A compensated dot product for vectors of floating-point expansions

Nicolas Louvet, UCB Lyon 1
Aric, LIP

RAIM 2018, 10th edition,
November 13, 2018

Current general purpose processors implement two IEEE 754 binary floating point arithmetics: Binary32 ($p = 24$) and Binary64 ($p = 53$).

The IEEE 754 also specifies Binary128 ($p = 113$); implemented in hardware by IBM: available on the z13 and on the POWER9 processor.

When more than 53 bits of precision are required, different software solutions exist:

- libraries based on integer arithmetic:
  - `__float128` in GCC for binary128.
  - the MPFR library [Lefèvre, Zimmermann 2017].
- libraries based on floating point expansions:
  - `dd_real` and `qd_real` in the `qd` library [Hida, Li, Bailey 2001],
  - `quad_float` in the NTL library (code from K. Briggs),
  - more recently : CAMPARY [Popescu 17].
- compensated algorithms: summation, dot products ([Kahan 65], [Møller 65], [Pichat 72], [Neumanier 74], [Priest 92], [Ogita-Rump-Oishi 05, 08], ...), polynomial evaluation ([Graillat-Langlois-L. 06], [Jiang-Barrio-Liao-Cheng 11]...)

How to trade off speed against a moderate loss of accuracy when computing with floating point expansions?

We do so by using compensation techniques, and by focusing on dot products.

Outline

Notation and assumptions

$\mathbb{F}$ denotes the set of binary floating-point numbers in precision p, with an unbounded exponent range (no over/underflow).

The floating point operations are rounded to the nearest even.
fl() indicates that the expression inside the parentheses is evaluated in f. p. arithmetic.

$u = 2^{-p}$ is the unit roundoff. Given $\mathrm{op} \in \{+, -, \times, /\}$, and $a, b \in \mathbb{F}$,

$$\mathrm{fl}(a \,\mathrm{op}\, b) = (1 + \varepsilon)(a \,\mathrm{op}\, b), \quad \text{with } |\varepsilon| \leqslant u.$$

Given $x, y \in \mathbb{R}^n$, we note $\langle x, y \rangle = \sum_{i=0}^{n} x_i y_i$.

$e_n = [1, \ldots, 1]^T \in \mathbb{R}^n$, and we will often write $\langle x, e_n \rangle$ for the sum of the elements in x.

Classical dot product algorithm

Let $x, y \in \mathbb{F}^n$ be two column vectors.
We assume a Fused Multiply-Add (FMA) instruction is available.

```
s = dotprod(x, y)
    s ← 0
    for i = 1 ... n − 1
        s ← fl(s + x_i y_i)
    return s
```

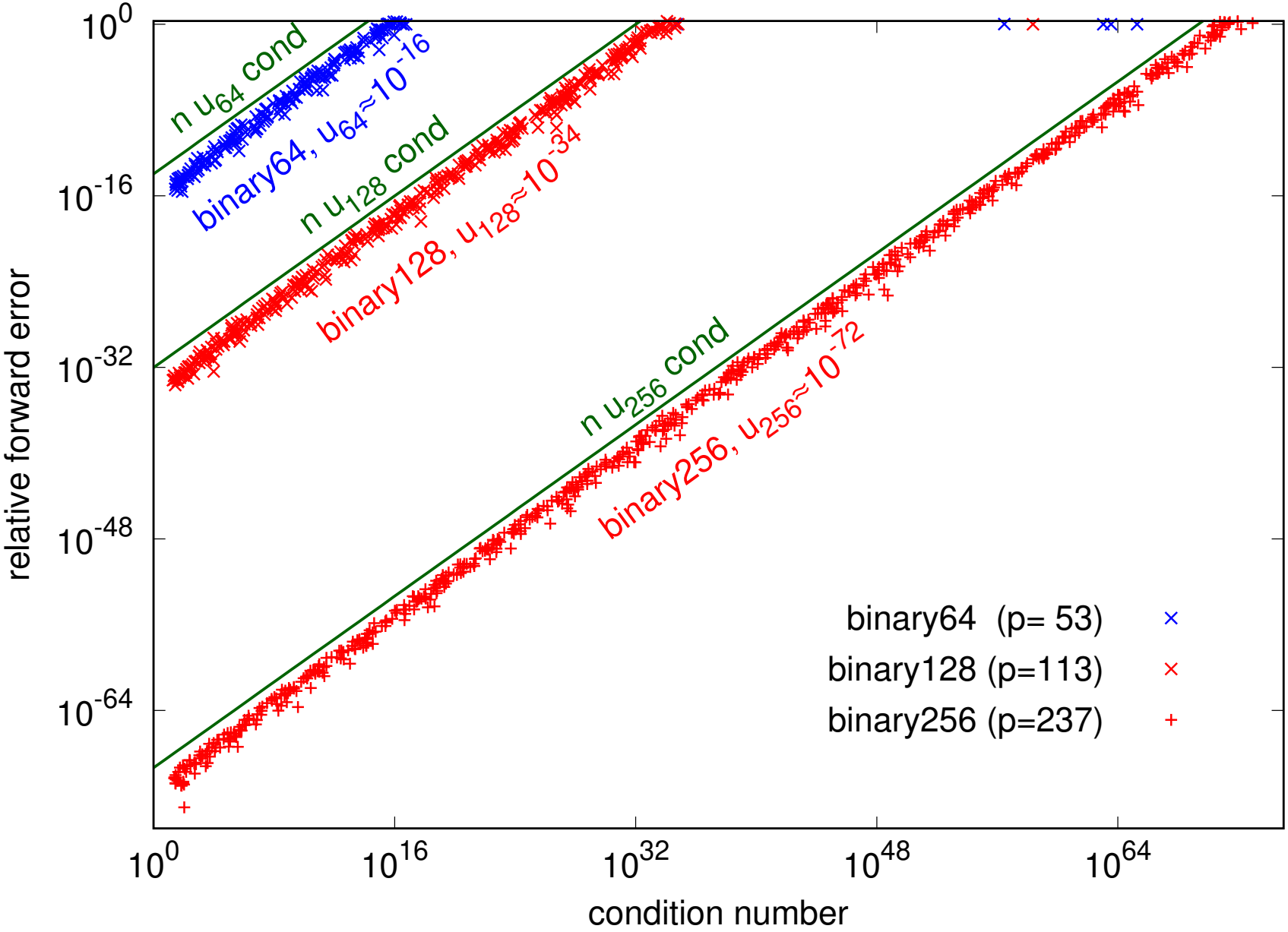
Relative error bound ($\langle x, y \rangle \neq 0$):

$$\frac{|s - \langle x, y \rangle|}{|\langle x, y \rangle|} \leqslant n\, u\, \mathrm{cond}(x, y),$$

where the condition number is given by

$$\mathrm{cond}(x, y) = \frac{\langle |x|, |y| \rangle}{|\langle x, y \rangle|} \qquad \left(= \frac{\sum_{i=1}^{n} |x_i y_i|}{\left| \sum_{i=1}^{n} x_i y_i \right|} \right).$$

$$|s/\langle x, y\rangle - 1| \leqslant nu\,\mathrm{cond}(x, y)$$



Floating point expansions of length $K > 2$

Such an expansion a is a sequence $(a_0, \ldots, a_{K-1})$ of elements in $\mathbb{F}$, representing $\sum_{i=0}^{K-1} a_i$.

