Fast, guaranteed-accurate sums of many floating-point numbers

Yong-Kang Zhu and Wayne B. Hayes
Department of Computer Science
University of California, Irvine
{yongkanz,wayne}@ics.uci.edu
Outline

1. Introduction
2. Related work
3. Algorithms
4. Results
5. Conclusions
Introduction

Example: (using IEEE 754 standard for binary floating-point arithmetic)

\[ a = 1000.13 \]
\[ b = 0.23 \]
\[ n = 2^{10} = 1024 \]
\[ s = a + n \cdot b \]

<table>
<thead>
<tr>
<th>Program 1</th>
<th>Program 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>double s1=a; for ( int i=1; i&lt;=n; i++ ) s1+=b;</td>
<td>double s2=a; s2+=n*b;</td>
</tr>
<tr>
<td>s1 = 1.235650000000000185 E 3</td>
<td>s2 = 1.235650000000000001 E 3</td>
</tr>
<tr>
<td>Mantissa of s1 = 1.4E99999999EB</td>
<td>Mantissa of s2 = 1.4E999999999A</td>
</tr>
</tbody>
</table>
Related work

- Recursive summation
  - ORS (Ordinary Recursive summation)
  - Recursive summation with orderings (increasing, decreasing, PSum)
  - Two other methods: Pairwise, Insertion

- Compensated summation and its variation

- Using high-precision accumulators
  - Demmel and Hida 2003

- Distillation algorithms
  - Anderson 1999
  - Ogita, et al., 2005
  - Zhu, et al., 2005
  - Rump, et al., 2006
Related work - A comparison

- No one method is uniformly more accurate than the other.
  (refer to recursive and compensated summations)  – Higham 1993
- No high-precision accumulators in typical computers.
- Distillation algorithms achieve a higher accuracy:

<table>
<thead>
<tr>
<th>methods</th>
<th>speed</th>
<th>accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anderson</td>
<td>Modified Deflation</td>
<td>slow</td>
</tr>
<tr>
<td>Ogita, et al.</td>
<td>SumK ((k = 3))</td>
<td>fast</td>
</tr>
<tr>
<td>Rump, et al.</td>
<td>AccSum</td>
<td>fast</td>
</tr>
<tr>
<td>Zhu, et al.</td>
<td>Zhu05</td>
<td>medium</td>
</tr>
</tbody>
</table>

\(R\) is the condition number: \(\sum_{i=1}^{n} |x_i| / \left| \sum_{i=1}^{n} x_i \right|\)
Algorithms

• Error-free addition: \( \{s, e\} \leftarrow \text{AddTwo}(a, b) \)
  
  \[
  \begin{align*}
  s + e &= a + b \\
  s &= \text{fl}(a + b) \\
  \text{fl}() : \text{standard floating-point operation}
  \end{align*}
  \]

• Assume \( \text{fl}() \) is correctly rounded (round-to-nearest)

Note: faithfully rounded

\[
\begin{cases}
  \text{fl}(x) = \text{either } a \text{ or } b, & \text{if } x \neq a \text{ and } x \neq b \\
  \text{fl}(x) = a, & \text{if } x = a \\
  \text{fl}(x) = b, & \text{if } x = b
\end{cases}
\]
Algorithms (Cont’d)

- FastSum is based on AddTwo

\[
\text{array } \begin{bmatrix} x_1 & x_2 & \ldots & \ldots & x_n \end{bmatrix}
\]

\[
\{s, x_i\} \leftarrow \text{AddTwo}(s, x_i)
\]

iteratively call AddTwo

- \( s = \text{fl}(x_1 + \text{fl}(x_2 + (\ldots \text{fl}(x_{n-1} + x_n) \ldots))) \)
- The errors are redistributed into the original array
- No significant digits are discarded
Algorithms (Cont’d)

• The basic idea of FastSum

Iteratively call AddTwo on all the numbers, to obtain $s$

Iteratively call AddTwo on positive and negative numbers separately, to obtain $s_p, s_n$

Estimate the sum of the remaining errors, say $e_m$

Call AddTwo to add $s_p$ and $s_n$ into $s$, and obtain two errors $e_1, e_2$

N

fl$(s + e_m) = s$ ?

