**

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 is usually represented in the form of ambiguous sentences such as *at least x*, *probably between 0.4 and 0.7*, etc. This particular kind of uncertainty is best modelled using fuzzy numbers, as they allow to deal with such type of quantifiers [24]. The fuzzy approach has been successfully applied in finance [23], transportation [3], supply chain management [6], load flow [17], [21]... for configuring expert systems to run under uncertainty scenarios. A fuzzy arithmetic based on the α-cut approach can be implemented using interval arithmetic [10], however this can be cumbersome in terms of computation intensity, as each fuzzy operation requires at least two interval operations to be computed. 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 or data 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 [22]. 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 [8], as it suppresses the overhead associated with the changes of rounding mode.

Applications relying on fuzzy logic have already been successfully ported on GPU [1], [16]. 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 will 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 specificities of Nvidia's GPU are greatly reducing 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:

- **Fuzzy arithmetic library.** We describe the implementation of a library of fuzzy arithmetic operations, callable from either CPU or GPU code optimized for each type of architecture. It includes two classes of fuzzy numbers, one based on the lower-upper encoding, the other one based on the midpoint-radius encoding. This library is based on C++ overload of basic operation making it easily extendible, portable and efficient.
- **Benefice of dedicated architecture.** GPUs are known as powerful vector coprocessor. We detail in this article, the benefice brought by GPUs and explain how TLP, ILP, static rounding mode and memory usage impact performance at a fine granularity of basic operations on complex format.
- **Representation format.** We propose a new representation format for fuzzy numbers with symmetric membership function which both reduces the number of operations and memory bandwidth compared to the traditional representation format.

The rest 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 present the performance of our library of fuzzy arithmetic. Section VI concludes the paper and introduces some ideas for future work.

II. Fuzzy numbers and their implementations

Fuzzy numbers and fuzzy arithmetic provide an answer where computations have to be done with imprecise parameters such as applications in fuzzy control, decision making,

approximate reasoning, optimization statistics with imprecise probabilities [10], [24]. Fuzzy arithmetic is often approached through the α-cut concept [10], which consists of representing the fuzzy number as a collection of intervals, one for each level of uncertainty in the model, called the α-cuts. Fuzzy operations are then performed by executing interval operations over the α-cuts of the same level. In this section we will recall basic concept of interval arithmetic, as well as fuzzy numbers and arithmetic.

A. Interval arithmetic

Interval arithmetic is a tool that accounts for one single level of uncertainty in numerical computation. It represents each quantity as a range of possibilities, and operates on these ranges, generating new ranges of possibilities for the results. Interval arithmetic is also used to provide certified results for operations subject to rounding errors such as floating-point operations. There can be represented using lower-upper encoding, one of its bounds and the diameter of the interval, or by the center of the interval and its radius. Most of the existing implementation such as Boost [4], MPFI [19] and GAOL [12], are based on the lower-upper encoding. Our proposition is based on midpoint-radius representation format. Let describes this two alternatives.

1) Lower-upper encoding: It uses two floating-point numbers, l and u, holding the lower and upper bounds of the interval. The interval $[l, u]$ is the set of elements comprised between l and u, i.e.

$$[l, u] = \{x \in \mathbb{R} : l \leq x \leq u\}$$

The interval operations are defined as

$$\begin{aligned}
[a, b] + [c, d] &= [\bigtriangledown(a + b), \bigtriangleup(c + d)], \\
[a, b] - [c, d] &= [\bigtriangledown(a - d), \bigtriangleup(b - c)], \\
[a, b] \cdot [c, d] &= [\min(\bigtriangledown(a \cdot c), \bigtriangledown(b \cdot d), \bigtriangledown(a \cdot d), \bigtriangledown(b \cdot c)), \\
&\qquad \max(\bigtriangleup(a \cdot c), \bigtriangleup(b \cdot d), \bigtriangleup(a \cdot d), \bigtriangleup(b \cdot c))]
\end{aligned}$$

where $\bigtriangledown$ is the rounding attribute toward $-\infty$ and $\bigtriangleup$, the rounding attribute toward $+\infty$. The changes of rounding mode guarantee that the resulting interval includes all possible solutions. For many architecture, such as CPU, the rounding mode corresponds to a processor state, which implies flushing the entire pipeline before being able to start an operation with a new rounding mode.

