Reproducibility of sparse matrix-vector product and sparse solvers

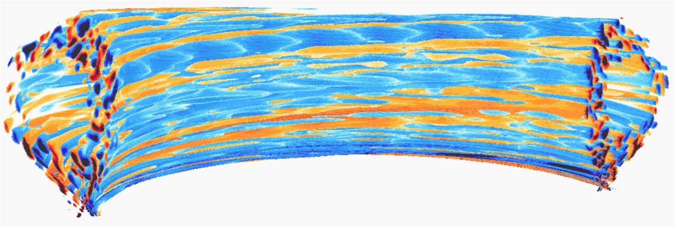
Roman Iakymchuk[1], Daichi Mukunoki[2], Stef Graillat[3], Takeshi Ogita[2]

[1]KTH Royal Institute of Technology, Sweden
[2]Tokyo Woman's Christian University, Japan
[3]Sorbonne University, France
riakymch@kth.se

June 27th-29th, 2018
Zürich, Switzerland

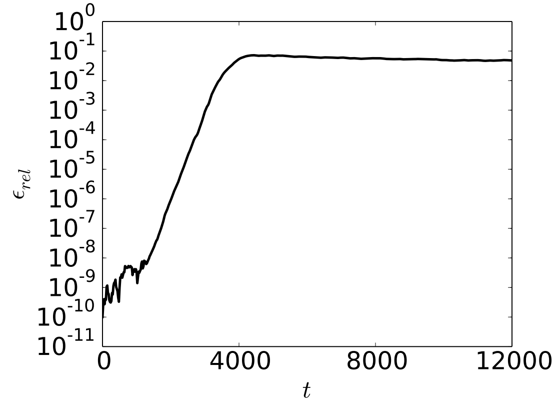
Motivation (1/2)

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

Motivation (2/2)



Accuracy and Reproducibility Issue

- Preconditioned Conjugate Gradient (PCG) to invert elliptic equation
- The issue is with computing residual: `dot`(a,b) and `dot`(a,b,c)
- But also `axpby` and probably `spmv`

Outline

Outline

Computer Arithmetic

Problems

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

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

Problems

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

$$2^{-53} \neq 0 \quad \text{in double precision}$$

Computer Arithmetic

Problems

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

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

- 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

Outline

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

- **Fix the Order of Computations**
  - Sequential mode: intolerably costly at large-scale systems
  - Fixed reduction trees: substantial communication overhead
    Example: Intel **C**onditional **N**umerical **R**eproducibility 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 **C**onditional **N**umerical **R**eproducibility 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
  - RARE-BLAS: Repr. Accur. Rounded and Eff. BLAS (Chohra, Langlois, Parello). 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)

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

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

Algorithm 2 ($|a| \geq |b|$)

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

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

Exact Multi-Level Parallel Reduction

- 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)

Function$[r, s]$ = `twosum`(a, b)

$r \leftarrow a + b$

$z \leftarrow r - a$

$s \leftarrow (a - (r - z)) + (b - z)$

Algorithm 2 $(|a| \geq |b|)$

Function$[r, s]$ = `twosum`(a, b)

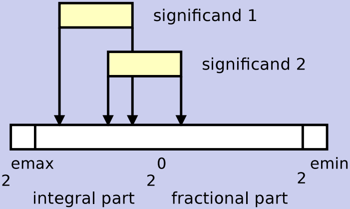
$r \leftarrow a + b$

$z \leftarrow r - a$

$s \leftarrow b - z$

- "Infinite" precision: reproducible independently from the inputs

→ Example: Kulisch accumulator (=16 FLOPs)



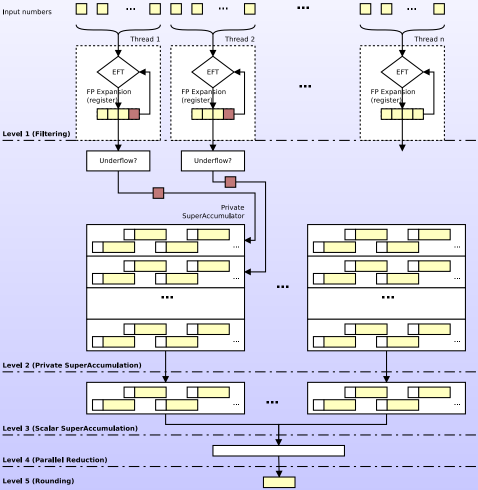
Requirements

- IEEE 754-2008 full or partial compliance $(+, -, *, /, \sqrt{})$
- 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

