Reproducibility of sparse matrix-vector product and sparse solvers

Roman Iakymchuk\textsuperscript{1}, Daichi Mukunoki\textsuperscript{2}, Stef Graillat\textsuperscript{3}, Takeshi Ogita\textsuperscript{2}

\textsuperscript{1}KTH Royal Institute of Technology, Sweden
\textsuperscript{2}Tokyo Woman’s Christian University, Japan
\textsuperscript{3}Sorbonne University, France
riakymch@kth.se

June 27th-29th, 2018
Zürich, Switzerland
FELTOR (Full-F ELectromagnetic code in TORoidal geometry)

- Both a numerical library and a scientific software package
- 2D and 3D drift- and gyrofluid simulations
- Discontinuous Galerkin methods on structured grids
- Platform independent code from laptop CPUs to hybrid CPU+GPU distributed memory systems
Accuracy and Reproducibility Issue

- Preconditioned Conjugate Gradient (PCG) to invert elliptic equation
- The issue is with computing residual: $\dot{\otimes}(a,b)$ and $\dot{\otimes}(a,b,c)$
- But also $axpby$ and probably $spmv$
1. Computer Arithmetic

2. ExBLAS: Exact BLAS

3. Sparse Matrix-Vector Multiplication

4. Performance Results

5. Discussion and Conclusions
Floating-point arithmetic suffers from rounding errors.

Floating-point operations (+, ×) are commutative but non-associative.

\((-1 + 1) + 2^{-53} \neq -1 + (1 + 2^{-53})\) in double precision.
### Problems

- Floating-point arithmetic suffers from **rounding errors**
- Floating-point operations \((+, \times)\) are commutative but **non-associative**

\[
2^{-53} \neq 0 \quad \text{in double precision}
\]
Floating-point arithmetic suffers from rounding errors

Floating-point operations (+, ×) are commutative but non-associative

\((-1 + 1) + 2^{-53} \neq -1 + (1 + 2^{-53})\) in double precision

Consequence: results of floating-point computations depend on the order of computation

Results computed by performance-optimized parallel floating-point libraries may be often inconsistent: each run returns a different result

Reproducibility – ability to obtain bit-wise identical and accurate results from run-to-run on the same input data on the same or different architectures
Sources of Non-Reproducibility

- **Changing Data Layouts:**
  - Data partitioning
  - Data alignment

- **Changing Hardware Resources**
  - Number of threads
  - Fused Multiply-Add support: $a \cdot b + c$
  - Intermediate precision (64 bits, 80 bits, 128 bits, etc)
  - Data path (SSE, AVX, GPU warp, etc)
  - Number of processors
  - Network topology
1. Computer Arithmetic
2. ExBLAS: Exact BLAS
3. Sparse Matrix-Vector Multiplication
4. Performance Results
5. Discussion and Conclusions
Accurate/ Reproducible Summation

Existing Solutions

- **Fix the Order of Computations**
  - Sequential mode: intolerably costly at large-scale systems
  - Fixed reduction trees: substantial communication overhead
    
    Example: Intel Conditional Numerical Reproducibility in MKL
    
    ($\sim 2x$ for datum, no accuracy guarantees)

- Eliminate/Reduce the Rounding Errors
  - Fixed-point arithmetic: limited range of values
  - Fixed FP expansions with Error-Free Transformations (EFT)
    
    Example: double-double or quad-double (Briggs, Bailey, Hida, Li)

- "Infinite" precision: reproducible independently from the inputs
  
  Example: Kulisch accumulator (considered inefficient)

- Libraries
  
  ReproBLAS: Reproducible BLAS (Demmel, Nguyen, Ahrens)
  
  For BLAS-1, GEMV, and GEMM on CPUs

  RARE-BLAS: Repr. Accur. Rounded and Eff. BLAS (Chohra, Langlois, Parello). For BLAS-1 and GEMV on CPUs
Accurate/ Reproducible Summation

Existing Solutions

- **Fix the Order of Computations**
  - Sequential mode: intolerably costly at large-scale systems
  - Fixed reduction trees: substantial communication overhead
    - Example: Intel Conditional Numerical Reproducibility in MKL
      ($\sim 2x$ for datum, no accuracy guarantees)

- **Eliminate/Reduce the Rounding Errors**
  - Fixed-point arithmetic: limited range of values
  - Fixed FP expansions with Error-Free Transformations (EFT)
    - Example: double-double or quad-double (Briggs, Bailey, Hida, Li)
      (work well on a set of relatively close numbers)
  - “Infinite” precision: reproducible independently from the inputs
    - Example: Kulisch accumulator (considered inefficient)
Accurate/ Reproducible Summation

