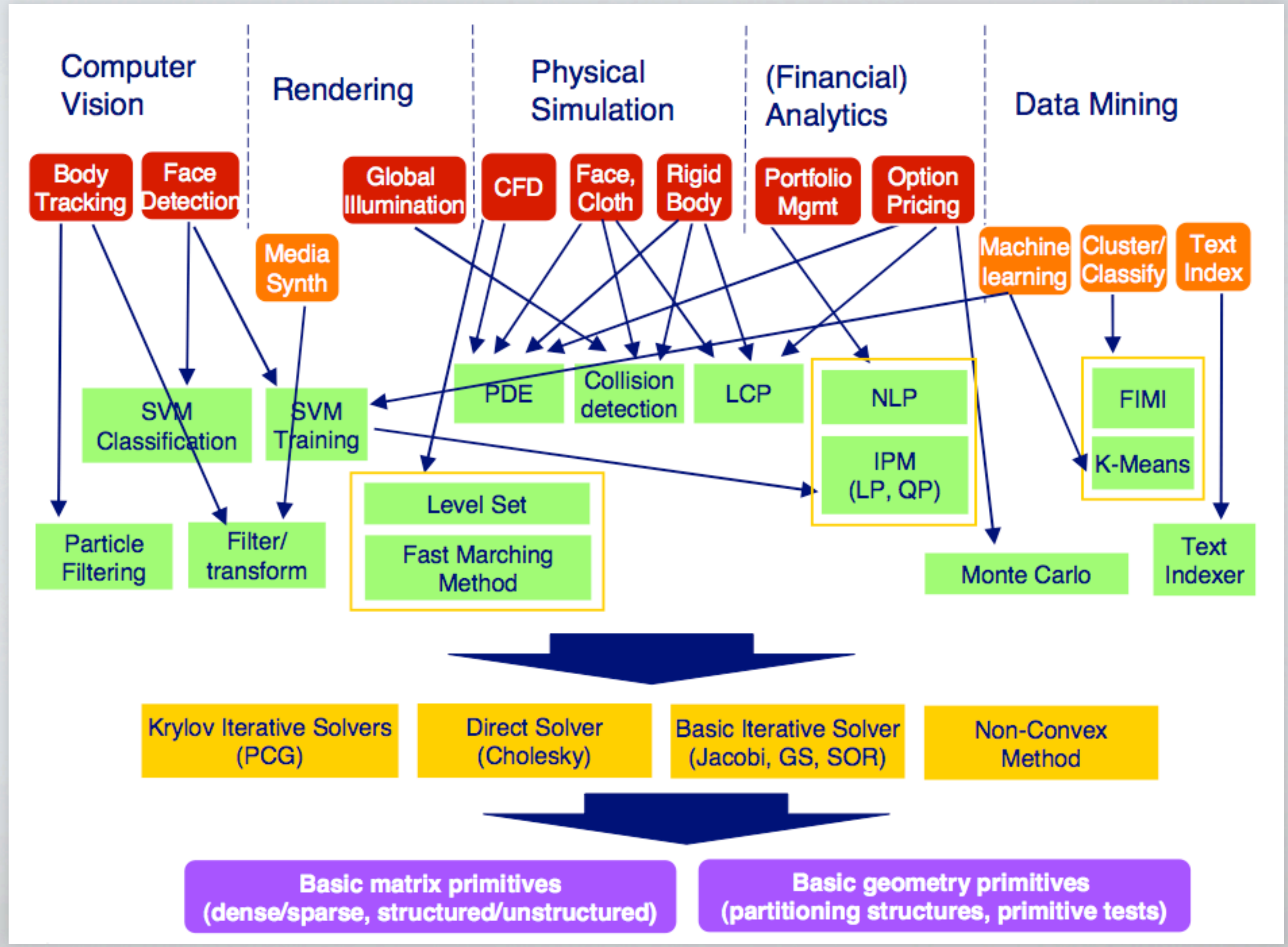
Interactions between parallelism and numerical stability, accuracy