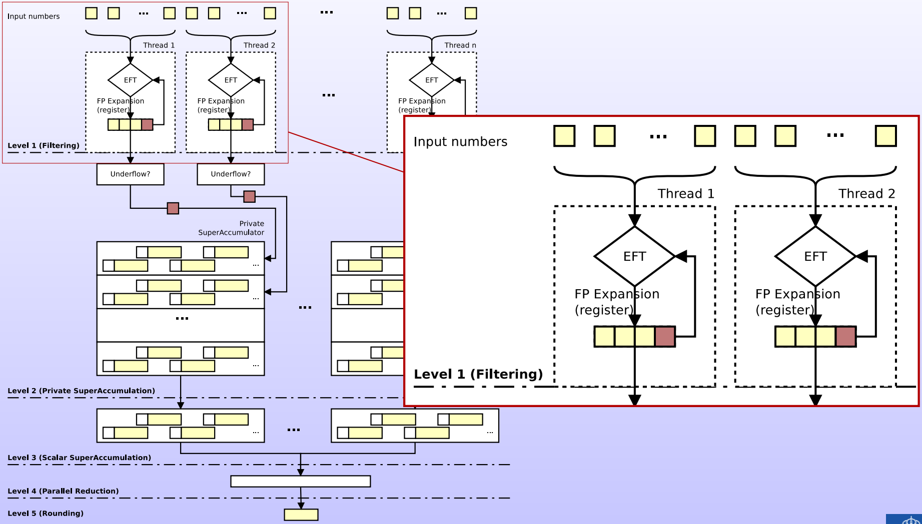
Exact Multi-Level Parallel Reduction



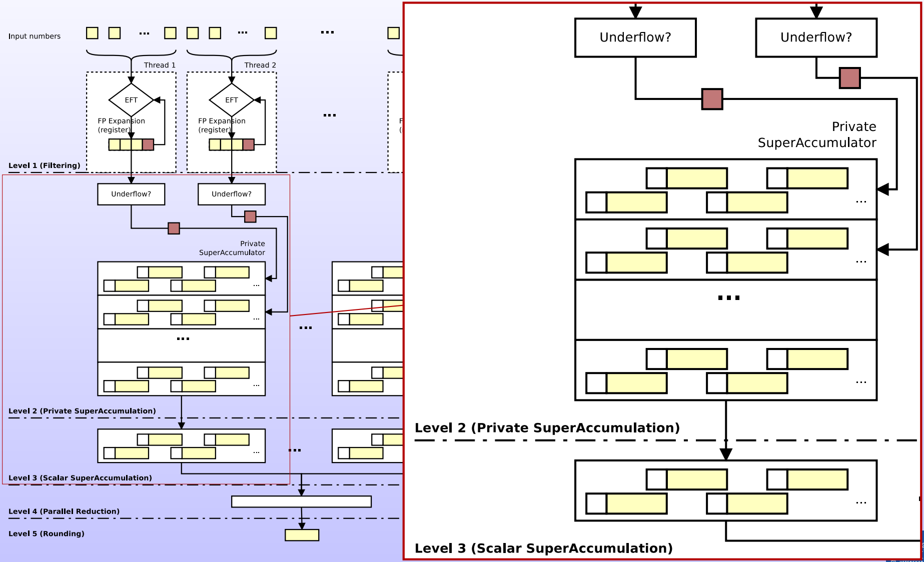
- Parallel algorithm with 5-levels
- Suitable for today's parallel architectures
- Based on FPE with EFT and Kulisch accumulator
- Guarantees "inf" precision