Existing Solutions

- **Fix the Order of Computations**
  - Sequential mode: intolerably costly at large-scale systems
  - Fixed reduction trees: substantial communication overhead
    Example: Intel Conditional Numerical Reproducibility in MKL
    \( \sim 2x \) for datum, no accuracy guarantees

- **Eliminate/Reduce the Rounding Errors**
  - Fixed-point arithmetic: limited range of values
  - Fixed FP expansions with Error-Free Transformations (EFT)
    Example: double-double or quad-double (Briggs, Bailey, Hida, Li)
    (work well on a set of relatively close numbers)
  - “Infinite” precision: reproducible independently from the inputs
    Example: Kulisch accumulator (considered inefficient)

- **Libraries**
  - **ReproBLAS**: Reproducible BLAS (Demmel, Nguyen, Ahrens)
    For BLAS-1, GEMV, and GEMM on CPUs
    For BLAS-1 and GEMV on CPUs
Fixed FP expansions (FPE) with Error-Free Transformations

Example: double-double or quad-double (Briggs, Bailey, Hida, Li)
(work well on a set of relatively close numbers)

Algorithm 1 (Dekker and Knuth)

\[
\text{Function } r, s = \text{twosum}(a, b)
\]

1: \( r \leftarrow a + b \)
2: \( z \leftarrow r - a \)
3: \( s \leftarrow (a - (r - z)) + (b - z) \)

Algorithm 2 (|a| ≥ |b|)

\[
\text{Function } r, s = \text{twosum}(a, b)
\]

1: \( r \leftarrow a + b \)
2: \( z \leftarrow r - a \)
3: \( s \leftarrow b - z \)
Exact Multi-Level Parallel Reduction

Preliminaries

- **Fixed FP expansions (FPE) with Error-Free Transformations**
  - Example: double-double or quad-double (Briggs, Bailey, Hida, Li)
    (work well on a set of relatively close numbers)

| Algorithm 1 (Dekker and Knuth) | Algorithm 2 \((|a| \geq |b|)\) |
|--------------------------------|--------------------------------|
| Function \([r, s] = \text{twosum}(a, b)\) | Function \([r, s] = \text{twosum}(a, b)\) |
| 1: \(r \leftarrow a + b\) | 1: \(r \leftarrow a + b\) |
| 2: \(z \leftarrow r - a\) | 2: \(z \leftarrow r - a\) |
| 3: \(s \leftarrow (a - (r - z)) + (b - z)\) | 3: \(s \leftarrow b - z\) |

- “Infinite” precision: reproducible independently from the inputs
  - Example: **Kulisch accumulator** \((=16\ FLOPs)\)
### Requirements

- IEEE 754-2008 full or partial compliance ($+, -, \times, \div, \sqrt{\cdot}$)
- Architecture support and compliance according to IEEE 754-2008 of rounding-to-nearest with breaking ties to even (correct rounding). This is a default widely used rounding mode

### Limitations

- Support for underflow numbers
- Exceptions and exception handling
Exact Multi-Level Parallel Reduction

- Parallel algorithm with 5-levels
- Suitable for today’s parallel architectures
- Based on FPE with EFT and Kulisch accumulator
- Guarantees “inf” precision

→ **bit-wise reproducibility**
Level 1: Filtering

Input numbers

Thread 1

EFT

FP Expansion (register)

Underflow?

Private SuperAccumulator

Level 2 (Private SuperAccumulation)

... ...

Level 3 (Scalar SuperAccumulation)

... ...

Level 4 (Parallel Reduction)

Level 5 (Rounding)

Input numbers

Thread 1

EFT

FP Expansion (register)

Thread 2

EFT

FP Expansion (register)
Level 2 and 3: Scalar Superaccumulator

Roman Iakymchuk (KTH)

June 27th-29th, 2018 14 / 35
Level 4 and 5: Reduction and Rounding
ExBLAS in brief

ExBLAS Status

- **ExBLAS-1**: `exsum^a`, `exscal`, `exdot`, `exaxpy`, ...
- **ExBLAS-2**: `exger`, `exgemv`, `extrsv`, `exsy`r, ...
- **ExBLAS-3**: `exgemm`, `extrsm`, `exsy`r2k, ...

