INTERVAL ARITHMETIC USING SSE-2 (DRAFT)

BRANIMIR LAMBOV

ABSTRACT. We present an implementation of double precision interval arithmetic using the single-instruction-multiple-data SSE-2 instruction and register set extensions. The implementation is part of a package for exact real arithmetic, which defines the interval arithmetic variation that must be used: incorrect operations such as division by zero cause exceptions, loose evaluation of the operations is in effect, and performance is more important than tightness of the produced bounds. The SSE2 extensions are suitable for the job, because they can be used to operate on a pair of double precision numbers and include separate rounding mode control and detection of the exceptional conditions. The paper describes the ideas we use to fit interval arithmetic to this set of instructions, shows a performance comparison with other freely available interval arithmetic packages, and discusses possible very simple hardware extensions that can significantly increase the performance of interval arithmetic.

1. INTRODUCTION

Interval arithmetic is a very useful tool that can be used to partially solve the problem of roundoff errors or as part of a complete solution in the form of exact real arithmetic.

RealLib relies on fast machine precision interval arithmetic for its first stage. The performance of the library in the cases that most frequently appear in practice, where machine precision suffices, depends only on the performance of the first stage. Thus it is crucial to have a very fast implementation of interval arithmetic.

The IEEE-754 standard for floating point arithmetic [9] has useful features to aid fast interval arithmetic, namely the directed rounding modes that should be present with every IEEE-754 implementation. Unfortunately, in some processor architectures, notably Intel's x86, it is non-trivial to effectively use them, as switching the rounding mode for an operation requires significantly more time than the operation itself. Even when one takes into account the fact that one of the directed rounding modes can be emulated by operations on negated values rounded in the other direction, an interval arithmetic package has to be aware that users may mix interval with standard floating point arithmetic and would still require repeatedly switching the rounding modes.

Fortunately, the newer generations of the x86 architecture provide an additional set of registers with its own rounding control, the SSE2 double-precision floating point registers [10]. They can coexist with the old x87-style floating point, which is still the register and instruction set used most widely. Thus, to serve all purposes, we can reserve the SSE2 register and operation set for interval arithmetic and leave x87-style floating point for any standard floating point operations that the user may be performing.

The SSE2 instruction set can also work on packed data, as every SSE2 register can contain and operate on a pair of double-precision floating point numbers. Since an interval is in fact a pair of bounds, one SSE2 register can be used to hold an interval, which nullifies the additional register pressure that interval arithmetic would normally exert.

Key words and phrases. interval arithmetic.

Dagstuhl Seminar Proceedings 06021
Reliable Implementation of Real Number Algorithms: Theory and Practice
http://drops.dagstuhl.de/opus/volltexte/2006/714

With this it is possible to develop a very fast machine precision interval arithmetic implementation. *RealLib* uses such an implementation which will be detailed in this chapter of the thesis.

As it is part of an exact real arithmetic package, the objective of this implementation is more oriented towards performance rather than accuracy, i.e. it prefers overestimating an interval rather than investing too much time in evaluating it tightly. We believe that this time would be better spent at the next iteration at higher precision, which would happen only if the computation actually requires it.

Additionally, the implementation ignores the portions of the argument of an operation that are outside its domain, e.g. the negative parts of the argument in a square root, meaning for example that $\sqrt{[-1,4]} = [0,2]$. This is the proper mode of operation to ensure that $\sqrt{0}$ is computable in exact real arithmetic.

2. KEY IDEAS

Normally, interval arithmetic based on floating point would use two rounding operations, Δ (rounding towards $+\infty$) and ∇ (rounding towards $-\infty$). By default IEEE-floating point uses rounding to nearest, which is not useful for our purposes.

We already mentioned that switching the rounding mode has a detrimental effect on the performance of floating point operations, thus we would want to avoid all rounding mode switches. We will only do this once, at the beginning of a computation[1], setting the rounding mode to rounding towards $-\infty$. To compute lower bounds of the results, we will directly use the floating point operation. To compute upper bounds, we will make sure that the result of the floating point operation is negated, thus making use of the identity

$$\Delta(x) = -\nabla(-x).$$

Seeing operations in the form above, compilers are usually overzealous[2] to fold the pair of negations and destroy the effect we want to achieve. To avoid this, at the same time keeping down the number of required operations, we make sure that we always keep the high bound of the interval negated, i.e. our representation of the interval $x = [\underline{x}, \overline{x}]$ is the pair $\langle \underline{x}, -\overline{x} \rangle$. (in the rest of this chapter we will assume every interval is represented in this fashion and will simply write x to mean $[\underline{x}, \overline{x}]$ and $\langle \underline{x}, -\overline{x} \rangle$)

