Level 1 Parallel RTN-BLAS: Implementation and Efficiency Analysis

Chemseddine Chohra
Philippe Langlois and David Parello

University of Perpignan Via Domitia (UPVD)

24 September 2014
Introduction and problematic

Limited machine precision

- Using floating point numbers as approximation.
- $x \rightarrow X = \text{fl}(x)$ if $x \not\in F$ or $x$ if $x \in F$.
- $X + Y \neq X \oplus Y = \text{fl}(X + Y)$.
- IEEE-754 standard defines several rounding modes.

![Diagram](image)

- $x + y$ and $x \oplus y$
Limited machine precision

- Using floating point numbers as approximation.
- \( x \rightarrow X = \text{fl}(x) \) if \( x \notin F \) or \( x \) if \( x \in F \).
- \( X + Y \neq X \oplus Y = \text{fl}(X + Y) \).
- IEEE-754 standard defines several rounding modes.

Non-associativity of addition

- \( A \oplus (B \oplus C) \neq (A \oplus B) \oplus C \).
- Catastrophic cancelation: \( M = 2^{53}; 0 = -M \oplus (M \oplus 1) \neq (-M \oplus M) \oplus 1 = 1 \).
Limited machine precision

- Using floating point numbers as approximation.
  \[ x \rightarrow X = \text{fl}(x) \text{ if } x \notin F \text{ or } x \text{ if } x \in F. \]
- \[ X + Y \neq X \oplus Y = \text{fl}(X + Y). \]
- IEEE-754 standard defines several rounding modes.

Non-associativity of addition

- \[ A \oplus (B \oplus C) \neq (A \oplus B) \oplus C. \]
- Catastrophic cancelation: \( M = 2^{53}; 0 = -M \oplus (M \oplus 1) \neq (-M \oplus M) \oplus 1 = 1. \)

Non-reproducibility of summation

- For a sum \( (\sum_{i=1}^{n} X_i) \), the final result depends on the order of the computations.
- In parallel programs, dynamic scheduling and reductions could change this order.
- Exascale computing.
  - Reached in 2020: \( 10^{18} \) flop/s, Millions of cores.
  - Reproducibility of results will be a challenge.
Introduction and problematic

Why numerical reproducibility is important?

- Problem for debugging.
  - We can not debug errors that we can not reproduce.

- Problem for validating results.
  - For contractual and legacy reasons.

- The problem arises in real applications.
  - Energetics (Villa and al., 2009).
  - Climate modeling (Y. He and al., 2001).
  - Molecular dynamics (P. Saponaro., 2010).
How to fix the numerical reproducibility problem?

- **Fix the computation order.**
  - Static scheduling.
  - Deterministic reduction (Katranov, 2012).

- **Deterministic error (Demmel and Nguyen, 2013).**
  - ReprodSum.
  - FastReprodSum.
  - 1-Reduction.

- **Enhanced precision.**
  - Higher precision (quadruple precision for instance).
    - Reduce the probability of non-reproducibility (Villa and al., 2009).
    - Get more reproducible bits.
  - Correctly rounded arithmetic.
    - Deterministic Bit-Accurate Parallel Summation (S. Collange and al., 2014).
Our aim

Guarantee the numerical reproducibility for BLAS (Basic Linear Algebra Subroutines)

- Level 1: max, min, scal, axpy, norm, asum, dot.
- dot can be transformed to a sum: \( \sum_{i=1}^{n} X_i \cdot Y_i = \sum_{i=1}^{2n} Z_i \).

Compute an accurate sum

- When the result is correctly rounded, then it is reproducible.
- Several algorithms available.
- Is the cost acceptable?
Table of contents

1 Introduction and problematic
2 How to compute a correctly rounded sum?
3 Preliminary step: optimization for the sequential case
4 Parallel RTN sum implementation
5 Conclusion
How to compute a correctly rounded sum?

Recent summation algorithms

Faithfully rounded (one of the floating-point neighbors)

- AccSum (Rump and al., 2008).
- FastAccSum (Rump, 2008).

$$\sum x_n$$
How to compute a correctly rounded sum?