Prof. Richard Vuduc

Georgia Institute of Technology

CSE/CS 8803 PNA: Parallel Numerical Algorithms

[L.13] Tuesday, February 19, 2008



Computer Vision

Rendering

Physical Simulation

(Financial) Analytics

Data Mining

Body Tracking

Face Detection

Global Illumination

CFD

Face, Cloth

Rigid Body

Portfolio Mgmt

Option Pricing

Media Synth

Machine learning

Cluster/ Classify

Text Index

SVM Classification

SVM Training

PDE

Collision detection

LCP

NLP

IPM (LP, QP)

FIMI

K-Means

Particle Filtering

Filter/ transform

Level Set

Fast Marching Method

Monte Carlo

Text Indexer

Krylov Iterative Solvers (PCG)

Direct Solver (Cholesky)

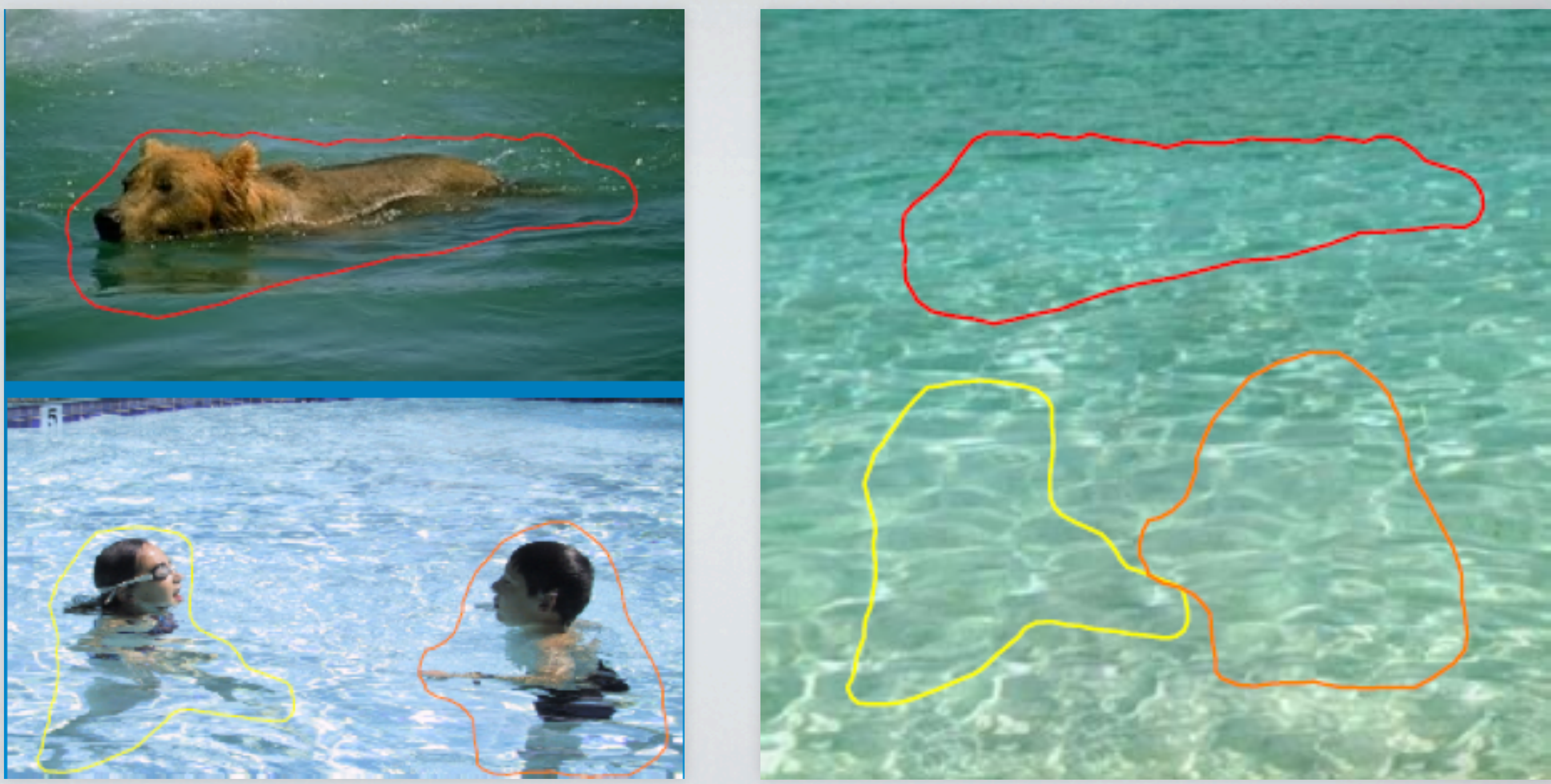
Basic Iterative Solver (Jacobi, GS, SOR)

Non-Convex Method

Basic matrix primitives (dense/sparse, structured/unstructured)

Basic geometry primitives (partitioning structures, primitive tests)

Source: Dubey, *et al*., of Intel (2005)



Problem: Seamless image cloning.