Three observations can be made directly from this:

- in this setting, the sum of x and y is evaluated by $\left\langle \nabla(\underline{x} + \underline{y}), -\nabla(-\overline{x} - \overline{y}) \right\rangle$ which is achieved by a single instruction, *_mm_add_pd*.
- changing the sign of an interval x is achieved by simply swapping the two bounds, i.e. $\langle -\overline{x}, \underline{x} \rangle$, achieved by a single instruction, *_mm_shuffle_pd*,
- joining two intervals (i.e. finding an interval containing all numbers in both, or finding the minimum of the lower bounds and the maximum of the higher bounds) is performed as $\left\langle \min(\underline{x}, \underline{y}), -\min((-\overline{x}), (-\overline{y})) \right\rangle$ in a single instruction, *_mm_min_pd*.

The latter is used extensively in the computation of multiplication, division and other operations.

---

[1] This is accomplished by the construction of a special object that also takes care of restoring the previous rounding mode after the interval computation has completed.

[2] The two negations have no effect on the rounding-to-nearest mode which is normally in place in *C/C++* code, and on which many standard functions rely, thus this optimization is perfectly legal. Only our specific (non-standard) use of floating point makes it unwanted.

3. OPERATIONS

In this section we will give short remarks on our implementation of the basic operations on intervals. The operations include the arithmetic operators, including the special cases $-x$, $\frac{1}{x}$, and x^2, absolute value and square root.

All the operations give tight bounds (i.e. the best possible enclosures after rounding).

3.1. **Addition.** Definition:

$$x + y = [\underline{x} + \underline{y}, \overline{x} + \overline{y}] \subseteq \left\langle \nabla(\underline{x} + \underline{y}), -\nabla((-\overline{x}) + (-\overline{y})) \right\rangle$$

Addition is implemented as a single *_mm_add_pd* instruction. The negated sign of the higher bound ensures the proper direction of the rounding.

3.2. **Sign change.** Definition:

$$-x = [-\overline{x}, -\underline{x}] = \langle -\overline{x}, \underline{x} \rangle$$

This is a single swap of the two values, implemented as a *_mm_shuffle_pd* instruction. No rounding is performed here.

3.3. **Substraction.** Definition:

$$x - y = [\underline{x} - \overline{y}, \overline{x} - \underline{y}] \subseteq \left\langle \nabla(\underline{x} + (-\overline{y})), -\nabla((-\overline{x}) + \underline{y}) \right\rangle$$

Substraction is implemented as $x + (-y)$, which corresponds to two processor instructions. This is the best that can be achieved with packed SSE2 instructions, because the formula requires a combination of the high bound of one of the arguments with the low bound of the other.

3.4. **Multiplication.** Definition:

$$xy = [\min(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}), \max(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y})] \tag{3.1}$$

Unfortunately, the rounding steps are inseparable parts of the operations, this the equation above requires 8 multiplications. Using the fact that $\Delta(\nabla(r) + \epsilon) \geq \Delta(r)$ (for ϵ being the smallest representable number), one can do with 4 multiplications at the expense of some accuracy.

In our implementation we chose a different approach where we use four multiplications without sacrificing accuracy, by selecting the multiples based on the signs of $\underline{x}$ and $\overline{x}$. More specifically, we use these observations:

$$xy = \begin{cases} [\min(\underline{x}\underline{y}, \overline{x}\underline{y}), \max(\overline{x}\overline{y}, \underline{x}\overline{y})], & \text{if} \quad 0 \leq \underline{x} \leq \overline{x} \\ [\min(\underline{x}\overline{y}, \overline{x}\underline{y}), \max(\overline{x}\overline{y}, \underline{x}\underline{y})], & \text{if} \quad \underline{x} < 0 \leq \overline{x} \\ [\min(\underline{x}\overline{y}, \overline{x}\overline{y}), \max(\overline{x}\underline{y}, \underline{x}\underline{y})], & \text{if} \quad \underline{x} \leq \overline{x} < 0 \end{cases} \tag{3.2}$$

to conclude that the formula

$$xy \subseteq \langle \min(\nabla(a\underline{x}), \nabla(b(-\overline{x}))), -\min(\nabla(c(-\overline{x})), \nabla(d\underline{x})) \rangle \,,$$

where

$$\begin{aligned} a &= \begin{cases} \underline{y} & \text{if } 0 \le \underline{x} \\ -(-\overline{y}) & \text{otherwise} \end{cases} \\ b &= \begin{cases} -\underline{y} & \text{if } (-\overline{x}) \le 0 \\ (-\overline{y}) & \text{otherwise} \end{cases} \\ c &= \begin{cases} -(-\overline{y}) & \text{if } (-\overline{x}) \le 0 \\ \underline{y} & \text{otherwise} \end{cases} \\ d &= \begin{cases} (-\overline{y}) & \text{if } 0 \le \underline{x} \\ -\underline{y} & \text{otherwise} \end{cases} \end{aligned}$$

computes the rounded results of the multiplication formula in (3.1). It uses more instructions than the direct implementation with 8 multiplications, but achieves better performance.