Recent summation algorithms

**Faithfully rounded (one of the floating-point neighbors)**
- AccSum (Rump and al., 2008).
- FastAccSum (Rump, 2008).

**Correctly rounded (according to the rounding mode)**
- NearSum (Rump and al., 2008).
- iFastSum (Zhu and Hayes, 2009).
- HybridSum (Zhu and Hayes, 2009).
- OnlineExact (Zhu and Hayes, 2010).
How to compute a correctly rounded sum?

Experimental framework of this work

Implementation

- Implemented using C language.

Hardware

- Xeon E5 socket.
- Cache L1 = 32 KB, L2 = 256 KB, L3 = 20 MB.
- Memory max bandwidth 51.2 GB/s.
- Turbo boost turned off.

Compiler

- Intel ICC 14.0.0.
How to compute a correctly rounded sum?

Implementation of algorithms

HybridSum and OnlineExact do not depend on the condition number.

Implementation

- Manually optimized version for all algorithms (see details in next section).

Legends.

X axis → Log2 of size.
Y axis → Runtime(cycles) / size.

- Cond = $10^8$
- Cond = $10^{16}$
- Cond = $10^{24}$
- Cond = $10^{32}$
How to compute a correctly rounded sum?

HybridSum and OnlineExact do not depend on the condition number.

Implementation
- Manually optimized version for all algorithms (see details in next section).

![Graphs showing performance of different algorithms](e) AccSum, (f) FastAccSum, (g) OnlineExact, (h) HybridSum

Legends.
- X axis $\rightarrow$ Log2 of size.
- Y axis $\rightarrow$ Runtime(cycles) / size.
- Cond = $10^8$
- Cond = $10^{16}$
- Cond = $10^{24}$
- Cond = $10^{32}$

Note
- HS and OLE: condition number independents.
How to compute a correctly rounded sum?

OnlineExact and HybridSum are faster for large vectors

(i) Condition Number = $10^8$

Legends.

X axis → Log2 of size.
Y axis → Runtime (cycles) / size.

- iFastSum
- AccSum
- FastAccSum
- OnlineExact
- HybridSum

(ii) Condition Number = $10^{32}$
How to compute a correctly rounded sum?

OnlineExact and HybridSum are faster for large vectors.

Legend:
- iFastSum
- AccSum
- FastAccSum
- OnlineExact
- HybridSum

Note:
- HS and OLE: linear to size.

X axis $\rightarrow$ Log2 of size.
Y axis $\rightarrow$ Runtime (cycles) / size.

(k) Condition Number = $10^8$

(l) Condition Number = $10^{32}$
Description of algorithm HybridSum (Zhu and Hayes, 2009)
How to compute a correctly rounded sum?

Implementation of algorithms

Description of algorithm HybridSum (Zhu and Hayes, 2009)
How to compute a correctly rounded sum?

Description of algorithm HybridSum (Zhu and Hayes, 2009)

$n < 2^{26}$

$C_{\text{exponent}}(\text{High}) += \text{High}$

$C_{\text{exponent}}(\text{Low}) += \text{Low}$

$C$ is an error-free transformation of $A$
How to compute a correctly rounded sum?

Implementation of algorithms

Description of algorithm HybridSum (Zhu and Hayes, 2009)

$n < 2^{26}$

$n$

$\text{Split}(A_i, \text{High}, \text{Low})$

$C_{\text{exponent}(	ext{High})} += \text{High}$

$C_{\text{exponent}(	ext{Low})} += \text{Low}$

$C$ is an error-free transformation of $A$

$\text{IFastSum}(C)$

2048
How to compute a correctly rounded sum?

Implementation of algorithms

Description of algorithm OnlineExact (Zhu and Hayes, 2010)
How to compute a correctly rounded sum?

