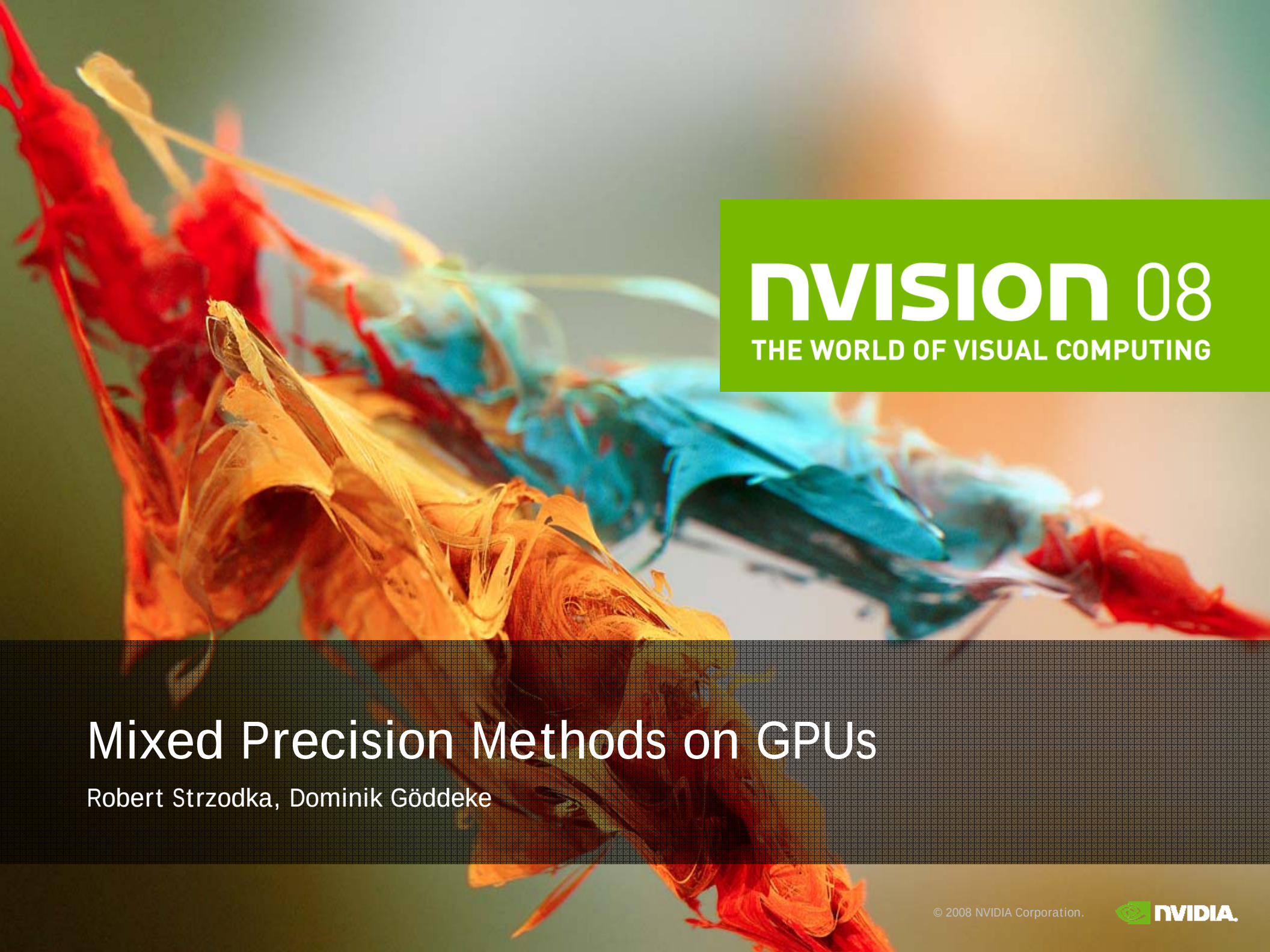


Mixed Precision Methods on GPUs

Robert Strzodka, Dominik Göddeke

Collaboration

Dominik Göddeke, Hilmar Wobker, Stefan Turek, FEAST Group
(Dortmund University of Technology)

Jamaludin Mohd-Yusof, Patrick McCormick
(Advanced Computing Lab, LANL)

Robert Strzodka, Max Planck Center
(Max Planck Institut Informatik)

What is a Mixed Precision Method?

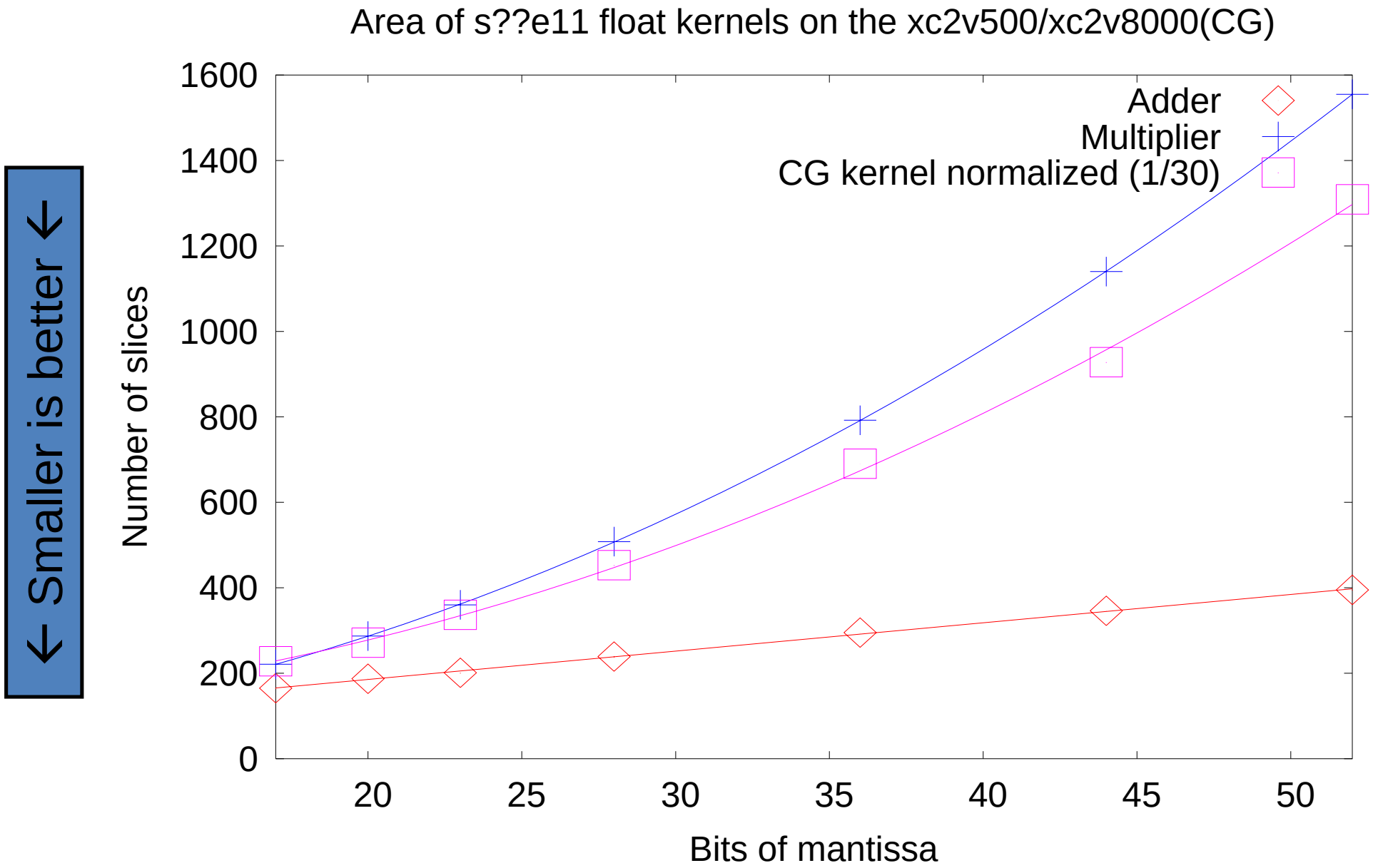
- **Definition**: A method that uses different precisions in its computations
- **Example**: **double**(a) + **double**(**float**(b) + **float**(c))
- **Typical usage**: Mix of single and double precision floating point computations
- **Goal**: Obtain the same accuracy but better performance with more low precision computations