A condition ensures that the elements are "sufficiently non-overlapping": a renormalization step is required at the end of each arithmetic operation (see [Popescu 17]).

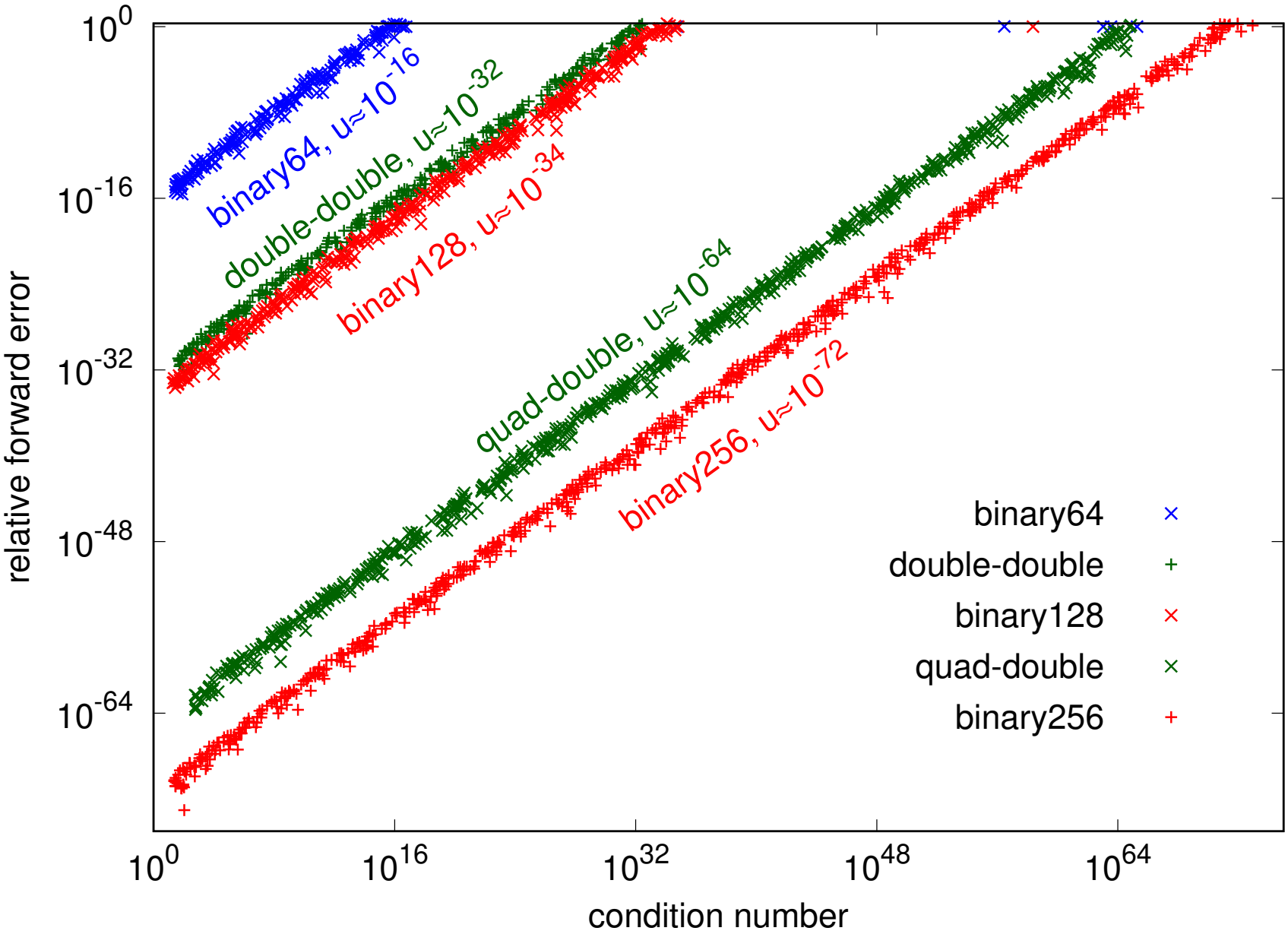
In this talk, we only assume that the floating point expansions we take as input satisfy

$$|a_i| \leqslant 2u|a_{i-1}|, \quad \text{for} \quad 0 < i < K,$$

and we use a renormalization algorithm from [Popescu 17] for the output.

In the qd library [Hida, Li, Bailey 2001]:

- a double-double is made of 2 Binary64 numbers (goes back to [Dekker 1971]).
  $\hookrightarrow$ 106 bits available, precision $\approx u^2$.
- a quad-double is the unevaluated sum of 4 Binary64 numbers.
  $\hookrightarrow$ 212 bits available, precision $\approx u^4$.


relative forward error
condition number
10^{0}
10^{-16}
10^{-32}
10^{-48}
10^{-64}
10^{0}
10^{16}
10^{32}
10^{48}
10^{64}
binary64, u≈10^{-16}
double-double, u≈10^{-32}
binary128, u≈10^{-34}
quad-double, u≈10^{-64}
binary256, u≈10^{-72}
binary64
double-double
binary128
quad-double
binary256

Error-free transformations (EFT)

- Given $\mathrm{op} = +$ or $\times$, and $x, y \in \mathbb{F}$, an EFT for op computes $r, \rho \in \mathbb{F}$ such that

$$x \,\mathrm{op}\, y = r + \rho \quad \text{with} \quad r = \mathrm{fl}(x \,\mathrm{op}\, y).$$