2) Midpoint-radius encoding: Midpoint-radius format holds the midpoint m and the radius ρ [20]. The interval $\langle m, \rho \rangle$ is the set of elements whose distance to m is not higher than ρ:

$$\langle m, \rho \rangle = \{x \in \mathbb{R} : |x - m| \leq \rho\}$$

Using this encoding, the interval operations are defined as

$$\begin{aligned}
\langle a, \alpha \rangle + \langle b, \beta \rangle &= \langle \square(a + b),\ \bigtriangleup(\epsilon' |\square(a + b)| + \alpha + \beta) \rangle \\
\langle a, \alpha \rangle - \langle b, \beta \rangle &= \langle \square(a - b),\ \bigtriangleup(\epsilon' |\square(a - b)| + \alpha + \beta) \rangle \\
\langle a, \alpha \rangle \cdot \langle b, \beta \rangle &= \langle \square(a \cdot b),\ \bigtriangleup(\eta + \epsilon' |\square(a \cdot b)| + \\
&\qquad (|a| + \alpha)\beta + \alpha|b|) \rangle
\end{aligned}$$

where $\bigtriangledown$ is the rounding attribute toward $-\infty$, $\bigtriangleup$ the rounding attribute toward $+\infty$ and $\square$ the rounding attribute toward the nearest. ϵ is the relative rounding error unit – $\epsilon' = \frac{1}{2}\epsilon$ –, and η is the smallest representable (denormalized) floating point number. In double precision, $\epsilon = 2^{-52}$ and $\eta = 2^{-1074}$.

B. Fuzzy arithmetic

While interval arithmetic operates on intervals, sets of one single level of uncertainty, fuzzy arithmetic operates on fuzzy numbers, which are sets of several levels of uncertainty. For each level of uncertainty we can obtain a corresponding interval or α-cut.

Every fuzzy number $\tilde{p}$ has a membership function, $\mu_{\tilde{p}}$, which assigns to each value in the real line a degree of membership to the fuzzy number. It exists a unique value $m \in \mathbb{R}$ for which $\mu_{\tilde{p}}(m) = 1$, the highest degree of membership. The membership function is a monotony around this value, i.e. $\mu_{\tilde{p}}(x)$ is increasing for $x < m$ and decreasing for $x \geq m$.

A fuzzy number is defined as

$$\tilde{p} = \{(x, \mu_{\tilde{p}}(x)) \mid x \in \mathbb{R}\}$$

An α-cut of $\tilde{p}$ is obtained by selecting all the elements with a degree of membership higher than a certain α_i, i.e.

$$p_{\alpha_i} = \{x \in \mathbb{R} \colon \mu_{\tilde{p}}(x) \geq \alpha_i\}$$

Fuzzy arithmetic operations are carried out by performing the operation on the α-cuts of the same level. That is, if $\tilde{p}$ and $\tilde{q}$ are fuzzy numbers with k levels of uncertainty; p_{α_i} and q_{α_i}, for $i \in \{0, \ldots, k-1\}$ are their k α-cuts; and $\oplus \in \{+, -, \times\}$, then

$$\tilde{r} = \tilde{p} \oplus \tilde{q}$$

is computed through k interval operations,

$$r_{\alpha_i} = p_{\alpha_i} \oplus q_{\alpha_i}$$

for $i \in \{0, \ldots, k-1\}$.

III. Proposed representation format of fuzzy number

An important characteristic of fuzzy numbers is the shape of the membership function. Depending on the application, fuzzy numbers may have a symmetric membership function and then all the α-cuts will be centred around the same midpoint [5], [7]. When fuzzy numbers exhibit symmetrical shape, we will prove in this section some properties which we will exploit for the proposed implementation. Let's first give the definition of a symmetric fuzzy number:

Definition (Symmetric fuzzy number)**.** *Let $\tilde{p}$ be a fuzzy number, $\mu_{\tilde{p}}$ its membership function, $m \in \mathbb{R}$ such that $\mu_{\tilde{p}}(m) = 1$. If $\mu_{\tilde{p}}$ is symmetric around m, i.e.*

$$\mu_{\tilde{p}}(m - x) = \mu_{\tilde{p}}(m + x), \forall x \in \mathbb{R}$$

then we call $\tilde{p}$ a symmetric fuzzy number.

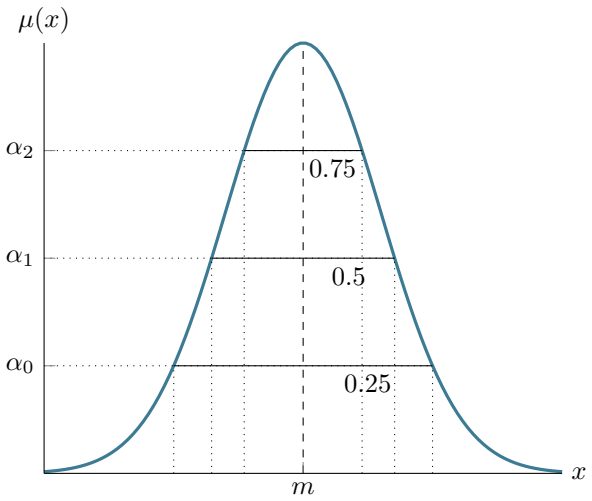


Fig. 1: Symmetric fuzzy number and α-cuts, all centered around the same midpoint.