Mixed Precision Performance Gains

- **Bandwidth bound algorithm**
  - 64 bit = 1 double = 2 floats
  - More variables per bandwidth (comp. intensity up)
  - More variables per storage (data block size up)
  - Applies to all memory levels:
    disc → main → device → local → register

- **Computation bound algorithm**
  - 1 double multiplier ≈ 4 float multiplier (quadratic)
  - 1 double adder ≈ 2 float adder (linear)
  - Multipliers are much bigger than adders
    → Quadrupled computational efficiency

Example resources on an FPGA



Overview

- Why Bother with Mixed Precision?
- **Precision and Accuracy**
- Floating Point Operations
- Mixed Precision Iterative Refinement

Roundoff and Cancellation

Roundoff examples for the **float s23e8** format

additive roundoff	$a = 1 + 0.00000004$	$= 1.00000004 =_{fl} 1$
multiplicative roundoff	$b = 1.0002 * 0.9998$	$= 0.99999996 =_{fl} 1$
cancellation $c \in \{a,b\}$	$(c - 1) * 10^8$	$= \pm 4 \quad =_{fl} 0$

Cancellation promotes the small error 0.00000004
to the absolute error 4 and a relative error of order one.

Order of operations can be crucial:

$1 + 0.00000004 - 1 =_{fl} 0$

$1 - 1 + 0.00000004 =_{fl} 0.00000004$

With the **double s52e11** format no problems above, but ...

An Instructive Example

Evaluating f(x,y) with powers as multiplications [S.M. Rump, 1988]

$$f(x,y) = (333.75 - x^2)\,y^6 + x^2(11x^2y^2 - 121y^4 - 2) + 5.5y^8 + x/(2y)$$

for $x_0 = 77617, \quad y_0 = 33096$ gives

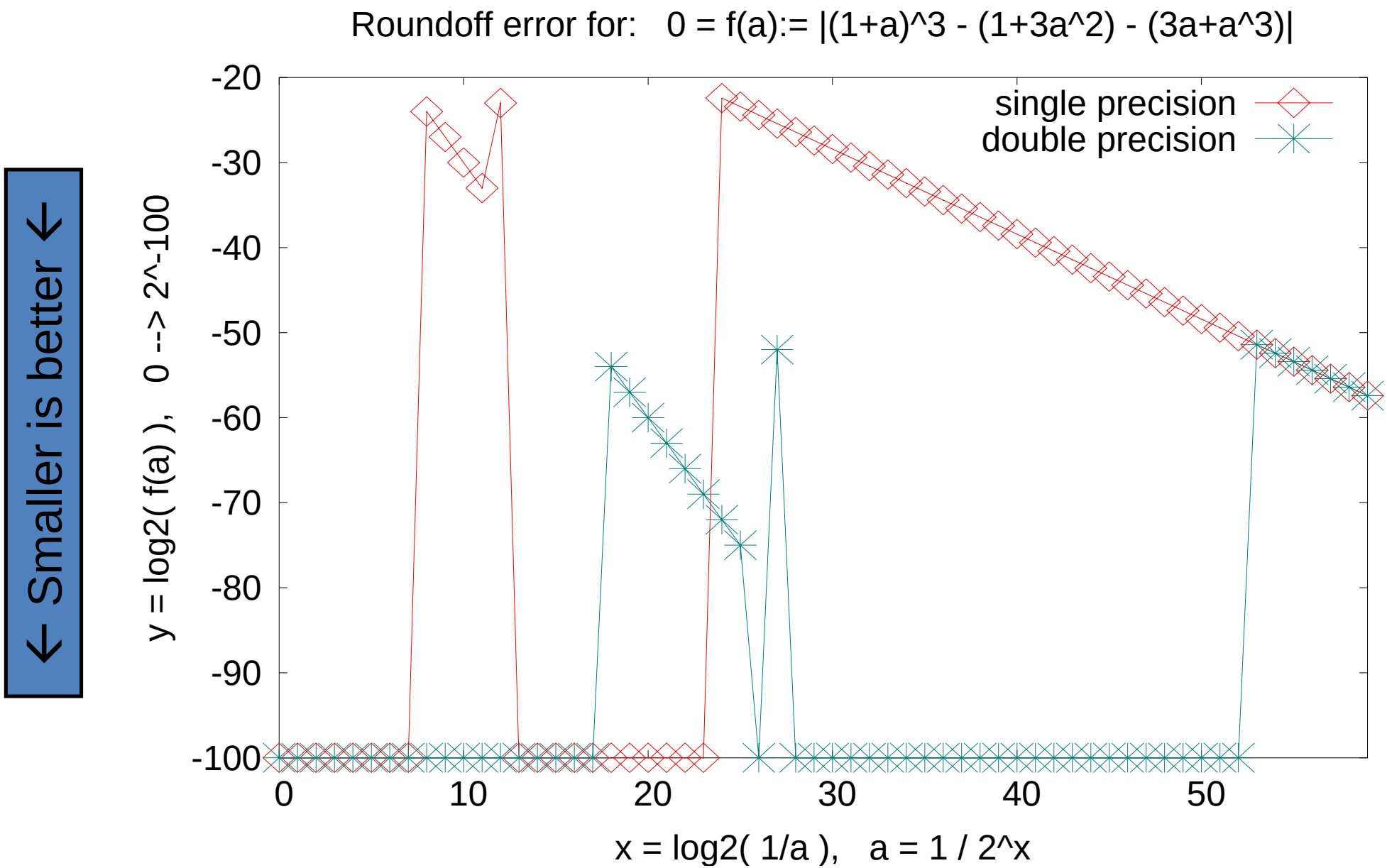
float s23e8	1.1726
double s52e11	1.17260394005318
long double s63e15	1.172603940053178631

This is all wrong, even the sign is wrong!! The correct result is

-0.827396059946821368141165095479816291...

