# SIPE: Small Integer Plus Exponent

Vincent LEFÈVRE

AriC, INRIA Grenoble – Rhône-Alpes / LIP, ENS-Lyon

Arith 21, Austin, Texas, USA, 2013-04-09

# Introduction: Why SIPE?

All started with floating-point algorithms in radix 2, assuming correct rounding to nearest:

- TwoSum: to compute a rounded sum $x_h = \circ(a + b)$ and the error term $x_\ell$;
- DblMult[1]: accurate double-FP multiplication $(a_h, a_\ell) \times (b_h, b_\ell)$;
- Kahan's algorithm to compute the discriminant $b^2 - ac$ accurately.

Valid with restrictions on the inputs, e.g.:

- no special datums (NaN, infinities);
- no non-zero tiny or huge values in order to avoid exceptions due to the bounded exponent range (overflow/underflow).

Questions about such algorithms: Correctness? Error bound? Optimality? ...

The answer may be difficult to find, and exhaustive tests in some domain may help to solve the problem. We need a tool for that...

[1] See *Computing Correctly Rounded Integer Powers in Floating-Point Arithmetic*, by P. Kornerup, Ch. Lauter, V. Lefèvre, N. Louvet, and J.-M. Muller, in TOMS, 2010.

# Introduction: Testing Floating-Point Algorithms

Exhaustive tests (in some domain) $\rightarrow$ proofs or reachable error bounds.

Drawbacks:

- valid only for the considered FP system (the chosen precision);
- and this may be possible only in very low precisions.

Still useful:

- try to generalize the results $\rightarrow$ conjectured error bounds or other properties for higher precisions;
- possibly leading to proofs;
- or counter-examples (in case of errors in pen-and-paper proofs).

**No need to take into account special data and exceptions** (or this could be optional if this doesn't slow down the generic cases).

# Introduction: Tools Existing Before SIPE

All of them in radix 2.

- GNU MPFR: correct rounding in any precision $p \geq 2$.
  OK concerning the behavior, but
  - very generic: not specifically optimized for a given precision;
  - we had to take into account that different precisions can even be mixed;
  - overhead due to exception handling and special data.

  $\rightarrow$ Cannot be as fast as specific software ignoring exceptions.

- GCC's *sreal* internal library. But
  - no support for negative numbers;
  - rounding is roundTiesToAway: to nearest, but not the usual even-rounding rule for the halfway cases (rounded away from zero);
  - the precision is more or less hard-coded;
  - overflow detection, unnecessary in our context;
  - no FMA support (needed for DblMult);
  - apparently, not very optimized.

# SIPE: Small Integer Plus Exponent

- Idea based on DPE (Double Plus Exponent) by Paul Zimmermann and Patrick Pélissier: a header file (`.h`) providing the arithmetic, where a finite floating-point number is represented by a pair of integers $(M, E)$, with the value $M \cdot 2^E$.
- Focus on efficiency:
  - code written in C (for portability), with some GCC extensions;
  - exceptions (in particular overflows/underflows) are ignored, and unsupported inputs are not detected;
  - restriction: the precision must be small enough to have a simple and fast implementation, without taking integer overflow cases into account. The maximal precision is deduced from the implementation (and the platform).
- Currently only the rounding attribute roundTiesToEven (rounding to nearest with the even rounding rule) is implemented.

# SIPE: Encoding

**Chosen encoding:**

- Structure of two native signed integers representing the pair $(M, E)$.
- If $M \neq 0$ (i.e. $x \neq 0$), the representation is normalized: $2^{p-1} \leq |M| \leq 2^p - 1$.
- If $M = 0$, then we require $E = 0$ (even though its real value doesn't matter, we need to avoid integer overflows, e.g. when two exponents are added).

**Bound on the precision:**

| FMA/FMS | 32-bit integers | 64-bit integers |
|---|---|---|
| No | 15 | 31 |
| Yes | 10 | 20 |

Alternative encodings that could have been considered:

- packed in a single integer or separate significand sign;
- fixed-point representation ($\rightarrow$ limited exponent range, unpractical);
- native floating-point format: native operations + Veltkamp's splitting, with double-rounding effect detection (second Veltkamp's splitting?)... But this effect cannot occur for $+$, $-$ and $\times$ with small enough $p$!

# SIPE: Implementation of Some Simple Operations

```
typedef struct { sipe_int_t i; sipe_exp_t e; } sipe_t;

static inline sipe_t sipe_neg (sipe_t x, int prec)
{ return (sipe_t) { - x.i, x.e }; }

static inline sipe_t sipe_set_si (sipe_int_t x, int prec)
{ sipe_t r = { x, 0 };
  SIPE_ROUND (r, prec);
  return r; }

static inline sipe_t sipe_mul (sipe_t x, sipe_t y, int prec)
{ sipe_t r;
  r.i = x.i * y.i;
  r.e = x.e + y.e;
  SIPE_ROUND (r, prec);
  return r; }
```