$+$	$(r, \rho) = \mathrm{TwoSum}(x, y)$	6 flops	[Knuth 74]
$+$	$(r, \rho) = \mathrm{FastTwoSum}(x, y)$ if $\lvert x\rvert \geqslant \lvert y\rvert$	3 flops	[Dekker 71]
$\times$	$(r, \rho) = \mathrm{TwoProd}(x, y)$ with FMA	2 flops	[Karp, Markstein 1997]

- EFT for the dot product of two floating point vectors $x, y \in \mathbb{F}^n$:

$(p, \pi) = \mathbf{VecTwoProd}(x, y)$
  for $i = 1 \ldots n$
    $(p_i, \pi_i) = \mathrm{TwoProd}(x_i, y_i)$

$\rightsquigarrow$ computes $p, \pi \in \mathbb{F}^n$ s. t.

$$\langle x, y \rangle = \langle p, e_n \rangle + \langle \pi, e_n \rangle.$$

- EFT for the summation of the elements of a vector $v \in \mathbb{F}^n$ [Ogita, Rump, Oishi 2005]:

$(s_n, v') = \mathbf{VecSum}(v)$
  $s_1 = v'_1 = 0$
  for $i = 2 \ldots n$
    $(s_i, v'_i) = \mathrm{TwoSum}(s_{i-1}, v_i)$

$\rightsquigarrow$ computes $s \in \mathbb{F}$ and $v' \in \mathbb{F}^n$ s. t.

$$\langle v, e_n \rangle = s_n + \langle v', e_n \rangle.$$

Outline

Compensation with double-double ($K = 2$) vectors

$x = x^{(0)} + x^{(1)}$ and $y = y^{(0)} + y^{(1)}$ are two double-double vectors of size n:

$$\langle x, y \rangle = \langle x^{(0)}, y^{(0)} \rangle + \langle x^{(1)}, y^{(0)} \rangle + \langle x^{(0)}, y^{(1)} \rangle + \langle x^{(1)}, y^{(1)} \rangle.$$

We relate the magnitude of all the terms to $\langle |x|, |y| \rangle$ (numerator in $\mathrm{cond}(x, y)$).

Using $|x^{(1)}| \leqslant 2u|x^{(0)}|$, $|y^{(1)}| \leqslant 2u|y^{(0)}|$:

$$\begin{array}{rcrcl}
|\langle x^{(0)}, y^{(0)} \rangle| & \leqslant & \langle |x|, |y| \rangle & + & \mathcal{O}(u), \\
|\langle x^{(1)}, y^{(0)} \rangle| & \leqslant & 2u\langle |x|, |y| \rangle & + & \mathcal{O}(u^2), \\
|\langle x^{(0)}, y^{(1)} \rangle| & \leqslant & 2u\langle |x|, |y| \rangle & + & \mathcal{O}(u^2), \\
|\langle x^{(1)}, y^{(1)} \rangle| & \leqslant & 4u^2\langle |x|, |y| \rangle & + & \mathcal{O}(u^3).
\end{array}$$

Compensation with double-double ($K = 2$) vectors

$x = x^{(0)} + x^{(1)}$ and $y = y^{(0)} + y^{(1)}$ are two double-double vectors of size n:

$$\langle x, y \rangle = \langle x^{(0)}, y^{(0)} \rangle + \langle x^{(1)}, y^{(0)} \rangle + \langle x^{(0)}, y^{(1)} \rangle + \langle x^{(1)}, y^{(1)} \rangle.$$

$$(p^{(0,0)}, \pi^{(0,0)}) = \mathrm{VecTwoProd}(x^{(0)}, y^{(0)})$$
$$(s_0, q^{(0)}) = \mathrm{VecSum}(p^{(0,0)})$$

$$\Longrightarrow \quad \langle x^{(0)}, y^{(0)} \rangle = s_0 + \sum_{\ell=1}^{n} \pi_\ell^{(0,0)} + \sum_{\ell=1}^{n} q_\ell^{(0)}$$

$$(p^{(1,0)}, \pi^{(1,0)}) = \mathrm{VecTwoProd}(x^{(1)}, y^{(0)})$$
$$(p^{(0,1)}, \pi^{(0,1)}) = \mathrm{VecTwoProd}(x^{(0)}, y^{(1)})$$
$$(s_1, q^{(1)}) = \mathrm{VecSum}([p^{(1,0)}; p^{(1,0)}; \pi^{(0,0)}; q^{(0)}])$$

$$\Longrightarrow \quad \langle x^{(0)}, y^{(0)} \rangle + \langle x^{(1)}, y^{(0)} \rangle + \langle x^{(0)}, y^{(1)} \rangle = s_0 + s_1 + \sum_{\ell=1}^{n} \left(\pi_\ell^{(1,0)} + \pi_\ell^{(0,1)} \right) + \sum_{\ell=1}^{4n} q_\ell^{(1)}$$

Overall, we get

$$\langle x, y\rangle = s_0 + s_1 + \langle x^{(1)}, y^{(1)}\rangle + \sum_{\ell=1}^{n} \left(\pi_\ell^{(1,0)} + \pi_\ell^{(0,1)}\right) + \sum_{\ell=1}^{4n} q_\ell^{(1)}$$

In the algorithm, we only keep what is needed to compute s_0 and s_1:

$$\begin{aligned}
&(r_0, r_1) = \textbf{DotComp2}(x, y)\\
&\quad (p^{(0,0)}, \pi^{(0,0)}) = \mathrm{VecTwoProd}(x^{(0)}, y^{(0)})\\
&\quad (s_0, q^{(0)}) = \mathrm{VecSum}(p^{(0,0)})\\
&\quad p^{(1,0)} = \mathrm{VecProd}(x^{(1)}, y^{(0)})\\
&\quad p^{(0,1)} = \mathrm{VecProd}(x^{(0)}, y^{(1)})\\
&\quad s_1 = \mathrm{Sum}([p^{(1,0)}; p^{(1,0)}; \pi^{(0,0)}; q^{(0)}])\\
&\quad (r_0, r_1) = \mathrm{Renorm}(s_0, s_1)
\end{aligned}$$

Bounding the magnitude of the $\mathcal{O}(u^2)$ terms gives:

If $\langle x, y\rangle \neq 0$, under some assumptions on u and n,

$$\frac{|\langle x, y\rangle - (r_0 + r_1)|}{|\langle x, y\rangle|} \leqslant (1 + 5u)(4 + 24n + 4n^2)u^2\,\mathrm{cond}(x, y).$$

