Correctness of Floating Point Programs: Exception Handling and Reproducibility

Jim Demmel, UC Berkeley

and many others …

Outline

- High level goals
  - Handle exceptions "consistently"
  - Get bitwise reproducible results
  - Don't cost "too much" more than "reckless" code
- Examples: where current BLAS not consistent
- Proposed BLAS Standard: consistent exception handling
- Examples: where LAPACK not consistent
- (Tentative) proposal for consistent exception handling in LAPACK
- Reproducible floating point summation, and BLAS
- Proposed update to IEEE 754 Floating Point Standard

Notation and Basic (Default) Exception Handling

- OV = overflow, UN = underflow thresholds
- NaNs (Quiet and Signalling), $\pm\infty$, ± 0
  - Operations defined, eg $1/\pm\infty = \pm 0$, 0/0 = QNaN
- Exceptions, which raise flags:
  - Invalid: SNaN + 3, $0 * \infty, \infty - \infty, \sqrt{-1}$, …
  - Divide by zero: 1 / $0 = \infty$
  - Overflow: OV * $2 = \infty$
  - Underflow: UN / 2
  - Inexact: 1/3

Inconsistent BLAS Exception Handling (1/4): ISAMAX: return index i of largest |A(i)|

- Code:

```
isamax = 1,  smax = abs(A(i))
for i = 2:n
   if (abs(A(i)) .gt. smax)  isamax = i, smax = abs(A(i))
```

- Inconsistency:
  – isamax([0, NaN, 2]) = 3
  – isamax([NaN, 0, 2]) = 1
- How to make consistent:
  – Point to NaN, if one exists, or (first) largest number?
  – We recommend NaN, reasons later
- Challenge: this (inconsistent) behavior is a standard!

Inconsistent BLAS Exception Handling (2/4): TRSV: Solve T*x = b, T triangular

- T can be upper (U) or lower (L), general or "unit" (T(i,i)=1)
- Inconsistency:
  - U1 = [1, NaN; 0, NaN], b1 = [1;0] ⇒ x1 = [1;0];
    - **NaNs do not propagate**; TRSV checks for trailing 0s in b, ignores cols of U
  - U2 = [1, NaN, 1; 0, 1, 1; 0, 0, 1], b2 = [2;1;1] ⇒ x2 = [1; 0; 1]
    - **NaNs do not propagate**; TRSV checks for 0s in x, does not multiply by them
  - L = U1^T, b = b1; solve L^T*x=b (**same as 1st example**) ⇒ x = [NaN; NaN]
    - **TRSV does not check for zeros in this case**
- How to make consistent: Depends on what NaN means
  - If NaN means some finite number, 0*NaN = 0 is ok
  - If NaN means "anything", 0 * NaN = NaN (IEEE 754 rules)
  - We recommend latter
- Challenge: this (inconsistent) behavior is a standard!
  - And potentially much faster, O(n) vs O(n^2), sometimes

Inconsistent BLAS Exception Handling (3/4): GER: $A = A + \alpha * x * y^T$

- Inconsistency:
  - If $\alpha = 0$, return A, ignore x and y, so ∞ or NaN in x or y do not propagate
  - If $y(i) = 0$, do not multiply by it, so if α or $x(j)$ is ∞ or NaN, it does not propagate
  - No checking for $x(j) = 0$, so if $y(i) = \infty$ or NaN, it propagates
- How to make consistent
  - Keep check for $\alpha = 0$, but not for $y(i) = 0$ (or $x(j) = 0$)
- Challenge: this (inconsistent) behavior is a standard!
  - And potentially much faster, O(n) vs O(n^2), sometimes

Inconsistent BLAS Exception Handling (4/4): GEMM: $C = \alpha * A * B + \beta * C$

- Inconsistency:
  - If $\alpha = \beta = 0$, return $C = 0$, do not propagate ∞ or NaN in A, B, C
  - If only $\beta = 0$ or $\alpha = 0$, analogous
  - Reference GEMM checks for $B(i, j) = 0$, but not A
  - Depending on summation order, may or may not get exceptions
    - Summing [OV, OV, -OV, -OV] could yield $0, +\infty, -\infty$ or NaN
- How to make consistent
  - Keep checks for $\alpha, \beta = 0$, expected by users
  - Provide reproducible BLAS (exceptions reproducible too)
- Challenge:
  - Vendors will tune GEMM as they see fit
  - Permit users to opt for reproducibility/consistency, at a cost

Inconsistent Language Exception Handling: Across programming languages/compilers

- Inconsistency: Complex multiply in C vs Fortran
  - $x = (\infty + i * 0),\ y = (\infty + i\ *\ \infty)$
  - Fortran: standard formula, yields $x * y =$ (NaN + $i *$ NaN)
  - C99 and C11: require $x * y = (\infty + i\ *\ \infty)$
    - 30+ line implementation provided
    - Similar rules, but no implementation, for division
- Inconsistency: max/min in different compilers
  - May or may not propagate NaNs
- How to make consistent
  - Outside our scope, live with it
- Challenge:
  - Devise ”consistent exception handling” rules that are flexible enough to accommodate such divergences

