A GPU library of interval arithmetic

Sylvain Collange, Jorge Flórez and David Defour

03/04/2008
Ray tracing on implicit surfaces

- Application in computer graphics
- Search of function roots using an interval Newton method

- Not Guaranteed (91 seconds)
- Guaranteed (137 seconds)
- Polygonal Approximation

- Interval operations required:
  $+, -, \times, \div, \sqrt{}, x_i, e^x$
Outline

- Interval arithmetic
- GPU architectural issues
- Implementing directed rounding
- Further optimizations
Interval arithmetic

- Redefinition of basic operators +, -, ∗, /,... on intervals
- Inclusion property: for \( X, Y, Z \) intervals, \( \circ \) any operator

\[
Z = X \circ Y \rightarrow (\forall x \in X, \forall y \in Y, z = x \circ y \rightarrow z \in Z)
\]

- Need to take rounding errors into account
- May return a wider interval
- Loss of variable dependency

- \( X^2 \neq X \times X \)
Outline

- Interval arithmetic
- GPU architectural issues
- Implementing directed rounding
- Further optimizations
Floating-point arithmetic on GPUs

NVIDIA GeForce 8/9 families, AMD Radeon HD 2000

- IEEE single precision
- Multiplication, addition correctly rounded to nearest
  - With Cuda, can round toward zero
  - No directed rounding
- Division, square root accurate to 2 ulps
- Multiply-Add with multiply rounded toward zero on NVIDIA

AMD Radeon HD 3000

- Division correctly rounded to nearest
SIMD execution

- The same instruction is run on multiple “threads”
- When a branch instruction occurs
  - If all threads follow the same path, do a real branch
  - Otherwise, execute both paths and use predication

- Diverging branches are expensive
Programming GPUs

- NVIDIA CUDA
  - For GeForce 8/9 GPUs only
  - Subset of C++ language
  - Templates not officially supported in CUDA 1.1
    - but their use is encouraged [Harris07]
    - in practice, most C++ features are supported

- Graphics API + shader language
  - Limited languages and features
  - Remains the only portable way
Outline

- Interval arithmetic
- GPU architectural issues
- Implementing directed rounding
- Further optimizations
Two libraries

- CUDA version
  - In C++
  - Adapted from the Boost Interval library
  - Provides various levels of consistency checking
- Cg version
  - Need GPU-specific routines
- We focus on the CUDA version
Implementing directed rounding

- We want round-down and round-up
- Cuda provides round-toward-zero
  - Equivalent to either round-down or round-up, depending on sign
- For the other way
  - Add one ulp to the rounded-toward-zero value
  - An integer addition on FP representation does this
    - Except in case of underflow/overflow
  - Or a multiplication by \((1 + \text{EPS})\) \(\text{EPS}=2^{-23}\) for single precision
    - No special cases for overflows/NaNs
Implementing directed rounding

- We want round-down and round-up
- Cuda provides round-toward-zero
  - Equivalent to either round-down or round-up, depending on sign
- For the other way
  - Add one ulp to the rounded-toward-zero value
  - An integer addition on FP representation does this
    - Except in case of underflow/overflow
  - Or a multiplication by \((1+\text{EPS})\) \(\text{EPS}=2^{-23}\) for single precision
    - No special cases for overflows/NaNs
Performance results

CUDA on a NVIDIA GeForce 8500 GT

- Throughput in cycles/warp

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>24</td>
</tr>
<tr>
<td>Multiplication</td>
<td>33 – 55 (*)</td>
</tr>
<tr>
<td>Square</td>
<td>20 – 25 (*)</td>
</tr>
<tr>
<td>$x^5$</td>
<td>141 – 148 (*)</td>
</tr>
</tbody>
</table>

* With uniform branching
Outline

- Interval arithmetic
- GPU architectural issues
- Implementing directed rounding
- Further optimizations
if (interval_lib::user::is_neg(xl))
    if (interval_lib::user::is_pos(xu))
        if (interval_lib::user::is_neg(yl))
            return I(::min(rnd.mul_down_neg(xl, yu), rnd.mul_down_neg(xu, yl)),
                    ::max(rnd.mul_up_pos (xl, yl), rnd.mul_up_pos (xu, yu)), true);
        else
            return I(rnd.mul_down_neg(xu, yl), rnd.mul_up_pos(xl, yl), true);
    else
        if (interval_lib::user::is_pos(yl))
            return I(rnd.mul_down_pos(xl, yu), rnd.mul_up_pos(xu, yu), true);
        else
            return I(static_cast<T>(0), static_cast<T>(0), true);
else
    if (interval_lib::user::is_pos(xu))
        if (interval_lib::user::is_neg(yl))
            if (interval_lib::user::is_pos(yu))
                return I(rnd.mul_down_neg(xu, yl), rnd.mul_up_pos(xu, yu), true);
            else
                return I(rnd.mul_down_neg(xu, yl), rnd.mul_up_neg(xl, yu), true);
        else
            if (interval_lib::user::is_pos(yu))
                return I(rnd.mul_down_pos(xl, yl), rnd.mul_up_pos(xu, yu), true);
            else
                return I(static_cast<T>(0), static_cast<T>(0), true);
Rewriting multiplication

- We do not want branches
- Starting from the general formula:

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

- We need to round all subproducts both up and down
  - Cost: \(4 \times (2 \text{ mul} + 1 \text{ min} + 1 \text{ max}) + 3 \text{ min} + 3 \text{ max}\)
- Or do we?
What is the sign of \([a, b] \times [c, d]\)?