3.5. **Multiplication by a positive number.** When one of the numbers is known to be positive (e.g. a known constant), one can use one of the cases in (3.2) directly:

$$xy \overset{x \ge 0}{=\!=} [\min(\underline{x}\underline{y}, \overline{x}\underline{y}), \max(\overline{x}\overline{y}, \underline{x}\overline{y})]$$

This is significantly faster than the general case multiplication, involving only 5 instructions (4 for constants).

3.6. **Multiplication of two positive numbers.** If both multiples are known to be positive, multiplication can be achieved by simply changing the sign of the higher bound of one of the arguments followed by *_mm_mul_pd*. If one of the numbers is a constant, one can prepare it in a suitable way to avoid the sign change and implement the multiplication as a single instruction.

3.7. **Division.** Definition:

$$\frac{x}{y} = \left[\min\left(\frac{\underline{x}}{\underline{y}}, \frac{\underline{x}}{\overline{y}}, \frac{\overline{x}}{\underline{y}}, \frac{\overline{x}}{\overline{y}}\right), \max\left(\frac{\underline{x}}{\underline{y}}, \frac{\underline{x}}{\overline{y}}, \frac{\overline{x}}{\underline{y}}, \frac{\overline{x}}{\overline{y}}\right)\right],$$

undefined if $0 \in y$.

Again, this computation would require 8 divisions. Unfortunately, division is a rather slow operation, that is why we would prefer to use as few divisions as possible. One way to do this is to use $\frac{x}{y} = x\frac{1}{y}$, using the definition below, which uses only two divisions but quite a few other operations.

A more efficient (as well as more accurate) approach turns out to be the use of case distinction similar to (3.2). By examining the divisor, we end up with fewer possible cases and easy recognition of the exceptional cases. More specifically, the operation becomes:

$$\frac{x}{y} = \begin{cases} \left[\min\left(\frac{\underline{x}}{\underline{y}}, \frac{\underline{x}}{\overline{y}}\right), \max\left(\frac{\overline{x}}{\underline{y}}, \frac{\overline{x}}{\overline{y}}\right)\right], & \text{if} \quad 0 < \underline{y} \le \overline{y} \\ \text{exception}, & \text{if} \quad \underline{y} \le 0 \le \overline{y} \\ \left[\min\left(\frac{\overline{x}}{\underline{y}}, \frac{\overline{x}}{\overline{y}}\right), \max\left(\frac{\underline{x}}{\underline{y}}, \frac{\underline{x}}{\overline{y}}\right)\right], & \text{if} \quad \underline{y} \le \overline{y} < 0 \end{cases} \tag{3.3}$$

The final formula we use is

$$\frac{x}{y} \subseteq \left\langle \min\left(\nabla\left(\frac{a}{\underline{y}}\right), \nabla\left(\frac{-a}{(-\overline{y})}\right)\right), -\min\left(\nabla\left(\frac{-b}{(-\overline{y})}\right), \nabla\left(\frac{b}{\underline{y}}\right)\right)\right\rangle,$$

where

$$a = \begin{cases} \underline{x} & \text{if } (-\overline{y}) \leq 0 \\ -(-\overline{x}) & \text{otherwise} \end{cases}$$
$$b = \begin{cases} (-\overline{x}) & \text{if } 0 \leq \underline{y} \\ -\underline{x} & \text{otherwise} \end{cases}$$

with an additional check to throw an exception if $\underline{y} \leq 0 \leq \overline{y}$.

3.8. **Reciprocal.** Definition:

$$\frac{1}{x} = \left[\frac{1}{\overline{x}}, \frac{1}{\underline{x}}\right] \subseteq \left\langle \nabla\left(\frac{-1}{(-\overline{x})}\right), \nabla\left(\frac{-1}{\underline{x}}\right) \right\rangle,$$

undefined if $0 \in x$.

This is implemented as a check if the argument contains zero, followed by division of -1 by the argument and swapping the two components.