High Level “consistent” Exception Handling Goals

- If NaNs or ∞s are provided as inputs, or created while running, then
  1. The program will still terminate
     - Undecidable in general, we refer to constructs that can fail if a NaN appears, but are assumed to terminate otherwise, like

       repeat … until (error < tolerance)
  2. Either
     - NaNs and ∞s propagate to the output in some way (either in a floating point output, or flag) so they are not “lost,” or
     - They are dealt with explicitly by the programmer, or
     - There are some simple, well-documented, “user-approved” cases where they do not propagate (ex: C = 0*A*B +0*C)
- Long term goal: Tools to help automate analysis

BLAS Standard Committee

- Reconvened in 2016 to address needs for updating BLAS to include reduced/mixed precision, batched, reproducible versions
  - bit.ly/Batch-BLAS-2017
- Led to discussions about error reporting and exception handling more generally
- Meant to be in addition to existing BLAS interface, not replace it
- Draft on-line, comments welcome
  - http://goo.gl/D1UKnw
- See also talk by Jason Riedy
  - BoF: Batched, Reproducible, and Reduced Precision BLAS
  - Wednesday, Nov 14, 12:15 – 1:15, Room C155/156

Design for exception handling recommended by BLAS Standard Committee (1/2)

- Meant to augment use of existing IEEE 754 exception flags, doesn't depend on them
- Too expensive to check all inputs and outputs for NaNs/∞s, but provide extra routines for checking this, that users could call
- Ok to check for and exploit zero scalars (eg $\alpha = \beta = 0$ in $C = \alpha * A * B + \beta * C$), but not zero entries of arrays

Design for exception handling recommended by BLAS Standard Committee (2/2)

- Make sure NaNs/∞s "propagate to output"
  - ISAMAX: should point to first NaN, else ∞, else largest finite number
    - Small oops: ICAMAX uses |Re(A(i))| + |Im(A(i))|, so if two overflow, get first, even if second correct
  - TRSV: no 0 checking in arrays ⟹ NaN or ∞ propagates
    - Possible significant performance loss
  - GER, GEMM:
    - Ok to check $\alpha = 0$ or $\beta = 0$, expected by users
    - No 0 checking in arrays ⟹ NaN or ∞ propagates
  - C vs Fortran semantics for multiply
    - Either NaN or ∞ may propagate, ok

What about Sca/LAPACK?

- Are BLAS with consistent exception handling sufficient, or just necessary, for consistency in higher level libraries?
- LAPACKE: C interface for LAPACK
  - High level interface adds optional, on-by-default checking all input arguments for NaNs, return with error flag
- Mathworks
  - Wants NaN propagation at "matrix level", not necessarily at each entry
  - May return error message if NaN or ∞ inputs
    - yes for eig, not necessarily for A\b

Inconsistent LAPACK Exception Handling (1/3): SGESV: Solve Ax=b

- Factor A = L*U, solve L*y=b, U*x=y
- A = [1,0; NaN, 2], b = [0,1]
- Combine previous BLAS inconsistencies:
  - **ISAMAX** chooses A(1,1)=1 as pivot, not A(2,1)=NaN
  - **GER** does not propagate NaN to A(2,2), so L = [1,0; NaN,1], U = [1,0; 0,2]
  - **TRSV** does not propagate NaN, so solution of L*y=b is y=b
  - Solution of U*x=y is x=[0;.5]
- **NaN does not propagate**

Inconsistent LAPACK Exception Handling (2/3): SGEEV: eig(A)

- A = [1, NaN; 0, 2]
- SGEEV recognizes this is a triangular matrix, so just returns diagonal entries 1,2
  - NaN does not propagate, same if ∞
- What do/should users expect?
- Matlab returns warning, that input has NaN or ∞

~~In~~consistent LAPACK Exception Handling (3/3): SSTEMR: counting eigenvalues

- SSTEMR counts the number of eigenvalues of a symmetric tridiagonal matrix T that are < shift
- D = diagonal entries of T, E = off diagonals
- Inner loop:

  pivot = (D(i+1) – shift) – E(i)**2/pivot

  if (pivot .le. 0) count = count + 1
- If some pivot = 0 (or tiny enough), next pivot $=-\infty$, next pivot = D(i+1)-shift, …
- Proven to be correct, much faster than checking to avoid division by zero / overflow
- No reason to propagate such exceptions
- See LAWN 172 or doi.org/10.1137/050641624 for details

Recall: High Level “consistent” Exception Handling Goals

- If NaNs or ∞s are provided as inputs, or created while running, then
  1. The program will still terminate
     - Undecidable in general, we refer to constructs that can fail if a NaN appears, but are assumed to terminate otherwise, like

       repeat … until (error < tolerance)
  2. Either
     - **NaNs and ∞s propagate to the output in some way (either in a floating point output, or flag) so they are not “lost,”** or
     - They are dealt with explicitly by the programmer, or
     - There are some simple, well-documented, “user-approved” cases where they do not propagate

(Tentative) proposal for LAPACK Exception Handling (1/3)

- Idea could apply to other libraries too
- Use one parameter (INFO) to report problems
- Current uses of INFO:
  - INFO = 0 means successful exit (common case)
  - INFO = -3 means 3rd parameter is "wrong", ex: negative dimension, return immediately
  - INFO = k > 0 means some numerical problem, ex: k eigenvectors failed to converge

(Tentative) proposal for LAPACK Exception Handling (2/3)