Implementation of algorithms

Description of algorithm OnlineExact (Zhu and Hayes, 2010)
1 Introduction and problematic

2 How to compute a correctly rounded sum?

3 Preliminary step: optimization for the sequential case
   - Optimization of HybridSum
   - Optimization of OnlineExact
   - Compare to dasum, ReprodSum and FastReprodSum
   - Overhead in the sequential case

4 Parallel RTN sum implementation

5 Conclusion
ALGORITHM HybridSum.
INPUT : A, an array of floating point summands.
OUTPUT : S, the correctly rounded sum of A.
BEGIN.

1. Declare an intermediate array C.
2. for i=1:n do.
   1. maskSplit(A[i], ah, al).
   2. e = exponent(ah).
   4. e = exponent(al).
   5. C[e] += al.

3. end for.
4. RETURN iFastSum(C).

END.
ALGORITHM HybridSum.
INPUT : A, an array of floating point summands.
OUTPUT : S, the correctly rounded sum of A.
BEGIN.
  1. Declare an intermediate array C.
  2. for i=1:n do.
      1. veltkampSplit(A[i], ah, al). step(1)
      2. e = exponent(ah).
      4. e = exponent(al).
      5. C[e] += al.
  end for.
  4. RETURN iFastSum(C).
END.
Preliminary step: optimization for the sequential case

Optimization of HybridSum

ALGORITHM HybridSum.
INPUT: A, an array of floating point summands.
OUTPUT: S, the correctly rounded sum of A.
BEGIN.

1. Declare an intermediate array C.
2. for i=1:n (unrolled) do. step(2)
   1. veltkampSplit(A[i], a_h, a_l). step(1)
   2. e = exponent(a_h).
   3. C[e] += a_h.
   4. e = exponent(a_l).
   5. C[e] += a_l.
3. end for.
4. RETURN iFastSum(C).
END.

Chemseddine Chohra, Philippe Langlois and DiLevel 1 Parallel RTN-BLAS: Implementation
Optimization of HybridSum

ALGORITHM HybridSum.
INPUT : A, an array of floating point summands.
OUTPUT : S, the correctly rounded sum of A.
BEGIN.

1. Declare an intermediate array C.
2. for i=1:n (unrolled) do. step(2)
   1. prefetch data. step(3)
   2. veltkampSplit(A[i], ah, al). step(1)
   3. e = exponent(ah).
   5. e = exponent(al).
   6. C[e] += al.

3. end for.
4. RETURN iFastSum(C).

END.
ALGORITHM HybridSum.
INPUT : A, an array of floating point summands.
OUTPUT : S, the correctly rounded sum of A.
BEGIN.

1. Declare an intermediate array C.
2. for i=1:n (unrolled) do. step(2)
   1. prefetch data. step(3)
   2. veltkampSplit(A[i], ah, a1). step(1)
   3. e = exponent(ah).
   5. e = e - 27. step(4)
3. end for.
4. RETURN iFastSum(C).

END.
Preliminary step: optimization for the sequential case

Optimization of HybridSum

Gain of 60% of runtime after optimization of HybridSum

Legends.

X axis → Log2 of size.
Y axis → Runtime (cycles) / naive.

- Naive implementation
- Step 1: replace the mask
- Step 2: unrolling loop
- Step 3: prefetching
- Step 4: compute exponent
Preliminary step: optimization for the sequential case

Optimization of HybridSum

Gain of 60% of runtime after optimization of HybridSum

Legends.

X axis → Log2 of size.
Y axis → Runtime (cycles) / naive.

- Naive implementation
- Step 1: replace the mask
- Step 2: unrolling loop
- Step 3: prefetching
- Step 4: compute exponent

Note

- We gain 60% of runtime.
ALGORITHM OnlineExact.
INPUT : A, an array of floating point summands.
OUTPUT : S, the correctly rounded sum of A.
BEGIN.
   1. Declare two intermediate arrays C1, C2.
   2. for i=1:n do.
      1. i = exponent(a).
      2. (C1[i], a) = 2Sum(C1[i], a).
   end for.
   3. RETURN iFastSum(C1 ∪ C2).
END.
ALGORITHM OnlineExact.
INPUT : A, an array of floating point summands.
OUTPUT : S, the correctly rounded sum of A.
BEGIN.

