Fast and Correct SIMD Algorithms for Interval Arithmetic

Frédéric Goualard

Université de Nantes, Nantes Atlantique Universités
CNRS, LINA, UMR 6241
2 rue de la Houssinière, BP 92208, F-44000 NANTES

Abstract. SIMD instructions on floating-point numbers have been readily available on ix86 computers for the past ten years or so. Almost from the start, they have been considered to compute the two bounds of intervals in parallel. Some authors reported marginal improvements over pure sequential computation, and those who reported otherwise had made choices that relinquished reliability. We discuss these works, pointing out their achievements and shortcomings, and then present data storage and SIMD algorithms that are both correct and much faster than the fastest routines in standard interval libraries.

1 Introduction

As microprocessor makers reach the limits imposed by physical laws, data parallelism becomes interesting to squeeze ever more performances from overstrained CPUs [19]. One of its strong points lies in that it may be used in otherwise purely sequential programs, thus avoiding the difficult task of properly synchronizing concurrent processes or threads. What is more, *Single Instruction, Multiple Data* (SIMD) instructions are now widely available (e.g., short-vector multimedia extensions AMD 3DNow! [1] and Intel SSE [9], or GPUs instruction sets [17]).

The Interval community has been quick to recognize the potential of SIMD floating-point instructions of ix86 processors to compute both bounds of an interval in parallel, thereby possibly achieving the same performances with interval arithmetic as with floating-point arithmetic. First reports of experiments by Wolff von Gudenberg [20, 21] were not enthusiastic, though, and the actual speed-up obtained proved marginal.

Part of the problem lies in that available SIMD instructions usually support only one rounding mode at a time for all operands, while interval bounds need to be rounded "outward" (that is, to opposite directions), in order to ensure the *containment principle* [16]. This problem can be circumvented, though it usually incurs some cost that may not be insignificant.

An original approach in this respect uses both Floating-Point Unit (FPU) instructions and Intel SSE instructions: Stolte [18] sets permanently the FPU and SSE rounding directions to opposite directions—as they may be set independently; he then computes left bounds and right bounds of intervals on different

units (either in FPU registers or in SSE registers). However, besides introducing another level of complexity to the implementation of interval libraries, such a scheme can only be used reliably in a carefully controlled environment with a well-known compiler, as some of them may choose to vectorize automatically some instructions that were intended for the FPU.

Some authors remove the problem entirely by not rounding bounds, thereby losing reliability. This is, however, considered a reasonable approach for specialized applications such as ray-tracing [11]. An orthogonal approach, sponsored by Malins et al. [15], tries to reduce the number of floating-point operations necessary to perform interval multiplications and divisions by recoursing to matrix multiplications of integers that symbolically represent interval bounds. According to the authors, the actual algorithm does not lead to good performances compared to traditional approaches, though.

Recently, Lambov [13] proposed a representation of intervals together with algorithms to better use SSE instructions[1]. However, we show below that their good performances are obtained at the cost of relinquishing reliability for intervals with null or infinite bounds.

We consider alternative algorithms based on previous works on purely sequential implementations of interval arithmetic, and we show it is indeed possible to achieve very good speed-ups while retaining reliability. In Section 4, the C++ library that embodies the ideas presented in this paper is compared with three (sequential) standard C or C++ interval arithmetic libraries and with two classes of SIMD algorithms from the points of view of speed and reliability. Lastly, we briefly discuss in Section 5 further extensions of our algorithms to evaluate in parallel an interval function for two different domains at the cost of one floating-point evaluation.

2 From Sequential to Data-Parallel Interval Arithmetic

The classical formulas to implement floating-point interval arithmetic are well known [16]:

$$[a,b] + [c,d] = [\mathrm{fl}_\triangledown(a+c)\,, \mathrm{fl}_\triangle(b+d)]$$
$$[a,b] - [c,d] = [\mathrm{fl}_\triangledown(a-d)\,, \mathrm{fl}_\triangle(b-c)]$$

$$[a,b] \times [c,d] = [\min(\mathrm{fl}_\triangledown(ac)\,, \mathrm{fl}_\triangledown(ad)\,, \mathrm{fl}_\triangledown(bc)\,, \mathrm{fl}_\triangledown(bd)),\\ \max(\mathrm{fl}_\triangle(ac)\,, \mathrm{fl}_\triangle(ad)\,, \mathrm{fl}_\triangle(bc)\,, \mathrm{fl}_\triangle(bd))]$$