If symmetric fuzzy numbers do not exhibit any interesting property for a representation based on lower-upper interval, this is not the case for midpoint-radius encoding. The proposed Theorem 1 is stating that all the α-cuts defining a symmetrical fuzzy numbers will share the same center.

Theorem 1. *Let $\tilde{p}$ be a fuzzy number, $\mu_{\tilde{p}}$ its membership function, $m \in \mathbb{R}$ such that $\mu_{\tilde{p}}(m) = 1$. Let $p_{\alpha_i} = \langle m_i, \rho_i \rangle$ be any arbitrary α-cut, expressed in the midpoint-radius encoding. Then*

$$\tilde{p} \text{ is symmetric } \iff m_i = m, \forall p_{\alpha_i}.$$

Proof. We will prove the reciprocal, i.e.

$$\tilde{p} \text{ is non-symmetric } \iff \exists p_{\alpha_i}, \text{with } m_i \neq m.$$

i)

$$\tilde{p} \text{ is non-symmetric } \implies \exists p_{\alpha_i}, \text{with } m_i \neq m.$$

If $\tilde{p}$ is non-symmetric, it means that exists $x_0 \in \mathbb{R}$ such that

$$\mu_{\tilde{p}}(m - x_0) \neq \mu_{\tilde{p}}(m + x_0) \tag{1}$$

Let $p_{\alpha_i} = [l_i, u_i] = \langle m_i, \rho_i \rangle$ be an α-cut, such that $l_i = m - x_0$.
By definition,

$$\mu_{\tilde{p}}(l_i) = \mu_{\tilde{p}}(u_i) = \alpha_i \tag{2}$$

Now, assume $m = m_i$. Since $l_i = m - x_0$, then $x_0 = \rho_i$ and $u_i = m + x_0$.

Replacing values in equation 2, we obtain

$$\mu_{\tilde{p}}(m - x_0) = \mu_{\tilde{p}}(m + x_0)$$

which contradicts equation 1. Hence, $m \neq m_i$.

ii)

$$\exists p_{\alpha_i}, \text{with } m_i \neq m \implies \tilde{p} \text{ is non-symmetric.}$$

Let $p_{\alpha_i} = [l_i, u_i] = \langle m_i, \rho_i \rangle$ be an α-cut with $m_i \neq m$. Again, by definition,

$$\mu_{\tilde{p}}(l_i) = \mu_{\tilde{p}}(u_i) = \alpha_i \tag{3}$$

Say $x_0 = m - l_i$. Since $m \neq m_i$, $m + x_0 \neq u_i$. Also, $\mu_{\tilde{p}}$ is monotonic, so $\mu_{\tilde{p}}(m + x_0) \neq \mu_{\tilde{p}}(u_i)$.
Replacing values in equation 3, we obtain

$$\mu_{\tilde{p}}(m - x_0) \neq \mu_{\tilde{p}}(m + x_0)$$

Hence, $\tilde{p}$ is non-symmetric.

□

Figure 1 shows an example of symmetric fuzzy number. We see that all the α-cuts have the same common midpoint. Symmetry is an important property for fuzzy numbers as it is preserved by common arithmetic operations such as $\{+, -, \times\}$ as stated by the proposed theorem 2.

Theorem 2. *Let $\tilde{p}$ and $\tilde{q}$ be two fuzzy numbers and $\oplus \in \{+, -, \times\}$. If $\tilde{p}$ and $\tilde{q}$ are both symmetric, then*

$$\tilde{r} = \tilde{p} \oplus \tilde{q}$$

is also symmetric.

Proof. Let's say, without loss of generality, that the system has k levels of uncertainty. We can express the k α-cuts of $\tilde{p}$, $\tilde{q}$ and $\tilde{r}$ in the midpoint-radius encoding as

$$\begin{aligned} p_{\alpha_i} &= \langle p_i, \beta_i \rangle \\ q_{\alpha_i} &= \langle q_i, \gamma_i \rangle \\ r_{\alpha_i} &= \langle r_i, \delta_i \rangle \end{aligned}$$

for $i \in \{0, \ldots, k-1\}$.