→ **bit-wise reproducibility**

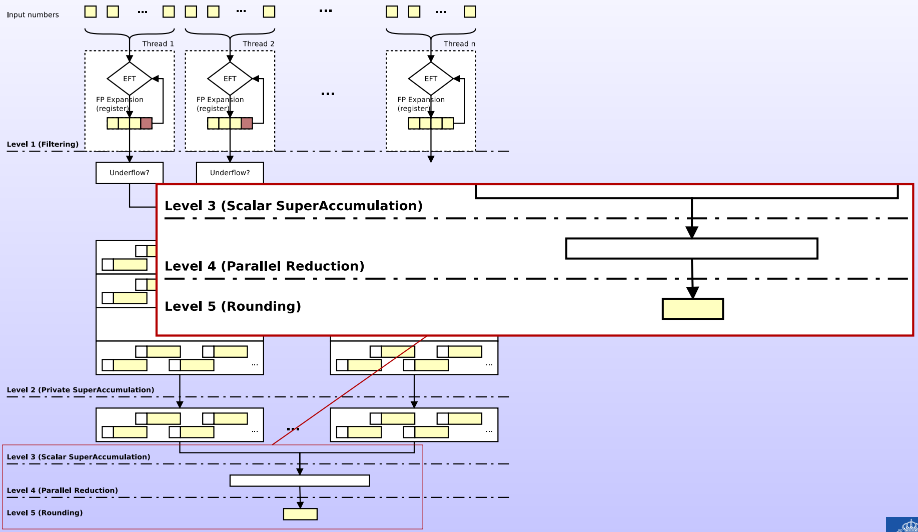
Level 1: Filtering



Level 2 and 3: Scalar Superaccumulator



Level 4 and 5: Reduction and Rounding



ExBLAS in brief

ExBLAS Status

- ExBLAS-1: `exsum`[a], `exscal`, `exdot`, `exaxpy`, ...
- ExBLAS-2: `exger`, `exgemv`, `extrsv`, `exsyr`, ...
- ExBLAS-3: `exgemm`, `extrsm`, `exsyr2k`, ...

[a] Routines in blue are already in ExBLAS

ExBLAS-1 Highlights

BLAS-1 routines

- Some are virtually built upon `exsum`

→ For instance, `exdot` = `twoprod` + 2`exsum`

→ `twoprod`(a,b) (= 3 FLOPs):

1: $res \leftarrow a \cdot b$,
2: $err \leftarrow \texttt{fma}(a, b, -res)$

ExBLAS-1 Highlights

BLAS-1 routines

- Some are virtually built upon `exsum`

→ For instance, `exdot` = `twoprod` + 2`exsum`

→ `twoprod`(a,b) (= 3 FLOPs):

1: $res \leftarrow a \cdot b$,
2: $err \leftarrow \texttt{fma}(a, b, -res)$

exaxpy

- $y := \alpha \cdot x + y$
- $\texttt{fma}(\alpha, x[i], y[i]) \rightarrow$ correctly rounded and reproducible

exscal

- $x := \alpha \cdot x \rightarrow$ correctly rounded and reproducible
- Within LU: $x := 1/\alpha \cdot x \rightarrow$ not correctly rounded
- `exinvscal`: $x := x/\alpha \rightarrow$ correctly rounded and reproducible

Outline

SpMV: CSR format

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

$$\begin{aligned} \texttt{ptr} &= \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix} \\ \texttt{indices} &= \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix} \\ \texttt{data} &= \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix} \end{aligned}$$

The CSR representation of A

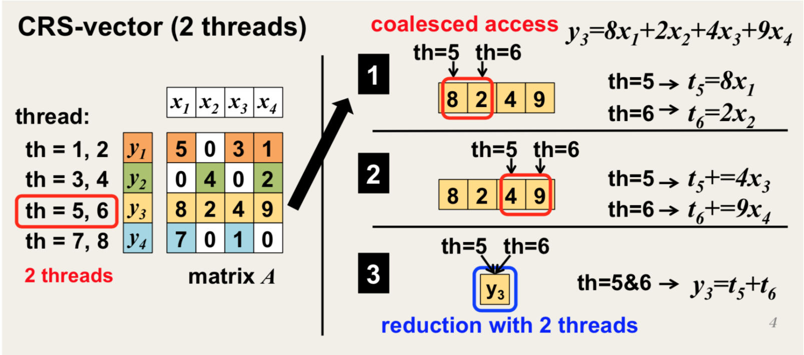
Listing 1: SpMV kernel for the CSR sparse matrix format (Bell and Garland 2008)

```
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[indeces[j]];

   y[row] += dot;
}
```

SpMV on GPUs (1/2)

CRS-vector (Bell and Garland 2008)