$$[a,b] \div [c,d] = [\min(\mathrm{fl}_\triangledown\left(\tfrac{a}{c}\right)\,, \mathrm{fl}_\triangledown\left(\tfrac{a}{d}\right)\,, \mathrm{fl}_\triangledown\left(\tfrac{b}{c}\right)\,, \mathrm{fl}_\triangledown\left(\tfrac{b}{d}\right)),\\ \max(\mathrm{fl}_\triangle\left(\tfrac{a}{c}\right)\,, \mathrm{fl}_\triangle\left(\tfrac{a}{d}\right)\,, \mathrm{fl}_\triangle\left(\tfrac{b}{c}\right)\,, \mathrm{fl}_\triangle\left(\tfrac{b}{d}\right))], \text{ with } 0 \notin [c,d]$$

where a, b, c, and d are floating-point numbers, and $\mathrm{fl}_\triangledown(r)$ (resp. $\mathrm{fl}_\triangle(r)$) returns the largest float smaller than (resp. the smallest float larger than) the real r.

[1] More accurately, SSE2 instructions, which are part of an extension of the original SSE instruction set.

Unfortunately, these formulas should not be used as a basis for implementing interval arithmetic for efficiency and reliability reasons:

1. Switching incessantly the rounding direction from $-\infty$ to $+\infty$ to compute left and right bounds slows down the execution as each switch requires emptying the FPU pipeline;
2. The constraint that a divisor not contain 0 is too strong for some applications, and should be lifted in any practical implementation;
3. The IEEE 754 standard [8] achieves an affine extension of the floating-point numbers set by defining the two infinities $-\infty$ and $+\infty$. It is, therefore, most desirable that infinite bounds be supported by the interval arithmetic operators as well, since infinities may arise from any seemingly innocuous computation involving intervals with finite bounds only. For a trivial example, consider the simple interval expression:

$$\mathbf{I} \times \exp(\mathbf{J}^2) \quad \text{with} \quad \mathbf{I} \in [0, 2], \mathbf{J} \in [0, 27].$$

Since $\mathbf{J}^2 = [0, 729]$ and e^{729} is greater than the largest double precision floating-point number, we have $\exp(\mathbf{J}^2) = [1, +\infty]$, which leads to having to consider infinities for the interval multiplication of $\mathbf{I}$ by the exponential of $\mathbf{J}^2$.
Infinities may also naturally appear when a divisor contains 0 (consider, for example, the interval division involved in an interval Newton-Raphson step when the function studied has multiple roots in the domain under scrutiny). With infinite bounds, some products in the formula of the multiplication and some quotients for the division may lead to undefined numbers—a.k.a. *Not a Number* (NaN) [8]. Consider, for example:

$$\begin{aligned}[-\infty, 2] \times [0, 3] = [\ &\min(-\infty \times 0, -\infty \times 3, 2 \times 0, 2 \times 3),\\ &\max(-\infty \times 0, -\infty \times 3, 2 \times 0, 2 \times 3)\ ].\end{aligned}$$

The product of $-\infty$ by 0 is an NaN. As NaNs are unordered, the final result will depend entirely on the implementation of the ""min" and "max" functions. It might be the correct one (i.e., $[-\infty, 6]$), or it might be the invalid interval $[\mathrm{NaN}, \mathrm{NaN}]$ if $\max(\mathrm{NaN}, a) = \max(a, \mathrm{NaN}) = \min(\mathrm{NaN}, a) = \min(a, \mathrm{NaN}) = \mathrm{NaN}$.
The same problem will occur for the division whenever we have to compute the quotient $\frac{0}{0}$.

Eliminating the need for switching the rounding direction is easy: use the fact that taking the opposite of a result does not introduce rounding errors to define new formulas that rely on one rounding direction only (say, to $+\infty$). For example, we would have:

$$[a, b] + [c, d] = [-\operatorname{fl}_{\triangle}(-a - c)\,, \operatorname{fl}_{\triangle}(b + d)].$$

The problem of undefined numbers cannot be discarded so cheaply. In order to compute correct domains for the multiplication and the division in the

presence of infinities and zeros, we have to test the bounds in advance, which leads to algorithms with 9 cases for the multiplication and up to 36 cases for the division [7].

Wolff von Gudenberg [20, 21] implemented interval arithmetic with SSE instructions by using the "opposite trick" above. He tested both the brute-force algorithms (computing all 8 rounded multiplications or quotients) and some algorithms with cases. Improvement over sequential algorithms was found marginal. It is interesting to note here that he made no reference to the fact that brute-force algorithms were flawed and might sometimes return wrong results.