We have to prove the theorem for the 3 basic operations. We will split the demonstration in two cases, the addition and subtraction first and the multiplication second.

i) $\oplus \in \{+, -\}$,

$$\begin{aligned} r_i &= \square(p_i \oplus q_i) \\ \delta_i &= \triangle(\epsilon' |r_i| + \beta_i + \gamma_i) \end{aligned}$$

From Theorem 1, since $\tilde{p}$ and $\tilde{q}$ are symmetric, we have

$$\begin{aligned} p_i &= p \\ q_i &= q \end{aligned}$$

Then

$$r_i = \square(p_i \oplus q_i) = \square(p \oplus q) = r$$

for all $i \in \{0, \ldots, k-1\}$. Hence, $\tilde{r}$ is symmetric.

ii) $\oplus = \times$,

$$r_i = \square(p_i \times q_i)$$
$$\delta_i = \triangle(\eta + \epsilon'|r_i| + (|p_i| + \beta_i)\gamma_i + \beta_i|q_i|)$$

Again, from Theorem 1, since $\tilde{p}$ and $\tilde{q}$ are symmetric, we have

$$p_i = p$$
$$q_i = q$$

Then

$$r_i = \square(p_i \times q_i) = \square(p \times q) = r$$

for all $i \in \{0, \ldots, k-1\}$. Hence, $\tilde{r}$ is symmetric.

□

Implementations of fuzzy arithmetic are available in various programming languages [2], [13], [15]. As for interval arithmetic, they are based on the lower-upper encoding for fuzzy numbers. The implementation we propose in this paper provides dedicated class to manipulate fuzzy numbers in both the lower-upper and the midpoint-radius encoding. The latter has been implemented making use of the result presented in Theorem 2. That is, for practical purposes, a symmetric fuzzy number can be represented as a unique midpoint, which is common to all the α-cuts, plus a set of radius. This allows, when performing fuzzy arithmetical operations, to save bandwidth and execution time as measured in section V.

IV. Implementation details

Our fuzzy arithmetic library is written in CUDA C and C++, and callable from either CPU and GPU code, transparently adapting itself to the underlying architecture. The source code is available at [9].

The implementation consists of successive wrappers. At the user-end, we have the two fuzzy classes, one for each fuzzy encoding. We decided to offer the two encoding, lower-upper representation format for general case and midpoint-radius when the user is dealing with symmetric fuzzy number. 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 stored as a unique midpoint, valid for all the α-cuts, plus a set of radius, making use of the results from the previous section. Ultimately, the rounded arithmetic class is a wrapper of C and CUDA compiler intrinsics with dedicated 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 an array of intervals which hold the lower and upper bounds for each α-cut. In the midpoint-radius encoding, `N` and `T` are the size and type of an array of scalar radius, 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 operations are carried out by a sequential **for** loop that iterates over the set of α-cuts. We decided not to spread the loop among different threads in GPU code, since most applications in real life do not involve more than three to four degrees of uncertainty in their data model [3], [6], [17], [21], [23]. This number is way too low to exploit thread level parallelism (TLP). However, TLP may be exploited in vector operations involving fuzzy numbers, through kernels that process each element of a fuzzy array in a different thread.

Performance of basic operations over complex datatype such as fuzzy numbers are impacted by numerous factors. Next, we will discuss and compare the two representation formats regarding the number of instructions, the memory requirement and the ILP.

A. Number of instructions

Table I shows the number of instructions, such as basic operation, minimum, maximum or absolute value, involved in the addition and multiplication for different data types, including fuzzy numbers. In the case of fuzzy data types, the number of α-cuts is represented by x. The lower-upper fuzzy requires only 2 operations less than midpoint-radius on the addition, but requires 2.8 times more operations when it comes to the multiplication. Therefore, by just comparing the number of instructions the midpoint-radius should bring a real speed-up over the lower-upper representation format.

TABLE I: Number of instructions per arithmetical operation, for different data types, where x corresponds to the number of α-cuts.

Data type	Number of instructions	
	Addition	Multiplication
Scalar	1	1
Lower-upper interval	2	14
Midpoint-radius interval	4	10
Lower-upper fuzzy	$2x$	$14x$
Midpoint-radius fuzzy	$2 + 2x$	$5 + 5x$

B. Memory usage

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

C. Instruction level parallelism (ILP)

Ideal ILP as defined in [11], 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,

TABLE II: Memory requirements for different data types, where x corresponds to the number of α-cuts.

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

the higher the amount of instructions that could be pipelined during the execution of the application. However, a bigger ideal ILP requires a larger amount of hardware resources to exploit it.