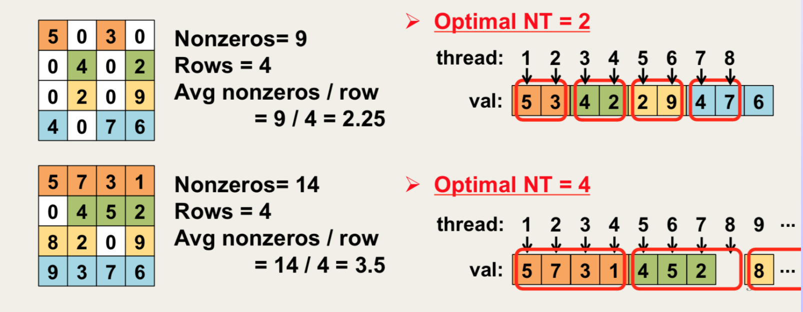
- Assigns multiple threads (e.g. 32 threads) to compute a single row of the matrix A
- Memory access to the matrix A is coalesced and thus it suites GPUs



SpMV on GPUs (2/2)

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



Reproducible and accurate SpMV

`exspmv` in brief

- Combine high performing algorithmic versions with `exdot`
- Invoke auto-tuning and optimization strategies

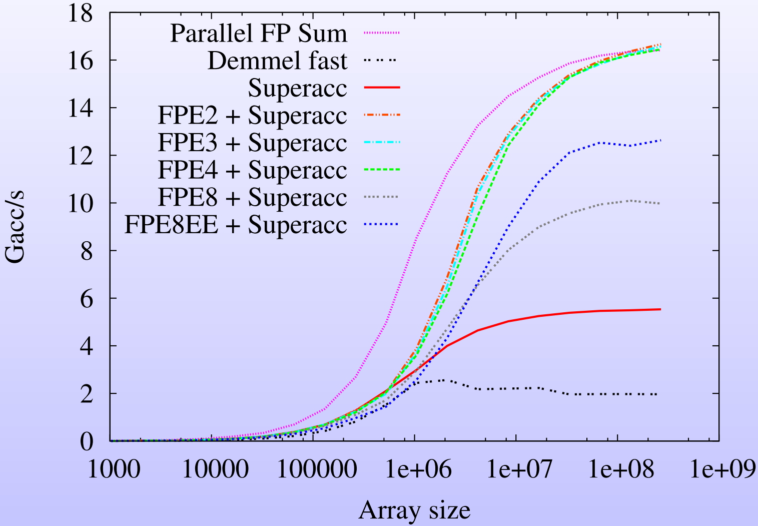
Optimization

- Determining the placement of long accumulators (eg 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

Outline

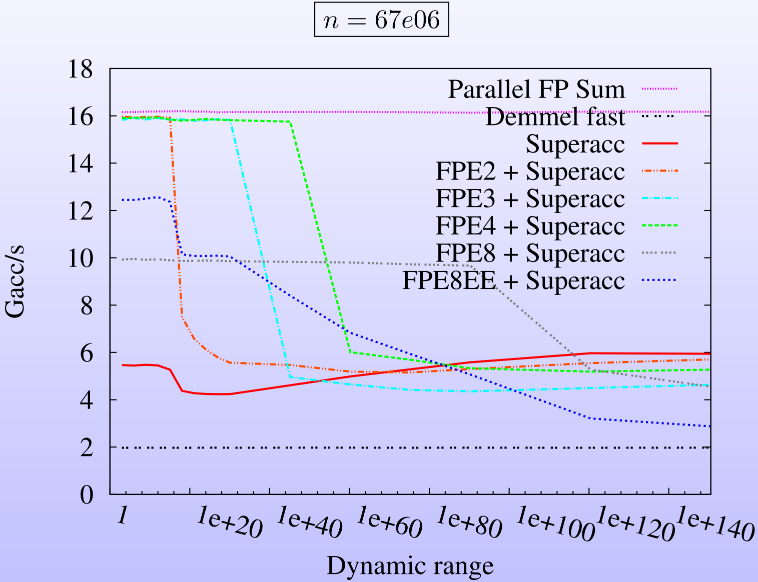
Parallel Reduction

Performance Scaling on NVIDIA Tesla K20c



Parallel Reduction

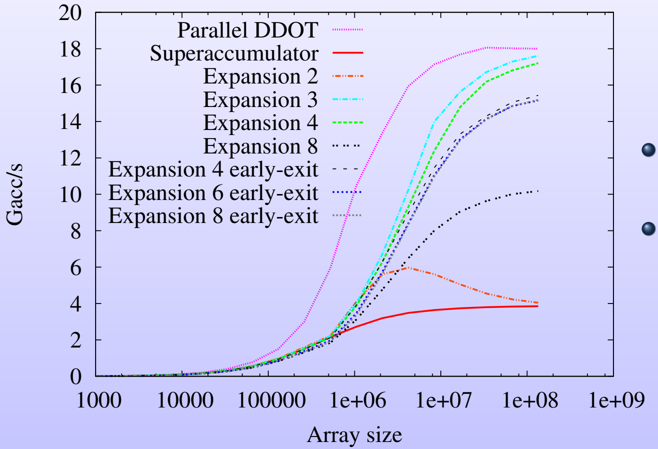
Data-Dependent Performance on NVIDIA Tesla K20c