Lesson learnt: **Computational Precision ≠ Accuracy of Result**

The Erratic Roundoff Error



← Smaller is better ←

The Dominant Data Error

- **Data error occurs when the exact value has to be truncated for storage in the binary format, e.g.**
  - π, $\sqrt{2}$, sin(2), exp(2), 1/3, ...
  - In fact, any value, e.g. 0.1, except combinations of 2^b

- **So more precision is usually better because**
  - for float s23e8: $1 + 4e\text{-}8 =_{fl} 1$
  - for double s52e11: $1 + 4e\text{-}15 =_{fl} 1$

- **How can float be better than double then?**
  - There is no data error in the operands
  - Alternatively, the errors cancel out themselves favorably

Overview

- Why Bother with Mixed Precision?
- Precision and Accuracy
- **Floating Point Operations**
- Mixed Precision Iterative Refinement

Understanding Floating Point Operations

- **Number representation s23e8**
  - a = | 1bit **sign** s_a | 23 bit **mantissa** m_a | 8 bit **exponent** e_a |

- **Multiplication a * b**
  - **Operations**: $s_a * s_b$, $m_a * m_b$, $e_a + e_b$
  - **Exact format**: s**46**e**9** = s23e8 * s23e8
  - **Main error**: Mantissa truncated from **46** bit to 23 bit

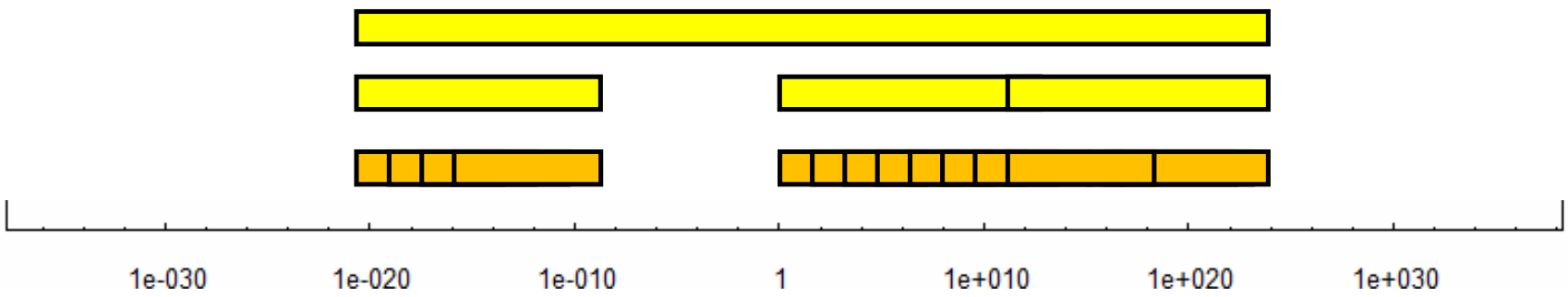
- **Addition a + b**
  - **Operations**: $e_{diff} = e_a - e_b$, $m_a + (m_b >> e_{diff})$, normalize
  - **Exact format**: s**278**e**8** = s23e8 + s23e8
  - **Main error**: Mantissa truncated from **278** bit to 23 bit

Commutative Summation

$$s = \sum_{i \in I} a_i$$

$$s = s_0 + s_1 + s_2$$

$$s_2 = \sum_{i \in I_2} a_i \qquad s_1 = \sum_{i \in I_1} a_i \qquad s_0 = \sum_{i \in I_0} a_i$$



Commutative Summation Example

- $1 + 0.00000004 =_{db} 1.00000004 =_{fl} 1$

- In **float s23e8**
  $s = \Sigma a_i = ½ + ½ + 0.00000004 - 0.00000003 =_{fl} 1$

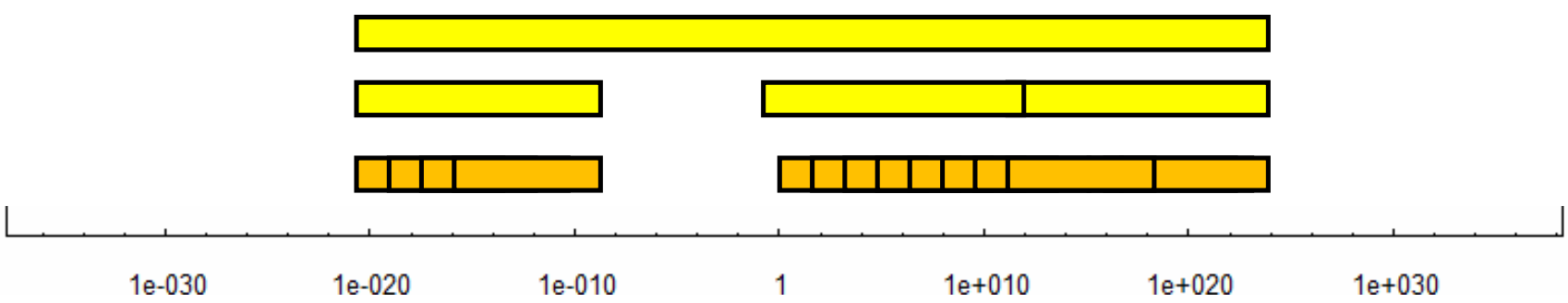
- In **double s52e11**
  $s = \Sigma a_i =_{db} 1.00000001$

- In **mixed double/float**
  $s_0 = \Sigma_0 a_i = ½ + ½ =_{fl} 1$
  $s_1 = \Sigma_1 a_i = 0.00000004 - 0.00000003 =_{fl} 0.00000001$
  $s = s_0 + s_1 =_{db} 1.00000001$

Dependent Summation

$$s =_{db} s_0 + s_1 + s_2$$

$$s_2 =_{fl} \sum_{i \in I_2} a_i \qquad s_1 =_{fl} \sum_{i \in I_1} a_i \qquad s_0 =_{fl} \sum_{i \in I_0} a_i$$



$$s =_{db} \sum_{i \in I} a_i, \qquad a_i = f_{db}(a_{i-1}), \quad \text{with slow double precision } f_{db}()$$

$$s_0 =_{fl} \sum_{i \in I_0} a_i, \quad s_1 =_{fl} \sum_{i \in I_1} a_i, \quad s_2 =_{fl} \sum_{i \in I_2} a_i, \quad s =_{db} s_0 + s_1 + s_2$$

$$a_i = f_{fl}(a_{i-1}), \quad \text{with fast single precision } f_{fl}()$$

High Precision Emulation

- **Combine two native floating point values**
  - Effectively doubles mantissa
  - Common exponent used to align mantissas
  - Normalisation required after each operation
  - Combined dynamic range slightly less than native range
  - Increases op count by 11x (ADD) and 18-32x (MUL)
  - Doubles bandwidth requirements

- **Example**
  - Two s23e8 yield a quasi s46e8 float with same storage requirements as a full s52e11 double
  - For large scale problems, we will need quad precision.

Example: Addition c=a+b

- **Compute high-order sum and error**

  t1 = a.hi + b.hi

  e = t1 – a.hi

- **Compute low order term including error and overflows**

  t2 = ((b.hi - e) + (a.hi - (t1 - e))) + a.lo + b.lo

- **Normalise to get final result**

  c.hi = t1 + t2

  c.lo = t2 – (c.hi – t1)

- **Computational costs**

  4 native reads, 2 native writes

  11 native primitive operations

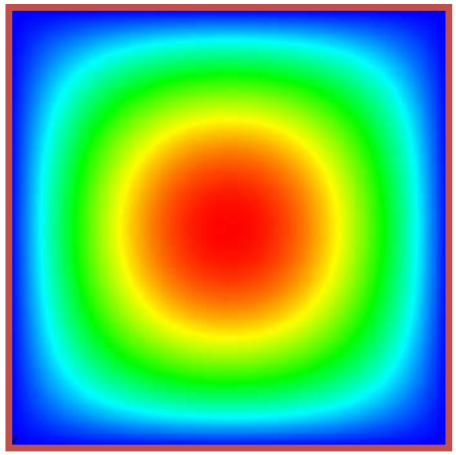
Details: Bailey et al., DSFUN90 library, http://crd.lbl.gov/~dhbailey/mpdist/index.html

Overview

- Why Bother with Mixed Precision?
- Precision and Accuracy
- Floating Point Operations
- **Mixed Precision Iterative Refinement**