Table III shows the ideal ILP of the addition and multiplication for different data types. This ILP is determined as the ratio of the total number of instructions to the number of levels in the dependency tree. The number of α-cuts in fuzzy data types does not affect the denominator, as each α-cut is independent from all others. Lower-upper fuzzy exhibits more ideal ILP than midpoint-radius on both the addition and the multiplication. However, ILP will be exploited differently depending on the GPU generation as well as the data type in use (single or double precision). For example, GPU 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 III: ILP per arithmetical operation, for different data types, where x correspond to the number of α-cuts..

Data type	ILP	
	Addition	Multiplication
Scalar	1	1
Lower-upper interval	2	14/3
Midpoint-radius interval	5/4	2
Lower-upper fuzzy	2x	4.6x
Midpoint-radius fuzzy	$0.5 + x/2$	$1 + x$

V. Tests and results

We have described our fuzzy arithmetic library and exposed the main differences between the two fuzzy encoding methods. Now we will present the tests we designed for the purpose of evaluating the performance of our fuzzy library on GPU under different circumstances. GPU performance is often driven by the balance between memory access and computational effort. In this sense, we consider two types of applications: compute-bound application, where arithmetic instructions are dominant over memory accesses; and memory-bound application.

The following architectures were used in this study:

- Intel Xeon E5645 CPU.
- NVIDIA GeForce GTX 480 GPU (compute capability 2.0).
- NVIDIA GeForce GTX 680 GPU (compute capability 3.0).

CUDA version is 4.2 and g++ version is 4.6.3 in all cases.

A. Compute-bound application: AXPY series

Listing 1 shows an AXPY kernel that computes the i-th element of the series $c \leftarrow ac+b$, where all the values are fuzzy numbers, through an iterative loop. All the threads read the same starting value, perform the recursive computation, and then write the result to a different position on an output vector. As we increment the number of iterations `n`, the number of floating point instructions grows; however, the number of global memory accesses, one for loading the starting value and one for storing the result, remains constant. In this way we can arbitrarily increment the number of floating-point instructions per accesses to global memory .

```
#include "fuzzy_lib.h"

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

int main(){
    fuzzy_lu<double, 4> * d_a, * d_b, * d_result;
    cudaMalloc((void**)&d_result, ...);
    ...
    compute_test<double, 4><<<GRID_SIZE, BLOCK_SIZE>>>(
         n_iters, d_a, d_b, d_result);
}
```

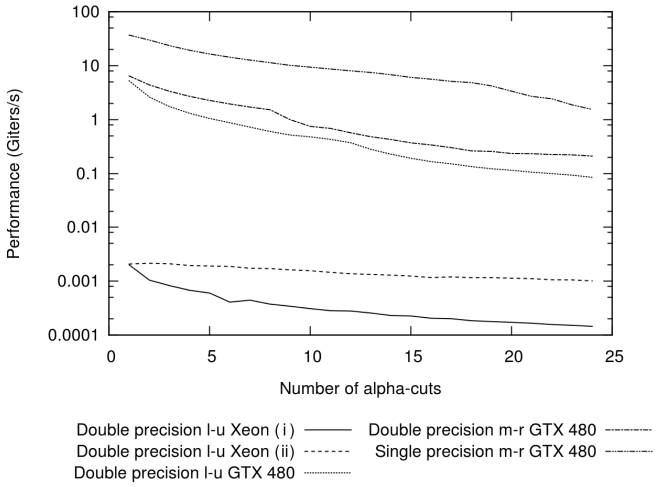
Listing 1: AXPY loop kernel (compute-bound test).

We measured the execution time of our kernel (as well as other performance indicators) under different parameter configurations, including:

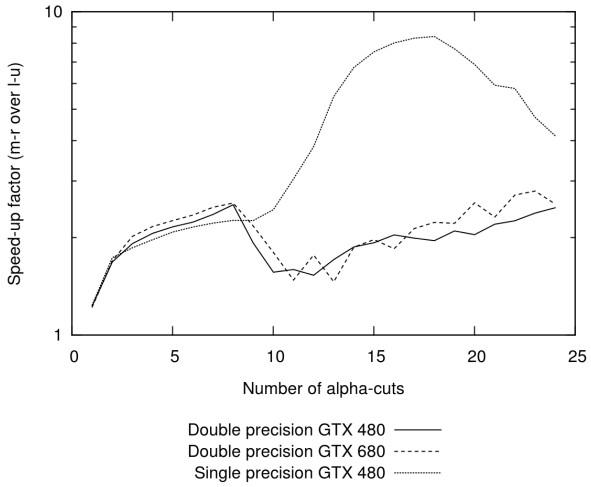
- lower-upper and midpoint-radius fuzzy encoding;
- a number of α-cuts from 1 to 24;
- single and double precision.