3.9. **Absolute value.** Definition:

$$|x| = [\max(\underline{x}, -\overline{x}, 0), \max(-\underline{x}, \overline{x})] = \langle \max(0, \underline{x}, (-\overline{x}))), -\min(\underline{x}, (-\overline{x}))\rangle\,.$$

3.10. **Square.** Implemented as $x^2 = |x||x|$, using multiplication of positive numbers.

3.11. **Square root.** Definition:

$$\sqrt{x} = \left[\sqrt{\underline{x}}, \sqrt{\overline{x}}\right]$$

only defined if $0 \leq \underline{x}$.

Since the rounding is an integral part of the square root operation, and in this case we cannot achieve a negated result, we need to use another method to ensure rounding in the correct direction. We use the fact already mentioned in the subsection on multiplication, $\Delta(r) \leq -\nabla(-\epsilon - \nabla(r))$.

The formula we use is:

$$\sqrt{x} \subseteq \begin{cases} \left\langle \nabla\left(\sqrt{\underline{x}}\right), -\nabla\left(\sqrt{-(-\overline{x})}\right) \right\rangle, \text{ if } \nabla\left(\nabla\left(\sqrt{-(-\overline{x})}\right)\right)^2 = -(-\overline{x}) \\ \left\langle \nabla\left(\sqrt{\underline{x}}\right), \nabla\left(\nabla\left(-\epsilon - \sqrt{-(-\overline{x})}\right)\right) \right\rangle, \text{ otherwise} \end{cases}$$

(where ϵ is the smallest representable positive number).

The condition for making the first choice in this formula is only satisfied if the result of $\sqrt{-(-\overline{x})}$ is exactly representable, in which case $\nabla\left(\sqrt{-(-\overline{x})}\right) = \Delta\left(\sqrt{-(-\overline{x})}\right)$. Otherwise the second choice adjusts the high bound to the next representable number.

Note that if we don't require tight bounds, using only the second choice in the equation above is sufficient to implement interval square root.

If the argument is entirely negative, the implementation will raise an exception. If it contains a negative part, the implementation will crop it to only its non-negative part, to allow that computations such as $\sqrt{0}$ can be carried on in exact real arithmetic.

4. TRANSCENDENTAL FUNCTIONS

If the implementations of transcendental functions in the standard *C/C++* libraries satisfied the requirements of IEEE-754 rounding, interval versions of them could be implemented in a manner similar to above. Unfortunately, the accuracy of these libraries (or hardware implementations) is notoriously unreliable. Moreover, it is almost never possible to find information about the error bounds of these functions, which vary from architecture to architecture and even with different compilers and different compiler versions on the same machine.

Thus we decided to implement transcendental functions on intervals that produce certified bounds enclosing the result. They do not try to give tight (correctly rounded) bounds, instead prefer to overestimate but compute quickly. The elaborate theory and complicated implementation required to give tight bounds are beyond the scope of the intended application of our interval arithmetic implementation.

All the implementations rely on polynomial approximations generated using an implementation of the Remez algorithm (see e.g. [1]) with exact computation and certified error bounds. However, instead of finding the best Chebyshev approximation and using interval coefficients containing the real ones, we use multi-step approximation (suggested by [3]) where we approximate, round the highest-order coefficient to a double-precision number and then approximate again with a lower-degree polynomial using this rounded value as a fixed coefficient. The final coefficient is taken as an interval, expanded to accommodate the approximation error and rounded outwards.

With this the approximation of the function in its primary interval only requires the computation of this polynomial with interval arithmetic (in fact we do a little bit better, discussed below). The fact that all coefficients except the final additive are double-precision numbers helps to reduce the growth of the intervals. We choose our primary intervals to contain only non-negative numbers, so that multiplication of intervals can be performed as the special case that requires only two multiplications in a single instruction. For an additional speed-up, the polynomial evaluation is done using Estrin's algorithm (see [7]) to maximize parallelism.

For a monotone function, we know that if $P(x) = c_n x^n + ... + c_1 x + c_0$ chosen so that $P(x) - e \le f(x) \le P(x) + e$ for all x in some non-negative interval $[a, b]$,

$$\begin{aligned} P(\underline{x}) - e &\le f(\underline{x}) &&\le P(\underline{x}) + e \\ P(\overline{x}) - e &\le f(\overline{x}) &&\le P(\overline{x}) + e \end{aligned}$$

but for any $x \in [\underline{x}, \overline{x}]$, $f(\underline{x}) \le f(x) \le f(\overline{x})$, thus

$$\underline{P}(\underline{x}) \le P(\underline{x}) - e \le f(x) \le P(\overline{x}) + e \le \overline{P}(\overline{x}),$$