Lambov [13] suggested that the "opposite trick" was partly to blame for poor performances in that it introduced the need for extra instructions to repeatedly take the opposite. He then proposed to directly store intervals with one of their bounds negated. This idea was already present for the same purpose in the (purely sequential) JAIL interval library [4]. He also proposed new data-parallel algorithms for the multiplication and the division with fewer cases, so as to make possible their implementation without any explicit branching.

Unfortunately, as we will see in Section 4, his algorithms suffer from the same unreliability as brute-force algorithms, even if to a lesser extent.

3 Fast and correct SIMD interval Arithmetic

As in JAIL, we propose to use the rounding direction towards $+\infty$ only. An interval $[a, b]$ is stored as a pair $\langle b : -a \rangle$ of double precision floating-point numbers in an 128 bits SSE register (see Fig. 1). We then use the formulas presented in Tables 1, 2, 3, and 4, which are directly mappable to C or C++ code with *intrinsic instructions* [10]. To better illustrate our point, the pseudo-code using intrinsics is given in Tables 1 and 2 for the addition and the subtraction. It is also graphically illustrated in Figure 2.

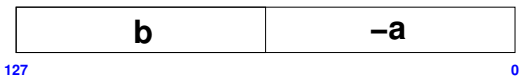

Fig. 1. Storing the interval $[a, b]$ in an 128 bits SSE register

Apart from the loading of the SSE registers, the interval addition requires only one instruction, while the subtraction requires two. The multiplication requires at most two floating-point multiplication SSE instructions and one "maximum" SSE instruction in the worst case, and one multiplication only in the best case, plus some additional instructions for testing the cases and swapping the bounds when necessary.

The interval division requires at most one floating-point SSE division, plus additional instructions for the cases handling. The tests performed before divid-

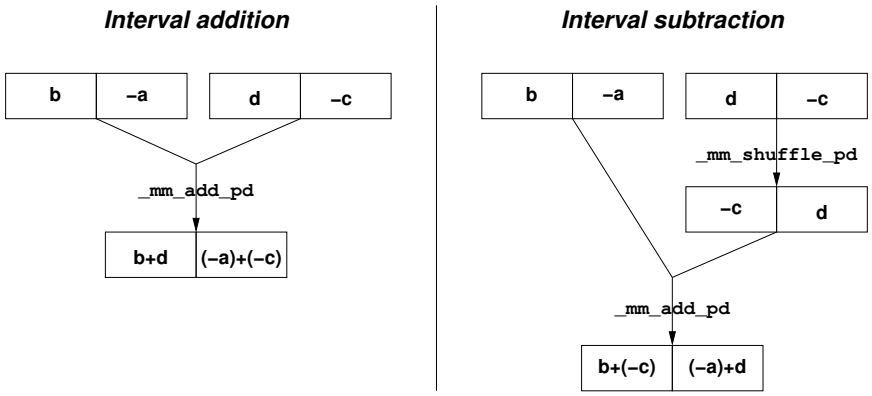

Fig. 2. Interval addition and subtraction with SSE registers and instructions

Table 1. Interval addition with SSE instructions on SSE registers

$\langle b : -a\rangle + \langle d : -c\rangle = \langle b + d : (-a) + (-c)\rangle$
`xmm0` $\leftarrow\langle b : -a\rangle$ `xmm1` $\leftarrow\langle d : -c\rangle$ `xmmres` $\leftarrow$`_mm_add_pd(xmm0,xmm1)`

Rounding set permanently towards $+\infty$

Table 2. Interval subtraction with SSE instructions on SSE registers

$\langle b : -a\rangle - \langle d : -c\rangle = \langle b + (-c) : (-a) + d\rangle$
`xmm0` $\leftarrow\langle b : -a\rangle$ `xmm1` $\leftarrow\langle d : -c\rangle$ `xmm1` $\leftarrow$`_mm_shuffle_pd(xmm1,xmm1,1)` `// xmm1` $\leftarrow \langle -c : d\rangle$ `xmmres` $\leftarrow$`_mm_add_pd(xmm0,xmm1)`

Rounding set permanently towards $+\infty$

ing the bounds are such that no situation that might lead to the creation of an NaN ever arises.