The kernel arguments were tuned for different runs in order to achieve maximum occupancy of the architecture. The results are plotted in figure 2. Specifically, figure 2a 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 [15][1]. We observe differences and performance gains coming from heterogeneous sources. First, there is a pure architectural gain 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. Finally, there is a gain

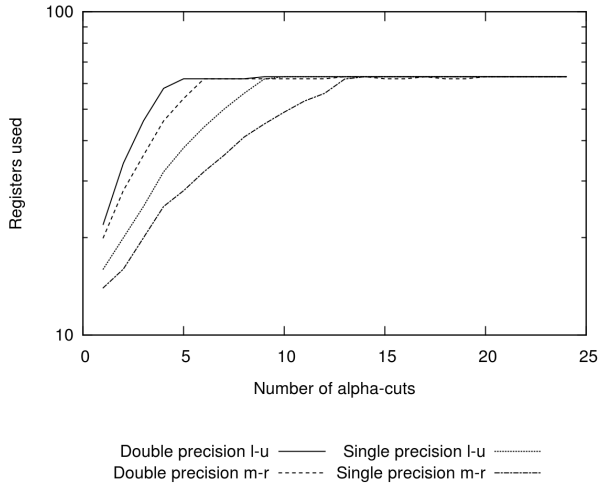
[1]Note that the library in [15] does not provide certified interval arithmetic, with correct rounding attributes, as our library does.



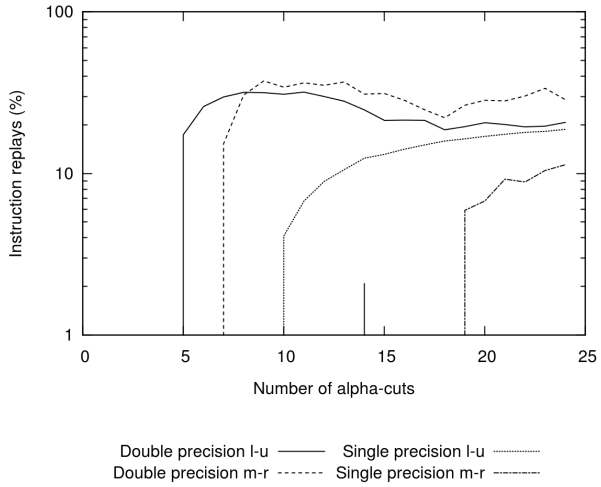
(a) Related performance of the AXPY kernel.



(b) Algorithmic gain.



(c) Registers used per implementation.



(d) Instruction replays due to local memory accesses.

Fig. 2: Compute-bound benchmark.

on passing from double to single precision in the 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 2b 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 8-9 α-cuts. In this range, the curve follows a growing pattern which is consistent with the theoretical analysis of number of operations per different type of fuzzy operations we made before. This analysis is summed up in table I. From this table, we can calculate the ratio of number of operations per cycle of the AXPY loop required by the midpoint-radius encoding, to the number of operations required by the lower-upper encoding, which is

$$\frac{16x}{7+7x}$$

This function perfectly describes the shape of the speed-up curve between 1 and 8-9 α-cuts, according to figure 2b. Moreover, this behaviour does not depend on the architecture neither on the precision. This result validates our theoretical analysis and also serves to illustrate the performance of the library when no limits on the architecture are being disturbed.

After 9-10 α-cuts, we have a complete different situation. Here, the effect of register spilling and local memory accesses with high latency becomes preponderant. Remember CUDA devices of compute capability 2.0 and 3.0 may allocate up to

63 registers of 32-bits per thread on a kernel execution. Double precision values are stored in two consecutive registers. When this limit is attained, the kernel starts using local memory to allocate extra data. Figure 2c shows the number of registers used by the AXPY kernel per implementation. Figure 2d shows the percentage of replayed instructions due to local memory accesses.

We observe that 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 immediately after, at 10 α-cuts. At this same point, in figure 2b, we observe an increase of the speed-up in single precision, 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. From this point on, 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 a little more complex. Both lower-upper and midpoint-radius fuzzy start spilling registers relatively early, between 6 and 8 α-cuts. In figure 2b, we observe that there is a slight drop in the speed-up in double precision, at about 9 α-cuts. The midpoint-radius fuzzy implementation is loosing performance in this zone and 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 it to calculate all the radius. If one midpoint is spilled to off-chip local memory, there might not be enough independent arithmetic instructions to hide the latency of this memory access. This effect cannot be appreciated in single precision as register spilling does not get too serious even for the higher number of α-cuts considered. This demonstrates that regarding memory, performance is mainly driven by the amount of memory required to store data and secondly by the temporal dependency among them.

