Author manuscript, published in "8th Conference on Real Numbers and Computers, Santiago de Compostela : Spain (2008)"

hal-00263670, version 2 - 21 Sep 2009

A GPU interval library based on Boost.Interval [*]

Sylvain Collange
Université de Perpignan Via Domitia
sylvain.collange@univ-perp.fr

Jorge Flórez
Universitat de Girona
jeflorez@silver.udg.edu

David Defour
Université de Perpignan Via Domitia
david.defour@univ-perp.fr

Abstract

Interval arithmetic is widely used in numerical algorithms requiring reliability. Ray tracing of implicit surface is one of these applications that use interval arithmetic to increase the quality of a produced image. However these applications are computationally demanding. One solution is to use graphics processing unit (GPU) in order to take advantage of its computational power. We describe in this paper a GPU implementation of interval operators based on the Boost library. We tested these operators on a ray tracing algorithms and observe several order of execution speed improvements over the CPU version with the same image quality.

1 Introduction

Graphics processing units (GPUs) are rising a lot of interest in the research community as these units are cost effective and offer more computing resources than available on general-purpose processors [15]. They can provide up to 400 speed-ups depending on the application. However, to reach this performance the application has to map well on this architecture by being heavily data-parallel with high arithmetic density. Even though the application may fit the previous requirement, there exists a class of applications that may not map well on the GPU due to floating point restrictions. The numerous floating point units of today's GPU are single precision, not fully IEEE-754 compliant and do not offer the entire set of rounding modes.

Applications such as ray tracing of implicit surfaces may suffer of reliability problems [2, 7, 13]. These problems occur with thin features of implicit surfaces which are not correctly rendered and may "disappear". This happen if the intersection test between an implicit function and a ray that consist in finding zero is missing a solution of the function. A common solution is to replace floating point arithmetic with interval arithmetic (IA). Ray tracing based on IA requires however much higher computational times [6]. A solution would be to execute this algorithm on the GPU. However, current implementations of interval arithmetic on SIMD architectures either rely on IEEE-754 rounding modes not available on GPUs [12], or just ignore rounding issues, potentially returning incorrect results [10]. Thus, due to hardware restrictions, there is currently no efficient GPU implementation of guaranteed interval arithmetic operators.

This article proposes to implement basic interval arithmetic operations for ray tracing on a GPU. We show how we took into consideration GPU specificities in terms of floating point properties, instruction

[*]This work has been partially funded by the EVA-Flo project of the French ANR. This work has been possible thanks to the kind help of M Sbert (GIlab, UDG) and M. Daumas (ELIAUS, UPVD).

hal-00263670, version 2 - 21 Sep 2009

scheduling and memory access patterns to develop reliable and efficient IA operators. Section 2 recalls some basics about GPUs, section 3 introduces interval arithmetic, section 4 presents our implementation of IA on GPUs and section 5 gives some results.

2 Graphics processing units

Original graphics accelerator cards were special-purpose hardware accelerators for graphics programming interfaces such as OpenGL or DirectX. These programming interfaces are used to describe a scene using geometrical objects made out of vertices. An image made out of pixels is then produced. In order to accelerate image rendering, original GPUs used to implement application-specific functionalities directly in hardware. Lately, operations performed on vertex and pixel objects became more flexible through programmable vertex and pixel units. Even though vertex and pixel shaders perform different kinds of operations, there are sharing a large portion of similar features. Therefore the DirectX 10 standard and compatible hardware provide a so-called unified architecture where vertex and pixel shaders share the same instruction set and/or units. Hardware implementing both shader types includes memory units such as texture access units, computational units such as multiply and add operators, and special hardware such as special function evaluators. In order to efficiently exploit data parallelism, GPU includes numerous copies of these units working in SIMD fashion.