In practice, we re-arrange the algorithm so that only one pass through the data is needed:

DotComp2 ($14n + \mathcal{O}(1)$ flops)

```
dd_real
DotComp2(dd_real *x, dd_real *y, int n) {
  double xh, xl, yh, yl, s, c, p1, e1, e2;

  s = c = 0.0;
  for(int i=0; i<n; i++) {
    xh = x[i]._hi(); xl = x[i]._lo();
    yh = y[i]._hi(); yl = y[i]._lo();

    TwoProd(p1, e1, xh, yh);
    c += xh*yl + xl*yh;


    TwoSumIn(s, e2, p1);
    c += e1 + e2;

  }
  TwoSumIn(s, c, c);
  return dd_real(s, c);
}
```

DotDD ($20n + \mathcal{O}(1)$ flops)

```
dd_real
DotDD(dd_real *x, dd_real *y, int n) {
  double xh, xl, yh, yl, sh, sl, ph, pl, e;

  sh = sl = 0.0;
  for(int i=0; i<n; i++) {
    xh = x[i]._hi(); xl = x[i]._lo();
    yh = y[i]._hi(); yl = y[i]._lo();

    TwoProd(ph, pl, xh, yh);
    pl += xh*yl + xl*yh;
    FastTwoSum(ph, pl, ph, pl);

    TwoSumIn(sh, e, ph);
    e += sl + pl;
    FastTwoSumIn(sh, sl, sh, e);
  }

  return dd_real(sh, sl);
}
```

DotComp avoids the renormalization steps used in **DotDD**: this exposes more instruction-level parallelism, which in general results in better performances (as in the case of CompHorner [L. 2007], and other compensated algorithms [Goossens *et al.* 2012]).

Compensated dot product with vectors of expansions of length $K > 2$

x and y are vectors of f. p. expansions: $x = \sum_{i=0}^{K-1} x_i$.

Same principles as in the case $K = 2$:

- all the renormalization steps are avoided,
- if K is fixed, "single pass" implementation.

Error bound of the form:

$$\frac{|r - \langle x, y\rangle|}{|\langle x, y\rangle|} \leqslant f_K(n) u^K \operatorname{cond}(x, y).$$

$(r_0, \ldots, r_{K-1}) = \mathbf{DotComp}(x, y, K)$
  for $k = 0 \ldots K-1$
    $s_k \leftarrow 0$
  end do
  for $\ell = 1 \ldots n$
    for i, j such that $0 \leqslant i + j \leqslant K-2$
      $(p, \pi) \leftarrow \mathrm{TwoProd}(x_\ell^{(i)}, y_\ell^{(j)})$
      $(s_{i+j}, p) \leftarrow \mathrm{TwoSum}(s_{i+j}, p)$
      for $k = i + j + 1 \ldots K-2$
        $(s_k, p) \leftarrow \mathrm{TwoSum}(s_k, p)$
        $(s_k, \pi) \leftarrow \mathrm{TwoSum}(s_k, \pi)$
      end do
      $s_{K-1} \leftarrow \mathrm{fl}\left(s_{K-1} + p + \pi\right)$
    end do
    for i, j such that $i + j = K-1$
      $s_{K-1} \leftarrow \mathrm{fl}\left(s_{K-1} + x_\ell^{(i)} y_\ell^{(j)}\right)$
    end do
  end do
  $(r_0, \ldots, r_{K-1}) = \mathrm{Renorm}(s_0, \ldots, s_{K-1})$

With quad-double inputs (K=4), $116n$ flops are required, and

$$f_4(n) = (1 + 5u)(96 + 768n + 41472un^3 + 1296n^4).$$

Outline

Numerical experiments

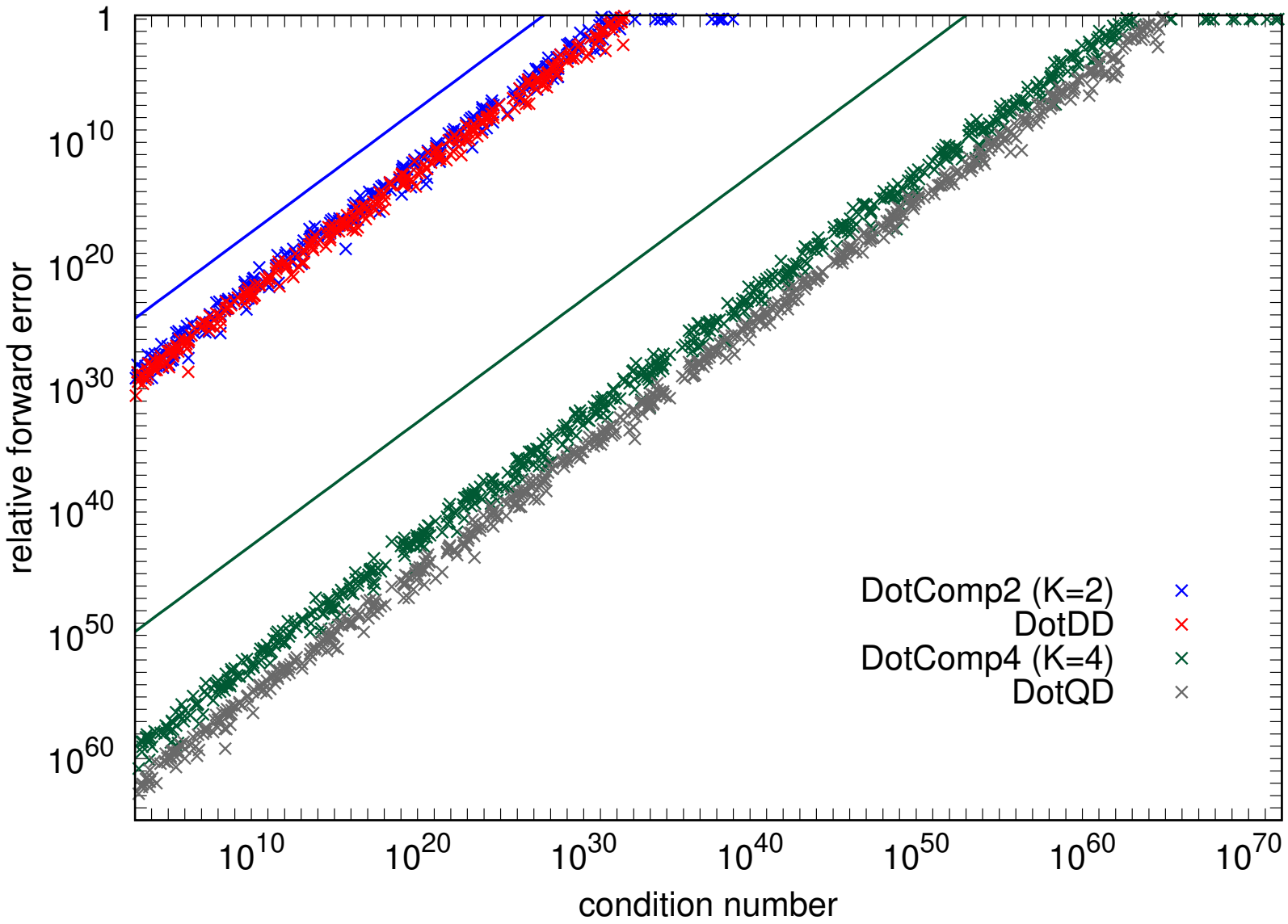
IEEE 754 Binary64 (double) is used as the working precision ($p = 53$, $u \approx 10^{-16}$).