1. Declare two intermediate arrays C1, C2.
2. for $i=1:n$ (unrolled) do. step(1)
   1. $i =$ exponent(a).
   2. $(C1[i], a) = 2\text{Sum}(C1[i], a)$.
   end for.
3. RETURN $i\text{FastSum}(C1 \cup C2)$.
END.
Optimization of OnlineExact

ALGORITHM OnlineExact.
INPUT : A, an array of floating point summands.
OUTPUT : S, the correctly rounded sum of A.
BEGIN.

1. Declare two intermediate arrays C1, C2.
2. for i=1:n (unrolled) do. step(1)
   1. prefetch data. step(2)
   2. i = exponent(a).
   3. (C1[i], a) = 2Sum(C1[i], a).

end for.

3. RETURN iFastSum(C1 ∪ C2).

END.
Optimization of OnlineExact

ALGORITHM OnlineExact.
INPUT : A, an array of floating point summands.
OUTPUT : S, the correctly rounded sum of A.
BEGIN.

1. Declare an intermediate arrays C. step(3)
2. for i=1:n (unrolled) do. step(1)
   1. prefetch data. step(2)
   2. i = exponent(a).
   3. (C[2*i], a) = 2Sum(C[2*i], a). step(3)

end for.

3. RETURN iFastSum(C). step(3)

END.
Preliminary step: optimization for the sequential case

Optimization of OnlineExact

Gain of 25% of runtime after optimization of OnlineExact

Legends.

X axis → Log2 of size.
Y axis → Runtime (cycles) / naive.

- Naive implementation
- Step 1: unrolling loop
- Step 2: prefetching
- Step 3: use one vector
Gain of 25% of runtime after optimization of OnlineExact

- Preliminary step: optimization for the sequential case
- Optimization of OnlineExact
- Gain of 25% of runtime after optimization of OnlineExact

Legends:
- X axis → Log2 of size.
- Y axis → Runtime (cycles) / naive.

- Naive implementation
- Step 1: unrolling loop
- Step 2: prefetching
- Step 3: use one vector

Note
- We gain 25% of runtime.
Preliminary step: optimization for the sequential case

Compare to dasum, ReprodSum and FastReprodSum

### Optimized sum
- dasum: optimized by Intel in the library MKL.

### Reproducible sum
- ReprodSum: guarantee reproducibility.
- FastReprodSum: faster than ReprodSum but requires direct rounding.
Preliminary step: optimization for the sequential case

Compare to dasum, ReprodSum and FastReprodSum

ReprodSum and FastReprodSum (Demmel and Nguyen, 2013)

Diagram showing the distribution of variables and processes:
- Variables: $x_1, x_2, x_3, x_4, x_5, x_6, \ldots$
- Processes: proc 1, proc 2, proc 3

Boundary: Bits discarded in advance

Levels of parallelism and efficiency analysis for RTN-BLAS.
Overhead in the sequential case

Sequential runtime VS absSum

- OnlineExact
- HybridSum
- ReprodSum
- FastReprod
- absSum

Runtime / absSum

log2(n)

14 16 18 20 22 24
Preliminary step: optimization for the sequential case

Overhead in the sequential case

Overhead in the sequential case

Sequential runtime VS absSum

- OnlineExact
- HybridSum
- ReprodSum
- FastReprod
- abssum

$\times 4$

log2(n)

Runtime / absSum
Preliminary step: optimization for the sequential case

Overhead in the sequential case

Sequential runtime VS absSum

- OnlineExact
- HybridSum
- ReprodSum
- FastReprod
- absSum

Runtime / absSum vs log2(n)

- x2
- x4
# Table of contents

1. **Introduction and problematic**

2. **How to compute a correctly rounded sum?**

3. **Preliminary step: optimization for the sequential case**

4. **Parallel RTN sum implementation**
   - Parallel algorithms
   - Experimental framework
   - Used libraries
   - Overhead for parallel RTN version