Dot Product

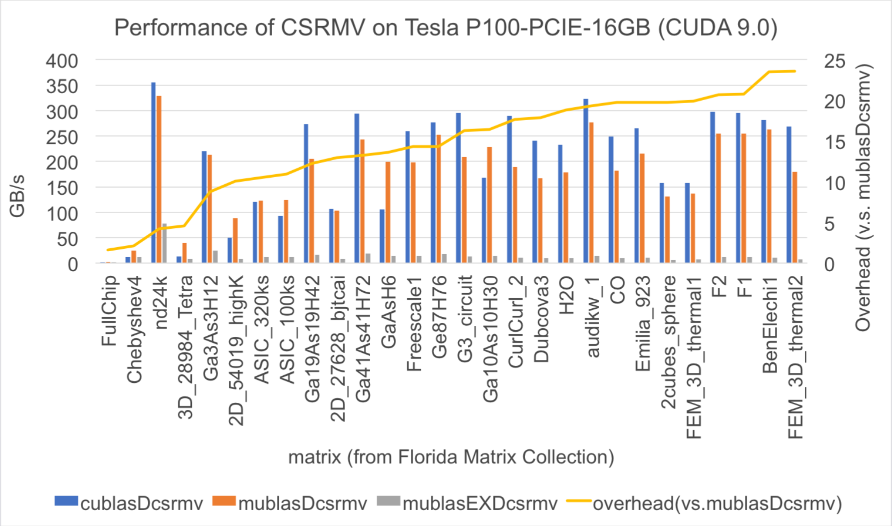
Performance Scaling on NVIDIA Tesla K20c

DDOT: $\alpha := x^T y = \sum_i^N x_i y_i$

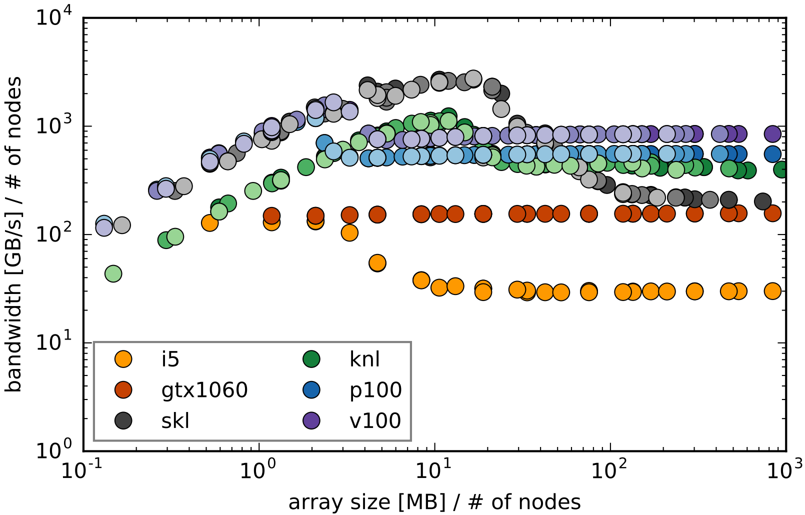


- Based on `exsum` and `twoprod`
- `twoprod`(a, b)
  1: $r \leftarrow a * b$
  2: $s \leftarrow$ `fma`$(a, b, -r)$