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>lower bound</th>
<th>upper bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>(ac)</td>
<td>(bd)</td>
</tr>
<tr>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>(ad)</td>
<td>(bd)</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>(ad)</td>
<td>(bc)</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>(bc)</td>
<td>(bd)</td>
</tr>
<tr>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>(ad)</td>
<td>(bd)</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>(bc)</td>
<td>(ad)</td>
</tr>
<tr>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>(bc)</td>
<td>(ac)</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>(bd)</td>
<td>(ac)</td>
</tr>
</tbody>
</table>

- \(ac\) and \(bd\) are always positive when used
- \(ad\) and \(bc\) are always negative when used
- We know in advance the rounding direction
  - No need to select it dynamically
Power to an integer: existing algorithm

T pow_dn(const T& x_, int pwr, Rounding& rnd) // x and pwr are positive
{
    T x = x_;  
    T y = (pwr & 1) ? x_ : 1;  
    pwr >>= 1;  
    while (pwr > 0) {
        x = rnd.mul_down(x, x);  
        if (pwr & 1) y = rnd.mul_down(x, y);  
        pwr >>= 1;  
    }
    return y;  
}

T pow_up(const T& x_, int pwr, Rounding& rnd) // x and pwr are positive
{
    [idem using mul_up]  
}

interval<T, Policies> pow(const interval<T, Policies>& x, int pwr)
{
    [...]  
    if (interval_lib::user::is_neg(x.upper())) { // [-2,-1]  
        T yl = pow_dn(-x.upper(), pwr, rnd);  
        T yu = pow_up(-x.lower(), pwr, rnd);  
        if (pwr & 1) // [-2,-1]^1  
            return I(-yu, -yl, true);  
        else // [-2,-1]^2  
            return I(yl, yu, true);  
    } else if (interval_lib::user::is_neg(x.lower())) { // [-1,1]  
        if (pwr & 1) { // [-1,1]^1  
            return I(-pow_up(-x.lower(), pwr, rnd), pow_up(x.upper(), pwr, rnd), true);  
        } else { // [-1,1]^2  
            return I(0, pow_up(::max(-x.lower(), x.upper()), pwr, rnd), true);  
        }
    } else { // [1,2]  
        return I(pow_dn(x.lower(), pwr, rnd), pow_up(x.upper(), pwr, rnd), true);  
    }
}
Improvements

- Exponent is constant: we can unroll the loop
  - Cuda 1.0 cannot do loop unrolling
  - Cuda 1.1 can, but without constant propagation and dead code removal
  - We use template metaprogramming instead

- We can compute \texttt{pow\_up} from \texttt{pow\_down}
  - Add a bound on the error: multiply by \((1+n \text{ EPS})\)
Performance results

- Cuda on a nVidia GeForce 8500 GT
  - Throughput in cycles/warp

<table>
<thead>
<tr>
<th>Operation</th>
<th>Original</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>Multiplication</td>
<td>33 – 55 (*)</td>
<td>36</td>
</tr>
<tr>
<td>$x^5$</td>
<td>141 – 148 (*)</td>
<td>23</td>
</tr>
</tbody>
</table>

* With uniform branching
Application to ray-tracing

- Xeon 3GHz vs 8800 GTX

<table>
<thead>
<tr>
<th>Surface</th>
<th>CPU (s)</th>
<th>GPU (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sphere</td>
<td>300</td>
<td>2</td>
</tr>
<tr>
<td>Kusner-Schmitt</td>
<td>720</td>
<td>2</td>
</tr>
<tr>
<td>Tangle</td>
<td>900</td>
<td>3</td>
</tr>
<tr>
<td>Gumdrop Torus</td>
<td>1080</td>
<td>3</td>
</tr>
</tbody>
</table>
Conclusions

- Conference paper preprint
  Sylvain Collange, Jorge Flórez, David Defour. A GPU interval library based on Boost Interval. hal-00263670. 2008

- Good algorithms on CPU are not always good choices for the GPU
  - Need to take GPU specificities into account
Earlier GPUs

- Non-compliant rounding for basic operations
- nVidia GeForce 7, multiplication
  - Faithful rounding
- AMD ATI Radeon X1x00, multiplication
  - Not toward zero, not faithful
Memory access issues

- Memory access units are shared among a SIMD block
- Fast memory accesses must obey specific patterns
  - Broadcast: all threads access the same address
  - Coalescing: all threads access consecutive addresses
- In Cuda
  - On-chip shared memory accessible in both modes
  - On-chip constant memory in broadcast mode
  - Off-chip global memory with coalescing capabilities