PDE Example: Poisson Problem

- $-\Delta u = f$
- Unit square [0,1]^2
- Bilinear conforming FEs (Q1)
- Regular quadrilateral grid
- Zero Dirichlet BCs
- Analytic test function x(1-x)y(1-y)

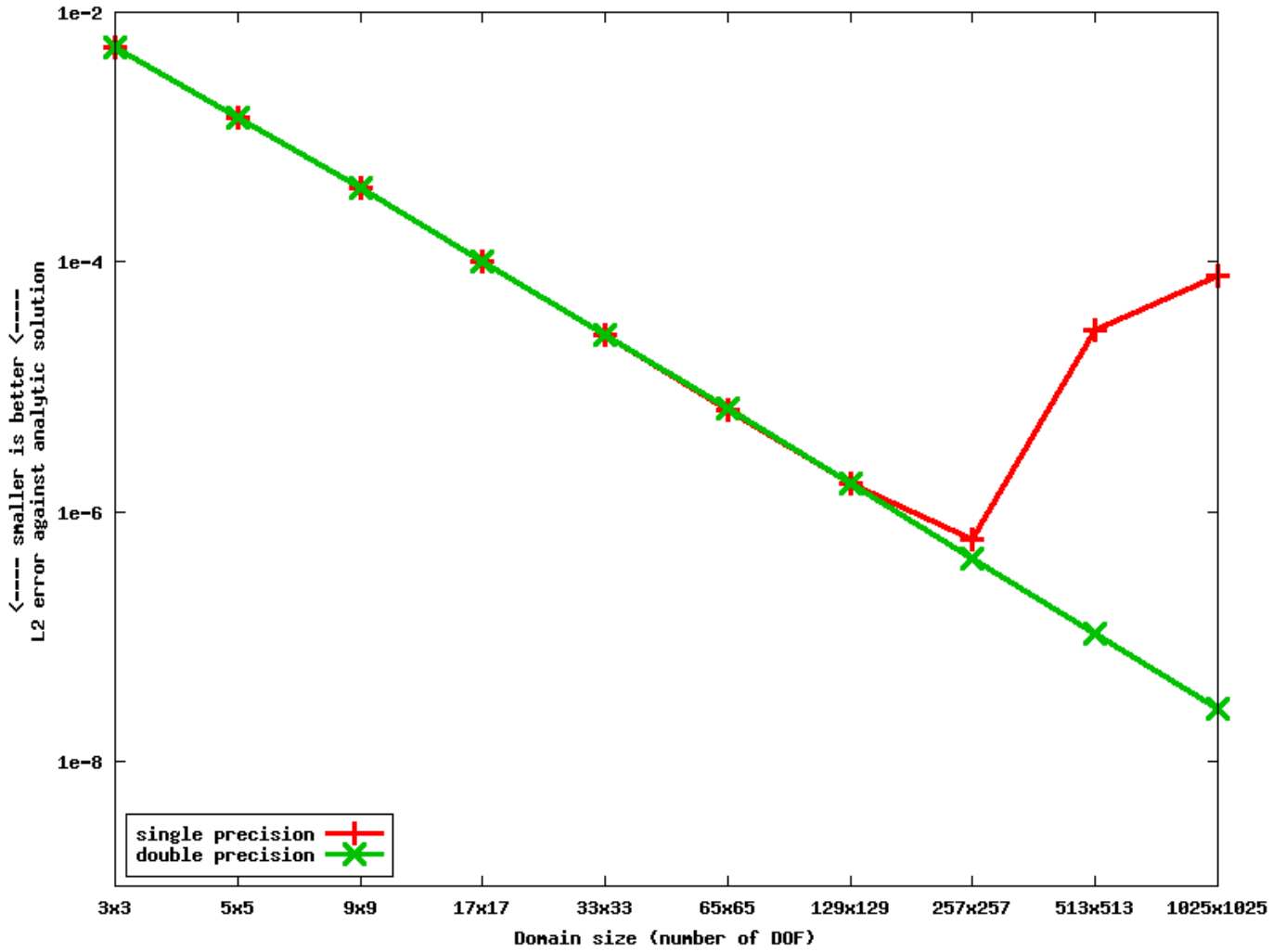


- Solved with multigrid until norms of residuals *indicate* convergence

PDE Example: Poisson Problem

- FEM theory: pure discretization error
- Expected error reduction of 4 (i.e. h^2) in each level

↙ Smaller is better ↙



PDE Example: Poisson Problem

- **Error reduction: single and double precision**

DOF	SINGLE	REDUCTION	DOUBLE	REDUCTION
3^2	5.208e-3		5.208e-3	
5^2	1.440e-3	3.62	1.440e-3	3.62
9^2	3.869e-4	3.72	3.869e-4	3.72
17^2	1.015e-4	3.81	1.015e-4	3.81
33^2	2.611e-5	3.89	2.607e-5	3.89
65^2	6.464e-6	4.04	6.612e-6	3.94
129^2	1.656e-6	3.90	1.666e-6	3.97
257^2	**5.927e-7**	**2.79**	4.181e-7	3.98
513^2	**2.803e-5**	**0.02**	1.047e-7	3.99
1025^2	**7.708e-5**	**0.36**	2.620e-8	4.00

Mixed Precision Iterative Refinement

- **Exploit the speed of low precision and obtain a result of high accuracy**

$\mathbf{d_k = b - Ax_k}$	**Compute** in **high** precision (cheap)
$\mathbf{Ac_k = d_k}$	**Solve** in **low** precision (fast)
$\mathbf{x_{k+1} = x_k + c_k}$	**Correct** in **high** precision (cheap)
$k = k+1$	Iterate until convergence in high precision

- **Low precision solution is used as a preconditioner in a high precision iterative method**
  - A is small and dense: Solve $\mathbf{Ac_k = d_k}$ directly
  - A is large and sparse: Solve (approximately) $\mathbf{Ac_k = d_k}$ with an iterative method itself

Direct Scheme for Small, Dense A

- **Algorithm**
  - Compute PA=LU once in single precision
  - Use LU decomposition to solve $Ly=Pd_k, Uc_k=y$ in each step