5. **Conclusion**
Parallel algorithm (2 processors case)

\[ \begin{array}{c}
  A_1 \\
  A_2 \\
  \vdots \\
  \vdots \\
  \vdots \\
  \vdots \\
  A_{n/2} \\
  \vdots \\
  \vdots \\
  \vdots \\
  \vdots \\
  A_n \\
\end{array} \]
Parallel algorithm (2 processors case)

\[
\begin{array}{c}
A_1 \\
A_2 \\
\vdots \\
A_{n/2} \\
A_n \\
\end{array}
\]
Parallel algorithm (2 processors case)

\[
\begin{array}{c|c}
A_1 & X_1_1 \\
A_2 & X_1_2 \\
\vdots & \vdots \\
\vdots & \vdots \\
A_{n/2} & X_1_{size} \\
\hline
A_n & X_2_1 \\
\vdots & \vdots \\
\vdots & \vdots \\
\vdots & \vdots \\
A_n & X_2_{size} \\
\end{array}
\]
Parallel algorithm (2 processors case)

\[
\begin{array}{c|c}
A_1 & X_{11} \\
A_2 & X_{12} \\
\vdots & \vdots \\
\vdots & \vdots \\
A_{n/2} & X_{1\text{size}} \\
\end{array}
\]

EFT

\[
\begin{array}{c|c}
X_{11} & X_{12} \\
\vdots & \vdots \\
& X_{1\text{size}} \\
\end{array}
\]

Size = 2048 for HybridSum
Size = 4096 for OnlineExact

\[
\begin{array}{c|c}
A_n/2 & X_{21} \\
A_n & X_{22} \\
\vdots & \vdots \\
\vdots & \vdots \\
\vdots & \vdots \\
\end{array}
\]

EFT

\[
\begin{array}{c|c}
X_{21} & X_{22} \\
\vdots & \vdots \\
& X_{2\text{size}} \\
\end{array}
\]
Parallel algorithm (2 processors case)

EFT

\[ X_1 \]
\[ X_1_2 \]
\[ \ldots \]
\[ X_{1\text{size}} \]

iFastSum

\[ S_{1\text{size}} \]

\[ S_{1_1} \]
\[ \ldots \]

\[ A_1 \]
\[ A_2 \]
\[ \ldots \]
\[ \ldots \]
\[ \ldots \]

\[ A_{n/2} \]

EFT

\[ X_2_1 \]
\[ X_2_2 \]
\[ \ldots \]
\[ X_{2\text{size}} \]

iFastSum

\[ S_{2\text{size}} \]

\[ S_{2_1} \]
\[ \ldots \]

Parallel algorithm (2 processors case)

\[
\begin{array}{ccccccccccc}
X_1 & X_2 & X_3 & X_4 & X_5 & X_6 & X_7 & X_8 & \ldots\. & X_{\text{size}} \\
\end{array}
\]

\[
\text{iFastSum}
\]

\[
S_1 \quad \text{Round-off errors \neq 0}
\]
### Parallel algorithm (2 processors case)

<table>
<thead>
<tr>
<th>X₁</th>
<th>X₂</th>
<th>X₃</th>
<th>X₄</th>
<th>X₅</th>
<th>X₆</th>
<th>X₇</th>
<th>X₈</th>
<th>........</th>
<th>Xₙᵢₙₑ</th>
</tr>
</thead>
</table>

- **S₁**
  - Round-off errors ≠ 0
  - iFastSum

- **S₂**
  - Round-off errors ≠ 0
  - iFastSum
Parallel RTN sum implementation

Parallel algorithm (2 processors case)
Parallel RTN sum implementation

Parallel algorithms

Parallel algorithm (2 processors case)
Parallel algorithm (2 processors case)

\[ A_1 \]

\[ A_2 \]

\[ \ldots \]

\[ \ldots \]

\[ \ldots \]

\[ \ldots \]

\[ A_{n/2} \]

\[ \begin{align*}
X_1_1 \\
X_1_2 \\
\ldots \\
X_1_{size}
\end{align*} \]

\[ \text{iFastSum} \]

\[ \begin{align*}
S_1_1 \\
\ldots \\
S_{1_{m1}}
\end{align*} \]

\[ m_1 < 40 \]

\[ \begin{align*}
\text{EFT} \\
\begin{align*}
X_2_1 \\
X_2_2 \\
\ldots \\
X_2_{size}
\end{align*} \\
\text{iFastSum}
\end{align*} \]

\[ \begin{align*}
S_2_1 \\
\ldots \\
S_{2_{m2}}
\end{align*} \]
Parallel RTN sum implementation