2.1 SIMD processor and memory

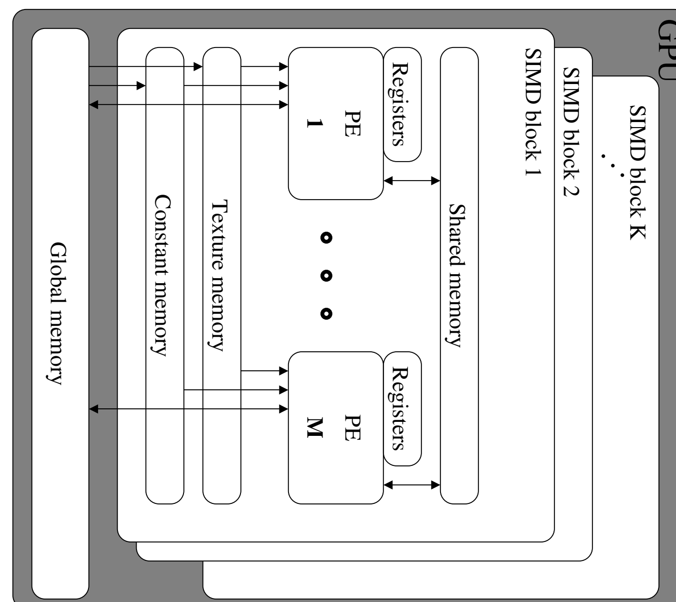
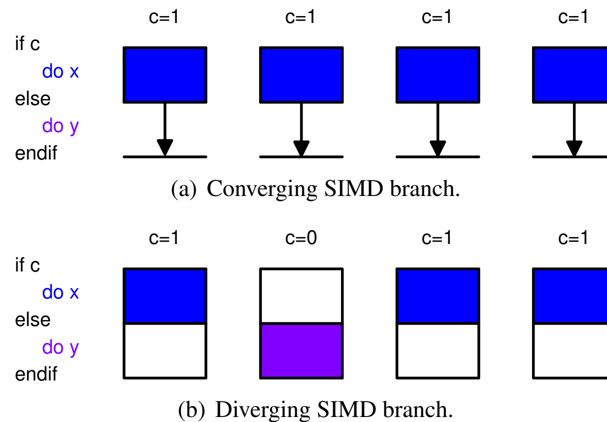


Figure 1. Unified architecture of a GPU

The unified architecture encouraged by the DirectX 10 standard has been implemented in hardware since the NVIDIA GeForce 8 and AMD ATI Radeon HD 2000 generations. This type of architecture is described in figure 1. The graphic processor is seen as a set of SIMD blocks. Each SIMD block is made of processing elements (PEs) that execute at each clock cycle the same instruction on different data. These SIMD blocks incorporate different kinds of memory such as a set of registers for each PE, memory shared among all the PE of a SIMD block, a constant memory and a read-only texture memory. In addition each PE can read or write in a global memory.

hal-00263670, version 2 - 21 Sep 2009



(a) Converging SIMD branch.

(b) Diverging SIMD branch.

Figure 2. Type of SIMD branching with 4 way.

The SIMD execution model implies two kinds of constraints. First, there are restrictions on the available memory access patterns (coalesced or broadcast), depending on the kind of memory (shared, constant or global). The second constraint is that the control flow has to remain the same for all the execution contexts within a SIMD block. Therefore a jump instruction is effectively executed as a jump if all the execution contexts follow the same path within a SIMD block, as seen in Figure 2(a). If one of the branches diverge, meaning one branch within the SIMD block is taking a different path, then it is necessary to run both branches of the conditional by applying a mask on the results as is the case in 2(b). This mechanism is called predication. Some SIMD architectures, including GPUs, are able to dynamically determine if a SIMD block can execute one side of the branch or if it is necessary to use predication and execute both sides of the branch.

2.2 Computational element

SIMD blocks integrate various computational units used to execute shaders (Figure 1). This includes general computation units that embed a Multiply and Add unit, texturing and filtering units, and a dedicated unit to evaluate general functions (sine, cosine, reciprocal, reciprocal square root, ...). The GeForce 8800 GTX from NVIDIA has 16 blocks and each SIMD block is composed of 8 PE and 2 units to evaluate general functions. The ATI Radeon HD 2900 XT from AMD embeds 4 SIMD blocks, each composed of 16 PE, capable of performing 5 different instructions simultaneously.

In the case of the GeForce 8, each processing element is able to perform floating-point operations such as addition and multiplication in IEEE-754 single precision. Both operations support round to nearest even and round toward zero via a statically defined rounding mode. Directed rounding toward $+/-$ infinity used for interval arithmetic is not supported. Additions and multiplications are often combined into a single multiply-and-add instruction ($a \cdot b + c$). In that case the result of the multiplication is truncated and the rounding is applied only on the last operation (the addition). There are other restrictions regarding the IEEE-754 standard such as uncorrectly rounded division and square root, the lack of denormalized numbers and no handling of floating point exceptions.

