# Faster and Tighter Evaluation of a Polynomial Range Through SIMD Interval Arithmetic

Frédéric Goualard

Université de Nantes
Laboratoire d'Informatique de Nantes-Atlantique, UMR CNRS 6241

# Interval Evaluation of Polynomials

Range of $p(x) = 10x - 1.5x^2 - 3x^3 + x^4$ for $x \in [-3, 7]$?

$$[a, b] + [c, d] = [\mathrm{dn}\{a + c\}, \mathrm{up}\{b + d\}]$$

$$[a, b] - [c, d] = [\mathrm{dn}\{a - d\}, \mathrm{up}\{b - c\}]$$

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

$[a, b]^n = \ldots$ *(consider sign of* $[a, b]$ *and parity of* $n$*)*

With:

$(a, b, c, d) \in \mathbb{F}^4$ : Floating-point numbers

$\mathrm{dn}\{x\}$ : Real $x$ rounded to float towards $-\infty$

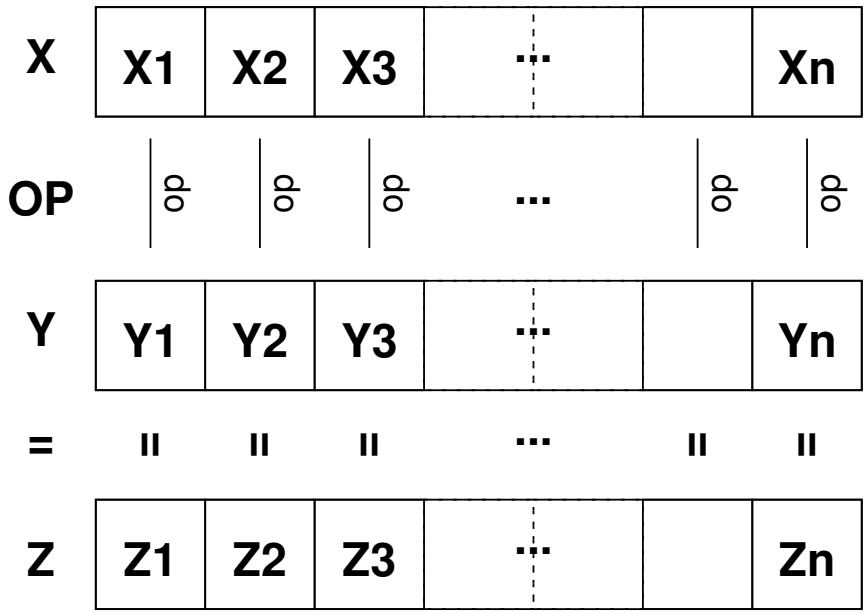
$\mathrm{up}\{x\}$ : Real $x$ rounded to float towards $+\infty$

Usual implementation: bounds of the result computed sequentially

# Parallel Interval Evaluation

[H. Sutter] *"The free lunch is over"*

- *S*ingle *I*nstruction *M*ultiple *D*ata (SIMD) model:
  - "Easy" and ubiquitous (GPU, SSE, ...)

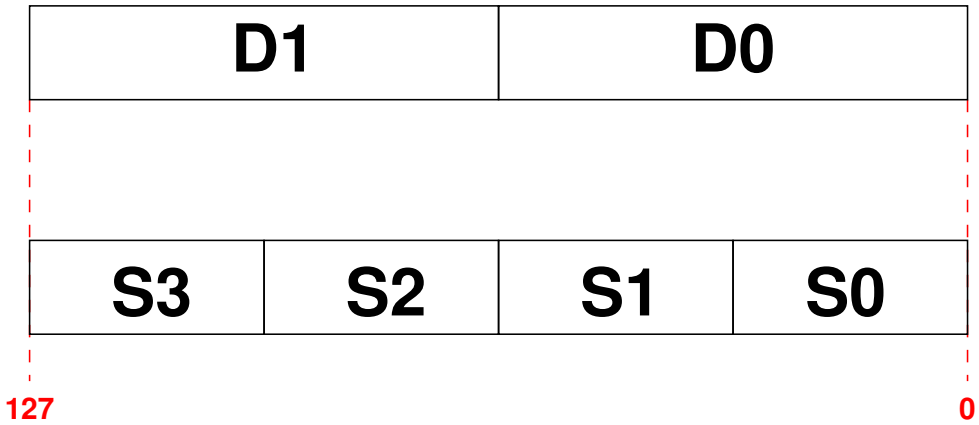



☺ Performing $n$ operations at (almost) the cost of one

☹ ~~Flow-control intensive algorithms~~

# SSE2

- Short-Vector instructions supported by ix86 processors since 2001 (Pentium IV and above)
- Instruction set on 128 bits registers