- Add error conditions INFO could signal
- Prioritize error conditions, signal “most important”:
  1. INFO = -k if input argument k is “wrong”, eg negative dimension (choose smallest k, return immediately)
  2. **INFO = -k, if input argument k contains NaN or ∞ (only when algorithm needs to return immediately)**
  3. INFO > 0, eg convergence failure
  4. **INFO = -k, if input argument k contains NaN or ∞ (and didn’t return immediately)**
  5. **INFO = some unique positive value, points to first output argument containing NaN or ∞**
  6. **INFO = some unique positive value, if exception only occurred internally (eg some subroutine call), to indicate where it (first) happened**
- Goal: should be implementable “bottom up”, starting with routines that don’t call any other routines
- Wish: tools to help automate this, eg analyzing whether a code section will/will not/might propagate an exceptional value
  - Abstract Interpretation

(Tentative) proposal for LAPACK Exception Handling (3/3)

- Example: Solving Ax=B with

  SGESV(N, NRHS, A, LDA, IPIV, B, LDB, INFO)

  1. INFO = -1 if N<0 (current)
  2. INFO = -2 if NRHS < 0 (and INFO not already set, current)
  3. INFO = -4 if LDA < max(1,N) (ditto)
  4. INFO = -7 if LDB < max(1,N) (ditto)
  5. INFO = $k, 1 \leq k \leq$ N, if k is first zero pivot (ditto)
  6. INFO = -3 if A contains NaN/∞ on input (and INFO not set, **new**)
  7. INFO = -6 if B contains NaN/∞ on input (ditto)
  8. INFO = N+3 if A contains NaN/∞ on output (ditto)
  9. INFO = N+6 if B contains NaN/∞ on output (ditto)
  10. INFO = N+7 if SGETRF reports a NaN/∞ (ditto)
  11. INFO = N+8 if SGETRS reports a NaN/∞ (ditto)

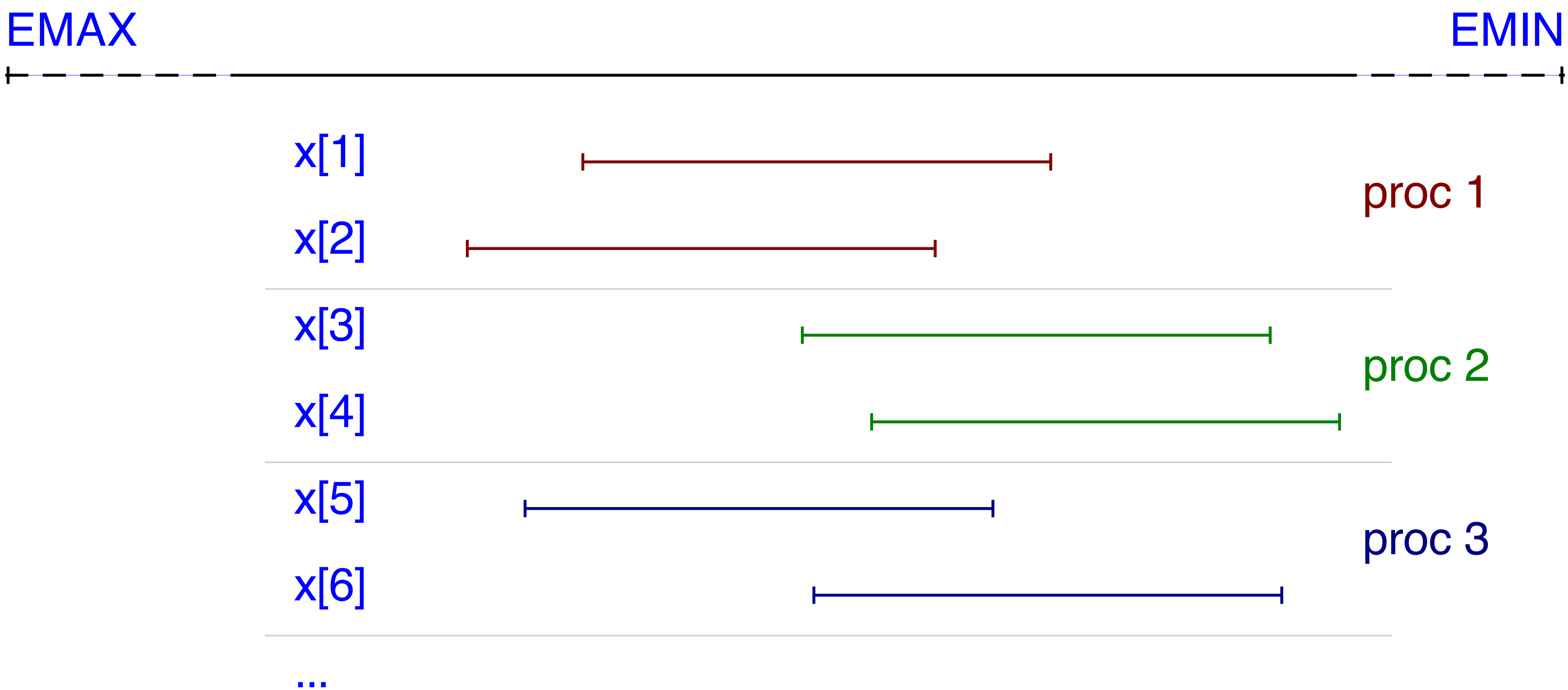
Reproducibility

- Motivation for reproducibility
  - Debugging, correctness, contractual requirements …
  - https://gcl.cis.udel.edu/sc15bof.php
- Reproducible floating point summation
  - Challenge: addition not associative, summation order can vary because of parallelism, changing hardware resources
  - Goal: bitwise reproducible summation
- IEEE 754 Floating Point Standard Committee
  - Voted to add new instructions to support both higher precision arithmetic and reproducible summation