These units can handle integer and floating point arithmetic and there is no overhead associated with mixing both operations. Each SIMD block of the GeForce 8 is able to execute a pack of 32 floating point additions, multiplications, multiply-and-adds or integer additions, bitwise operations, compares, or evaluate the minimum or maximum of 2 numbers in 4 clock cycles. As there is no 32-bits integer multiplication in hardware, evaluating such an operation requires 16 clock cycles for a pack.

hal-00263670, version 2 - 21 Sep 2009

3 Interval arithmetic

Using interval arithmetic, it is possible to take into account uncertainties in data and return a reliable bound including the true result of a calculation. The basis of interval arithmetic is to replace each number by an interval surrounding it and to execute computations on intervals.

The most frequently used representation for intervals is the infimum-supremum representation. Interval variables will be noted using brackets "$[\cdot]$" to represent bounded intervals

$$[a_1, a_2] = \{x : a_1 \leq x \leq a_2\} \quad \text{for some} \quad a_1 \leq a_2,$$

We can define the set of bounded interval $I(R)$:

$$I(R) = \{[a_1, a_2] : (a_1, a_2) \in R^2, a_1 \leq a_2\}$$

from which we can extend the usual operations $(+, -, \times, /)$ from R to $I(R)$. Let $A = [a_1, a_2]$ and $B = [b_1, b_2]$, we define

$$\begin{aligned} A + B &= [a_1 + b_1, a_2 + b_2] \\ A - B &= [a_1 - b_2, a_2 - b_1] \\ A \times B &= [min(a_1b_1, a_2b_2, a_1b_2, a_2b_1), max(a_1b_1, a_2b_2, a_1b_2, a_2b_1)] \\ A/B &= [min(a_1/b_1, a_2/b_2, a_1/b_2, a_2/b_1), max(a_1/b_1, a_2/b_2, a_1/b_2, a_2/b_1)] \text{ when } 0 \notin B \end{aligned} \tag{1}$$

Common implementations of interval arithmetic on current processors are based on floating-point numbers. These processors follow the IEEE-754 standard that requires that the result of one of the four basic operations has to correspond to the rounding of the exact result. There are 4 rounding modes required by the standard: rounding to the nearest used by default as it is the most precise, rounding toward zero, and rounding toward plus or minus infinity used for interval arithmetic. Throughout this paper, the rounded values toward plus and minus infinity of a value a are denoted respectively $\overline{a}$ and $\underline{a}$. These last two rounding modes surround the exact result with floating point numbers. They can be implemented in hardware when available as it is the case on IEEE-754 compliant general purpose processors, or emulated in software. Therefore a floating point implementation of interval arithmetic is implemented as follow:

$$\begin{aligned} A + B &= [\underline{a_1 + b_1}, \overline{a_2 + b_2}] \\ A - B &= [\underline{a_1 - b_2}, \overline{a_2 - b_1}] \\ A \times B &= [min(\underline{a_1b_1}, \underline{a_2b_2}, \underline{a_1b_2}, \underline{a_2b_1}), max(\overline{a_1b_1}, \overline{a_2b_2}, \overline{a_1b_2}, \overline{a_2b_1})] \\ A/B &= [min(\underline{a_1b_1}, \underline{a_2b_2}, \underline{a_1b_2}, \underline{a_2b_1}), max(\overline{a_1b_1}, \overline{a_2b_2}, \overline{a_1b_2}, \overline{a_2b_1})] \\ A/B &= [min(\underline{a_1/b_1}, \underline{a_2/b_2}, \underline{a_1/b_2}, \underline{a_2/b_1}), max(\overline{a_1/b_1}, \overline{a_2/b_2}, \overline{a_1/b_2}, \overline{a_2/b_1})] \text{ when } 0 \notin B \end{aligned}$$

An error is associated with each rounding operation which generates larger interval for each operation that needs to be rounded compared to exact arithmetic.