In these experiments, we use:

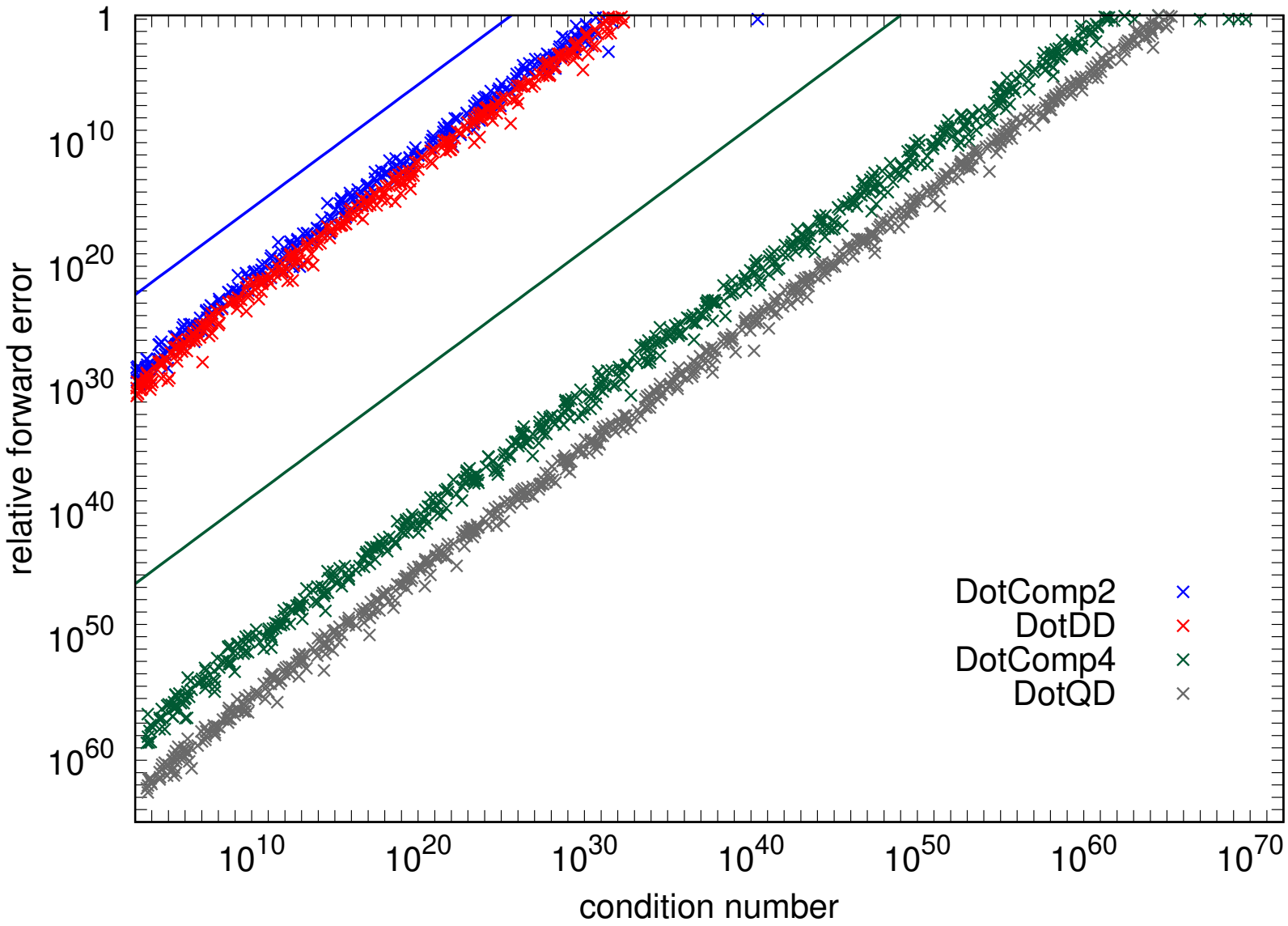
- DotDD and DotQD implements the naive dot product using the `qd` library,
- DotComp2 and DotComp4 implements our algorithm with $K = 2$ and 4 respectively; the implementations are compatible with the `qd` library.

The algorithms have been implemented in C++, compiled with GCC 6.3.0 (with options `-fno-associative-math` and `-ffp-contract=off`).

$n = 100$


relative forward error
1
10^{10}
10^{20}
10^{30}
10^{40}
10^{50}
10^{60}
10^{10}
10^{20}
10^{30}
10^{40}
10^{50}
10^{60}
10^{70}
condition number
DotComp2 (K=2)
DotDD
DotComp4 (K=4)
DotQD

$n = 1000$


relative forward error
1
10^{10}
10^{20}
10^{30}
10^{40}
10^{50}
10^{60}
10^{10}
10^{20}
10^{30}
10^{40}
10^{50}
10^{60}
10^{70}
condition number
DotComp2
DotDD
DotComp4
DotQD

Practical performances

We report experiments in two environments:

- **broadwell**: Intel CPU with micro-architecture Broadwell, GCC 6.3.0 (`-march=broadwell` `-mavx2` and `-mfma`).
- **skylake**: Intel CPU with micro-architecture Skylake, GCC 6.3.0 (`-march=skylake` `-mavx512f` and `-mfma`).