Parallel algorithms

Parallel algorithm (2 processors case)
Parallel algorithms

Parallel algorithm (2 processors case)

Sequential phase

EFT

iFastSum

S1₁

... S₁ₘ₁

C₁

... C₁ₘ₁+ₘ₂

iFastSum(C)

A₁

A₂

... ...

X₁₁

X₁₂

... ...

X₁_size

Aₙ/₂

... ...

EFT

iFastSum

S₂₁

... S₂ₘ₂

Aₙ

... ...

X₂₁

X₂₂

... ...

X₂_size

UNION

Chemseddine Chohra
Philippe Langlois and DiLevel 1 Parallel RTN-BLAS: Implementation 24 September 2014 22 / 34
Hardware

- Two Xeon E5 sockets.
- 8 cores on each socket.
- Multi-threading is turned off.
### Parallel RTN sum implementation

#### Used libraries

#### Implementation details

**Machine (128GB)**

<table>
<thead>
<tr>
<th>NUMANode P#0 (64GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket P#0</td>
</tr>
<tr>
<td>L2 (256KB)</td>
</tr>
<tr>
<td>L2 (256KB)</td>
</tr>
<tr>
<td>L2 (256KB)</td>
</tr>
<tr>
<td>L2 (256KB)</td>
</tr>
<tr>
<td>L2 (256KB)</td>
</tr>
<tr>
<td>L2 (256KB)</td>
</tr>
<tr>
<td>L2 (256KB)</td>
</tr>
<tr>
<td>L2 (256KB)</td>
</tr>
<tr>
<td>L2 (256KB)</td>
</tr>
<tr>
<td>L2 (256KB)</td>
</tr>
<tr>
<td>L2 (256KB)</td>
</tr>
<tr>
<td>Core P#0</td>
</tr>
<tr>
<td>Core P#1</td>
</tr>
<tr>
<td>Core P#2</td>
</tr>
<tr>
<td>Core P#3</td>
</tr>
<tr>
<td>Core P#4</td>
</tr>
<tr>
<td>Core P#5</td>
</tr>
<tr>
<td>Core P#6</td>
</tr>
<tr>
<td>Core P#7</td>
</tr>
<tr>
<td>PU P#0</td>
</tr>
<tr>
<td>PU P#2</td>
</tr>
<tr>
<td>PU P#4</td>
</tr>
<tr>
<td>PU P#6</td>
</tr>
<tr>
<td>PU P#8</td>
</tr>
<tr>
<td>PU P#10</td>
</tr>
<tr>
<td>PU P#12</td>
</tr>
<tr>
<td>PU P#14</td>
</tr>
</tbody>
</table>

**NUMANode P#1 (64GB)**

<table>
<thead>
<tr>
<th>Socket P#1</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2 (256KB)</td>
</tr>
<tr>
<td>L2 (256KB)</td>
</tr>
<tr>
<td>L2 (256KB)</td>
</tr>
<tr>
<td>L2 (256KB)</td>
</tr>
<tr>
<td>L2 (256KB)</td>
</tr>
<tr>
<td>L2 (256KB)</td>
</tr>
<tr>
<td>L2 (256KB)</td>
</tr>
<tr>
<td>L2 (256KB)</td>
</tr>
<tr>
<td>L2 (256KB)</td>
</tr>
<tr>
<td>L2 (256KB)</td>
</tr>
<tr>
<td>L2 (256KB)</td>
</tr>
<tr>
<td>Core P#0</td>
</tr>
<tr>
<td>Core P#1</td>
</tr>
<tr>
<td>Core P#2</td>
</tr>
<tr>
<td>Core P#3</td>
</tr>
<tr>
<td>Core P#4</td>
</tr>
<tr>
<td>Core P#5</td>
</tr>
<tr>
<td>Core P#6</td>
</tr>
<tr>
<td>Core P#7</td>
</tr>
<tr>
<td>PU P#1</td>
</tr>
<tr>
<td>PU P#3</td>
</tr>
<tr>
<td>PU P#5</td>
</tr>
<tr>
<td>PU P#7</td>
</tr>
<tr>
<td>PU P#9</td>
</tr>
<tr>
<td>PU P#11</td>
</tr>
<tr>
<td>PU P#13</td>
</tr>
<tr>
<td>PU P#15</td>
</tr>
</tbody>
</table>