Programmers can use interval arithmetic through libraries that define an interval type and a set of operations on this type. There exist several implementations of interval arithmetic such as [4, 8, 9], each with specific characteristics such as MPFI [16] that offers multiprecision intervals.

However it is very tedious to directly use these libraries as it is necessary to precisely know how there are working. The easiest is to use an extension of the programming language that integrates the type *interval*. Such extensions are available in C++ as for example with the Boost library, which includes an interval library [1]. This library exposes an interval template class and operators with some restriction on the execution environment (compiler, system, processor).

hal-00263670, version 2 - 21 Sep 2009

4 Implementation of interval arithmetics on GPU

In order to run our ray tracing algorithm on a GPU we developed an interval arithmetic library for graphics processors. This consists in describing operations that will be executed by shader processing units. Among all the available languages, shaders can be written in the Cg language [5] which gives an execution environment for both AMD ATI and NVIDIA chips or in CUDA [14] which is a C/C++ environment for NVIDIA chips. We implemented our interval library for both programming environments.

CUDA is a language to address GPU programming for general purpose computing. It consists in a development framework based on the C++ language and a GPU architecture. We took the Boost interval library [1] as a starting point to design our interval library for the GPU. This version allows programmers to benefit from C++ features such as templates with the computational power of GPUs through CUDA.

CUDA is however not supported on AMD ATI chips, nor on earlier NVIDIA GPUs, while Cg is a portable programming environment to develop shaders for graphics applications. Therefore to address portability issues, we implemented a version of the interval arithmetic algorithms in Cg. The Cg version exhibits some differences with the CUDA version as it is not possible to control the rounding mode in Cg programs.

In order to implement interval arithmetic on GPUs, it is necessary to know precisely how arithmetic operators behave in hardware. DirectX 10 requires IEEE-754 single precision without any requirement about rounding. GeForce 8 follows the standard and additionally provides correct rounding with some restrictions (see section 2.2). For other hardware, specific tests have to be done as in [3] as public information is lacking. This leads to various performance tradeoffs depending on requirements such as including tests for special values or dealing with exact values.

4.1 GPUs specific considerations

4.1.1 Rounding

GPUs do not support rounding modes toward +/- infinity used for interval arithmetic. They offer instead a round to nearest even mode and/or a round toward zero mode for multiplication and addition depending on the hardware (Section 2.2).

To preserve hardware compatibility, we choose to implement interval arithmetic using faithful rounding for our Cg implementation. This rounding mode can be accessed easily for addition and multiplication on NVIDIA GeForce 7 and GeForce 8, and AMD ATI Radeon HD 2000 and with additional work for ATI Radeon X1000 series [3].

Under CUDA, the rounding mode can be statically set for each operation using a flag in the opcode encoding. We perform our computations in round-toward-zero in our CUDA implementation. Depending on the sign of the value, this provides either the rounded-down or rounded-up result. The other rounding direction is then obtained by adding one ulp to the rounded-down result or subtracting one ulp to the rounded-up result. However this software rounding leads to an overestimation of the error when the result is exact.

There are two known tricks based on the IEEE-754 floating point representation format to add/substract an ulp to the result. The first solution consists in incrementing the binary representation of the floating-point value, and the second in multiplying the results by $1 + 2^{-23}$ rounded toward 0. Let us call this operation *NextFloat*. For performance reasons, on a CPU *NextFloat* is usually implemented as an incrementation operation over integer plus some extra instructions to handle special cases such as denormals and infinities. On GPU, floating-point multiplication is less expensive than it is on a CPU and the multiplication throughput can be even higher than the addition throughput on GeForce 8. We tested both on the GPU and noticed that the solution based on a multiplication, which does not requires extra care for special cases, was more suitable for an execution on a GPU.

hal-00263670, version 2 - 21 Sep 2009

4.1.2 Branches in a SIMD architecture

As GPUs are Single-Instruction Multiple-Data (SIMD) architectures, diverging branches are expensive. Whenever one Processing Element (PE) of the SIMD array takes a different path, the hardware has to execute sequentially both code paths for all PEs, using predication to mask results.

On a CPU efficient branch prediction mechanisms make algorithms involving less operations and more branches attractive. For example, the multiplication or division can be done by choosing the operation to be done depending on the sign of each operand. On a GPU, when there is a risk of divergence in the execution on each PE within a SIMD block, a different algorithm with linear code is more suitable.