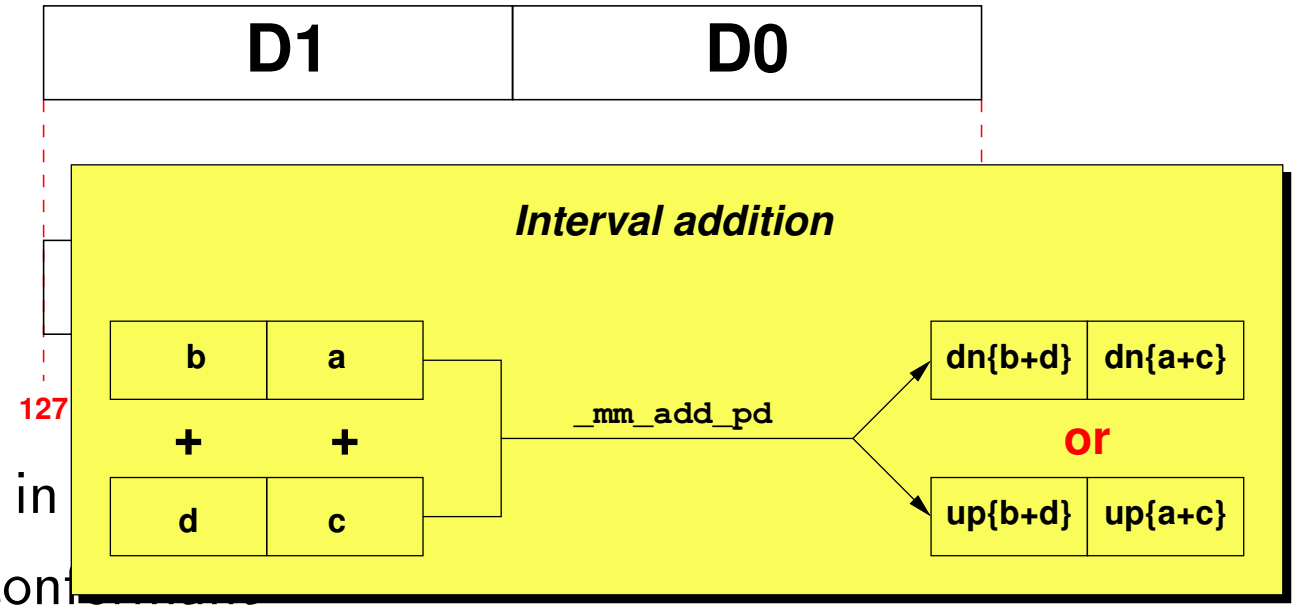



- Operations in parallel on 4 SP or 2 DP numbers
- IEEE 754 conformant
  (rounding direction switch independent of FPU's)

  Rounding direction set *per instruction*

# SSE2

- Short-Vector instructions supported by ix86 processors since 2001 (Pentium IV and above)
- Instruction set on 128 bits registers



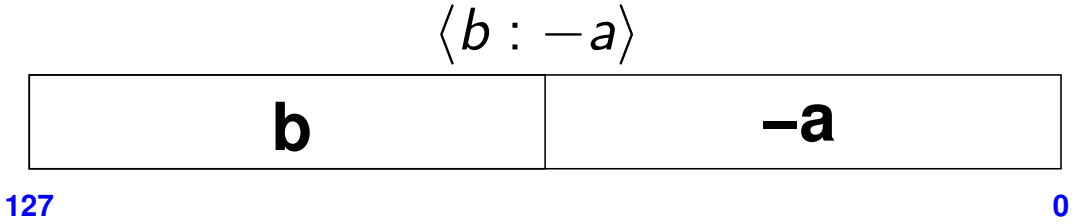

- Operations in
- IEEE 754 cont
  (rounding direction switch independent of FPU's)
  - Rounding direction set *per instruction*

# Practical SSE2 Implementation

- No rounding direction switch: *"opposite trick"*

$$\begin{aligned} \mathrm{dn}\{a+c\} &= -\,\mathrm{up}\{-a-c\} \\ \mathrm{dn}\{a-c\} &= -\,\mathrm{up}\{c-a\} \\ \mathrm{dn}\{ac\} &= -\,\mathrm{up}\{(-a)c\} \end{aligned}$$

- Rounding permanently set to $+\infty$
- SSE2 register layout for $[a, b]$:



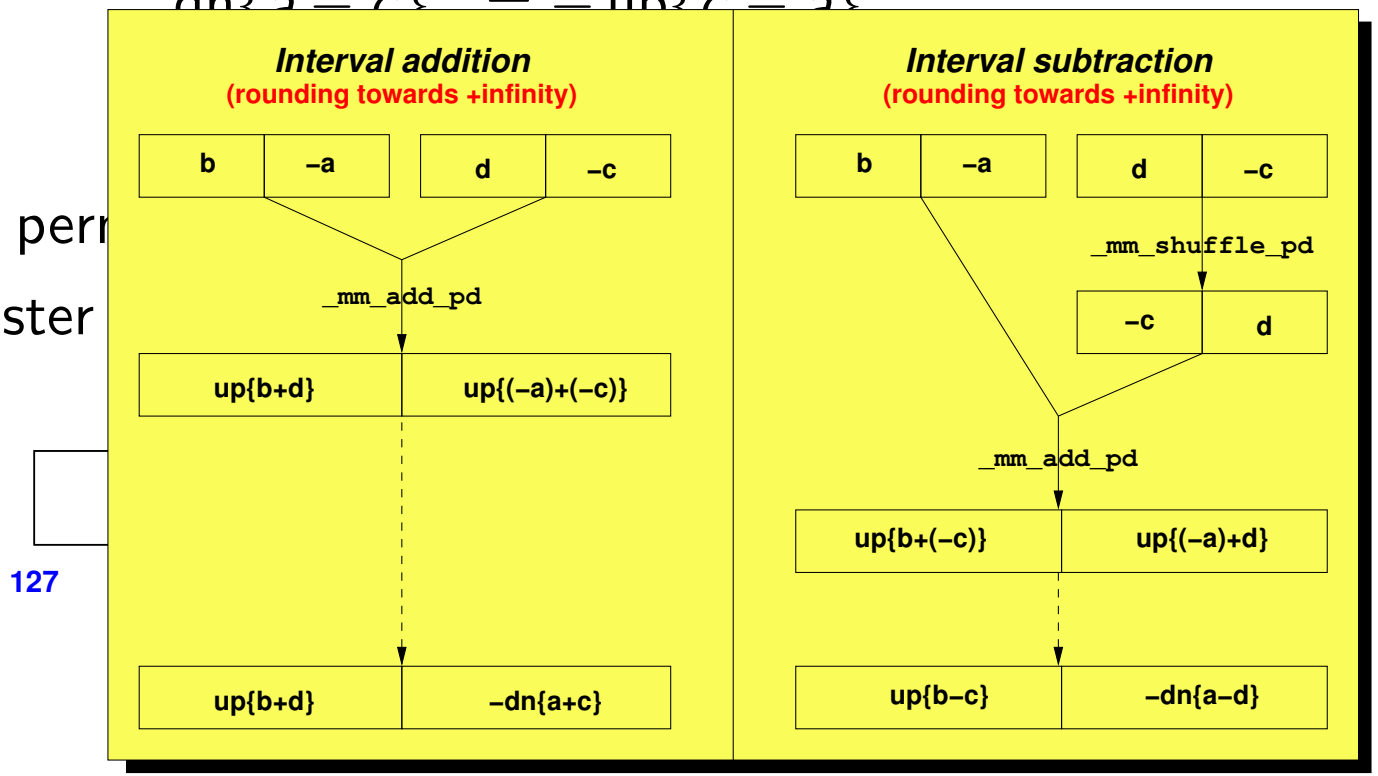
# Practical SSE2 Implementation

- No rounding direction switch: *"opposite trick"*

$$\mathrm{dn}\{a+c\} = -\,\mathrm{up}\{-a-c\}$$

$$\mathrm{dn}\{a-c\} = -\,\mathrm{up}\{c-a\}$$

- Rounding perr
- SSE2 register




127

# SSE2 Interval Multiplication

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

- Brute-force: compute all 8 products (4 SSE2 mult.)
- [Wolff v. Gudenberg, 2000]: 9 cases on bound signs (at most 4 products, i.e. 2 SSE2 mult.)
- [Lambov, 2006]: branchless (always 4 products, i.e. 2 SSE2 mult.)

# SSE2 Brute-Force Multiplication

```
interval interval::operator*(const interval& I1, const interval& I2)
{
  // Saving SSE control word and setting rounding direction to +∞
  // [...]
  __m128d r1  = _mm_shuffle_pd(I1.bounds,I1.bounds,1); // r1  = <-a, b>
  __m128d r2  = _mm_xor_pd(r1,lbrbsignmask());         // r2  = <a, -b>
  __m128d r3  = _mm_xor_pd(I2.bounds,lbsignmask());    // r3  = <d, c>
  __m128d r4  = _mm_shuffle_pd(r3,r3,1);               // r4  = <c, d>
  __m128d r5  = _mm_unpacklo_pd(I2.bounds,I2.bounds);  // r5  = <-c, -c>
  __m128d r6  = _mm_unpackhi_pd(I2.bounds,I2.bounds);  // r6  = <d, d>
  __m128d r7  = _mm_mul_pd(I1.bounds,r3);              // r7  = <bd, (-a)c>
  __m128d r8  = _mm_mul_pd(r1,r5);                     // r8  = <ac, b(-c)>
  __m128d r9  = _mm_mul_pd(I1.bounds,r4);              // r9  = <bc, (-a)d>
  __m128d r10 = _mm_mul_pd(r2,r6);                     // r10 = <ad,(-b)d>
  __m128d r11 = _mm_max_pd(r7, r8);
  __m128d r12 = _mm_max_pd(r9, r10);
  __m128d r13 = _mm_max_pd(r11, r12);

  // Restoring SSE control word
  // [...]
  return interval(r13);
}
```

- C++ and Intel intrinsic instructions

# Interval multiplication (Cases on bounds)

Considering bound signs to reduce multiplications:

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

9 cases depending on signs

# Interval multiplication (Cases on bounds)

Considering bound signs to reduce multiplications:

| $\langle b : -a\rangle \times \langle d : -c\rangle$ | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| 00 | $\langle bd : a(-c)\rangle$ | $\langle bd : b(-c)\rangle$ | $\langle 0 : -0\rangle$ | $\langle ad : b(-c)\rangle$ |
| 01 | $\langle bd : (-a)d\rangle$ | $\langle \max((-a)(-c), bd) : \max((a)d, b(-c))\rangle$ | $\langle 0 : -0\rangle$ | $\langle (-a)(-c) : b(-c)\rangle$ |
| 10 | $\langle 0 : -0\rangle$ | $\langle 0 : -0\rangle$ | $\langle 0 : -0\rangle$ | $\langle 0 : -0\rangle$ |
| 11 | $\langle bc : (-a)d\rangle$ | $\langle (-a)(-c) : (-a)d\rangle$ | $\langle 0 : -0\rangle$ | $\langle (-a)(-c) : (-b)d\rangle$ |

16 cases depending on sign bits for $a$, $b$, $c$, and $d$

# Lambov's algorithm

- Branchless code at the cost of more multiplications

```
interval interval::operator*(const interval& I1, const interval& I2) {
  // Saving SSE control word and setting rounding direction to +∞
  // [...]
  __m128d A = _mm_xor_pd(I1.bounds,interval::lbsignmask()); // < b, a >
  __m128d B = _mm_shuffle_pd(I2.bounds,I2.bounds,1);        // < -c, d >
  __m128d C = _mm_cmplt_pd(A,interval::m128_zero());        // < b<0, a<0 >
  __m128d D = _mm_xor_pd(B,interval::lbrbsignmask());       // < c, -d >
  __m128d G = _mm_shuffle_pd(C,C,1);                        // < a<0, b<0 >
  __m128d E = _mm_and_pd(C,D);              // < (b<0)?c:0, (a<0)?-d:0 >
  __m128d F = _mm_andnot_pd(C,I2.bounds);   // < (b>=0)?d:0, (a>=0)?-c:0 >
  __m128d K = _mm_andnot_pd(G,I2.bounds);   // < (a>=0)?d:0, (b>=0)?-c:0 >
  __m128d I = _mm_and_pd(D,G);              // < (a<0)?c:0, (b<0)?-d:0 >
  __m128d H = _mm_or_pd(E,F);               // < (b<0)?c:d, (a<0)?-d:-c >
  __m128d L = _mm_shuffle_pd(A,A,1);        // < a, b >
  __m128d M = _mm_or_pd(I,K);               // < (a<0)?c:d, (b<0)?-d:-c >
  __m128d J = _mm_mul_pd(A,H);              // < (b<0)?bc:bd, (a<0)?a(-d):a(-c) >
  __m128d N = _mm_mul_pd(L,M);              // < (a<0)?ac:ad, (b<0)?b(-d):b(-c) >
  __m128d P = _mm_max_pd(J,N);              // < (b<0)?max(ac,bc)
                                            //       :(a<0)?max(ac,bd)
                                            //             :max(ad,bd),
                                            //   (b<0)?max(b(-d),a(-d))
                                            //       :(a<0)?max(b(-c),a(-d))
                                            //             :max(b(-c),a(-c)) >
  // Restoring SSE control word
  // [...]
  return interval(P);
}
```

# Performances of Multiplication Algorithms

- Tests on 10,000,000 pairs of random intervals (repeated 10 times)
- Intel Core2 Duo T5600 1.83GHz, Linux 32bits

| Algorithm | Time (in s.) |
|---|---|
| Brute force | 40.2 |
| Lambov | 29.9 |
| Test on 9 cases | 32.9 |
| Test on 16 cases | 24.5 |

# Interval Exponentiation $[a, b]^n$ (1)

- Left-to-right binary exponentiation

```
1 function y = powdn(r ∈ F+, n ∈ N*):   % n = (b_t b_{t-1} ... b_1 b_0)_2
2     y ← −r
3     for i in t − 1, ..., 0:
4         y ← up{y × (−y)}
5         if b_i = 1:
6             y ← up{y × r}
7     return y   % y = − dn{r^n}
```

```
1 function y = powup(r ∈ F+, n ∈ N*):   % n = (b_t b_{t-1} ... b_1 b_0)_2
2     y ← r
3     for i in t − 1, ..., 0:
4         y ← up{y × y}
5         if b_i = 1:
6             y ← up{y × r}
7     return y   % y = up{r^n}
```

SSE2 implementation:

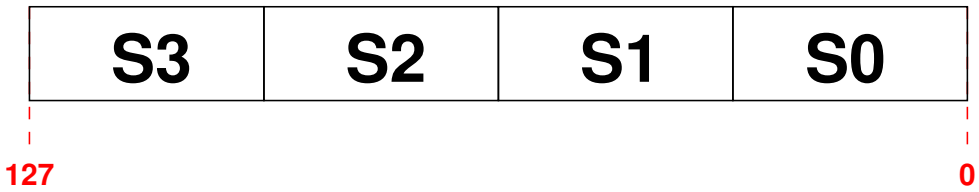
- **powdnup**$(\langle b : a \rangle, n) = \langle \mathrm{up}\{b^n\} : -\mathrm{dn}\{a^n\} \rangle$
- **powupup**$(\langle b : a \rangle, n) = \langle \mathrm{up}\{b^n\} : \mathrm{up}\{a^n\} \rangle$

# Interval Exponentiation $[a, b]^n$ (2)

| $[a, b]$<br>$\langle b : -a \rangle$ | even($n$) | odd($n$) |
|---|---|---|
| $b \leqslant 0$ | $[b^n, a^n]$<br>**powdnup**$(\langle -a : b \rangle, n)$ | $[a^n, b^n]$<br>$-$**powdnup**$(\langle -a : b \rangle, n)$ |
| $a \leqslant 0 \leqslant b$ | $[0, \max(a^n, b^n)]$<br>$\langle d : c \rangle =$ **powupup**$(\langle b : -a \rangle, n)$<br>$\rightsquigarrow \langle \max(c, d) : 0 \rangle$ | $[a^n, b^n]$<br>**powupup**$(\langle b : -a \rangle, n)$ |
| $0 \leqslant a$ | $[a^n, b^n]$<br>**powdnup**$(\langle b : -a \rangle, n)$ | $[a^n, b^n]$<br>**powdnup**$(\langle b : -a \rangle, n)$ |

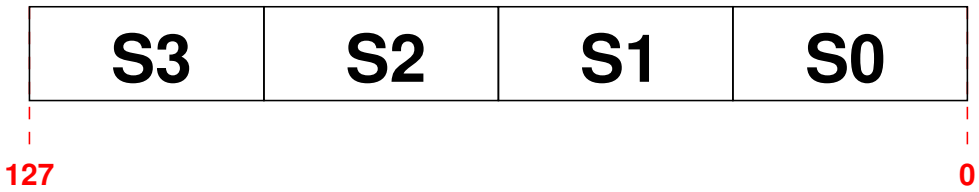
- $i$-th power of $a$ and $b$ computed in parallel

# Even more parallelism



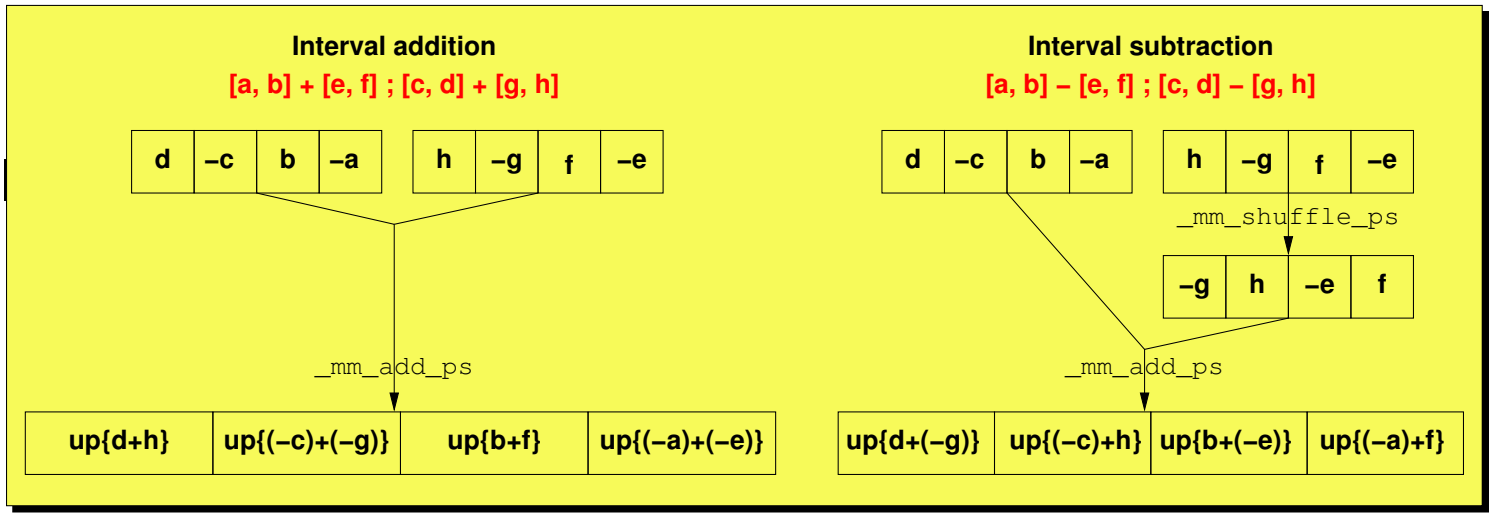

- ☺ Two interval operations in parallel
- ☹ Single precision bounds

# Even more parallelism




☺ Two

☹ Sing



# Interval multiplication

Multiplications in parallel:

$$[a, b] \times [e, f] \quad || \quad [c, d] \times [g, h]$$

- Brute-force (very convenient: almost same code as before)
  4 SSE2 multiplications
- Cases on bounds signs (8 bits): 256 cases
  At most 2 SSE2 multiplications