i.e. $f[x] \subseteq \left[\underline{P}(\underline{x}), \overline{P}(\overline{x})\right]$, where $\underline{P}$ (and $\overline{P}$) is P computed in such a way that it gives a lower bound for $P(x) - e$ (resp. a higher bound for $P(x) + e$). If all the coefficients are positive, this can be accomplished by simply rounding all coefficients down (resp. up), with the exception of c_0, which would also have to accommodate e, i.e. $\underline{c_0} = \nabla(c_0 - e)$ (resp. $\overline{c_0} = \Delta(c_0 + e)$). In our special case where all coefficients except c_0 are exactly represented in double precision, the coefficients of $\underline{P}$ and $\overline{P}$ coincide with the coefficients of P except for the very last one, c_0.

Unfortunately, in the presence of inexact operations, the evaluation of the polynomial is not so easy to do if the coefficients are not all positive. A negative coefficient requires an upper bound for x^i, which would be a nuisance to compute and would add up to the uncertainty of the result. However, in the cases we actually use we have patterns that can

be exploited, e.g. alternation between positive and negative coefficient. In the latter case, in the computation of $\underline{P}$ we can assume $\underline{x}$ is given exactly, thus we can compute pairs $c_{i+1}x + c_i$ rounded in the correct direction (these pairs are actually required by Estrin's algorithm). If we, additionally, know that all these pairs are positive (e.g. if $0 < x \leq 1$ and $-c_{i+1} \leq c_i$), the computation can proceed from there using only lower bounds for the even powers of x.

The transcendental functions in the current implementation of the interval arithmetic package of *RealLib* are not very precise, i.e. they overestimate the output intervals. The main reason for this is the use of Estrin's algorithm, which was chosen for its superior performance. At the moment we are considering improving the accuracy of these functions whenever such improvements would not drastically influence their performance.

On the other hand, since the functions do provide correct enclosures in very little time, and overestimation is one of the principles on which the exact real number library *RealLib* is based, the current transcendental functions completely serve their purpose as part of the library.

4.1. **Sine and cosine.** Sine and cosine are non-monotonic functions, which means that one cannot simply use $\sin x = [\sin \underline{x}, \sin \overline{x}]$. Instead, we use the fact that both functions are non-expansive and thus

$$\sin x = \left[\sin\left[\frac{\underline{x}+\overline{x}}{2}\right] + \frac{\underline{x}-\overline{x}}{2}, \sin\left[\frac{\underline{x}+\overline{x}}{2}\right] + \frac{\overline{x}-\underline{x}}{2}\right],$$

where by $\sin[x]$ we mean evaluation of $\sin$ on the interval $[x, x]$, returning an interval containing the result.

The latter we compute by a polynomial approximation of the function $\sin \frac{x}{3}$ on the interval $[-\pi, \pi]$ by an 8-coefficient polynomial, such that $\sin \frac{x}{3} \approx xP(x^2)$. Before we can apply this, we use range reduction (which can be safely performed as x is a real number and not an interval) to make sure $x \in [-\pi, \pi]$. To get the final value of $\sin x$, we use the identity $\sin(3x) = (3\sin^2 x - 4)\sin x$.

The computation of cosine is done in a very similar manner, the only significant difference is that the approximation used is $\cos \frac{x}{3} \approx P(x^2)$, i.e. the computation requires one multiplication less.

4.2. **Arctangent.** Two versions of this function are used in practice, one is the simple arctangent and the other one takes two arguments and gives a result that depends on the signs of both of them so that it can be directly used to compute polar coordinates or arguments of complex numbers.

Both are implemented using case distinctions and a common function that computes the arctangent for the primary interval $[0, 1]$. For the cases that contain numbers on the boundaries (e.g. $[-0.9, 1.1]$), we use the fact that arctangent is lipschitz continuous with constant one, this we simply return a constant interval expanded to accommodate the width of the input interval.

The computation on the primary interval is done simply by a polynomial approximation with 20 coefficients of alternating sign.

4.3. **Exponent.** Since the floating point representation of numbers uses a base-2 exponent, the easiest way to perform exponentiation is to transform the argument to base 2 (i.e. simply multiply by $\log_2 e$), separate the integer and fractional part, use some bit operations to construct a number with the integer part as the exponent, approximate the exponent of the fractional part and combine the two components using multiplication.

We do this separately for the lower and upper bounds of the interval. The extraction of integer and fractional part is an exact operation, but any other step in the computation requires that we round in the appropriate direction. The 12-coefficient polynomial approximation of 2^x for $x \in [0, 1)$ we use contains only positive coefficients, thus it presents no problem. The initial and final multiplication are done according to the rules of interval multiplication with one (resp. two) positive multiples.