4.1.3 Truncation of multiplication in MAD operation

GPUs are usually collapsing a consecutive multiplication and addition into a single operation called MAD. This leads to smaller code and faster execution. However up to the GeForce 8 or R500 executing the result of a MAD can be different from the result of the multiplication followed by an addition. This difference lies in the internal design of this operator. A MAD is usually done in hardware with a modified multiplier combined with an extra addition done in the last stage before the rounding.

The GeForce 8 is designed in such a way that a multiplication or an addition alone can be rounded independently (rounded to the nearest or truncated). When there are grouped together in a MAD, the result of the multiplication is first truncated and the result of the addition is then rounded to the nearest or truncated. The GPU driver can reorder and group the instructions together and the programmer does not have control over it. However our implementation is based on truncation and the sequence of operations involved do not suffer from this problem.

4.2 Implementation issues

4.2.1 Addition

An implementation of the addition or subtraction of two intervals on a CPU uses directed rounding modes. However these two rounding modes are not available on the GPU. We have to emulate them using the round-toward-zero mode and the *NextFloat* function. This give the following CUDA code:

Listing 1. Interval addition in CUDA

```
__device__ interval sumI(interval x, interval y) {
        interval result;
        float a = __fadd_rz(x.inf, y.inf);
        float b = __fadd_rz(x.sup, y.sup);

        result.inf = min(a, next_float(a));
        result.sup = max(b, next_float(b));
        return result;
}
```

The situation is different as we address the Cg implementation. The result of the computation correspond to the rounding to the nearest or next-to nearest. With this rounding mode, there is no efficient solution to determine whether the exact result is greater or lower than the rounded result. The lower and upper bound of a reliable interval correspond to the rounded result plus or minus one ulp. This increases the length of the interval by one ulp compared to the direct rounding solution in CUDA. A solution to avoid this extra ulp is to determine the sign of the rounding error with a modified *Fast2Sum* procedure [11]. The Cg code for the addition is given in Listing 2.

hal-00263670, version 2 - 21 Sep 2009

Listing 2. Interval addition in Cg

```
float2 sumI(float2 x, float2 y) {
        float2 result = x + y;

        float one_minus_2_23 = 1 - pow(2.0,-23);
        float one_plus_2_23 = 1 + pow(2.0,-23);

        float2 to_zero = result *  one_minus_2_23;
        float2 to_inf = result * one_plus_2_23;
        float lower = min(to_zero.x, to_inf.x);
        float upper = max(to_zero.y, to_inf.y);
        return float2(lower,upper);
}
```

4.2.2 Multiplication

An implementation of interval multiplication without branches can be written along the general formula:

$$[a, b] \times [c, d] = [\min(\underline{ac}, \underline{ad}, \underline{bc}, \underline{bd}), \max(\overline{ac}, \overline{ad}, \overline{bc}, \overline{bd})]$$

A naive implementation would emulate each directed rounding using round-toward-zero and the *NextFloat* function. For each subproduct xy, we would have to compute:
$xy_0 = rz(x \times y)$
$xy_\infty = NextFloat(xy_0)$
$\underline{xy} = \min(xy_0, xy_\infty)$
$\overline{xy} = \max(xy_0, xy_\infty)$

This would require a total of 4 multiplications, 4 *NextFloat* calls and 14 min and max operations. However, it is possible to reduce the number of operation by studying the sign of each subproduct according to the signs of a, b, c and d. All cases are depicted in table 1, using '+' for non-negative numbers and '-' for non-positive numbers.

Table 1. Signs of upper and lower bounds of $[a, b] \times [c, d]$.

a	b	c	d	lower bound		upper bound	
+	+	+	+	$\underline{ac}$	+	$\overline{bd}$	+
-	+	+	+	$\underline{ad}$	-	$\overline{bd}$	+
-	-	+	+	$\underline{ad}$	-	$\overline{bc}$	-
+	+	-	+	$\underline{bc}$	-	$\overline{bd}$	+
-	+	-	+	$\min(\underline{bc}, \underline{ad})$	-	$\max(\overline{ac}, \overline{bd})$	+
-	-	-	+	$\underline{ad}$	-	$\overline{ac}$	+
+	+	-	-	$\underline{bc}$	-	$\overline{ad}$	-
-	+	-	-	$\underline{bc}$	-	$\overline{ac}$	+
-	-	-	-	$\underline{bd}$	+	$\overline{ac}$	+