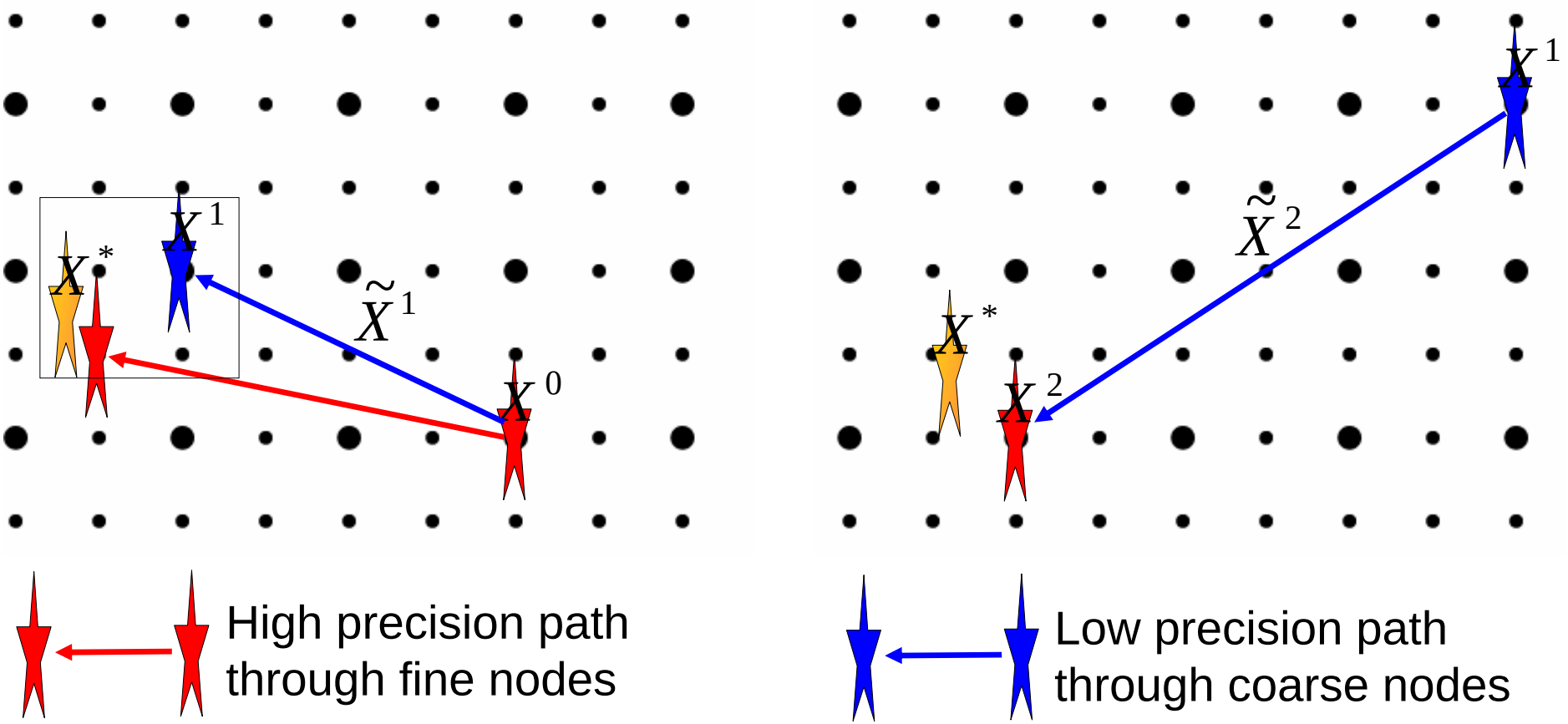
- **Main reasons for speedup**
  - Computation of LU decomposition is O(n^3)
  - Computation of LU is much faster in single than in double
  - Solution using LU for several RHS is only O(n^2)

- **Upper bound for iteration count**
  - $\text{ceil}(t_d/(t_s-K))$, where K, t_d, t_s are log10 of matrix condition and double and single precision (e.g. t_d approx 16)

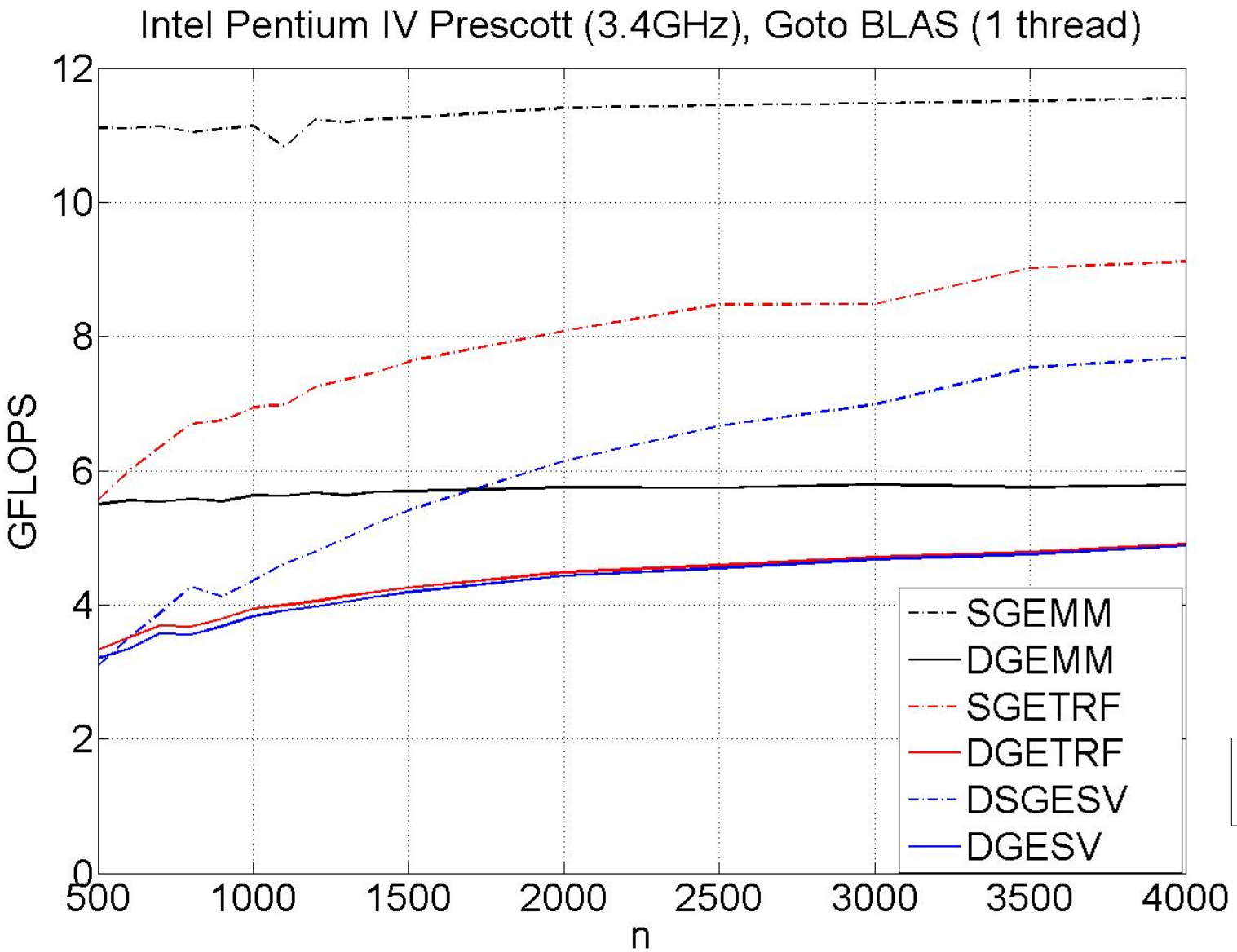
Iterative Refinement: 1st and 2nd Step

We solve $AX = B$ with iterative refinement

$$B^{k+1} := B - AX^{k}, \qquad A\tilde{X}^{k+1} = B^{k+1}, \qquad X^{k+1} := X^{k} + \tilde{X}^{k+1}$$



CPU SSE Results: LU Solver



→ Larger is better →

chart courtesy of Jack Dongarra

[Langou et al. *Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems)*, SC 2006]

Iterative Scheme for Large, Sparse A

- **Algorithm**
  - Inner solver: Conjugate Gradients, Multigrid
  - Correction loop can run on CPU or on GPU (old GPUs: emulated precision; new GPUs: true double precision)
  - Terminate inner solver after fixed number of iterations, fixed error reduction or convergence