4.4. **Logarithm.** The range reduction in the logarithm case is the inverse of the work done for exponentiation, with a few additional steps.

The mantissa and exponent are separated using a few bit operations, to produce a mantissa in the range $[0.5, 1)$. Unfortunately the direct approximation of the function $\ln x$ on this interval does not give us a polynomial which can be safely evaluated separately for the lower and upper bounds of an interval.

Instead, we approximate $\ln(1 - x)$ where $x \in (0, 1 - 2^{-\frac{1}{4}}]$, using two steps of range reduction to limit the number of coefficients to 14 (all positive). The range reduction is accomplished by choosing x or $x2^{2^{-i}}$ (for $i = 1, 2$) depending on whether the latter is smaller than one, adjusting the exponent by adding 2^{-i} if that is the case.

The result of the polynomial approximation is finally added to the (adjusted) exponent, multiplied by $\ln 2$.

5. Performance

We compare the performance of this implementation to the performance of two other packages for interval arithmetic freely available on the internet: the interval part of the *Boost* project (version 1.33.0, [8]) and the library *filib++* (version 2.0, [5]). For the latter, we tried the macro version as well as two of the available rounding policies, *multiplicative* and *native_onesided_global*, the latter corresponding most closely to our method of rounding.

The results of the benchmark are summarized in the following table, showing the ratio between the performance of the respective library and double precision floating point:

operation	*filib++*, macro	*filib++*, onesided	*filib++*, multiplicative	*Boost*	*RealLib3*
+	6.52	2.64	6.84	10.72	1.11
*	7.86	3.40	7.93	113.47	5.50
/	12.42	3.98	10.06	9.60	3.80
$\sqrt{\cdot}$	25.43	63.97	63.17	16.54	2.00
$\lvert\cdot\rvert$	27.11	20.23	20.21	1.61	2.62
$\sin$	2.91	2.63	2.73	-	0.63
$\cos$	2.92	2.75	2.74	-	0.58
$\arctan$	2.26	6.20	6.42	-	1.27
$e^{\cdot}$	3.45	22.25	40.30	-	0.86
$\ln$	3.86	5.91	6.13	-	0.94
$\sum_{i=1}^{1000000} \frac{1}{i}$	3.19	1.51	2.74	4.80	1.53

(Pentium-M 1.8GHz, Windows XP + Cygwin, GCC 3.4.4)

Several cells are blank, because *Boost* does not provide transcendental operations.

RealLib is faster almost everywhere, with the notable exception of multiplication in *filib++*'s *native_onesided_global* mode. In this case *filib++* uses a case distinction, which in our test only reaches the shortest of the 9 possible paths. We prefer not to explore the performance of *filib++*'s multiplication in cases where the signs of the arguments change in an unpredictable manner. Our implementation does not have such a problem as it only

uses one execution path for all multiplications, thus the ratio given in the table is both best and worst case performance.

6. INTEL'S SSE3

The latest multimedia extension set introduced by Intel, the SSE3 [12], aimed at improving complex number computations, does not provide any benefit for interval computations. Intel chose to improve complex multiplications and divisions by introducing the instruction *_mm_addsub_pd*, which combines two packed registers by adding one of the two components and subtracting the other [11]. Unfortunately, the use of this instruction leads to incorrect results if a directed rounding mode is in effect, because the multiplication that precedes the substraction is rounded in the wrong direction.

A better handling of complex multiplications would have been the introduction of a multiplication instruction "*mulpn*" (for multiply positive negative) that changes the sign of one of the components of one of the arguments. This would require the same effort that the instruction *_mm_addsub_pd* required, but would have the correct behavior in directed rounding modes, i.e. complex multiplication code using *mulpn* would yield upper bounds for the result of the multiplication if rounding towards $+\infty$ is in effect, and lower bounds in the case of rounding towards $-\infty$.

Unlike *_mm_addsub_pd*, a *mulpn* instruction would have been useful and advantageous for interval arithmetic. Multiplication of two positive numbers could be implemented as a single *mulpn*, which would also speed up the implementations of transcendental interval functions.

7. SUGGESTIONS FOR A HARDWARE IMPLEMENTATION

We hope that the presentation until this point has convinced the reader that the use of the storage $\langle \underline{x}, -\overline{x} \rangle$ for intervals in SSE2 registers is clearly superior to the traditional method of storing intervals as simply the pair of the two bounds. This mode of storage avoids the need for special rounding modes in a hardware implementation, and even turns some existing instructions into meaningful interval operations.