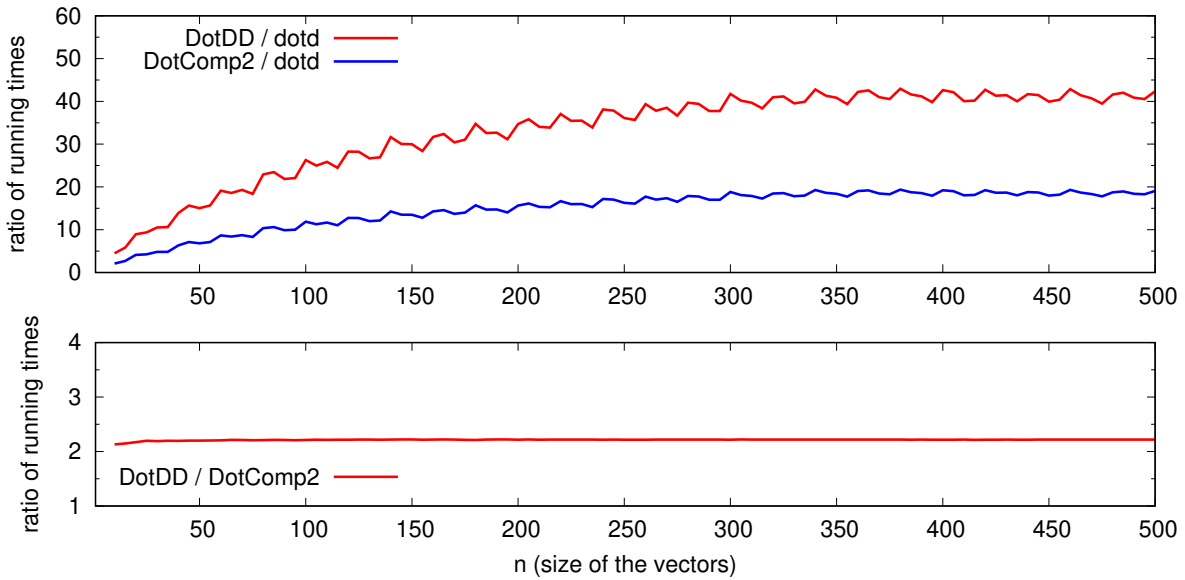
We measured the computing times in clock-cycles using the RDTSC and RDTSCP: we repeat each measure 400 times, ignore the 10% largest values, and take the average.

In all the case, the caches are warm (and the data fit within the L1 32 kB cache).

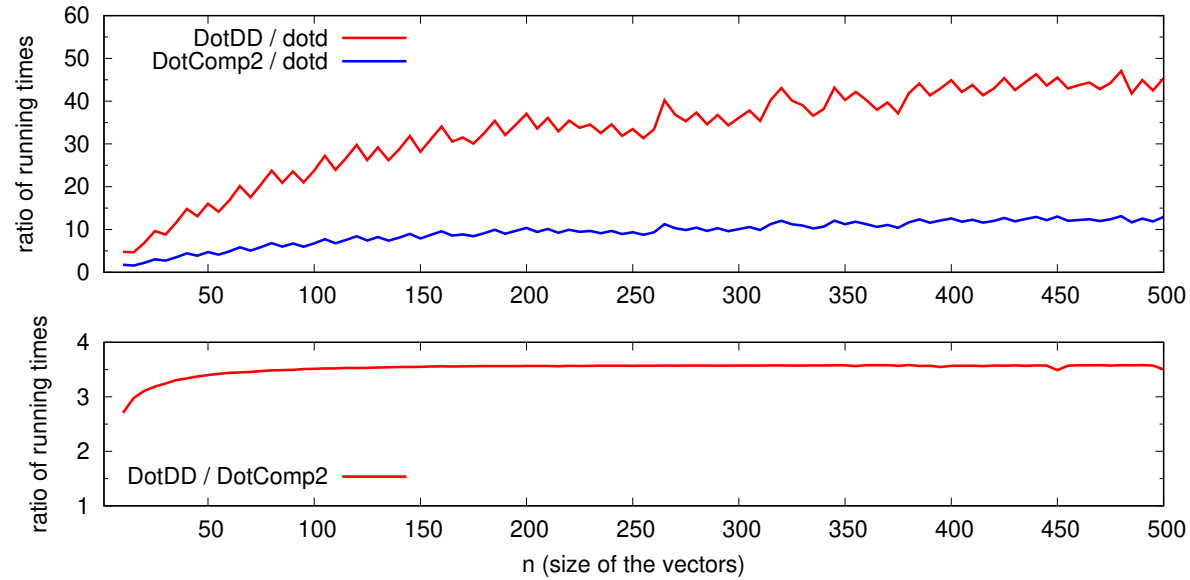
dotd is the naive algorithm in Binary64 arithmetic, vectorized by GCC.
DotDD, DotQD, DotComp2, DotComp4 are non-vectorized.