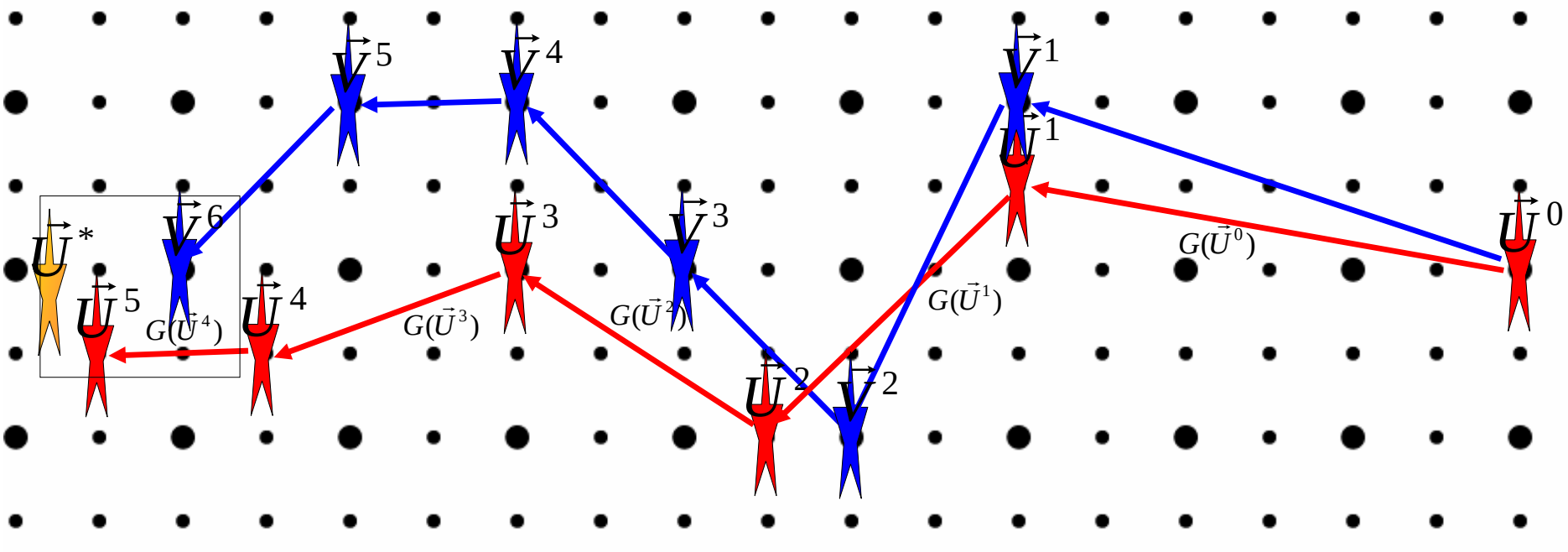
- **Main reason for speedup**
  - Inner solver on the GPU runs almost at peak bandwidth

- **Applicability**
  - Works even for very ill-conditioned matrices

Iterative Convergence: 1st Correction

Convergent iterative scheme

$$\vec{U}^{k+1} := \vec{U}^{k} + G(\vec{U}^{k}), \quad \left(\vec{U}^{k}\right)_k \xrightarrow{k\to\infty} \vec{U}^{*}, \quad \left(G(\vec{U}^{k})\right)_k \xrightarrow{k\to\infty} 0$$



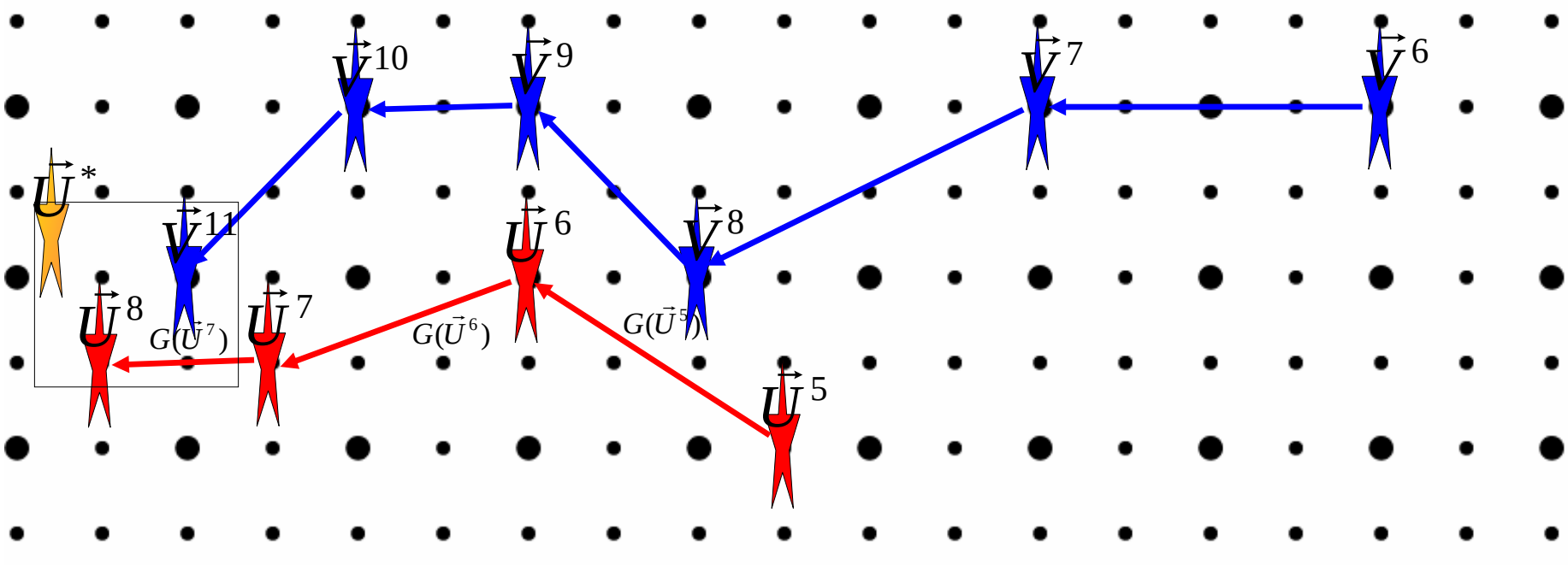
High precision path through fine nodes

Low precision path through coarse nodes

Iterative Convergence: 2nd Correction

Convergent iterative scheme

$$\vec{U}^{k+1} := \vec{U}^{k} + G(\vec{U}^{k}), \quad \left(\vec{U}^{k}\right)_k \xrightarrow{k\to\infty} \vec{U}^{*}, \quad \left(G(\vec{U}^{k})\right)_k \xrightarrow{k\to\infty} 0$$