# SIPE: Implementation of Addition and Subtraction

```
#define SIPE_DEFADDSUB(OP,ADD,OPS)                                  \
  static inline sipe_t sipe_##OP (sipe_t x, sipe_t y, int prec) \
  { sipe_exp_t delta = x.e - y.e;                               \
    sipe_t r;                                                   \
    if (SIPE_UNLIKELY (x.i == 0))                               \
      return (ADD) ? y : (sipe_t) { - y.i, y.e };               \
    if (SIPE_UNLIKELY (y.i == 0) || delta > prec + 1)           \
      return x;                                                 \
    if (delta < - (prec + 1))                                   \
      return (ADD) ? y : (sipe_t) { - y.i, y.e };               \
    r = delta < 0 ?                                             \
      ((sipe_t) { (x.i) OPS (y.i << - delta), x.e }) :          \
      ((sipe_t) { (x.i << delta) OPS (y.i),   y.e });           \
    SIPE_ROUND (r, prec);                                       \
    return r; }

SIPE_DEFADDSUB(add,1,+)
SIPE_DEFADDSUB(sub,0,-)
```

# SIPE: Provided Functions

Header file `sipe.h` providing:

- a macro `SIPE_ROUND(X,PREC)`, to round and normalize any pair $(i, e)$;
- initialization: via `SIPE_ROUND` or `sipe_set_si`;
- `sipe_neg`, `sipe_add`, `sipe_sub`, `sipe_add_si`, `sipe_sub_si`;
- `sipe_nextabove` and `sipe_nextbelow`;
- `sipe_mul`, `sipe_mul_si`, `SIPE_2MUL`;
- `sipe_fma` and `sipe_fms` (optional, see slide 6);
- `sipe_eq`, `sipe_ne`, `sipe_le`, `sipe_lt`, `sipe_ge`, `sipe_gt`;
- `sipe_min`, `sipe_max`, `sipe_minmag`, `sipe_maxmag`, `sipe_cmpmag`;
- `sipe_outbin`, `sipe_to_int`, `sipe_to_mpz`.

**New (2013-04-07/08):**
Second implementation, using the native floating-point encoding.
→ All the above functions except `sipe_fma` and `sipe_fms`.

# Example: Minimality of TwoSum in Any Precision

Based on the article *On the computation of correctly-rounded sums*, by Peter Kornerup, Vincent Lefèvre, Nicolas Louvet and Jean-Michel Muller, IEEE Transactions on Computers, 2012.

Full version on `http://hal.inria.fr/inria-00475279` [RR-7262 (2010)].

**Algorithm TwoSum***

$$
\begin{aligned}
s &= RN(a+b) \\
b' &= RN(s-a) \\
a' &= RN(s-b') \\
\delta_b &= RN(b-b') \\
\delta_a &= RN(a-a') \\
t &= RN(\delta_a+\delta_b)
\end{aligned}
$$

* due to Knuth and Møller.

- Floating-point system in radix 2.
- Correct rounding in rounding to nearest.
- Two finite floating-point numbers $a$ and $b$.

→ Assuming no overflows, this algorithm computes two floating-point numbers $s$ and $t$ such that:

$$s = RN(a+b) \quad \text{and} \quad s+t = a+b.$$

**Is this algorithm minimal (number of operations + and −, and depth of the computation DAG) in any precision $p \geq 2$?**

# Example: Minimality of TwoSum in Any Precision [2]

The idea: choose the pairs of inputs in some form so that one can prove that a counter-example in one precision yields a counter-example in all (large enough) precisions. Choices after testing various pairs, where $\uparrow x$ denotes nextUp($x$), i.e. the least floating-point number that compares greater than $x$:

$$\begin{aligned} a_1 &= \uparrow 8 & b_1 &= \uparrow^3 1 \\ a_2 &= \uparrow^5 1 & b_2 &= \uparrow 8 \\ a_3 &= 3 & b_3 &= \uparrow 3 \end{aligned}$$

In precision $p \geq 4$, this gives, where $\varepsilon = \mathrm{ulp}(1) = 2^{1-p}$:

$$\begin{aligned} a_1 &= 8 + 8\,\varepsilon & b_1 &= 1 + 3\,\varepsilon \\ a_2 &= 1 + 5\,\varepsilon & b_2 &= 8 + 8\,\varepsilon \\ a_3 &= 3 & b_3 &= 3 + 2\,\varepsilon \end{aligned}$$

Precisions 2 to 12 are tested. Results in precisions $p \geq 13$ can be deduced from the results in precision 12.

# Gain by Using SIPE Instead of GNU MPFR?

Expected gain by using SIPE instead of GNU MPFR?

Timing of individual operations: could be interesting information, but in practice, one needs to consider the whole program.

Indeed, in real-world tests: need to process each SIPE final result, and this may take time.

For the proof of minimality (optimality) of TwoSum: rather fast.

- Pre-computation step: generation of all the algorithms (DAG's).
- For each SIPE final result: 1 to 4 comparisons with *constant* values.

# Gain by Using SIPE Instead of GNU MPFR? [2]

For the computation of error bounds, for each input:

1. Compute the FP result with SIPE. High speed-up here.
2. Compute the exact value or a good approximation.
3. Compare the results. For a relative error, needs a division.