Table 3. Interval multiplication with SSE instructions on SSE registers

$\langle b:-a\rangle \times \langle d:-c\rangle$	$d \leqslant 0$	$c < 0 < d$	$0 \leqslant c$
$b \leqslant 0$	$\langle(-a)(-c) : (-b)d\rangle$	$\langle(-a)(-c) : (-a)d\rangle$	$\langle bc : (-a)d\rangle$
$a < 0 < b$	$\langle(-a)(-c) : b(-c)\rangle$	$\langle\max((-a)d, b(-c)) :$ $\max((-a)(-c), bd)\rangle$	$\langle bd : (-a)d\rangle$
$0 \leqslant a$	$\langle ad : b(-c)\rangle$	$\langle bd : b(-c)\rangle$	$\langle bd : a(-c)\rangle$

Rounding set permanently towards $+\infty$

Table 4. Interval division with SSE instructions on SSE registers

$\frac{\langle b:-a\rangle}{\langle d:-c\rangle}$	$d < 0$	$c < 0, d = 0$	0	$c < 0 < d$	$c = 0, 0 < d$	$0 < c$
$b < 0$	$\langle \frac{a}{d} : \frac{b}{-c}\rangle$	$\langle \infty : \frac{b}{-c}\rangle$	$\varnothing$	$\langle \infty : \infty\rangle$	$\langle \frac{b}{d} : \infty\rangle$	$\langle \frac{b}{d} : \frac{-a}{c}\rangle$
$a < 0, b = 0$	$\langle \frac{a}{d} : -0\rangle$	$\langle \infty : -0\rangle$	$\varnothing$	$\langle \infty : \infty\rangle$	$\langle 0 : \infty\rangle$	$\langle 0 : \frac{-a}{c}\rangle$
$a = b = 0$	$\langle 0 : -0\rangle$	$\langle 0 : -0\rangle$	$\varnothing$	$\langle 0 : -0\rangle$	$\langle 0 : -0\rangle$	$\langle 0 : -0\rangle$
$a < 0 < b$	$\langle \frac{a}{d} : \frac{-b}{d}\rangle$	$\langle \infty : \infty\rangle$	$\varnothing$	$\langle \infty : \infty\rangle$	$\langle \infty : \infty\rangle$	$\langle \frac{b}{c} : \frac{-a}{c}\rangle$
$a = 0, 0 < b$	$\langle 0 : \frac{-b}{d}\rangle$	$\langle 0 : \infty\rangle$	$\varnothing$	$\langle \infty : \infty\rangle$	$\langle \infty : -0\rangle$	$\langle \frac{b}{c} : -0\rangle$
$0 < a$	$\langle \frac{-a}{-c} : \frac{-b}{d}\rangle$	$\langle \frac{-a}{-c} : \infty\rangle$	$\varnothing$	$\langle \infty : \infty\rangle$	$\langle \infty : \frac{-a}{d}\rangle$	$\langle \frac{b}{c} : \frac{-a}{d}\rangle$

Rounding set permanently towards $+\infty$
Note: this is *functional* division and not the *relational* one, hence the results for the division by O

4 Experiments

To evaluate our algorithms, we randomly generated $20,000,000$ non-empty intervals. Our generator allows us to specify the probability of generating a bound

Table 5. Assessing performances of interval operators

Probabilities	0:0.2:0.2:0.6				0.05:0:0:0.95				0.05:0.05:0.05:0.85			
Operation	+	−	×	÷	+	−	×	÷	+	−	×	÷
BIAS 2.0.4	48.1	46.4	54.7	43.1	37.6	37.3	54.5	46.5	41.0	41.3	55.8	45.4
boost 1.34.1	35.8	35.5	45.7	20.6	30.2	29.8	49.9	40.4	32.7	32.8	50.6	24.4
filib++ 210905	62.4	60.9	69.5	61.0	52.0	52.9	77.4	71.6	58.0	57.3	77.8	71.5
bf	2.6	2.5	9.4	18.0	4.8	4.9	27.9	37.3	4.8	4.9	24.4	34.2
lambov	2.6	2.5	7.3	8.9	4.8	4.9	18.3	18.0	4.8	4.9	16.5	16.9
cell	2.6	2.5	9.0	9.1	4.8	4.9	19.4	20.9	4.8	4.9	18.8	20.5

Times in seconds on an Intel Core2 Duo T5600 1.83GHz (whetstone 100 000=1111 MIPS) for 200 000 000 operations on non-empty random intervals.

that is a denormal number, zero, an infinity, or a normal number. Half of these intervals were added, subtracted, multiplied or divided by the other half. In order to avoid misguided optimizations by the compiler, each result was added to an accumulator whose value was eventually printed. Hence, each test performs $10,000,000$ interval operations corresponding to the operator tested, and $10,000,000$ interval additions. To obtain larger runtimes in Table 5, these $20,000,000$ operations are repeated 10 times, leading to a total of $200,000,000$ operations per operator tested. Lastly, all tests were run several times with various probabilities, including one test where no interval had 0 or an infinity as one of its bounds.