As we stated in section IV, real life applications typically involve 3 to 4 α-cuts. In this range, register spilling and local memory accesses with high latency is not an issue. The above results were obtained using the default 16 KB L1 cache configuration. Nevertheless, if pressure over the register file is too high, the user of this library has the possibility to tune the size of L1 cache and shared memory [18]. On one hand, increasing the size of the L1 cache allows to use more α-cuts before seeing the impact of register spilling; on the other hand, increasing shared memory does not affect the number of α-cuts for which the spilling occurs, but allows to reduce its overall impact on performance.

B. Memory-bound application: sort by keys

Listing 2 shows a thrust [14] program that sorts a vector of fuzzy numbers on the device. We observe in this example 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 will 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 will be read and copied into a new position. This is a typical case of memory-bound application.

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

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

Listing 2: Thrust sort by keys (memory-bound test).

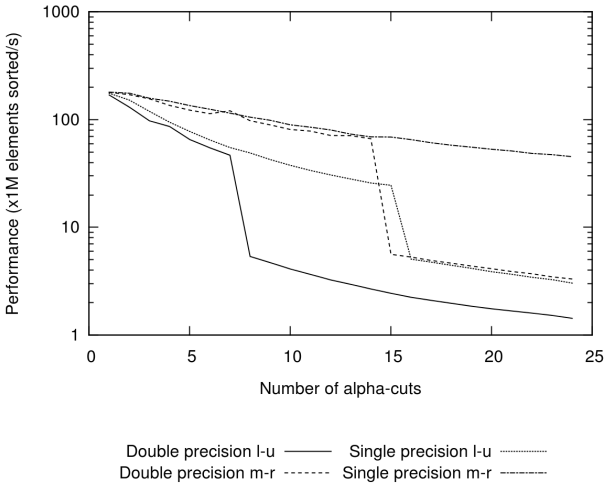
We measured the GPU sorting time for the lower-upper and midpoint-radius fuzzy encoding, with a number of α-cuts ranging between 1 and 24 and for single and double precision.

The time spent in transferring data between host and device is not included in our measure. Results for the GTX 480 are plotted in figure 3. Specifically, figure 3a shows performance for different fuzzy encoding and precision, in terms of millions of sorted elements per second. Figure 3b presents the same information in terms of speed-up of midpoint-radius over lower-upper.

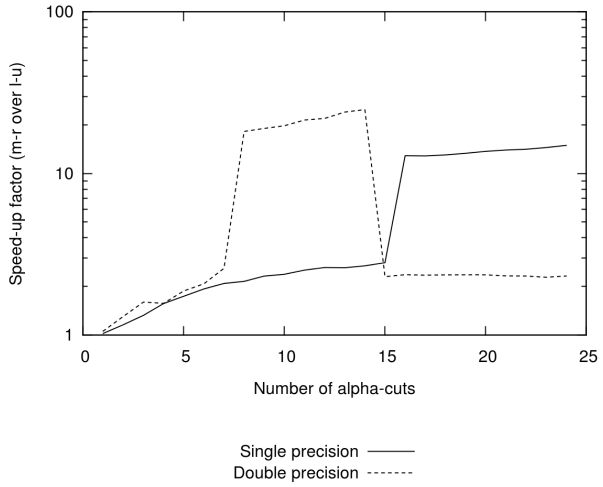
We observe in figure 3 that for the same number of α-cuts, midpoint-radius encoding allows to sort twice the amount of fuzzy numbers than lower-upper encoding, whether we are in single or double precision. For this application, the size of the data array being sorted becomes the main factor driving performance. 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 size is given by table II:

$$\frac{2x}{1+x}$$

where x is the number of α-cuts. We observe that the speed-up curves in figure 3b follow approximately this ratio 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 sort uses shared memory to speed-up the sorting process. When the values being treated by threads within one SM do not longer fit in shared memory, it starts using global memory with a direct impact over performance. As we stated earlier, 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.



(a) Thrust's sort by keys performance.



(b) Algorithmic gain.

Fig. 3: Memory-bound benchmark.