# Interval multiplication

Multiplications in parallel:

$$[a, b] \times [e, f] \quad || \quad [c, d] \times [g, h]$$

- Brute-force (very convenient: almost same code as before)
  4 SSE2 multiplications
- Cases on bounds signs (8 bits): 256 cases
  At most 2 SSE2 multiplications

| | Brute force | 256 cases |
|---|---|---|
| Multiplication | 20.9 | 10.8 |

Times in seconds, 10,000,000 multiplications

# Interval Exponentiation $[a, b]^n \,||\, [c, d]^n$

- 16 cases depending on sign bits of bounds
- Only three basic cases:
  - **powdnupdnup**$(\langle d : c : b : a \rangle, n) =$
    $\langle \mathrm{up}\{d^n\} : -\mathrm{dn}\{c^n\} : \mathrm{up}\{b^n\} : -\mathrm{dn}\{a^n\} \rangle$
  - **powupupupup**$(\langle d : c : b : a \rangle, n) =$
    $\langle \mathrm{up}\{d^n\} : \mathrm{up}\{c^n\} : \mathrm{up}\{b^n\} : \mathrm{up}\{a^n\} \rangle$
  - **powupupdnup**$(\langle d : c : b : a \rangle, n) =$
    $\langle \mathrm{up}\{d^n\} : \mathrm{up}\{c^n\} : \mathrm{up}\{b^n\} : -\mathrm{dn}\{a^n\} \rangle$

# Benchmarking

- Intel Core 2 Duo, Ubuntu 32 bits
- $10,000,000$ operations on random intervals repeated 10 times
- Exponentiation: $n \in \{1, \ldots, 100\}$

| Package | `add` | `sub` | `mul` | `pow` |
|---|---|---|---|---|
| Bias 2.0.8 [DP] | 46.2 | 46.2 | 59.0 | 691.4 |
| boost 1.43.0 [DP] | 29.6 | 29.7 | 44.8 | 170.1 |
| fi_lib++ 2.0 [DP] | 58.3 | 58.3 | 89.2 | 206.1 |
| gaol 4.0 (2d) [DP] | 15.2 | 14.5 | 24.1 | 60.1 |
| gaol 4.0 (4f) [SP] | 7.2 | 7.1 | 13.4 | 38.9 |

- Rounding direction reset after each interval operation for all libraries

# Applications

Computation of two interval operations in parallel:

- Tighter evaluation of polynomials:

$$f([a,b]) = f([a,c]) \cup f([c,b]), \quad c \in [a,b]$$

with $f([a,c])$ and $f([c,b])$ evaluated in parallel

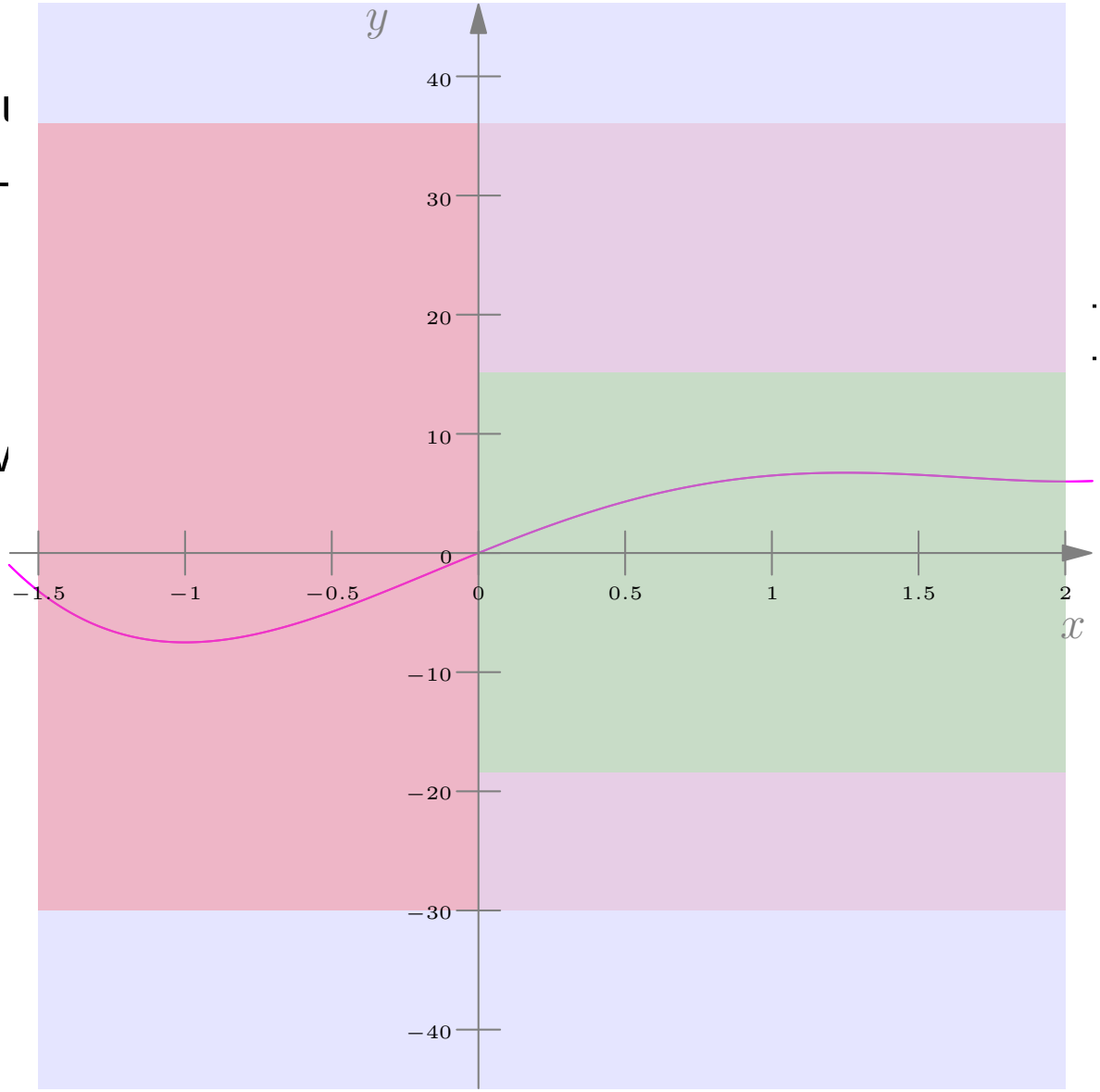
# Applications

Compı

- T

$[a, b]$

w



# Applications

Computation of two interval operations in parallel:

- Tighter evaluation of polynomials:

$$f([a,b]) = f([a,c]) \cup f([c,b]), \quad c \in [a,b]$$

  with $f([a,c])$ and $f([c,b])$ evaluated in parallel

- Branch and prune:

$$0 \in f([a,b])?$$

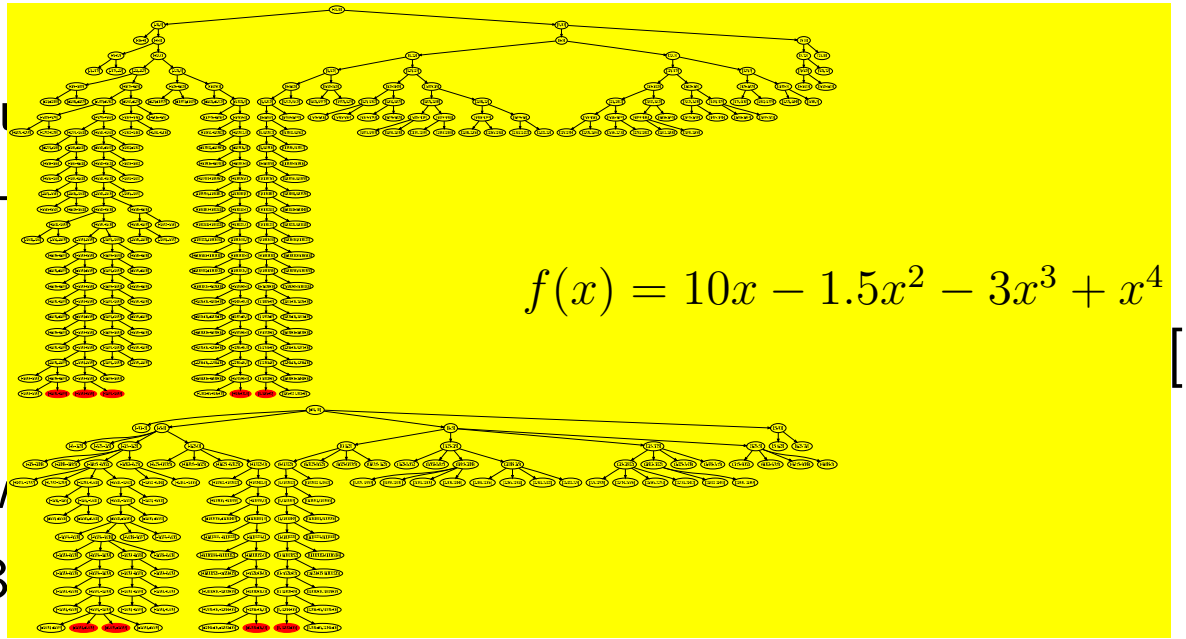
$$0 \in f([a,c])? \,||\, 0 \in f([c,b])?, \quad c \in [a,b]$$

  Validate solution nodes with double precision interval arithmetic

# Applications

Compu

- T



$$f(x) = 10x - 1.5x^2 - 3x^3 + x^4$$

$[a, b]$

w

- B

$$0 \in f([a, b])?$$

$$0 \in f([a, c])? \,||\, 0 \in f([c, b])?, \quad c \in [a, b]$$

Validate solution nodes with double precision interval arithmetic

# Applications

Computation of two interval operations in parallel:

- Tighter evaluation of polynomials:

$$f([a,b]) = f([a,c]) \cup f([c,b]), \quad c \in [a,b]$$

  with $f([a,c])$ and $f([c,b])$ evaluated in parallel

- Branch and prune:

$$0 \in f([a,b])?$$

$$0 \in f([a,c])? \,||\, 0 \in f([c,b])?, \quad c \in [a,b]$$

  Validate solution nodes with double precision interval arithmetic

- [Goualard & Goldsztejn, PDCAT 2008] Root finding by parallel shaving

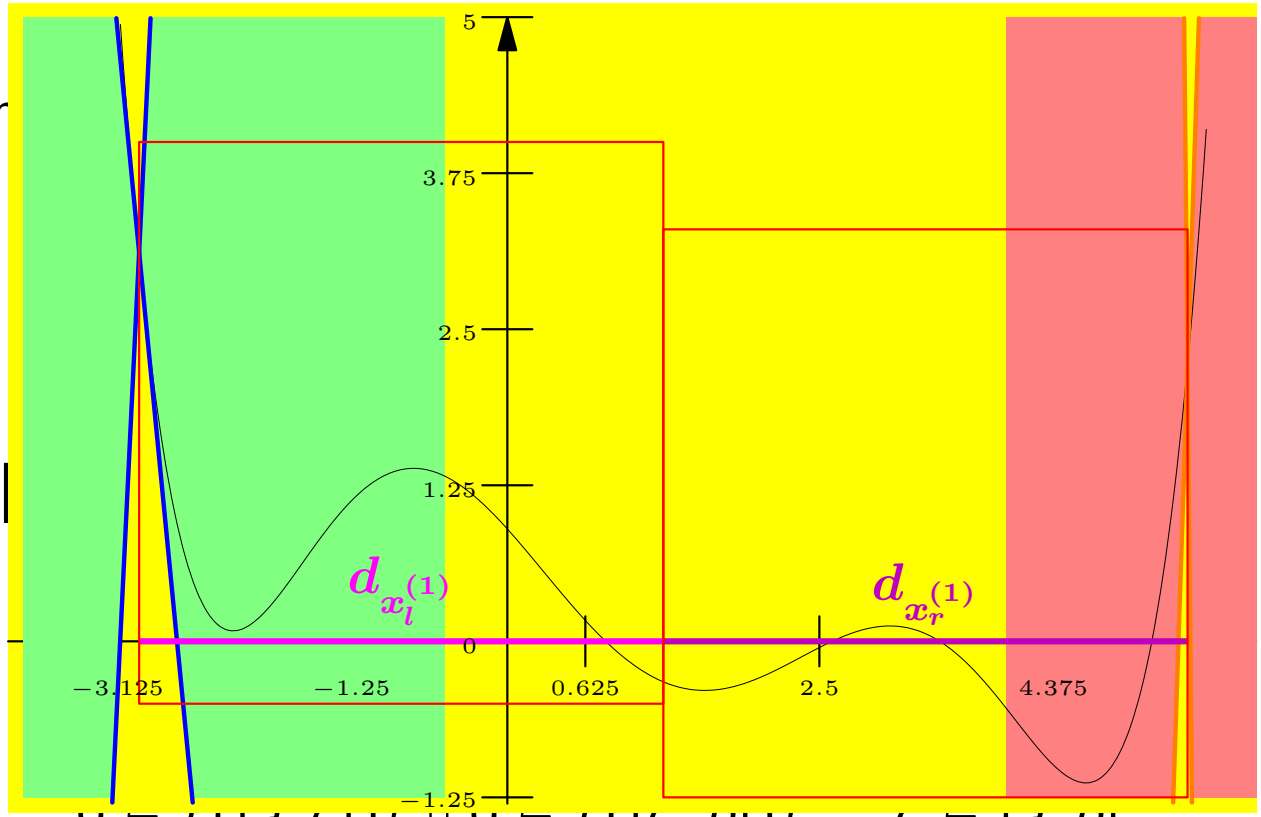
# Applications

Computation

- Tighter

with $f($

- Branch




$0 \in f([a, c])\,?\;||\;0 \in f([c, b])\,?,\quad c \in [a, b]$

Validate solution nodes with double precision interval arithmetic

- [Goualard & Goldsztejn, PDCAT 2008] Root finding by parallel shaving

# Conclusion

- Good performances with SSE2 instructions in double precision
- Very good performances in single precision
  - May serve as a fast pre-processing step followed by double precision computation
- Intel AVX: single precision algorithms should be usable directly for very good performances in double precision
- Interval division already implemented
- Future work: other operators

# Faster and Tighter Evaluation of a Polynomial Range Through SIMD Interval Arithmetic

Frédéric Goualard

Université de Nantes
Laboratoire d'Informatique de Nantes-Atlantique, UMR CNRS 6241