DotComp2 runs between 2 and 3.5 times faster than DotDD

broadwell

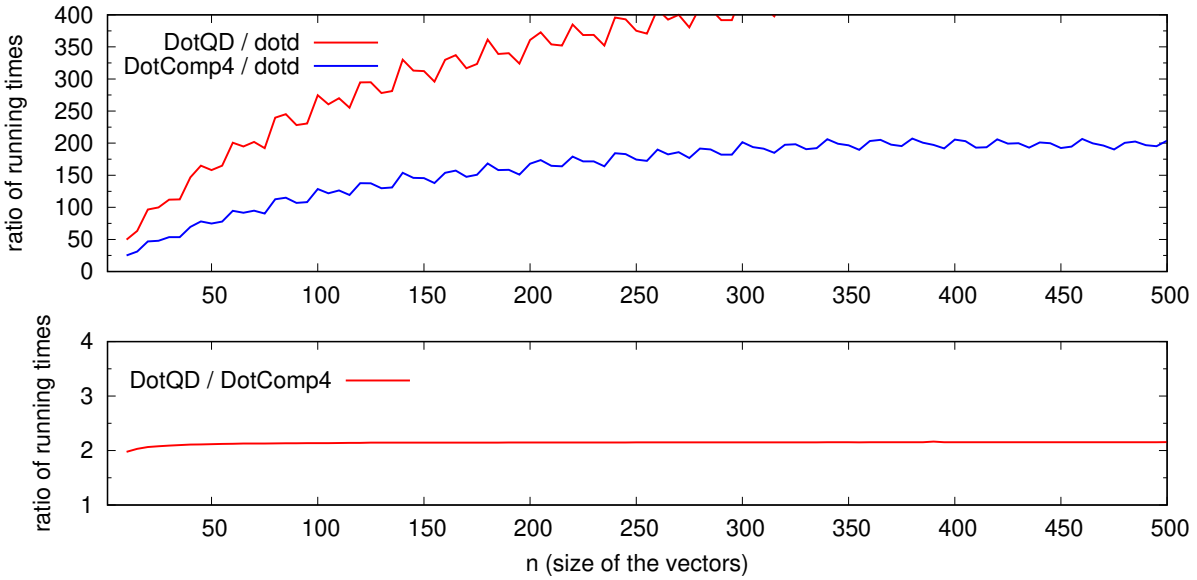


skylake

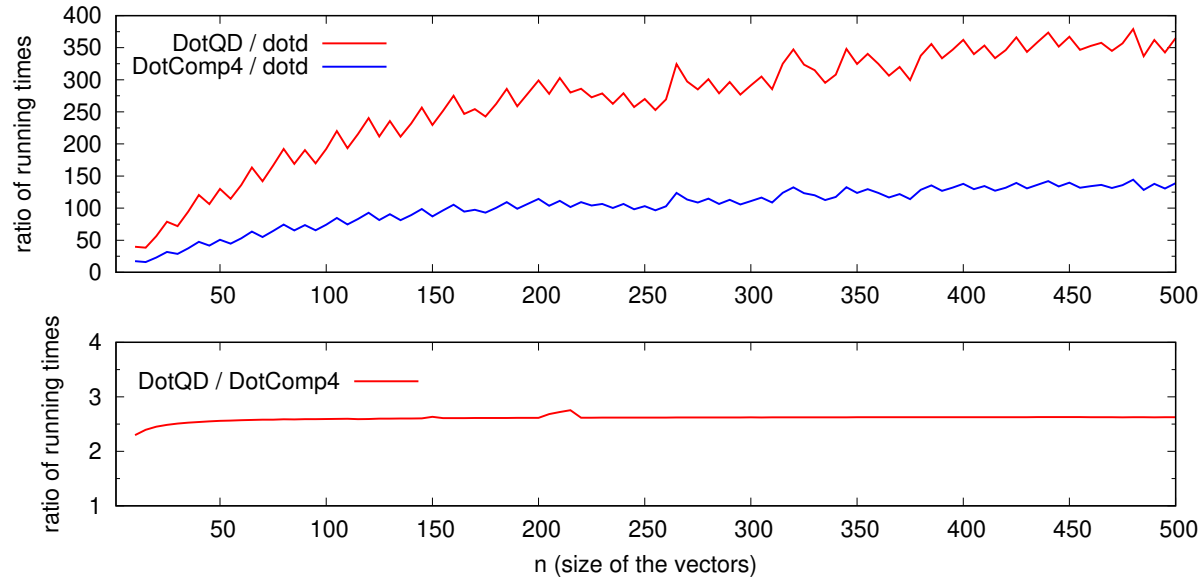


DotComp4 runs between 2 and 2.5 times faster than DotQD

broadwell



skylake



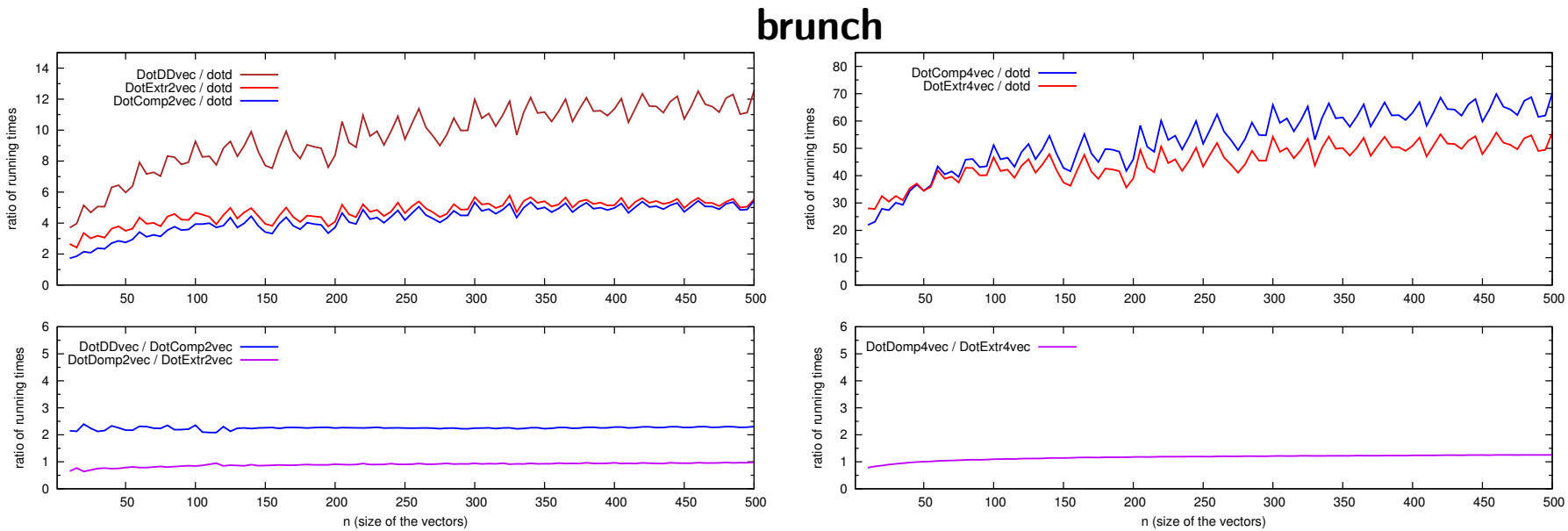
Conclusion

DotComp2 and DotComp4 are interesting alternatives to the use of the qd library: slightly less accurate algorithms, but a least two times faster.

We also experimented using PAPI (Performance Application Programming Interface) to confirm that the compensated algorithms provide more instruction level parallelism.

Other EFTs for summation of floating point numbers can be used in DotComp: "ExtractVector" from [Rump 2009] leads to better performances when $K = 4$.