We can observe that, regardless of the rounding direction, ac and bd are always non-negative, while ad and bc stay non-positive whenever they are used in the result. This information helps us to statically set the rounded-up and rounded-down of ac, bd, ad, bc. For these values we can define:
$\underline{ac} = ac_0$, $\overline{ac} = ac_\infty$,
$\underline{bd} = bd_0$, $\overline{bd} = bd_\infty$,
$\underline{ad} = ad_\infty$, $\overline{ad} = ad_0$,
$\underline{bc} = bc_\infty$, $\overline{bc} = bc_0$.

This leads to the following simplifications in the algorithm:
$\max(\overline{ac}, \overline{bd}) = \max(ac_\infty, bd_\infty)$
$\min(\underline{ad}, \underline{bc}) = \min(ad_\infty, bc_\infty)$

We can further reduce the number of operation by noticing that the function *NextFloat* which add one ulp is an increasing function that preserves the ordering on positive values. Therefore, we can safely execute the *NextFloat* operation after computation of min/max. This leads to the following simplifications:
$\max(\overline{ac}, \overline{bd}) = NextFloat(\max(ac_0, bd_0))$
$\min(\underline{ad}, \underline{bc}) = NextFloat(\min(ad_0, bc_0))$

We developped two versions of the multiplication which can be statically selected at compile time. The first handles NaN and overflows and the other one without these tests is used for the raytracing algorithm. Both version include the previous simplifications based on the study of the sign of the results according to the sign of the inputs. The multiplication algorithm used for ray tracing requires 4 multiplications, 2 *NextFloat* and 6 min and max. This is less than half the number of *NextFloat* and mix/max operations in the original version.

4.2.3 Power to a constant natural

Powering by a small constant integer such as square, cube or fourth power is an operation widely used in ray-tracing of implicit surfaces. When this operation is done with dedicated interval algorithm, it is possible to exploit the fact that variable are dependant, resulting in a smaller interval. For example if we want to compute the square of an interval, we know that the solution is non-negative and that the resulting interval cannot include negative numbers. This property cannot be obtained if the square is computed with an interval multiplication.

Table 2. Signs of upper and lower bounds of $[a, b]^n$.

a	b	n even				n odd			
		lower		upper		lower		upper	
+	+	$\underline{a^n}$	+	$\overline{b^n}$	+	$\underline{a^n}$	+	$\overline{b^n}$	+
-	+	0	+	$\max(\overline{a^n}, \overline{b^n})$	+	$\underline{a^n}$	-	$\overline{b^n}$	+
-	-	$\underline{b^n}$	+	$\overline{a^n}$	-	$\underline{a^n}$	-	$\overline{b^n}$	-

Similarly to interval multiplication, we studied the different possibilities for the results depending on the signs of the operands and the parity of the exponent in table 2. This leads to a reduction of the number of instructions as well as the resulting interval for intervals that include 0.

We defined and implemented on the GPU algorithms for the evaluation of $\underline{a^n}$ and $\overline{a^n}$ for small and statically defined values of n. We first evaluate $\underline{a^n}$ by successive multiplications in round-toward-zero mode using the binary method describes by Knuth ([11], page 461). Then we deduce $\overline{a^n}$ by adding to $\underline{a^n}$ a bound on the maximum rounding error performed at each multiplication with the same technique used for *NextFloat*. On a GeForce 8 this is done as follows:

$$\overline{a^n} = \underline{a^n}(1 + n \cdot 2^{-23})$$

This is valid as long as $n < 2^{22}$ which make this algorithm safe as it is used only for small values of n. Likewise, underflow only needs to be detected at the end of the power computation.

On current GPU architectures, looping constructs are expensive. Small loops with a constant number of iterations should therefore be unrolled. However, CUDA 1.0 does not support loop unrolling, and although CUDA 1.1 does, it lacks the capability to perform constant propagation and dead code removal after unrolling in the case of the power function. Therefore, we completely unroll the loop and propagate constants at language level using C++ template metaprogramming.

hal-00263670, version 2 - 21 Sep 2009

Table 3. Number of instructions of an interval computation at various levels of generality.