(Source: Pérez, *et al.*, SIGGRAPH 2003)

… then reconstruct.

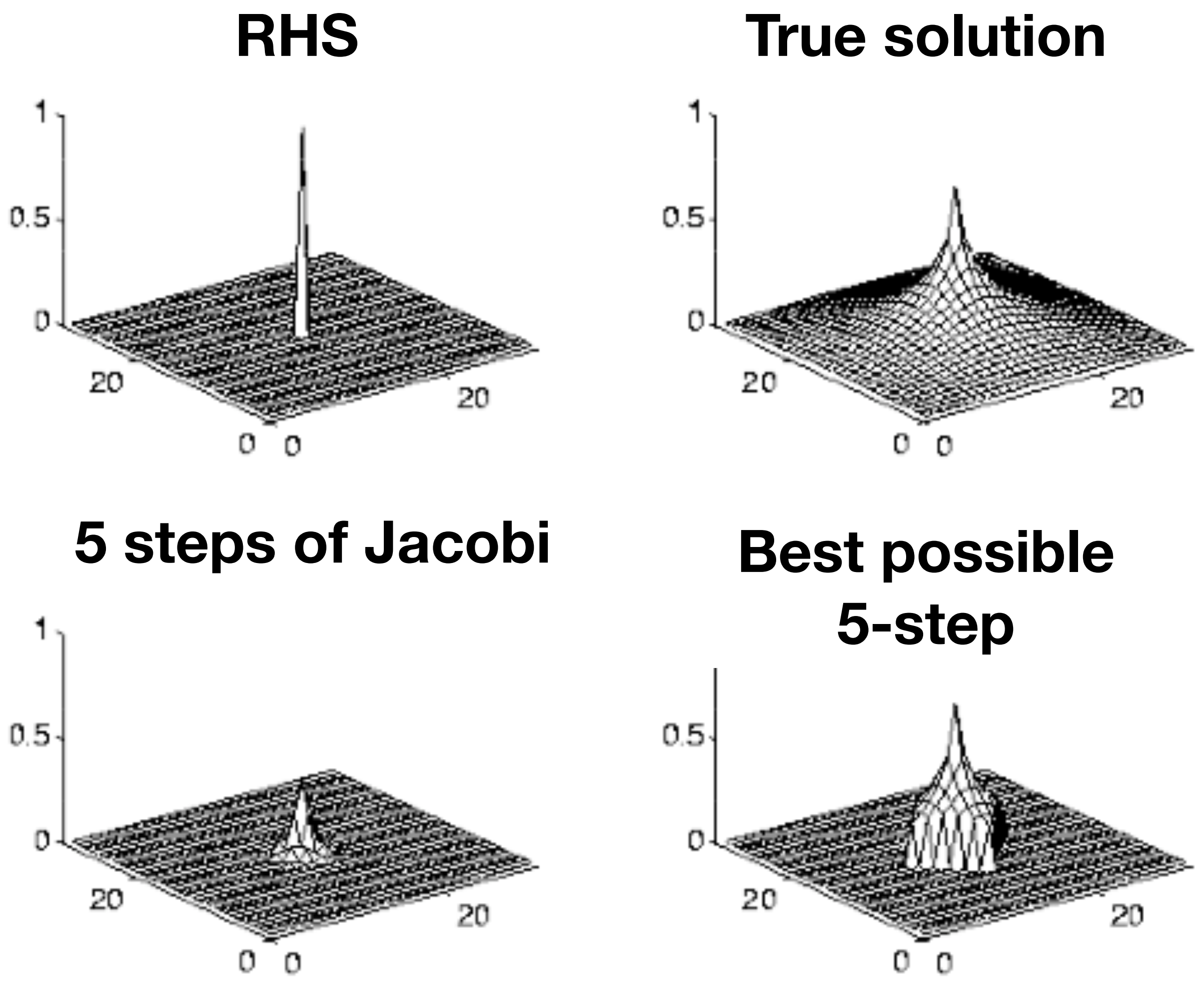
(Source: Pérez, *et al.*, SIGGRAPH 2003)