High precision path through fine nodes

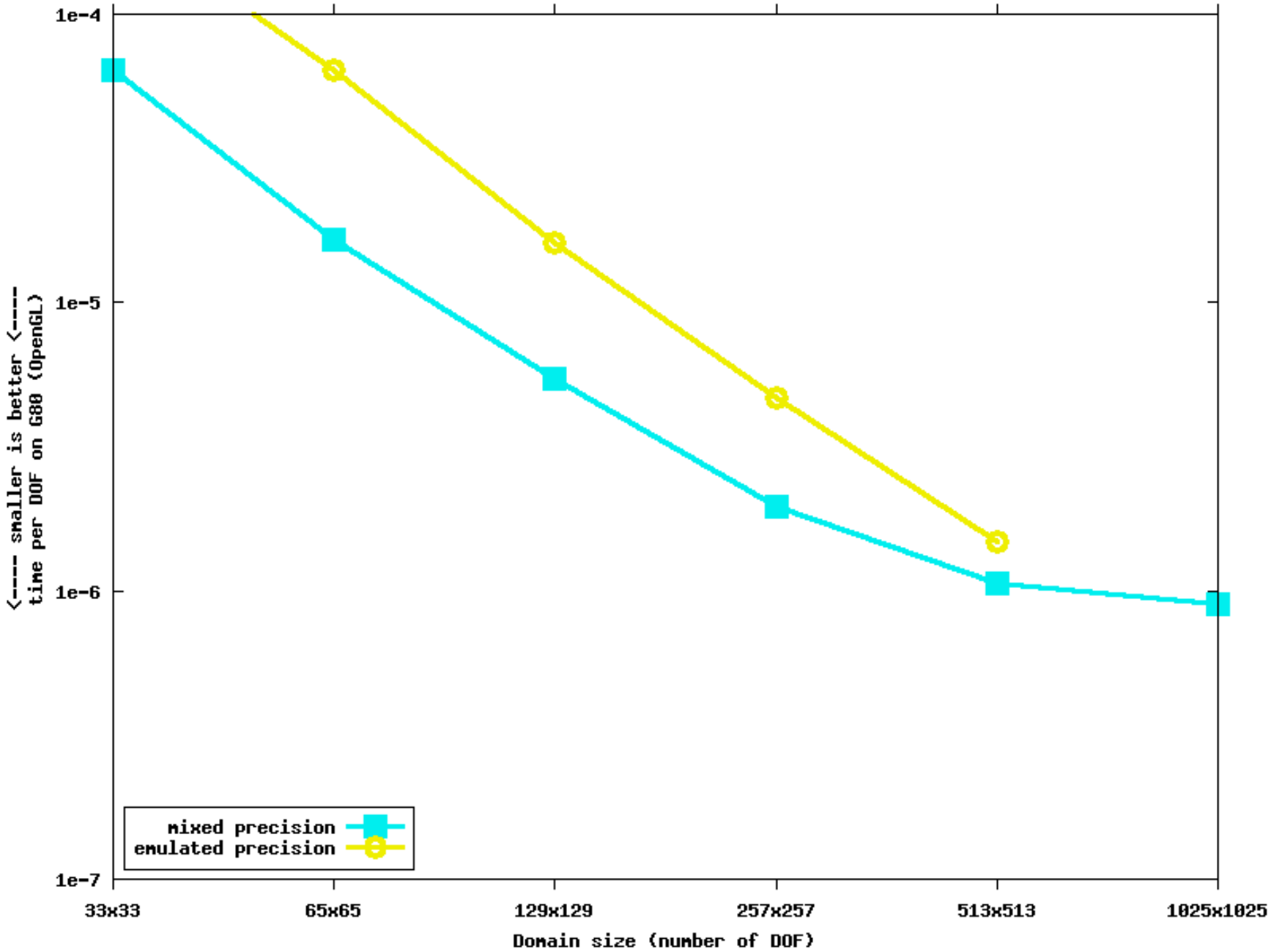
Low precision path through coarse nodes

GPU Performance Results

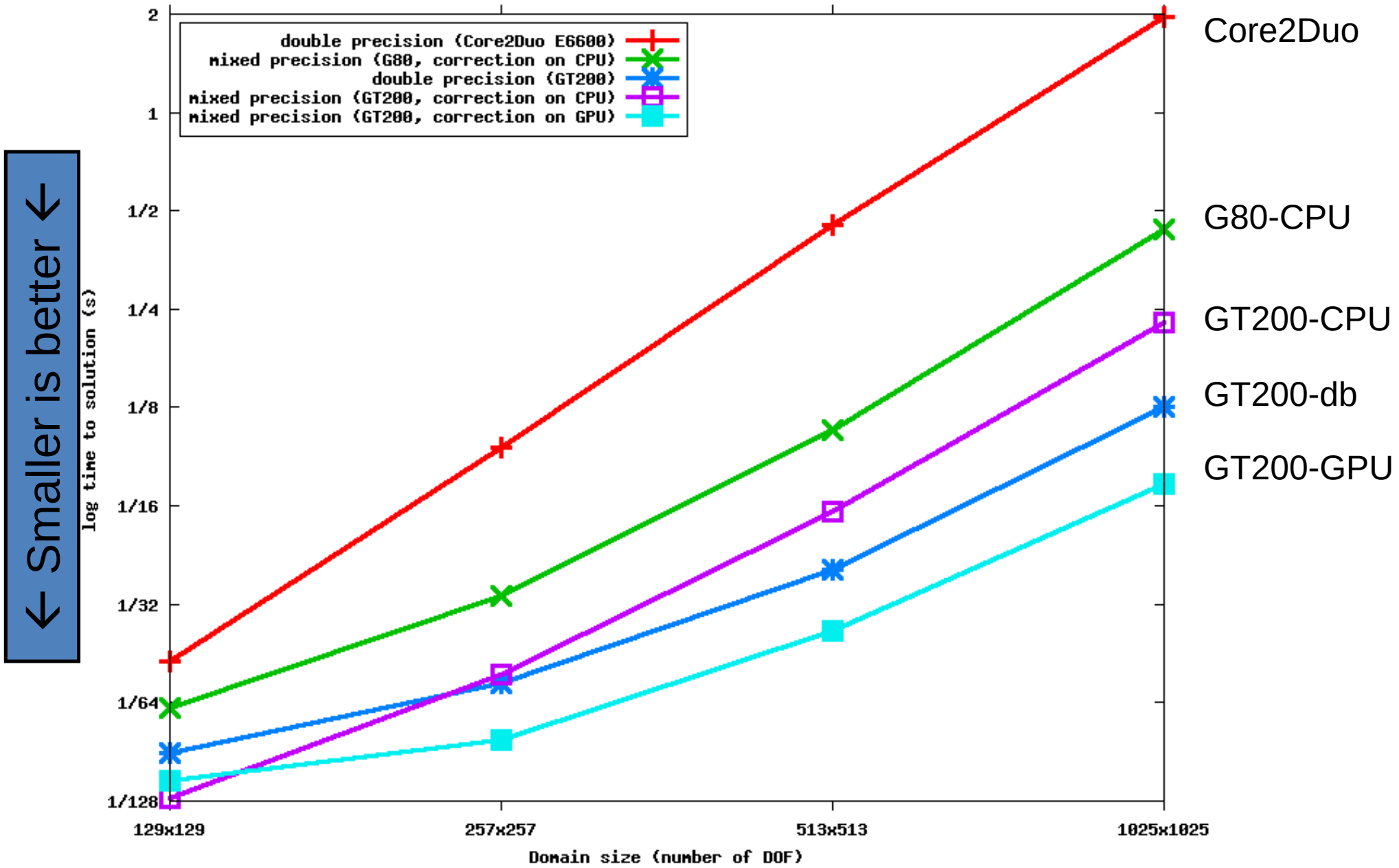
- **Test problem**
  - Poisson on unit square
  - Multigrid solver
  - N=33^2 to N=1025^2 DOF (=mesh points for Q1 FE)

- **Solver combination parameter space**
  - CPU implementation (Core2Duo E6600, SSE-optimized, double)
  - GPGPU implementation (GeForce 8800 GTX, OpenGL)
    - Mixed precision, correction on CPU
    - Emulated precision
  - CUDA implementation (GeForce 8800 GTX and GeForce GTX 280)
    - Mixed precision, correction on CPU (G80 and GT200)
    - Native double precision (GT200 only)
    - Mixed precision, correction on GPU (GT200 only)