We propose this storage to be adopted as the preferred storage format for intervals in hardware implementations.

To further speed up computations on intervals, we propose the introduction of a special selection instruction we call *ivchoice* (for interval choice) that can be used to prepare the arguments for multiplication and division. The action of this instruction should correspond to the following function:

```
__m128d ivchoice(__m128d a, __m128d b) {
    a = _mm_xor_pd(a, _mm_set_pd(0.0, -0.0));
    a = _mm_shuffle_pd(a, a, _mm_movemask_pd(b));
    return a;
}
```

This is pseudocode, because *_mm_shuffle_pd* cannot be performed based on a non-const integer. A software implementation of the above requires a switch statement, which can slow the execution considerably, especially in cases where the signs of the multiples cannot be predicted.

If such an instruction is available, the multiplication algorithm becomes:

```
__m128d IntervalMul(__m128d x,
__m128d y) {
    __m128d a = _mm_shuffle_pd(x, x, 1);
```

```
    __m128d b = _mm_shuffle_pd(y, y, 1);
    __m128d c = ivchoice(b, x);
    __m128d d = ivchoice(y, a);
    __m128d e = _mm_mul_pd(c, x);
    __m128d f = _mm_mul_pd(d, a);
    __m128d g = _mm_min_pd(e, f);
    return g;
}
```

If the latency of the proposed instruction can be the same as the latency of *_mm_shuffle_pd*, this sequence of instructions will run about 30% faster than the current implementation.

Moreover, since the multiplications above only use the results of *ivchoice* with the same second argument, it is even possible to fuse *ivchoice* with the multiplication that is applied to the result. The extent to which such fusion can be beneficial depends on the actual hardware implementation. If the latency of *ivchoice* can be folded completely (which seems possible) or partially, interval multiplication using the fused "*ivmul*" could reach a latency close to the latency of two dependant double precision multiplications.

Apart from an additional test if the divisor contains zero and the use of *_mm_div_pd* instead of *_mm_mul_pd*, the division code is identical to the multiplication one:

```
 __m128d IntervalDiv(__m128d y, __m128d x) {
    if (_mm_movemask_pd(x)==3)
        throw exception;
    __m128d a = _mm_shuffle_pd(x, x, 1);
    __m128d b = _mm_shuffle_pd(y, y, 1);
    __m128d c = ivchoice(b, x);
    __m128d d = ivchoice(y, a);
    __m128d e = _mm_div_pd(c, x);
    __m128d f = _mm_div_pd(d, a);
    __m128d g = _mm_min_pd(e, f);
    return g;
}
```

Fused *ivchoice* and division ("*ivdiv*") is also possible.

Of course, one would prefer to have a complete hardware implementation of interval arithmetic that provides instructions for the four basic operations on intervals. In our mode of operation addition already has a hardware implementation as a single instruction. Substraction would require a fusion of swapping and addition ("*ivsub*") which should be easy to accomplish in hardware without extra latency compared to addition.

On the other hand, multiplication and division seem too complex to be directly implemented. A pure hardware implementation of multiplication may be able to choose execution paths without the delays associated with incorrect branch predictions, thus probably the preferable hardware design would examine the signs of the four components to choose one of 9 possible combinations and perform a single pair of multiplications in 8 of the possible cases. In the 9'th case, however, the operation would require the same amount of work as the function *IntervalMul* above.

Since the worst-case latency would be the same as the algorithm above, the latter should not be ignored as a possible basis for a pure hardware implementation of interval multiplication.

To conclude, we suggest that hardware assistance for interval computations should be provided as the adoption of the $\langle \underline{c}, -\overline{c} \rangle$ storage format and the introduction of the instructions of one of the following three levels:

basic	*mulpn*, *ivsub*, *ivchoice*
advanced	*mulpn*, *ivsub*, *ivmul*, *ivdiv*
full	*ivsub*, *IntervalMul*, *IntervalDiv*

The advanced level seems to be the best combination of feasibility and performance.

8. Related work

In [4], von Gudenberg discusses the efficiency of implementations of interval arithmetic using the multimedia extensions Intel's SSE, AMD's 3DNow! and Motorola's AltiVec. The paper concludes that the use of multimedia extensions only leads to a very modest improvement in multiplication with Intel's SSE in comparison to standard floating point, and only due to the fact that four single-precision operations can be executed in parallel.

Unlike SSE, the double precision second version of the extensions, SSE2, is a natural candidate for interval arithmetic because the packed registers hold two double precision values.