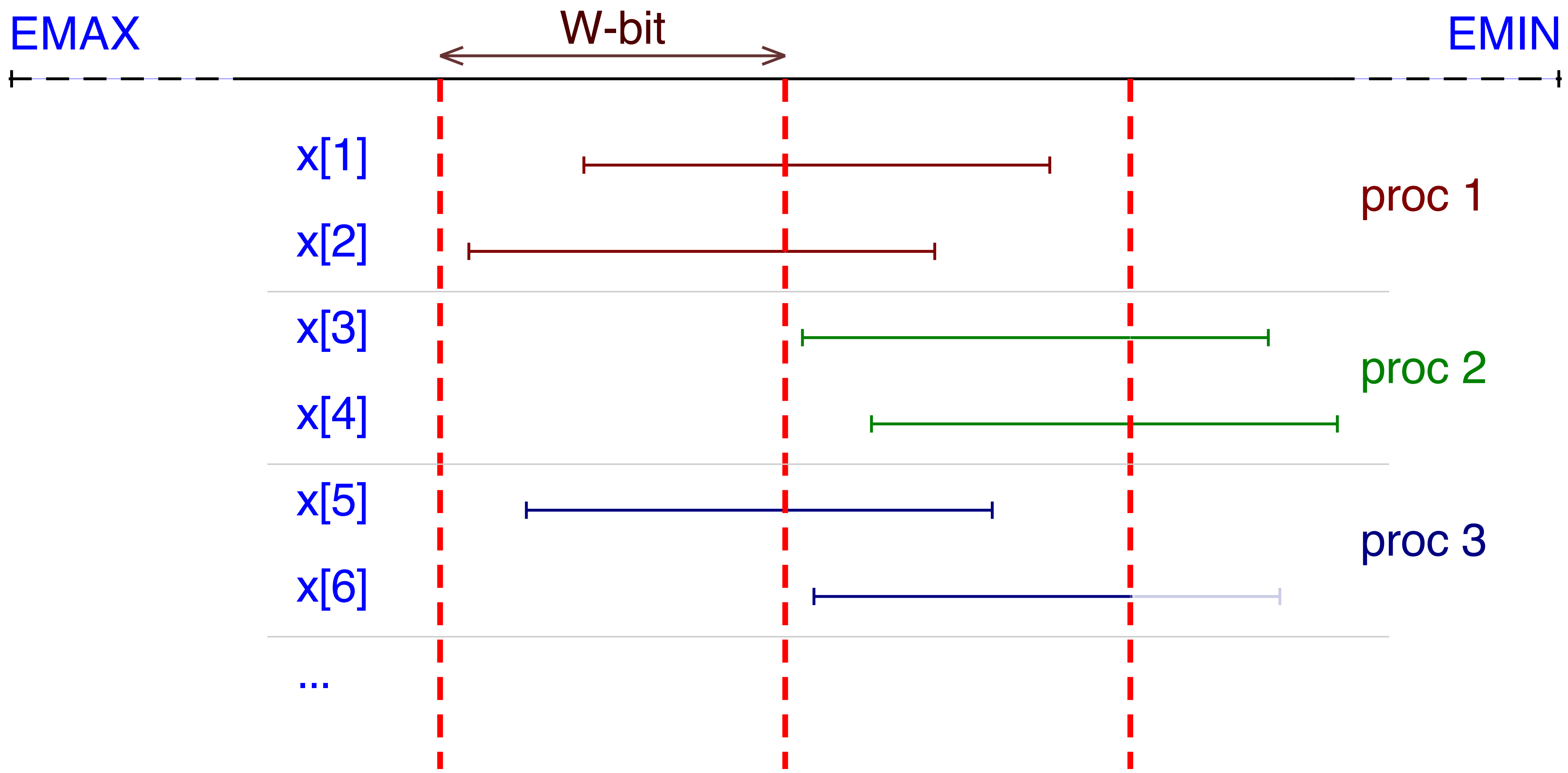
Design Goals for Reproducible Sum (high level)

1. Reproducible sum, independent of order, assuming a subset of IEEE 754
   – Limits: #summands at most 2^{64} in double, 2^{33} in single
2. Accuracy at least as good as conventional, and tunable
   – Default: 80 bit accuracy
3. Handle exceptions reproducibly
4. One read-only pass over summands
5. One reduction
6. Use as little memory as possible, to enable tiling BLAS
   – Default: one "reproducible accumulator" is 6 floats
7. Modular design, for various use cases