VI. Conclusion

In this article, we presented a fuzzy arithmetic library written in CUDA and C++, based on the α-cuts concept and relying on interval and rounded floating point arithmetic optimized for different hardware. Our implementation offers two fuzzy containers. The first one uses a traditional lower-upper encoding for intervals and allows to represent any fuzzy number, regardless of the shape of the membership function. The second one offers a new optimized midpoint-radius encoding for fuzzy numbers having a symmetric membership function, which notably improves computational efficiency. The optimization consists of using a single variable to store all the α-cuts' common midpoint. We also detailed the performance impact of TLP, ILP and memory usage for recent NVIDIA's GPU regarding operations over fuzzy number's for the proposed representation format. We have shown through a theoretical study and real measures for memory and compute bound kernels that a gain of 2 to 20 can be obtained by preferring the midpoint-radius encoding over the traditional lower-upper encoding. However, accuracy of the new representation format has not been evaluated and is part of the future works.

References

[1] D. Anderson, R. Luke, and J. Keller. Speedup of fuzzy clustering through stream processing on graphics processing units. *Fuzzy Systems, IEEE Transactions on*, 16(4):1101–1106, 2008.

[2] A. Anile, S. Deodato, and G. Privitera. Implementing fuzzy arithmetic. *Fuzzy Sets and Systems*, 72(2):239 – 250, 1995.

[3] S. Bonvicini, P. Leonelli, and G. Spadoni. Risk analysis of hazardous materials transportation: evaluating uncertainty by means of fuzzy logic. *Journal of Hazardous Materials*, 62(1):59 – 74, 1998.

[4] H. Brnnimann, G. Melquiond, and S. Pion. The design of the boost interval arithmetic library. *Theoretical Computer Science*, 351(1):111 – 118, 2006.

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

[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*, 102(2):289 – 301, 2006.

[7] M.-Y. Chen and D. Linkens. Rule-base self-generation and simplification for data-driven fuzzy models. In *Fuzzy Systems, 2001. The 10th IEEE International Conference on*, volume 1, pages 424–427, 2001.

[8] S. Collange, M. Daumas, and D. Defour. Chapter 9 - interval arithmetic in cuda. In W. mei W. Hwu, editor, {*GPU*} *Computing Gems Jade Edition*, pages 99 – 107. Morgan Kaufmann, Boston, 2012.

[9] D. Defour and M. Marin. Fuzzygpu: a fuzzy arithmetic library for gpu. http://code.google.com/p/fuzzy-gpu/, 2013.

[10] D. Dubois and H. Prade. Operations on fuzzy numbers. *International Journal of Systems Science*, 9(6):613–626, 1978.

[11] B. Goossens and D. Parello. Limits of instruction-level parallelism capture. In *ICCS*, pages 1664–1673, 2013.

[12] F. Goualard. Gaol 3.1. 1: Not just another interval arithmetic library. *Laboratoire d'Informatique de Nantes-Atlantique*, 4, 2006.

[13] M. Hanss. The transformation method for the simulation and analysis of systems with uncertain parameters. *Fuzzy Sets and Systems*, 130(3):277 – 289, 2002.

[14] J. Hoberock and N. Bell. Thrust: A parallel template library, 2010. Version 1.7.0.

[15] N. Kolarović. Fuzzy numbers and basic fuzzy arithmetics (+, -, *, /, 1/x) implementation written in java. http://fuzzyarith.sourceforge.net, 2013.

[16] M. Martnez-Zarzuela, F. Daz Pernas, J. Dez Higuera, and M. Rodrguez. Fuzzy art neural network parallel computing on the gpu. In F. Sandoval, A. Prieto, J. Cabestany, and M. Graa, editors, *Computational and Ambient Intelligence*, volume 4507 of *Lecture Notes in Computer Science*, pages 463–470. Springer Berlin Heidelberg, 2007.

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

[18] NVIDIA. *NVIDIA CUDA C Programming Guide 5.0*. 2013.

[19] N. Revol and F. Rouillier. The mpfi library, 2001.

[20] S. M. Rump. Fast and parallel interval arithmetic. *BIT*, 39(3):534–554, 1999.

[21] A. Saber and G. Venayagamoorthy. Resource scheduling under uncertainty in a smart grid with renewables and plug-in vehicles. *Systems Journal, IEEE*, 6(1):103–109, 2012.

[22] V. Volkov. Better performance at lower occupancy. In *Proceedings of the GPU Technology Conference, GTC*, volume 10, 2010.

[23] Y. Yoshida, M. Yasuda, J. ichi 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*, 157(19):2614 – 2626, 2006.

[24] L. Zadeh. The role of fuzzy logic in the management of uncertainty in expert systems. *Fuzzy Sets and Systems*, 11(1–3):197 – 198, 1983.