Y

Return $s$ if faithfully rounded; otherwise recursively call FastSum

Question 1

Question 2
Algorithms (Cont’d)

Question 1: can $\text{fl}(s+e_m)=s$ fail to be satisfied?

• Claim: iteratively calling AddTwo will, in finite loops, converge to the following stable state:
  – The array is sorted by increasing magnitude
  – the mantissas of adjacent elements are non-overlapping
  – thus AddTwo does nothing
  – and the sum is constant
Algorithms (Cont’d)

• $e_m$ is calculated by:

$$e_m \leftarrow count \cdot \text{ulp}(\max(|s_p|, |s_n|))$$

  number of non-zero errors  ulp (unit in last place)

• If such a stable state arrives, then
  – $count$ equals the number of non-overlapping floating-point numbers
  – The maximum of non-overlapping floating-point numbers is limited by the arithmetic
Algorithms (Cont’d)

Compute this maximum:

- \( \text{fl}(s + s_p) = s, \text{fl}(s + s_n) = s \quad \implies \quad \max(|s_p|, |s_n|) < \text{ulp}(s) \)
- \( \text{fl}(s + e_m) \neq s \quad \implies \quad e_m > \text{ulp}(s) \)
- Recall: \( e_m \leftarrow \text{count} \cdot \text{ulp}(\max(|s_p|, |s_n|)) \)
- Thus, \( \text{count} \cdot \text{ulp(ulp(s))} > \text{ulp}(s) \quad \implies \quad \text{count} > \beta^t \)
- But in IEEE754 double,
  exponent = 11 bits, mantissa = 53 bits
  so, \( \text{count} < 2^{11} / 53 \approx 38 \ll \beta^t = 2^{53} \)

**Answer 1:** \( \text{fl}(s + e_m) = s \) can be satisfied in finite loops
Algorithms (Cont’d)

**Question 2:** Is $s$ faithfully rounded when recursively calling FastSum?

- Recall $e_1$ and $e_2$
- It is possible that $\text{fl}(s + e_m) = s$, but $\text{fl}(s \pm e_m + e_1 + e_2) \neq s$
- In this case, name the current $s_{s_1}$, and recursively call FastSum on the remaining numbers to obtain $s_2$, etc.
- Note that $s_i$ and $s_{i+1}$ overlap by at most 1 digit
- $s = s_1 + s_2 + \ldots + s_m$, $m$ is finite, and $s_m$ is faithfully rounded
Algorithms (Cont’d)

• The basic idea of FastSum

- Iteratively call AddTwo on all the numbers, to obtain $s$
- Iteratively call AddTwo on positive and negative numbers separately, to obtain $s_p, s_n$
- Estimate the sum of the remaining errors, say $e_m$

Call AddTwo to add $s_p$ and $s_n$ into $s$, and obtain two errors $e_1, e_2$

\[ \text{fl}(s + e_m) = s? \]

- **Question 1**
  - Return $s$ if faithfully rounded;
  - otherwise recursively call FastSum

- **Question 2**
Question 2: Is $s$ faithfully rounded when recursively calling FastSum?

- Recall $e_1$ and $e_2$
- It is possible that $\text{fl}(s + e_m) = s$, but $\text{fl}(s \pm e_m + e_1 + e_2) \neq s$
- In this case, name the current $s$ $s_1$, and recursively call FastSum on the remaining numbers to obtain $s_2$, etc.
- Note that $s_i$ and $s_{i+1}$ overlap by at most 1 digit.
- $s = s_1 + s_2 + \ldots + s_m$, $m$ is finite, and $s_m$ is faithfully rounded.