Binned Summation



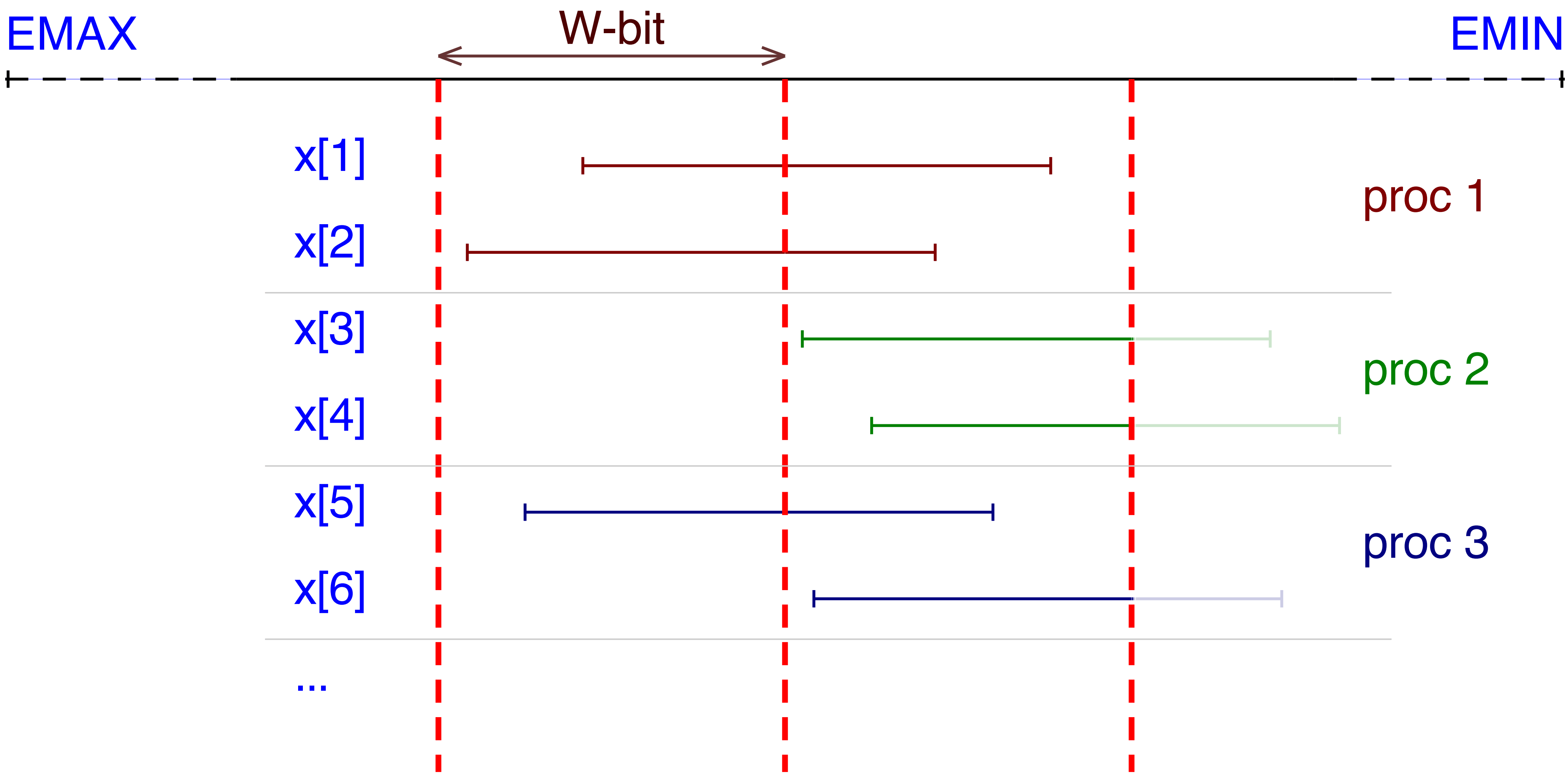
Binned Summation



- Boundaries predetermined

K = 2 bins

Binned Summation



- Only keep top K bins, don't compute, or discard, the rest

Inner loop: “Two-Sum”

- Update one bin, obtain remainder for next one
- Using current IEEE 754: Uses round-to-nearest-even (ToEven)
  - h = ToEven(x + (y|1)) … last bit of y set to 1
  - t = (x-h)+y … exact error
- Needed property for reproducibility:
  tie-breaking direction independent of mantissa of x
  - Above formula breaks ties in direction sign(y)
- Proposed new instruction in IEEE 754:
  - h = ToZero(x+y) … round-to-nearest-ties-toward-zero
  - t = (x-h)+y … exact error
  - Standard also includes Two-Subtract and Two-Multiply

Error Bounds in Double (W=40,K=3)

- Notation:
  - $T = \Sigma_i \, x(i)$, S = computed sum, $M = \max_i |x(i)|$, $\varepsilon = 2^{-53}$
- New error bound:

  $$| S - T | \le n \, 2^{-80} M + 7 \varepsilon |T|$$

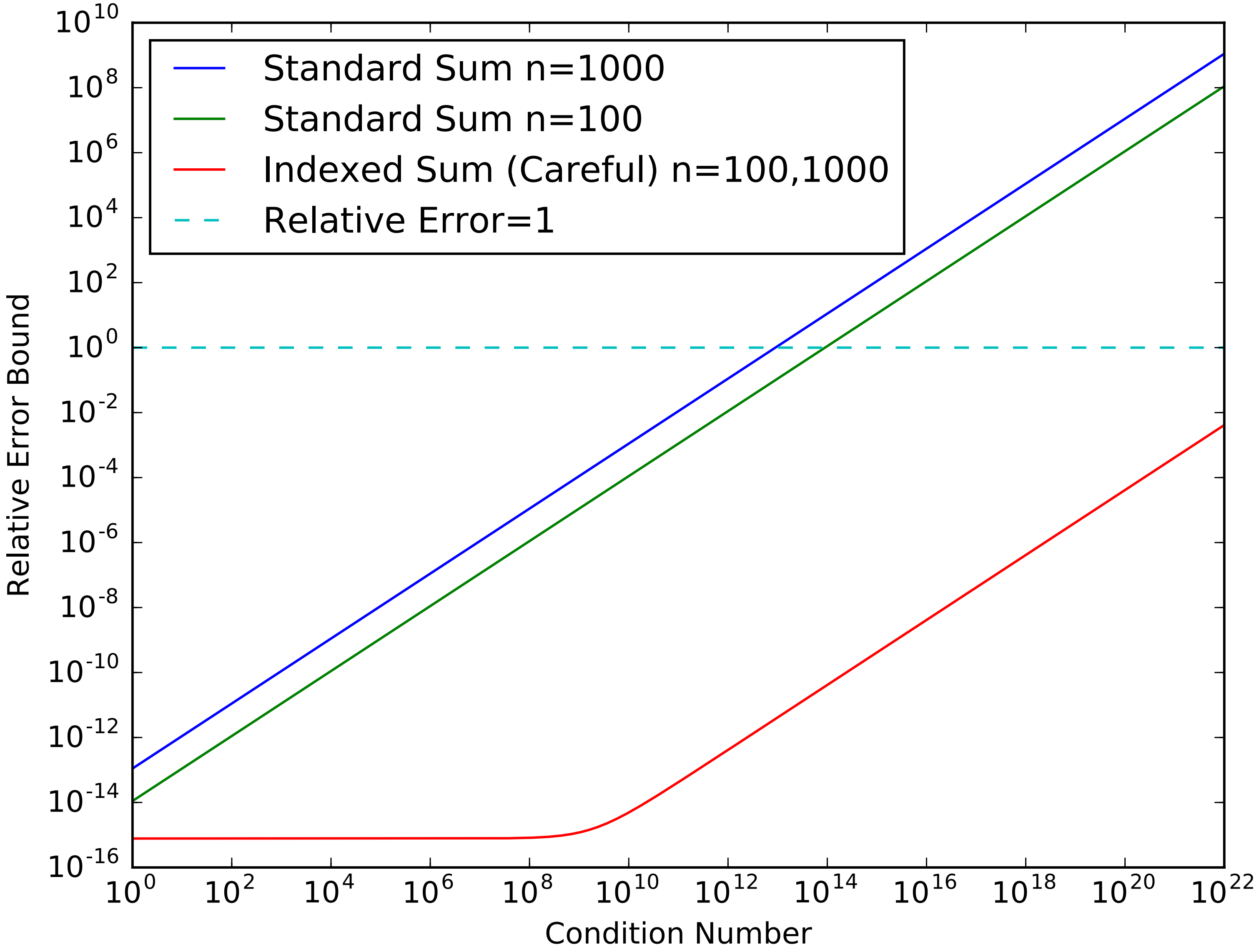
- Standard error bound:

  $$| S - T | \le n \varepsilon \Sigma_i | x(i) | \le n^2 \varepsilon M$$

- New bound up to 10^8 x smaller when lots of cancellation ($|T| << M$)

Relative Error Comparison

Condition number = $n*\max_i |x(i)| / | \Sigma_i\ x(i) |$



Cost to sum n numbers reproducibly

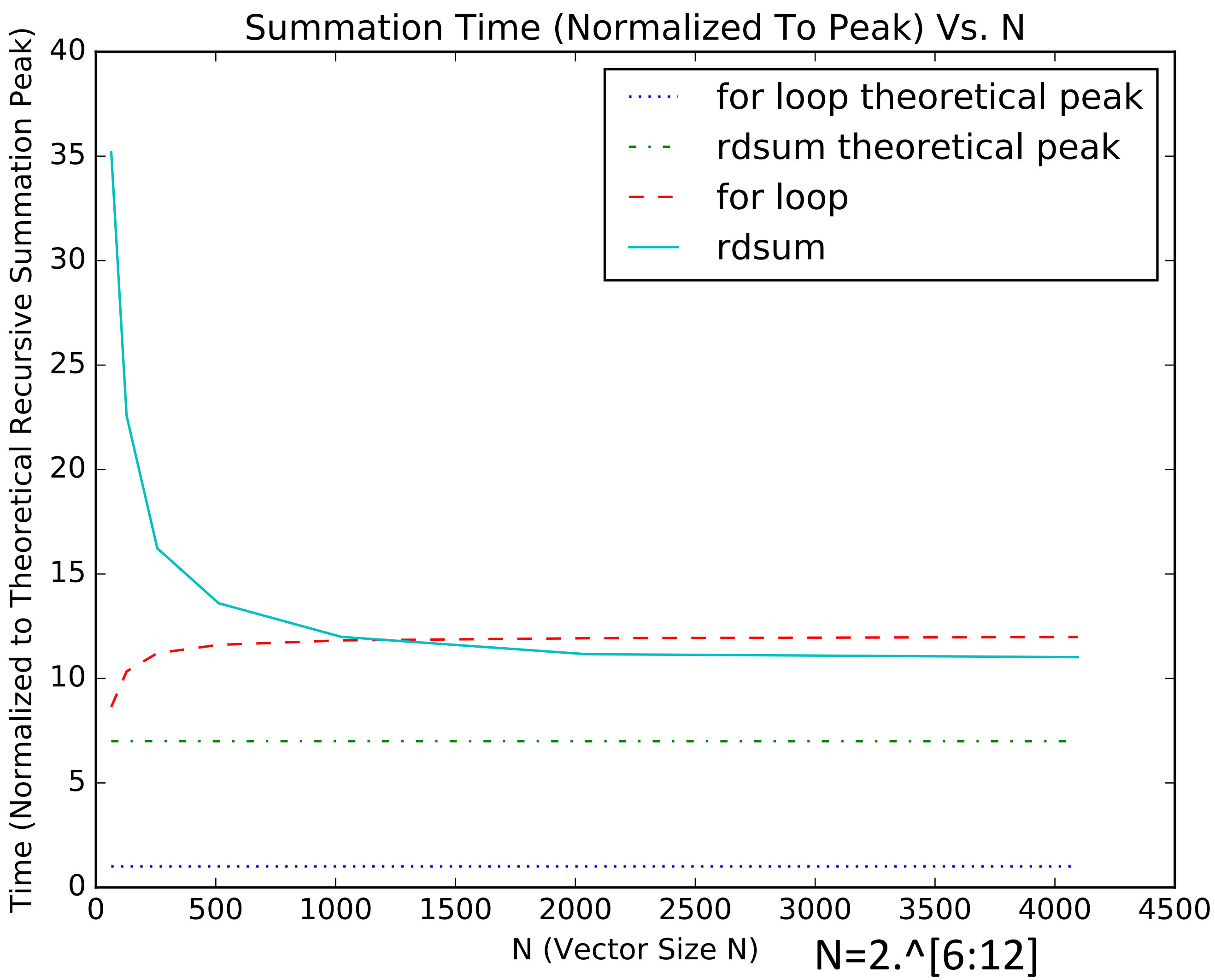
- Count arithmetic only
  - Using existing 754 standard: 7n ops
  - Using new operation: 3n ops
    - Counting (h,t) = x+y as one operation
- Additional common operations:
  - n abs, n max
- Same cost (plus O(1)) for higher precision too
  - Needs larger reproducible accumulator