**Host: daliserv2**

- **OpenMP**
- **MPI**
Strong scaling of HybridSum and OnlineExact

(m) HybridSum

(n) OnlineExact

Note: Good scalability up to 16 cores at least.
Strong scaling of HybridSum and OnlineExact

(o) HybridSum

(p) OnlineExact

Note

- Good scalability up to 16 cores at least.
4 cores parallel results

![Graph showing four cores runtime versus absSum](image)
4 cores parallel results

Four cores runtime VS absSum

- OnlineExact
- HybridSum
- ReprodSum
- FastReprod
- absSum

Runtime / absSum vs log2(n)
4 cores parallel results

Four cores runtime VS absSum

- OnlineExact
- HybridSum
- ReprodSum
- FastReprod
- absSum

$x2$
$x5$
4 cores parallel results

![Four cores runtime VS absSum](chart)

- OnlineExact
- HybridSum
- ReprodSum
- FastReprod
- absSum

Log2(n) vs Runtime / absSum

- x3
- x2
- x5
16 cores parallel results

Sixteen cores runtime VS absSum

- OnlineExact
- HybridSum
- ReprodSum
- FastReprod
- absSum

Runtime / absSum vs log2(n)
Sixteen cores runtime VS absSum

- OnlineExact
- HybridSum
- ReprodSum
- FastReprod
- absSum

Runtime / absSum vs \log_2(n)

x2
16 cores parallel results

![Graph showing runtime vs absSum for different algorithms with a note indicating a factor of 2 improvement](image-url)
Strong scalability using 1 core, 2 cores, 4 cores, 8 cores and 16 cores.
Strong scalability using 1 core, 2 cores, 4 cores, 8 cores and 16 cores.

Note
- dasum, ReprodSum and FastReprodSum: bandwidth limit.
HybridSum and OnlineExact are not limited by bandwidth.
HybridSum and OnlineExact are not limited by bandwidth.

**Note**

- HybridSum and OnlineExact: no bandwidth limit.
Parallel RTN sum implementation

Overhead for parallel RTN version

Summarization

![Graphs showing performance comparisons between different algorithms](image)

- Sequential runtime vs. absSum
- Four cores runtime vs. absSum
- Sixteen cores runtime vs. absSum

- OnlineExact
- HybridSum
- ReprodSum
- FastReprod
- absSum

Arrows indicate performance improvements:
- x2
- x4
- x3
- x2
- x5
- < 2
## Table of contents

1. Introduction and problematic
2. How to compute a correctly rounded sum?
3. Preliminary step: optimization for the sequential case
4. Parallel RTN sum implementation
5. Conclusion
Conclusion

Our paradigm
- If we are accurate enough, then we are reproducible.
- How much does it cost?

The used algorithms are convincingly
- Not depending on condition number.
- Only one pass through the input vector.
- No reuse of data, so not depending on cache.
- Suitable to shared memory and NUMA architectures.
- Scale correctly until 16 cores using hybrid parallel programming.

The use could be restricted
- RTN sum have up to 5 times overhead.
- Use on applications with no strict temporary limits.
- Use for debugging and validating steps.
Future work

- Test on a large-scale system.
- Compare to 1-Reduction algorithm (adapted for large-scale systems).
- Upgrade to BLAS level 2 and 3.
THANK YOU FOR YOUR ATTENTION