SpMV: High performing version

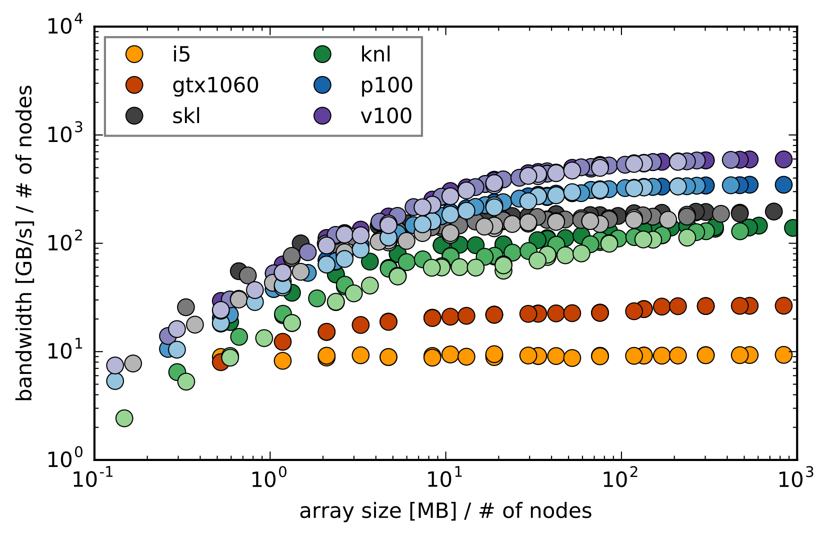


axpby: $y := \alpha x + \beta y$

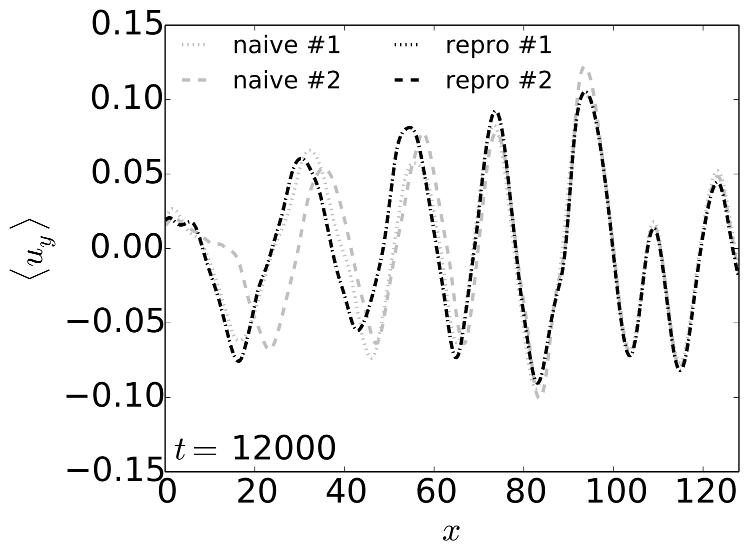


Feltor: dot

dot: $\alpha := x^T y = \sum_i^N x_i y_i$



Feltor: Reproducibility and Accuracy



Outline

Discussion

S1: Compute the preconditioner $A \to M \approx LU$	
S2: Initialize $x_0, r_0, z_0, d_0, \beta_0, \tau_0$	
S3: $k := 0$	
S4: **while** $(\tau_k > \tau_{\max})$	**Iterative PCG solve**
S5: $w_k := Ad_k$	(SPMV)
S6: $\rho_k := \beta_k / d_k^T w_k$	(DOT product)
S7: $x_{k+1} := x_k + \rho_k d_k$	(AXPY)
S8: $r_{k+1} := r_k - \rho_k w_k$	(AXPY)
S9: $z_{k+1} := M^{-1} r_{k+1} \approx U^{-1} L^{-1} r_{k+1}$	Apply preconditioner
S10: $\beta_{k+1} := r_{k+1}^T z_{k+1}$	(DOT product)
S11: $\alpha_k := \beta_{k+1} / \beta_k$	
S12: $d_{k+1} := z_{k+1} + \alpha_k d_k$	(AXPY-like)
S13: $\tau_{k+1} := \parallel r_{k+1} \parallel_2$	(2-norm)
S14: $k := k + 1$	
S15: **endwhile**	

Feltor: Reproducible PCG

- Missing components: `spmv` and `nrm2`
- But `spmv` with their specific format

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 $+, -, *$ (aka `twosum` and `twoprod`)
  - Considered but dropped from 754-2008
  - Pending hardware implementations encouraged put them back
- Importance: extended-precision/ reproducible computations

Conclusions and Future Work

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** `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

TODO List

- Optimization and auto-tuning of `csrmv`
- Reproducible Jacobi and Conjugate Gradient methods

Thank you for your attention!

Publications: `pdc.kth.se/~riakymch/pubs`

Code: `https://exblas.lip6.fr`

Soon on GitHub