All experiments were conducted on an Intel Core2 Duo T5600 1.83GHz. The Whetstone test [3] for this machine reports 1111 MIPS with a loop count equal to $100,000$.

We compared "cell", the library embodying the ideas presented in this paper, with three freely available interval libraries: BIAS [12], Boost Interval [2] and Filib++ [14]. None of these uses SIMD instructions. We also compared "cell" with Lambov's and SIMD brute-force algorithms. For all libraries, we chose the mode in which the rounding direction is set only once at the beginning of the program and never changed again, if available. We also chose settings that allowed 0 in the divisor of a quotient.

Tables 5 and 6 report our findings. Note that "bf" corresponds to the brute-force algorithm with SSE instructions, and "lambov" implements the algorithms by Lambov. Row "Probabilities" corresponds to the probabilities of generating denormal numbers, zeros, infinities, or normal numbers as bounds, in that order.

Table 5 presents the time in seconds necessary to performs the $200,000,000$ aforementioned tests.

Table 6. Assessing the correctness of interval operators

Probabilities	0:0.2:0.2:0.6				0.05:0:0:0.95				0.05:0.05:0.05:0.85			
Operation	×		÷		×		÷		×		÷	
Result (Larger/Wrong)	L	W	L	W	L	W	L	W	L	W	L	W
BIAS 2.0.4	0%	11%	8%	42%	0%	0%	0%	25%	0%	1%	3%	29%
boost 1.34.1	0%	0%	5%	0%	0%	0%	0%	0%	0%	0%	1%	0%
filib++ 210905	20%	0%	17%	1%	0%	0%	0%	0%	2%	0%	1%	1%
bf	0%	17%	2%	43%	0%	0%	0%	44%	0%	1%	1%	40%
lambov	0%	16%	13%	11%	0%	0%	0%	0%	0%	2%	3%	2%
cell	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%

Table 6 reports the percentage of the $10,000,000$ test operations[2] with larger than expected (L), or plain wrong (W) results. Wrong results originate from mishandlings of bounds (NaNs), as discussed above; too large intervals come from insufficiently discriminating algorithms (e.g., dividing $[-\infty, 0]$ by $[-\infty, 0]$ leading to $[-\infty, +\infty]$ instead of $[0, +\infty]$). The correct results against which all libraries are tested were computed using the algorithms of Hickey et al. [7] as implemented in the "gaol" library [5]. Correctness results for the addition and the subtraction are not given as all libraries passed successfully the tests.

From both tables, we see that Lambov's carefully tuned algorithms are indeed the fastest; "cell" is, however, not so far behind in terms of pure performances. As for correctness, "cell" is definitely the winner since it is the only library that does not compute wrong results, nor intervals larger than expected.

Side note 1: The performances of the boost library are very good without recoursing to SIMD instructions. As already noted by the author for the JAIL library [4], this might be explained by its heavy use of templated classes and functions, which allow extensive optimization by the C++ compiler.

Side note 2: Table 5 clearly shows the negative impact of denormal numbers on performances for SSE floating-point instructions (and, to a lesser extent, for FPU instructions as well): runtimes double when the probability of creating denormal bounds increases from 0 to 0.05. There is nothing new here, though: slowness of denormal numbers handling is already amply documented in the literature. Still, such a great impact should lead us to reconsider the true importance of denormal numbers for interval arithmetic as processors always give the option of not using them (e.g., by flushing too small numbers to 0).

[2] The $10,000,000$ additions performed to avoid misguided compiler optimizations are not considered here.

5 Discussion

Our tests show that it is possible to take advantage of widely available SIMD floating-point instructions to speed-up the computation of the interval operators $+, -, \times, \div$. As already suggested by Lambov [13], the same approach could be used to speed-up other operators such as trigonometric, hyperbolic or otherwise transcendental operators. Still, it is not yet clear whether comparable speed-ups are achievable for those operators, as their proper implementation is not straightforward.

On the other hand, it is possible to further accelerate interval computation at the cost of less accuracy by packing two intervals with single precision bounds (instead of double precision bounds) in one 128 bits SSE register (see Fig. 3).