One may think that (2) and (3), which cannot use SIPE, would take most of the time, so that the speed-up would remain limited. However...

- Case of an exhaustive search: if the function is numerically regular enough, the exact value might be determined very quickly from the previous one, like in the search for the hardest-to-round cases.
- But here, in very low precision, this may not work well, as input intervals contain much fewer FP values per binade.
- For (3): division not always needed (filtering, low precision, consecutive inputs...).

# Timings

Example with "best" optimizations (Intel Xeon E5520, GCC 4.7.1 + LTO/PGO):

| | | timings (in seconds) | | | | ratios | |
|---|---|---|---|---|---|---|---|
| args | g | double | MPFR | SIPE/0 | SIPE/1 | S/D | M/S |
| 1 2 6 | – | 0.50 | 6.45 | 1.99 | 1.99 | 4.0 | 3.2 |
| 1 2 6 | 2 | 0.41 | 6.79 | 1.64 | 1.69 | 4.1 | 4.1 |
| 1 2 6 | 4 | 0.43 | 6.80 | 1.66 | 1.68 | 3.9 | 4.1 |
| 1 2 6 | 6 | 0.48 | 6.85 | 1.71 | 1.73 | 3.6 | 4.0 |
| 1 4 6 | – | 5.20 | 49.66 | 14.94 | 14.87 | 2.9 | 3.3 |
| 1 4 6 | 2 | 6.99 | 53.30 | 14.27 | 14.48 | 2.1 | 3.7 |
| 1 4 6 | 4 | 4.78 | 52.75 | 13.35 | 13.55 | 2.8 | 3.9 |
| 1 4 6 | 6 | 6.74 | 51.90 | 13.48 | 13.40 | 2.0 | 3.9 |
| 1 6 5 | – | 0.20 | 1.37 | 0.42 | 0.41 | 2.1 | 3.3 |
| 1 6 5 | 2 | 0.25 | 1.48 | 0.41 | 0.42 | 1.7 | 3.6 |
| 1 6 5 | 4 | 0.20 | 1.49 | 0.41 | 0.42 | 2.1 | 3.6 |
| 1 6 5 | 6 | 0.23 | 1.40 | 0.38 | 0.38 | 1.7 | 3.7 |

# Timings [2]

The above timings:

- For the proof of the minimality of TwoSum (number of operations), i.e. only add/sub are currently tested.
- Thus include the overhead for the input data generation and the tests of the results.
- Tests with other GCC versions and other machines (see article).

From all these tests, the use of SIPE is

- between 1.2 and 6 times as slow as the use of the `double` C floating-point type, i.e. for $p = 53$ (incomplete for the proof in precisions $p \leq 11$);
- between 2 and 6 times as fast as the use of MPFR for precision 12.

# Timings [3]

With the new version of SIPE on Intel Xeon E5520, GCC 4.7.2 (no LTO):

| | | timings (in seconds) | | | | | |
|---|---|---|---|---|---|---|---|
| args | g | double | MPFR | SIPE/0 | SIPE/1 | SIPE/D | SIPE/L |
| 1 2 6 | – | 0.54 | 8.88 | 2.02 | 2.04 | 0.53 | 0.92 |
| 1 2 6 | 2 | 0.40 | 8.78 | 1.69 | 1.72 | 0.54 | 0.82 |
| 1 2 6 | 4 | 0.38 | 8.83 | 1.84 | 1.86 | 0.50 | 0.85 |
| 1 2 6 | 6 | 0.44 | 9.01 | 1.86 | 1.84 | 0.48 | 0.89 |
| 1 4 6 | – | 5.19 | 64.44 | 14.85 | 14.67 | 5.61 | 12.18 |
| 1 4 6 | 2 | 7.92 | 67.49 | 14.57 | 14.50 | 8.42 | 12.45 |
| 1 4 6 | 4 | 6.52 | 65.78 | 15.64 | 16.05 | 7.13 | 11.73 |
| 1 4 6 | 6 | 7.00 | 65.84 | 15.20 | 15.40 | 7.08 | 12.99 |
| 1 6 5 | – | 0.19 | 1.73 | 0.41 | 0.40 | 0.20 | 0.40 |
| 1 6 5 | 2 | 0.31 | 1.94 | 0.43 | 0.41 | 0.32 | 0.42 |
| 1 6 5 | 4 | 0.28 | 1.89 | 0.48 | 0.50 | 0.28 | 0.40 |
| 1 6 5 | 6 | 0.27 | 1.76 | 0.45 | 0.45 | 0.26 | 0.44 |

# Conclusion

SIPE (now Sipe): a "library" whose purpose is to do simple operations in binary floating-point systems in very low precisions with correct rounding to nearest.

Web page: `http://www.vinc17.net/research/sipe/`

Future work:

- Other applications, e.g. minimal DblMult error bound.
  $\rightarrow$ Pen-and-paper proof (currently almost done for most cases).
  $\rightarrow$ New timings, where multiplication is now involved.
- Try the floating-point solution. **Done on 2013-04-08 except fma/fms.**

In the long term, support for:

- other operations (e.g. division, square root);
- directed rounding;
- decimal arithmetic.