^aRoutines in **blue** are already in ExBLAS
ExBLAS-1 Highlights

BLAS-1 routines

- Some are virtually built upon \texttt{exsum}
  - For instance, \texttt{exdot} = \texttt{twoprod} + 2\texttt{exsum}
  - \texttt{twoprod}(a,b) (= 3 FLOPs):
    1: \texttt{res} ← \texttt{a} \cdot \texttt{b},
    2: \texttt{err} ← \texttt{fma}(\texttt{a}, \texttt{b}, -\texttt{res})
ExBLAS-1 Highlights

**BLAS-1 routines**
- Some are virtually built upon `exsum`
  - For instance, `exdot = twoprod + 2exsum`
  - `twoprod(a,b) (= 3 FLOPs):`
    1. `res ← a · b`,
    2. `err ← fma(a, b, −res)`

**exaxpy**
- `y := α · x + y`
- `fma(α, x[i], y[i]) → correctly rounded and reproducible`

**exscal**
- `x := α · x → correctly rounded and reproducible`
- Within LU: `x := 1/α · x → not correctly rounded`
- `exinvscal: x := x/α → correctly rounded and reproducible`
SpMV: CSR format

The CSR representation of $A$

\[
A = \begin{bmatrix}
1 & 7 & 0 & 0 \\
0 & 2 & 8 & 0 \\
5 & 0 & 3 & 9 \\
0 & 6 & 0 & 4 \\
\end{bmatrix}
\]

\[
\text{ptr} = [0 \ 2 \ 4 \ 7 \ 9] \\
\text{indices} = [0 \ 1 \ 1 \ 2 \ 0 \ 2 \ 3 \ 1 \ 3] \\
\text{data} = [1 \ 7 \ 2 \ 8 \ 5 \ 3 \ 9 \ 6 \ 4]
\]

Listing 1: SpMV kernel for the CSR sparse matrix format (Bell and Garland 2008)

```c
for (int row = 0; row < num_rows; i++) {
    double dot = 0.0;
    int row_start = ptr[row];
    int row_end = ptr[row+1];
    for (int j = row_start; j < row_end; j++)
        dot += data[j] * x[indices[j]];
    y[row] += dot;
}
```
CRS-vector (Bell and Garland 2008)

- Assigns multiple threads (e.g. 32 threads) to compute a single row of the matrix $A$
- Memory access to the matrix $A$ is coalesced and thus it suits GPUs.

![CRS-vector (2 threads)]

- **thread:**
  - th = 1, 2
  - th = 3, 4
  - **th = 5, 6**
  - th = 7, 8

- **matrix $A$**

- **coalesced access**
  - $y_3 = 8x_1 + 2x_2 + 4x_3 + 9x_4$
  - th = 5 $\rightarrow$ $t_5 = 8x_1$
  - th = 6 $\rightarrow$ $t_6 = 2x_2$

- **reduction with 2 threads**
  - th = 5 $\rightarrow$ $t_5 = 4x_3$
  - th = 6 $\rightarrow$ $t_6 = 9x_4$
  - th = 5 & 6 $\rightarrow$ $y_3 = t_5 + t_6$
CRS-vector (Reguly and Giles 2012)

- Selecting the suitable number of threads (NT) in proportion to the average number of non-zeros per row
- Reduce NT if the number of non-zeros is less than 32

**Example:**

- **Nonzeros = 9**
  - Rows = 4
  - Avg nonzeros / row = 9 / 4 = 2.25
  - Optimal NT = 2

- **Nonzeros = 14**
  - Rows = 4
  - Avg nonzeros / row = 14 / 4 = 3.5
  - Optimal NT = 4
Reproducible and accurate SpMV

**exspmv in brief**
- Combine high performing algorithmic versions with `exdot`
- Invoke auto-tuning and optimization strategies

**Optimization**
- Determining the placement of long accumulators (e.g., shared memory)
- Using read-only data cache to store the vector $x$
- Avoiding outermost loop on the number of rows
- Using shuffle instructions for load/store
Parallel Reduction
Performance Scaling on NVIDIA Tesla K20c

![Graph showing performance scaling for different parallel reduction methods.](image)
Parallel Reduction
Data-Dependent Performance on NVIDIA Tesla K20c

\[ n = 67e06 \]

![Graph showing performance of different parallel FP sum methods](image)

- Parallel FP Sum
- Demmel fast
- Superacc
- FPE2 + Superacc
- FPE3 + Superacc
- FPE4 + Superacc
- FPE8 + Superacc
- FPE8EE + Superacc

Dynamic range: 1 to 1e+140

Gacc/s: 0 to 18
Dot Product
Performance Scaling on NVIDIA Tesla K20c

DDOT: $\alpha := x^T y = \sum_{i}^{N} x_i y_i$

- Based on `exsum` and `twoprod`
  - `twoprod(a, b)`
    1: $r \leftarrow a \ast b$
    2: $s \leftarrow \text{fma}(a, b, -r)$
SpMV: High performing version

Performance of CSRMM on Tesla P100-PCIE-16GB (CUDA 9.0)

matrix (from Florida Matrix Collection)