GPU Performance Results: GPGPU



GPU Performance Results: CUDA



← Smaller is better ←

GeForce GTX 280 Results

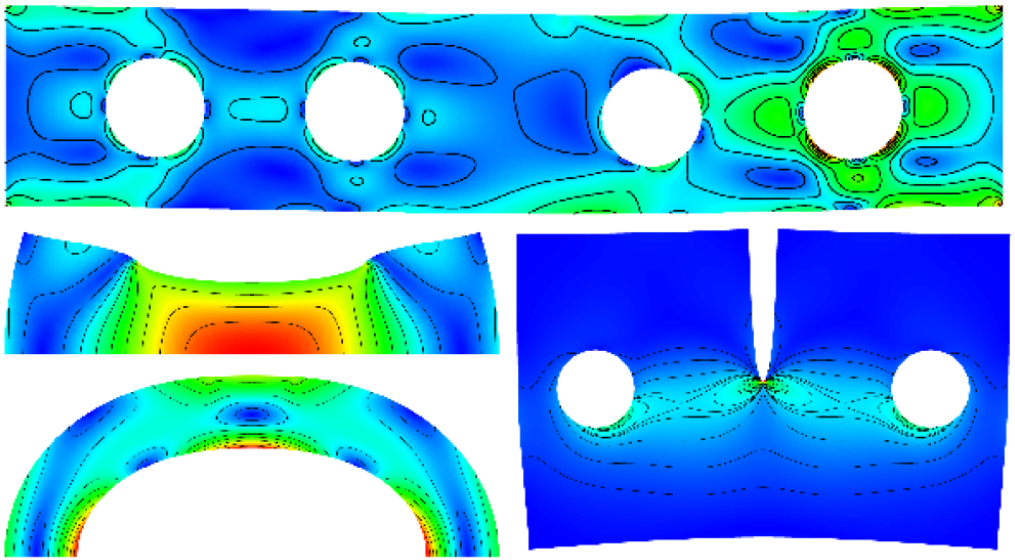
- **Mixed precision on GPU favorable**
  - Outperforms CPU by **27x**
  - Outperforms double-GPU by **1.7x**
  - Mixed precision with correction on CPU is limited by CPU for update and PCIe for transfer of intermediate results
  - Detailed timings for N=1025^2 (PCIe transfers not included in GByte/s metric):

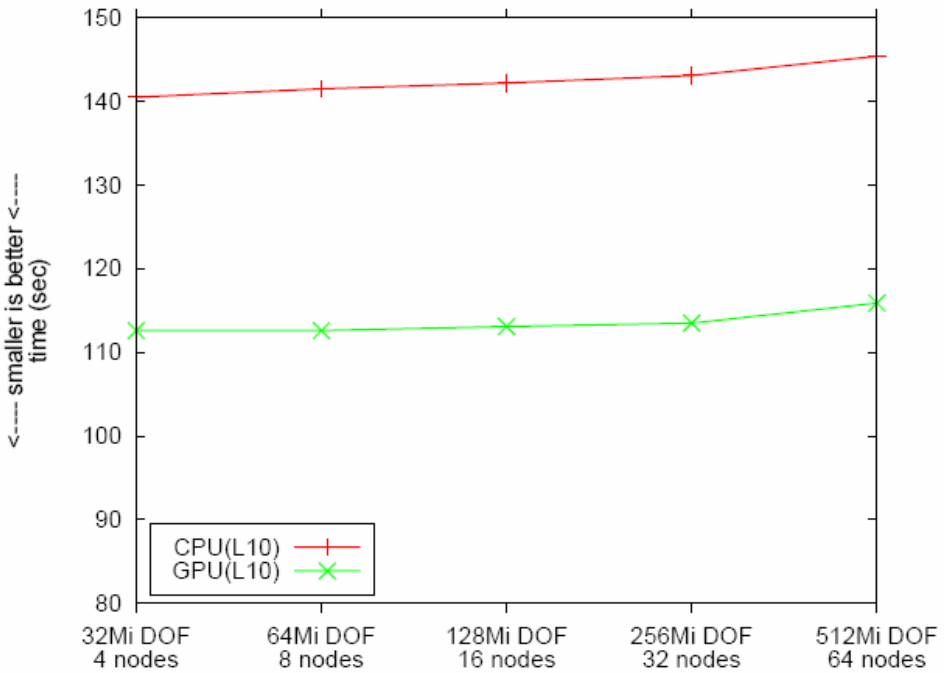
Variant	Time (s)	GFLOP/s	GByte/s
double, on CPU	1.962	0.8	4.6
double, on GPU	0.125	11.9	74.9
mixed, correction on CPU	0.227	6.9	23.9
mixed, correction on GPU	0.073	21.4	74.0

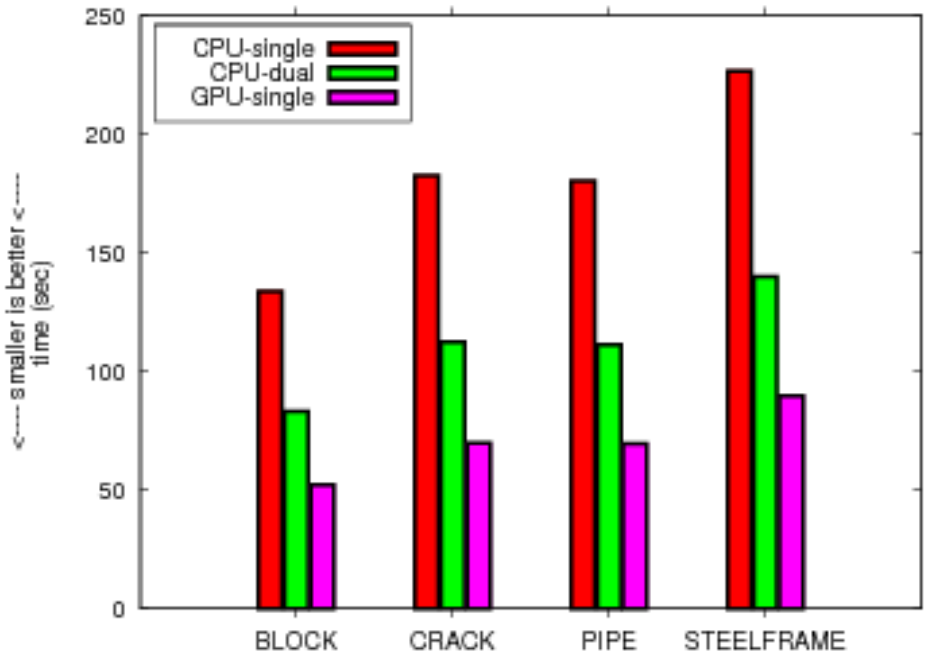
Mixed Precision on GPU Clusters

Total speedup of 2.7x for Quadro 5600 vs. AMD Santa Rosa (16 node cluster)

Good weak scalability on up to 64 nodes (dual Xeon, Quadro 1400 GPUs)







Conclusions

- The relation between computational precision and final accuracy is complicated but analyzable
- When single precision alone fails iterative refinement recovers the full accuracy with few double precision ops
- Mixed precision methods benefit bandwidth and even more computation bound algorithms
- Double precision GPUs are best utilized in mixed precision mode achieving outstanding performance and accuracy
- The benefits also extend to GPU clusters

Robert Strzodka, Max Planck Center, Saarbrücken, Germany
Dominik Göddeke, Dortmund University of Technology, Germany

Mixed Precision Methods on GPUs

www.mpii.de/~strzodka/

www.mathematik.tu-dortmund.de/~goeddeke/

D. Göddeke, R.Strzodka and S.Turek: *Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations*, IJPEDS, 2007

D. Göddeke and R.Strzodka, *Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations (Part 2: Double Precision GPUs)*, TU Dortmund, Technical Report, 2008

D. Göddeke, H. Wobker, R. Strzodka, J. Mohd-Yusof, P. McCormick, and S. Turek. *Co-processor acceleration of an unmodified parallel solid mechanics code with FEASTGPU*, IJCSE, 2008