Implementation	Add	Mul (optimized)	Mul (original algorithm)	Square	x^5
General	17	21	93	26	22
No NaN	10	14	86	23	14
No NaN, no underflow	6	12	80	20	10

Table 4. Measured performance of an interval computation at various levels of generality, in cycles/warp.

Implementation	Add	Mul (optimized)	Mul (original algorithm)	Square	x^5
General	46	55	49 – 117	30 – 35	71
No NaN	36	45	45 – 63	24 – 29	34
No NaN, no underflow	24	36	33 – 55	20 – 25	23

hal-00263670, version 2 - 21 Sep 2009

5 Results

5.1 Interval library

We did specific tests on the interval library to measure its performance. We generated assembly code with the CUDA 1.1 compiler provided in the programming environment from NVIDIA. Then we used the Decuda toolset written by Wladimir J. van der Laan [1] to disassemble the NVIDIA CUDA binary (.cubin) generated and looked in detail how our interval arithmetic operators are compiled for the GPU. This let us determine precisely the number of instruction necessary for each version of our algorithm and study assembly-level bottlenecks. Results are given in table 3.

We also did some timing measures on a NVIDIA Geforce 8500 GT for the proposed implementation of IA operators, which are summarized in table 4. For each given measure we ran 2^{24} iterations with 8 warps/block which represent 256 threads/block and 4 blocks per grid. We observed a variation of less than 0.2% in timing results and deduced from these results the number of cycles per iteration by subtracting the time required to execute an "empty" loop.

5.2 Reliable ray tracing

We also tested the IA library within a GPU implementation of the reliable ray tracing algorithm describes in [6]. The algorithm was tested on a GeForce 8800 GPU. The resolution selected for the images was 1024 x 1024. The first surfaces tested are a Drop (figure 3a) and a Tri-thrumpet (figure 3c). We compared the rendered images with images rendered without interval arithmetic. We observe that the thin parts of those surfaces are correctly rendered with a quality similar to the CPU version of the same reliable ray tracing.

We compared the execution time of the GPU implementation with a CPU version. The tests were done on DELL 670 Workstation with a 3Ghz Xeon processor, 3 Gigabytes of RAM and a GeForce 8800 GTX GPU. The execution time measured is the time necessary to load the data and instructions, execute

[1] http://www.cs.rug.nl/~wladimir/decuda/

the program and get the final results which is an image of 1024x1024 pixels on both CPU and GPU version. Results are given in table 5. We observe that the time required to render these surfaces with GPU is divided by a factor ranging from 100 to 300.