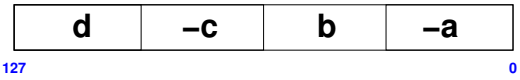

Fig. 3. Packing the two intervals $[a, b]$ and $[c, d]$ in one SSE register

Extending the definition of interval operators accordingly allows to compute an interval function **f** for two different domains in parallel at the cost of one floating-point evaluation of f. A preliminary implementation of this idea has already been successfully tested to design a data-parallel algorithm to solve nonlinear constraint systems [6].

Acknowledgments. Alexandre Goldsztejn made insightful comments on a preliminary version of this paper that hopefully helped to improve it.

References

1. Advanced Micro Devices, Inc. *3DNow! Technology Manual*, March 2000.
2. Hervé Brönnimann, Guillaume Melquiond, and Sylvain Pion. The design of the Boost interval arithmetic library. *Theor. Comput. Sci*, 351:111–118, 2006.
3. H. J. Curnow and Brian A. Wichmann. A synthetic benchmark. *Comput. J.*, 19(1):43–49, 1976.
4. Frédéric Goualard. Towards good C++ interval libraries: Tricks and traits. http://hal.archives-ouvertes.fr/hal-00430568/fr/, December 2000.
5. Frédéric Goualard. *GAOL 3.1.1: Not Just Another Interval Arithmetic Library.* Laboratoire d'Informatique de Nantes-Atlantique, 4.0 edition, October 2006. Available at `http://sourceforge.net/projects/gaol`.
6. Frédéric Goualard and Alexandre Goldsztejn. A data-parallel algorithm to reliably solve systems of nonlinear equations. In *Procs. of the 9th Intl. Conf. on Parallel and Distributed Computing, Applications and Technologies*, pages 39–46. IEEE Computer Society, 2008.

7. Timothy Hickey, Qun Ju, and Maarten Van Emden. Interval arithmetic: from principles to implementation. *Journal of the ACM*, 48(5):1038–1068, September 2001.
8. IEEE. IEEE standard for binary floating-point arithmetic. Technical Report IEEE Std 754-1985, Institute of Electrical and Electronics Engineers, 1985.
9. Intel. Intel 64 and IA-32 architectures software developer's manual: Vol. 1, basic architecture. Manual 253665-025US, Intel Corporation, November 2007.
10. Intel. Intel C++ intrinsic reference. Technical Report 312482-003US, Intel Corporation, 2007.
11. Aaron Knoll, Younis Hijazi, Charles Hansen, Ingo Wald, and Hans Hagen. Interactive ray tracing of arbitrary implicits with SIMD interval arithmetic. In *Proceedings of the 2nd IEEE/EG Symposium on Interactive Ray Tracing*, pages 11–18. IEEE, 2007.
12. Olaf Knüppel. PROFIL/BIAS—a fast interval library. *Computing*, 53:277–287, 1994.
13. Branimir Lambov. Interval arithmetic using SSE-2. In Peter Hertling, Christoph M. Hoffmann, Wolfram Luther, and Nathalie Revol, editors, *Reliable Implementation of Real Number Algorithms: Theory and Practice*, Dagstuhl Seminar Procs, 2006.
14. Michael Lerch, German Tischler, Jürgen Wolff Von Gudenberg, Werner Hofschuster, and Walter Krämer. Filib++, a fast interval library supporting containment computations. *ACM Trans. Math. Softw*, 32:299–324, 2006.
15. Eoin Malins, Marek Szularz, and Bryan Scotney. Vectorised/semi-parallel interval multiplication. In *Proceedings of SCAN 2006*, September 2006.
16. Ramon Edgar Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, N. J., 1966.
17. John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Tim Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.
18. Nilo Stolte. Arbitrary 3D resolution discrete ray tracing of implicit surfaces. In Eric Andres, Guillaume Damiand, and Pascal Lienhardt, editors, *Proceedings of the International Conference on Discrete Geometry for Computer Imagery*, volume 3429 of *Lecture Notes in Computer Science*, pages 414–426. Springer-Verlag, 2005.
19. Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005.
20. Jürgen Wolff von Gudenberg. Multimedia architectures and interval arithmetic. RR 265, Lehrstuhl für Informatik II. Universität Würzburg, October 2000.
21. Jürgen Wolff von Gudenberg. Interval arithmetic on multimedia architectures. *Reliable Computing*, 8:307–312, 2002.