Summation Performance

- Compare to gcc –O3 applied to:

  res=0; for (j=0; j<N; j++) { res += X[j]; }

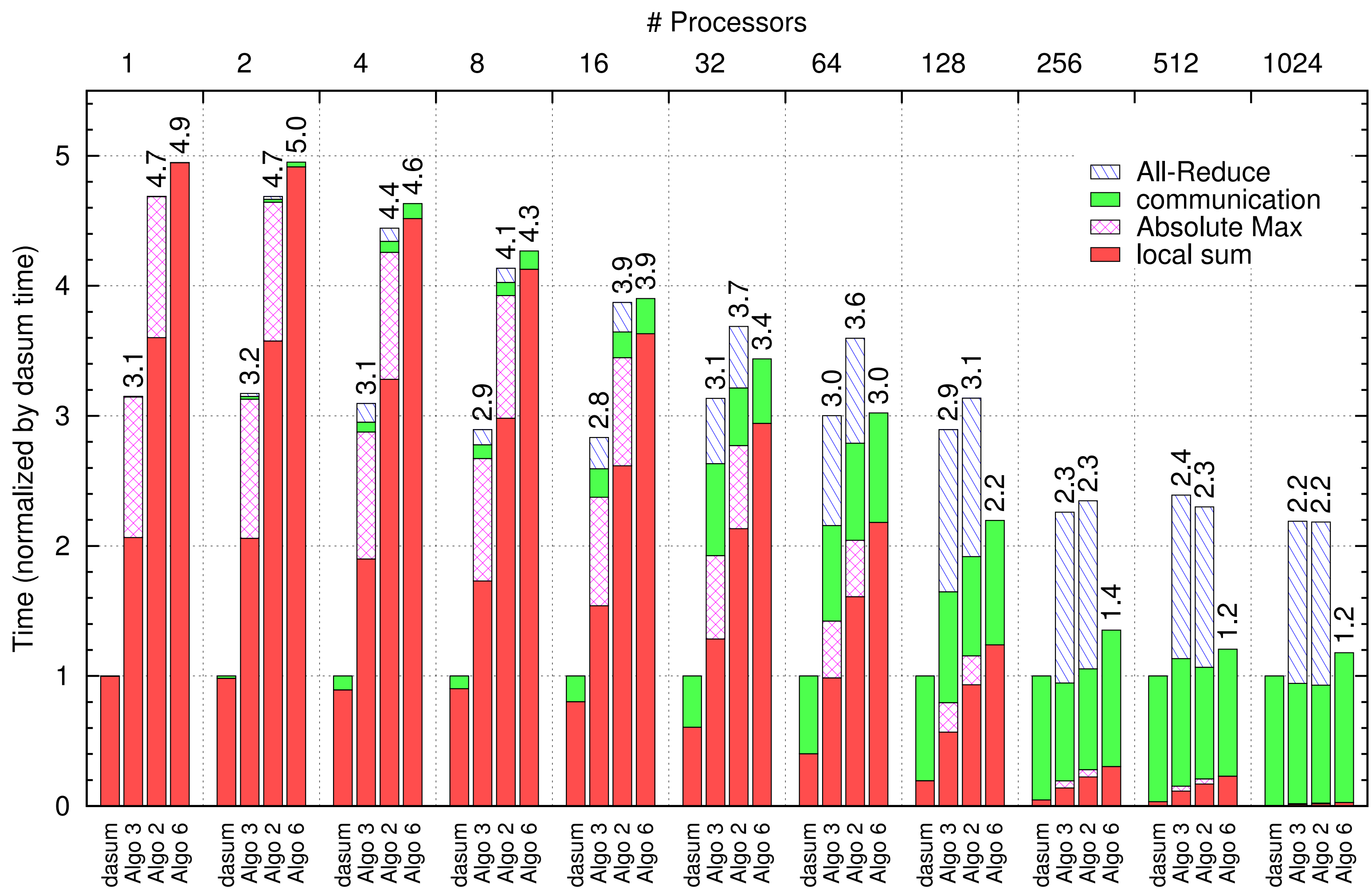
- Reproducible sum faster for large N !