Even better speedups can be obtained using SIMD vectorization:



Appendix

Compensated dot product [Neumanier 74, Pichat 76, Ogita-Rump-Oishi 05]

Given $x, y \in \mathbb{F}^n$, assume we start the evaluation of $\langle x, y \rangle$ as follows:

$$(p, \pi) = \mathrm{VecTwoProd}(x, y) \qquad // \ \langle x, y \rangle = \langle p, e_n \rangle + \langle \pi, e_n \rangle$$
$$(s_n, \sigma) = \mathrm{VecSum}(p) \qquad // \ \langle p, e_n \rangle = s_n + \langle \sigma, e_n \rangle$$

Since $\langle x, y \rangle = s_n + \langle \pi + \sigma, e_n \rangle$, we evaluate $\langle \pi + \sigma, e_n \rangle$ to compensate the errors in s_n.

$r = \mathbf{dot2}(x, y)$
$(p, \pi) = \mathrm{VecTwoProd}(x, y)$
$(s_n, \sigma) = \mathrm{VecSum}(p)$
$c = \mathrm{Sum}(\mathrm{fl}(\pi + \sigma))$
$r = \mathrm{fl}(s_n + c)$

Classical algorithm [Neumaier 1974, Pichat 1976].

Error bound from [Ogita, Rump, Oishi 2005]:

$$\frac{|r - x^T y|}{|x^T y|} \leqslant u + n^2 u^2 \, \mathrm{cond}(x, y) + \mathcal{O}(u^3).$$

Error-free transformations (EFT)

- Given $\mathrm{op} = +$ or $\times$, and $a, b \in \mathbb{F}$, an EFT for op computes $x, y \in \mathbb{F}$ such that

$$x + y = a \,\mathrm{op}\, b \quad \text{with} \quad x = \mathrm{fl}(a \,\mathrm{op}\, b).$$

Known EFT using only floating point operations in $\mathbb{F}$:

$+$	$(x, y) = \mathrm{TwoSum}(a, b)$	6 flops	[Knuth 74]
$+$	$(x, y) = \mathrm{FastTwoSum}(a, b)$ if $\lvert a\rvert \geqslant \lvert b\rvert$	3 flops	[Dekker 71]
$\times$	$(x, y) = \mathrm{TwoProd}(a, b)$ with FMA	2 flops	[Karp, Markstein 1997]

(x,y) = FastTwoSum(a,b)
$x = \mathrm{fl}(a + b)$
$y = \mathrm{fl}(\mathrm{fl}(a - x) + b)$

(x,y) = TwoProd(a,b)
$x = \mathrm{fl}(a \times b)$
$y = \mathrm{fl}(a \times b - x)$

Basic blocs for extending the working precision:

- expansions: [Dekker 71], [Priest 91], [Hida, Li, Bailey 01], . . . , [Popescu 17], [van der Hoeven 17]
- compensated summation and dot product: [Kahan 65], [Møller 65], [Pichat 72], [Neumanier 74], [Priest 92], [Ogita-Rump-Oishi 05, 08], [Rump 09]

- EFT for the dot product of two floating point vectors $x, y \in \mathbb{F}^n$:

$$
\begin{array}{l}
(p, \pi) = \textbf{VecTwoProd}(x, y) \\
\quad \text{for } i = 1 \ldots n \\
\qquad (p_i, \pi_i) = \text{TwoProd}(x_i, y_i)
\end{array}
$$

VecTwoProd computes $p, \pi \in \mathbb{F}^n$ s. t.

$$\langle x, y \rangle = \langle p, e_n \rangle + \langle \pi, e_n \rangle \quad \text{and} \quad |\langle \pi, e_n \rangle| \leqslant u \langle |x|, |y| \rangle.$$

- EFT for the summation of the elements in $v \in \mathbb{F}^n$:

$$
\begin{array}{l}
(s_n, v') = \textbf{VecSum}(v) \\
\quad s_1 = v'_1 = 0 \\
\quad \text{for } i = 2 \ldots n \\
\qquad (s_i, v'_i) = \text{TwoSum}(s_{i-1}, v_i)
\end{array}
$$

VecSum computes $s \in \mathbb{F}$ and $v' \in \mathbb{F}^n$ s. t.

$$\langle v, e_n \rangle = s_n + \langle v', e_n \rangle \quad \text{and} \quad \langle |v'|, e_n \rangle \leqslant nu \langle |v|, e_n \rangle.$$

Compensated dot product : some details

Three pass version of the algorithm:

$$
\begin{aligned}
r &= \mathbf{dot2}(x, y) \\
&\quad (p, \pi) = \mathrm{VecTwoProd}(x, y) \\
&\quad (s_n, \sigma) = \mathrm{VecSum}(p) \\
&\quad c = \mathrm{Sum}(\mathrm{fl}(\pi + \sigma)) \\
&\quad r = \mathrm{fl}(s_n + c)
\end{aligned}
$$

Single pass version of the algorithm:

$$
\begin{aligned}
r &= \mathbf{dot2}(x, y) \\
&\quad s_0 = c_0 = 0; \\
&\quad \text{for } i = 1 \ldots n \\
&\qquad (p_i, \pi_i) = \mathrm{TwoProd}(x_i, y_i) \\
&\qquad (s_i, \sigma_i) = \mathrm{TwoSum}(s_{i-1}, p_i) \\
&\qquad c_i = \mathrm{fl}(c_{i-1} + \mathrm{fl}(\pi_i + \sigma_i)) \\
&\quad r = \mathrm{RN}(s_n + c_n)
\end{aligned}
$$

go back

Further experiments with the PAPI library

- We used PAPI to measure the Instruction Per Cycle (IPC) rate of each implementation :

$$\text{IPC} = \frac{\#\text{instr}}{\#\text{cycles}}.$$

- Example with vectors of length $n = 500$:

	broadwell			**skylake**		
	#instr	#cycles	IPC	#instr	#cycles	IPC
DotDD	11816	12690	0.9	11816	16329	0.7
DotCmp2	8455	5017	1.7	8455	3833	2.2
DotQD	140585	115581	1.2	125586	103076	1.2
DotCmp4	67689	53180	1.3	66687	40235	1.7

- When we move from **broadwell** to **skylake**:
  - the IPC of DotDD decreases ☹
  - the IPC of CompDot2 increases ☺

Renormalization steps in DotDD = 2 × 3 ADD/SUB in series in the loop :

	instruction	ports	throughput	latency
broadwell	ADD/SUB	p1	1	3
skylake	ADD/SUB	p0 or p1	2	4

- Cheat sheet concerning the floating point instructions (`http://www.agner.org/`):

	instruction	ports	throughput	latency
broadwell	ADD/SUB	p1	1	3
	MUL/FMA	p0 or p1	2	3
skylake	ADD/SUB	p0 or p1	2	4
	MUL/FMA	p0 or p1	2	4