Von Gudenberg used a variety of rounding policies, the fastest of which is global onesided rounding, the method we use, but did not store one of the components negated in memory. Consequently, handling the negations required to perform rounding in the proper direction increases the number of instructions needed for every operation. If we were to use SSE2 in a similar mode of operation, the required number of instructions for addition would be four instead of one, for sign change – two instead of one, for substraction – five instead of two, and for multiplication of positive intervals – three instead of two.

Additionally, instead of 9-case branching on the signs of the 4 components, we prefer to use 4 multiplications with selected arguments (the selection is branch-free), which gives us stable performance that is not affected by branch mispredictions or longer latency execution paths, although with a slightly worse best-case performance.

In [6], Kolla, Vodopivec and von Gudenberg discuss the possibility of hardware extensions supporting interval arithmetic similar to the multimedia extensions 3DNow!, via packed storage of single precision numbers in a double precision register. For addition and substraction they require special instructions that round each component of the pair in the appropriate direction, and for multiplication they describe a case selection method that can easily be implemented and be very efficient for 8 of the 9 possible cases and requires a sequence of operations and longer latency for the (rare) 9'th case.

We are quite skeptical about the chances of such a complicated multiplication instruction ever being implemented in hardware. Instead, we give a much more modest proposal that can also lead to very good performance at the cost of little extra hardware. It also has the benefit that one of the operations, addition, already has a hardware implementation.

In [2], Ershov and Kashevarova report on implementations of transcendental functions, based on the Chebyshev and Taylor approximations of these functions. They note that three sources of error have to be accounted for in the computation of approximating polynomials:

- the error caused by finitely approximating an infinite sequence,
- the error in the approximation of the coefficients of the polynomial,
- the error caused by inexact operations.

The use of rounded coefficients influencing the choice of approximating polynomials in our approach nearly invalidates the need to consider the second source of error above. Our approximating polynomials only contain coefficients that are correctly representable as double precision floating point numbers, with the exception of the first coefficient, whose interval representation could be modified so that it also covers the first source of error in the list above.

REFERENCES

[1] Cheney, E. W. *Introduction to Approximation Theory*, 2nd ed. Providence, RI: Amer. Math. Soc. (1999).

[2] Ershov, A.G., Kashevarova, T.P., *Interval Mathematical Library Based on Chebyshev and Taylor Series Expansion*. Reliable Computing Vol. **11**, No. 5 (2005).

[3] Green, R., *Faster Math Functions*. Game Developers Conference (2002).
available at http://www.research.scea.com/research/pdfs/RGREENfastermath_GDC02.pdf

[4] von Gudenberg, J.W., *Interval Arithmetic on Multimedia Architectures*. Reliable Computing Vol. **8** No. 4 (2002).

[5] Hofschuster, W., Krämer, W., Lerch, M., Tischler G., von Gudenberg, J.W., *The Interval Library fi_lib++ 2.0 Design, Features and Sample Programs*. Preprint 2001/4,Universität Wuppertal, (2001).
available at http://www.math.uni-wuppertal.de/wrswt/preprints/prep_01_4.pdf

[6] Kolla, R., Vodopivec, A., von Gudenberg, J.W., *The IAX Architecture – Interval Arithmetic Extension*. Universität Würzburg, Institut für Informatik, Techn. Report TR225, April 1999.
available at http://www2.informatik.uni-wuerzburg.de/mitarbeiter/wvg/Public/iax.ps.gz

[7] Knuth, D., *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Third Edition. Reading, Massachusetts: Addison-Wesley, xiv+762pp. (1997).

[8] *Boost Interval Arithmetic Library*.
available at http://www.boost.org/libs/numeric/interval/doc/interval.htm

[9] IEEE Standards Committee 754, *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*. Institute of Electrical and Electronics Engineers, New York (1985). Reprinted in SIGPLAN Notices, 22(2):9–25 (1987).

[10] Intel Corp., *IA-32 Intel Architecture Software Developer's Manual, Volumes 1-3*.
available at http://developer.intel.com/design/pentium4/manuals/index_new.htm

[11] Intel Corp., *Using SSE3 Technology in Algorithms with Complex Arithmetic*.
available at http://www.intel.com/cd/ids/developer/asmo-na/eng/dc/pentium4/optimization/66717.htm

[12] Intel Corp., *Next Generation Intel Processor: Software Developers Guide*.
available at http://www.intel.com/cd/ids/developer/asmo-na/eng/dc/pentium4/optimization/66756.htm

BRICS, UNIVERSITY OF AARHUS, IT PARKEN, 8200 AARHUS N, DENMARK
E-mail address: barnie@brics.dk