N=2.^[6:12]

Performance results on 1024 proc Cray XC30

1.2x to 3.2x slowdown vs fastest (nonreproducible) code dasum data for n=1M summands on up to p=1024 processors
3 reproducible sum algorithms compared, best one depends on n, p
code and papers at bebop.cs.berkeley.edu/reproblas



Future Work

- BLAS Standard
  - Get community input
  - Converge on final version
  - Reference implementations (using current IEEE 754 standard)
- Reproducible summation
  - Shepherd 754 standard through voting process
  - Provide reference (software) implementation of new operations
  - Use to provide new ReproBLAS
    - Partial implementation at bebop.cs.berkeley.edu/reproblas
- Consistent exception handling for other libraries
  - Sca/LAPACK, ,…
  - Develop tools to help automate consistency analyses: Abstract Interpretation?

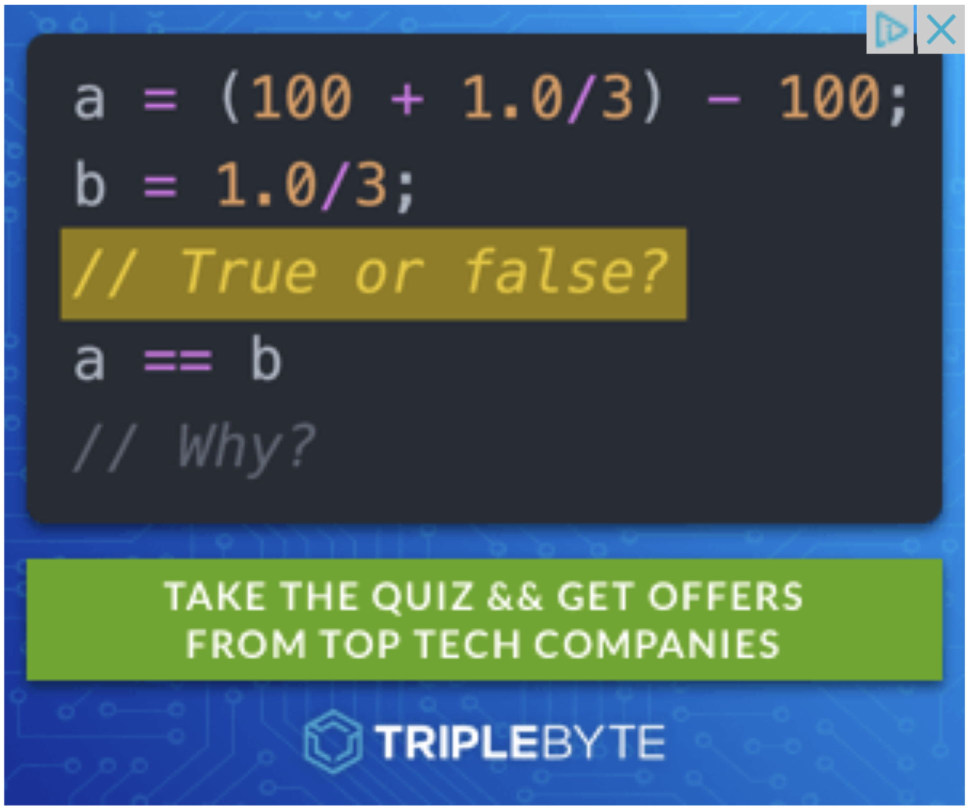
References / Other presentations

- BoF: Batched, Reproducible, and Reduced Precision BLAS, Wednesday, Nov 14, 12:15 – 1:15, C155/156
- Draft BLAS Standard – comments welcome
  http://goo.gl/D1UKnw
- Previous BLAS Standard meetings:
  http://bit.ly/Batch-BLAS-2017
- Reproducibility papers and software:
  http:/bebop.cs.berkeley.edu/reproblas/
- SC15 BoF on Reproducibility:
  – https://gcl.cis.udel.edu/sc15bof.php
- SIAM News article, Oct 2018
  – Reproducible BLAS: Make Addition Associative Again!

Collaborators

- Reproducible summation + ReproBLAS
  - Peter Ahrens, Hong Diep Nguyen
  - Wen Rui Liau, Swapnil Das, James Park
- IEEE 754 Standards Committee
  - Chair: David Hough
  - Jason Riedy
- BLAS Standards Committee
  - Chair: Jack Dongarra
  - Mark Gates, Greg Henry, Xiaoye Li, Jason Riedy, Peter Tang

Want a job?


a = (100 + 1.0/3) - 100;
b = 1.0/3;
// True or false?
a == b
// Why?
TAKE THE QUIZ && GET OFFERS
FROM TOP TECH COMPANIES
TRIPLEBYTE