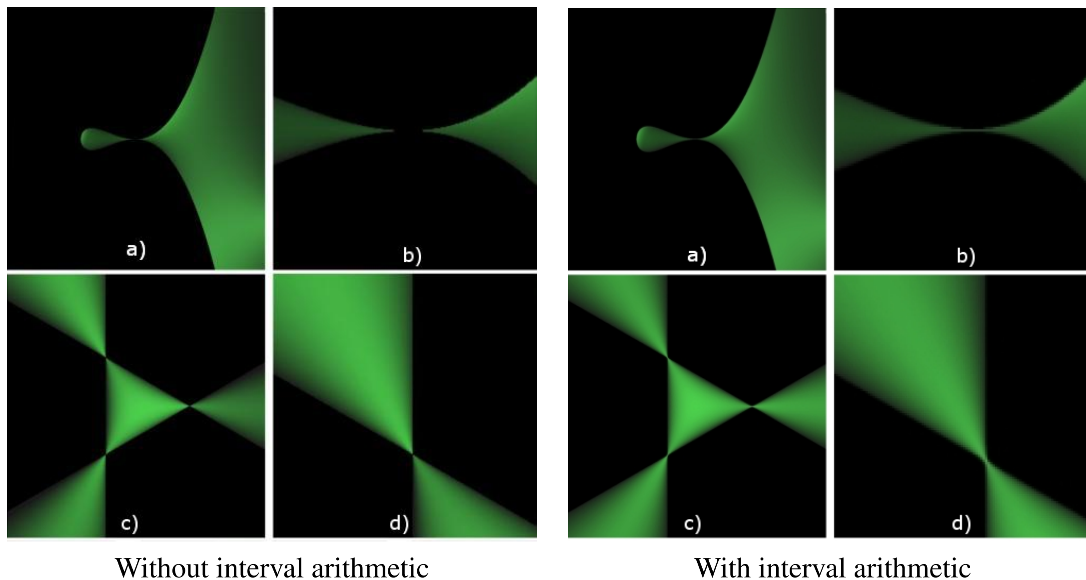


Figure 3. Comparison of surfaces rendered without interval arithmetic (left) and with interval arithmetic (right). A Drop surface (a) and a Tri-trumpet surface (c) and the details of there corresponding thin section in (b) and (d).

Table 5. Comparison of CPU Times vs. GPU times for four surfaces (in seconds).

Surface	CPU	GPU
Sphere	300	2
Kusner-Schmitt	720	2
Tangle	900	3
Gumdrop Torus	1080	3

6 Conclusions

We described how to implement in CUDA and CG common operators for interval arithmetic on a GPU. We took into consideration the GPU specificities in order to provide efficient operators. These operators are provided to end-users through the Boost Interval library. This library opens up new areas of improvement through the use of the computational horse-power of GPUs to critical applications requiring reliability. We tested this library with a reliable ray tracing algorithm on implicit surfaces and obtained a speed-up of 100 to 300 compared to a similar algorithm executed on a CPU.

In the near future we are planning to complete the library with other operations like square root and division. We are also planning to compare this implementation of interval arithmetic with other representations of intervals which may be more suitable with vector and parallel processors such as the midpoint-radius representation.

hal-00263670, version 2 - 21 Sep 2009

References

[1] Hervé Brönnimann, Guillaume Melquiond, and Sylvain Pion. The design of the boost interval arithmetic library. *Theor. Comput. Sci.*, 351(1):111–118, 2006.

[2] O. Capriani, L. Hvidegaard, M. Mortensen, and T. Schneider. Robust and efficient ray intersection of implicit surfaces. *Reliable Computing*, 1(6):9–21, 2000.

[3] Sylvain Collange, Marc Daumas, and David Defour. Line-by-line spectroscopic simulations on graphics processing units. *Computer Physics Communications*, 2007.

[4] George F. Corliss. INTPAK for interval arithmetic in Maple : introduction and applications. Article soumis au Journal of Symbolic Computation.

[5] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[6] J. Flórez, Mateu Sbert, Miguel A. Sainz, and Josep Vehí. Improving the interval ray tracing of implicit surfaces. *Lecture Notes in Computer Science*, 4035:655–664, 2006.

[7] D. Kalra and A. Barr. Guaranteed ray intersection with implicit surfaces. *Computer Graphics (Siggraph proceedings)*, 23:297–206, 1989.

[8] Ralph Baker Kearfott, M. Dawande, K. S. Du, and C. Y. Hu. Algorithm 737: INTLIB : a portable Fortran 77 interval standard function. *ACM Transactions on Mathematical Software*, 20(4):447–459, 1994.

[9] R. Klatte, Ulrich W. Kulisch, A. Wiethoff, C. Lawo, and M. Rauch. *C-XSC – a C++ class library for extended scientific computing*. Springer-Verlag, 1993.

[10] Aaron Knoll, Younis Hijazi, Charles D Hansen, Ingo Wald, and Hans Hagen. Interactive ray tracing of arbitrary implicits with SIMD interval arithmetic. In *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing*, 2007.

[11] Donald E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*. Addison-Wesley, 1997. Third edition.

[12] Branimir Lambov. Interval arithmetic using SSE-2. In *Reliable Implementation of Real Number Algorithms: Theory and Practice*, number 06021 in Dagstuhl Seminar Proceedings, 2006.

[13] Don Mitchell. Robust ray intersection with interval arithmetic. *Proceedings on Graphics interface '90*, pages 68–74, 1990.

[14] nVidia. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide, Version 1.0*, 2007.

[15] Matt Pharr, editor. *GPUGems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, 2005.

[16] Nathalie Revol and Fabrice Rouillier. Motivations for an arbitrary precision interval arithmetic and the mpfi library. *Reliable Computing*, 11(4):275–290, 2005.

hal-00263670, version 2 - 21 Sep 2009