Answer 2: $s$ is faithfully rounded
Results

- Running time for 4 algorithms

Three ways to generate the original data

- Data No.1
  - well-conditioned
  - condition number $R = 1$, i.e., all positive or negative

- Data No.2
  - ill-conditioned
  - subtract the mean from each summand

- Data No.3
  - ill-conditioned data
  - condition number $R = +\infty$
  - generate pairs of equal numbers with opposite signs
    (the final sum is exactly zero)
Results (cont’d)

- Running time for 4 algorithms

<table>
<thead>
<tr>
<th>Data No.1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10 ($\times 10^6$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum3</td>
<td>3.9</td>
<td>4.5</td>
<td>6.8</td>
<td>6.0</td>
<td>5.5</td>
</tr>
<tr>
<td>Zhu05</td>
<td>11.7</td>
<td>11.6</td>
<td>17.1</td>
<td>16.0</td>
<td>14.9</td>
</tr>
<tr>
<td>AccSum</td>
<td>4.0</td>
<td>3.5</td>
<td>4.9</td>
<td>4.7</td>
<td>4.3</td>
</tr>
<tr>
<td>FastSum</td>
<td>4.0</td>
<td>4.5</td>
<td>6.3</td>
<td>5.9</td>
<td>5.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data No.2</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10 ($\times 10^6$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum3</td>
<td>5.2</td>
<td>4.0</td>
<td>6.3</td>
<td>5.7</td>
<td>5.6</td>
</tr>
<tr>
<td>Zhu05</td>
<td>15.7</td>
<td>14.1</td>
<td>21.9</td>
<td>18.9</td>
<td>16.6</td>
</tr>
<tr>
<td>AccSum</td>
<td>6.3</td>
<td>7.0</td>
<td>9.8</td>
<td>8.7</td>
<td>9.5</td>
</tr>
<tr>
<td>FastSum</td>
<td>4.1</td>
<td>4.5</td>
<td>6.8</td>
<td>5.9</td>
<td>5.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data No.3</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10 ($\times 10^6$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum3</td>
<td>4.2</td>
<td>8.8</td>
<td>6.6</td>
<td>6.0</td>
<td>7.2</td>
</tr>
<tr>
<td>Zhu05</td>
<td>15.6</td>
<td>29.3</td>
<td>22.7</td>
<td>20.3</td>
<td>24.7</td>
</tr>
<tr>
<td>AccSum</td>
<td>6.3</td>
<td>13.7</td>
<td>10.5</td>
<td>9.0</td>
<td>11.4</td>
</tr>
<tr>
<td>FastSum</td>
<td>5.2</td>
<td>10.7</td>
<td>8.1</td>
<td>7.3</td>
<td>8.8</td>
</tr>
</tbody>
</table>
Results (cont’d)

• Why choosing Dekker’s algorithm for AddTwo

<table>
<thead>
<tr>
<th></th>
<th>additions</th>
<th>branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dekker’s</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Knuth’s</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

– Will branches slow down the speed significantly?

• Numerical test

<table>
<thead>
<tr>
<th></th>
<th>with function calls</th>
<th>without function calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using Dekker’s algorithm</td>
<td>437</td>
<td>375</td>
</tr>
<tr>
<td>Using Knuth’s algorithm</td>
<td>469</td>
<td>391</td>
</tr>
</tbody>
</table>
Results (cont’d)

- Running time of FastSum is linear with $n$

<table>
<thead>
<tr>
<th>$n$</th>
<th>times</th>
<th>running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000,000</td>
<td>1</td>
<td>410 ms</td>
</tr>
<tr>
<td>1,000</td>
<td>10,000</td>
<td>$\approx 410$ ms</td>
</tr>
</tbody>
</table>

![Graphs showing running times](image-url)
Results (cont’d)

• In our test, for Data No. 3: \( R = +\infty \), exact sum = 0
  – Zhu05, AccSum, FastSum can always generate correct results, if no overflow occurs
  – Sum3 fails when \( \Delta E > 90 \)
  – \( \Delta E > 2000 \), AccSum produces an overflow
  – FastSum requires at most 48 loops when \( \Delta E \) is big, although it is not realistic that \( \Delta E > t \)
Conclusions

- **FastSum** is as fast as the existing algorithms.
- **FastSum** can guarantee the accuracy, independent of both $n$ and the condition number $R$.
- For floating-point arithmetic other than IEEE754, **FastSum** works as long as an effective **AddTwo** exists.
- The running time is linear with $n$, if generating the original data with the same attribute.
- In our environment, branches are not important.