- cublasDcsrmv
- mublasDcsrmv
- mublasEXDcsrmv
- overhead (vs. mublasDcsrmv)
Feltor: \text{axpby}

\text{axpby}: y := \alpha x + \beta y

Roman Iakymchuk (KTH)
June 27th-29th, 2018 28 / 35
Feltor: dot

dot: $\alpha := x^T y = \sum_{i}^{N} x_i y_i$
Feltor: Reproducibility and Accuracy

$$\langle u_y \rangle_{t = 12000}$$

- naive #1
- naive #2
- repro #1
- repro #2

Roman Iakymchuk (KTH)
June 27th-29th, 2018
Outline

1. Computer Arithmetic
2. ExBLAS: Exact BLAS
3. Sparse Matrix-Vector Multiplication
4. Performance Results
5. Discussion and Conclusions
## Discussion

**Feltor: Reproducible PCG**

- **Missing components:** \texttt{spmv} and \texttt{nrm2}
- **But** \texttt{spmv} with their specific format

### Iterative PCG solve

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1:</td>
<td>Compute the preconditioner ( A \rightarrow M \approx LU )</td>
</tr>
<tr>
<td>S2:</td>
<td>Initialize ( x_0, r_0, z_0, d_0, \beta_0, \tau_0 )</td>
</tr>
<tr>
<td>S3:</td>
<td>( k := 0 )</td>
</tr>
<tr>
<td>S4:</td>
<td>\textbf{while} (( \tau_k &gt; \tau_{\text{max}} ))</td>
</tr>
<tr>
<td>S5:</td>
<td>( w_k := A d_k )</td>
</tr>
<tr>
<td>S6:</td>
<td>( \rho_k := \beta_k / d_k^T w_k )</td>
</tr>
<tr>
<td>S7:</td>
<td>( x_{k+1} := x_k + \rho_k d_k )</td>
</tr>
<tr>
<td>S8:</td>
<td>( r_{k+1} := r_k - \rho_k w_k )</td>
</tr>
<tr>
<td>S9:</td>
<td>( z_{k+1} := M^{-1} r_{k+1} \approx U^{-1} L^{-1} r_{k+1} )</td>
</tr>
<tr>
<td>S10:</td>
<td>( \beta_{k+1} := r_{k+1}^T z_{k+1} )</td>
</tr>
<tr>
<td>S11:</td>
<td>( \alpha_k := \beta_{k+1} / \beta_k )</td>
</tr>
<tr>
<td>S12:</td>
<td>( d_{k+1} := z_{k+1} + \alpha_k d_k )</td>
</tr>
<tr>
<td>S13:</td>
<td>( \tau_{k+1} := | r_{k+1} |_2 )</td>
</tr>
<tr>
<td>S14:</td>
<td>( k := k + 1 )</td>
</tr>
<tr>
<td>S15:</td>
<td>\textbf{endwhile}</td>
</tr>
</tbody>
</table>
IEEE 754-2018 (revised)

History

1985 was a hardware standard – hoping for hardware adoption

2008 was a meta-standard for programming languages – hardware adopted, hoping for languages

2018 is a bug fix release – catching up with C and searching for other languages

Updates

- Augmented operations $+, -, \ast$ (aka twosum and twoprod)
  - Considered but dropped from 754-2008
  - Pending hardware implementations encouraged put them back

Importance: extended-precision/ reproducible computations
Conclusions

- Leveraged a long accumulator and EFTs to design **reproducible and correctly-rounded** `exsum` and `exdot`.
- Delivered reproducible and accurate BLAS-1 routines like `axpy`, `scal`, and `invscal`.
- Designed **high performance algorithmic variants for csrmv**.
- Ensured reproducibility and accuracy of `csrmv` through `exdot`.
- Provided **bit-to-bit reproducible** results independently from:
  - Data permutation, data assignment, partitioning/blocking
  - Thread scheduling
  - Reduction trees
Conclusions and Future Work

Conclusions

- Leveraged a long accumulator and EFTs to design **reproducible and correctly-rounded** $\text{exsum}$ and $\text{exdot}$
- Delivered reproducible and accurate BLAS-1 routines like $\text{axpy}$, $\text{scal}$, and $\text{invscal}$
- Designed **high performance algorithmic variants for $\text{csr}mv$**
- Ensured reproducibility and accuracy of $\text{csr}mv$ through $\text{exdot}$
- Provided **bit-to-bit reproducible** results independently from
  - Data permutation, data assignment, partitioning/blocking
  - Thread scheduling
  - Reduction trees

TODO List

- Optimization and auto-tuning of $\text{csr}mv$
- Reproducible Jacobi and Conjugate Gradient methods
Thank you for your attention!

Publications: pdc.kth.se/~riakymch/pubs
Soon on GitHub