Review: Multigrid

Exploiting structure to obtain fast algorithms for 2-D Poisson

- **Dense LU**: Assume no structure ⇒ O(n^6)
- **Sparse LU**: Sparsity ⇒ O(n^3), need extra memory, hard to parallize
- **CG**: Symmetric positive definite ⇒ O(n^3), a little extra memory
- **RB SOR**: Fixed sparsity pattern ⇒ O(n^3), no extra memory, easy to parallelize
- **FFT**: Eigendecomposition ⇒ $O(n^2 \log n)$
- ***Multigrid***: Eigendecomposition ⇒ $O(n^2)$ [Optimal!]

Problem: Slow convergence



RHS

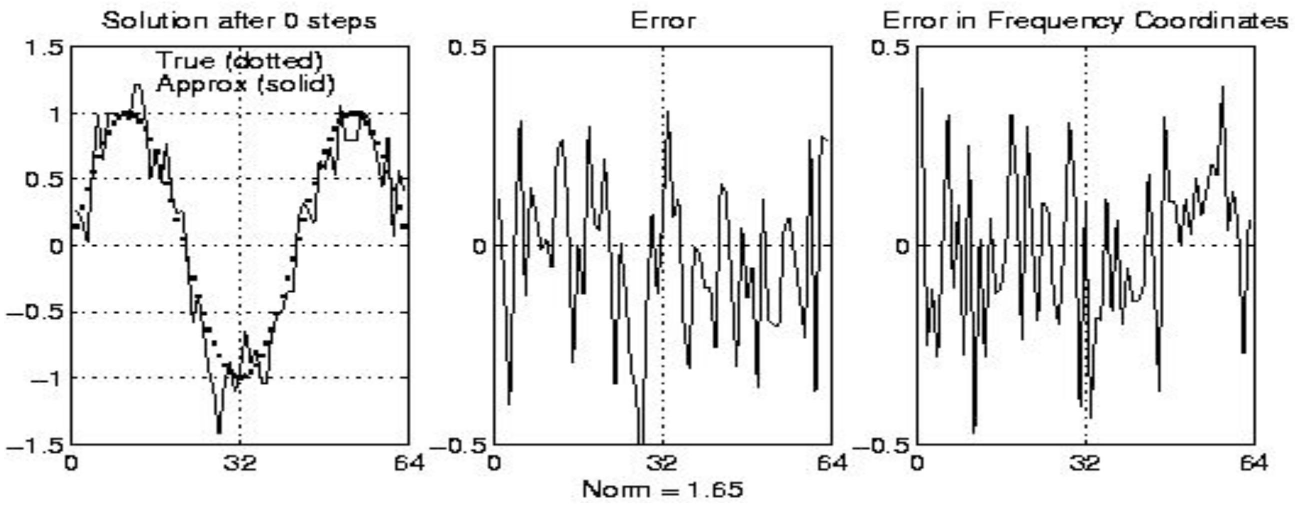
True solution

5 steps of Jacobi

Best possible 5-step

Error “frequencies”

$$
\begin{aligned}
\epsilon^{(t)} = R_w^t \cdot \epsilon^{(0)} &= (I - \frac{w}{2} Z\Lambda Z^T)^t \cdot \epsilon^{(0)} \\
&= Z\left(I - \frac{w}{2}\Lambda\right)^t Z^T \cdot \epsilon^{(0)} \\
&\Downarrow \\
Z^T \cdot \epsilon^{(t)} &= \left(I - \frac{w}{2}\Lambda\right)^t Z^T \cdot \epsilon^{(0)} \\
\left(Z^T \cdot \epsilon^{(t)}\right)_j &= \left(I - \frac{w}{2}\Lambda\right)^t_{jj} \left(Z^T \cdot \epsilon^{(0)}\right)_j
\end{aligned}
$$

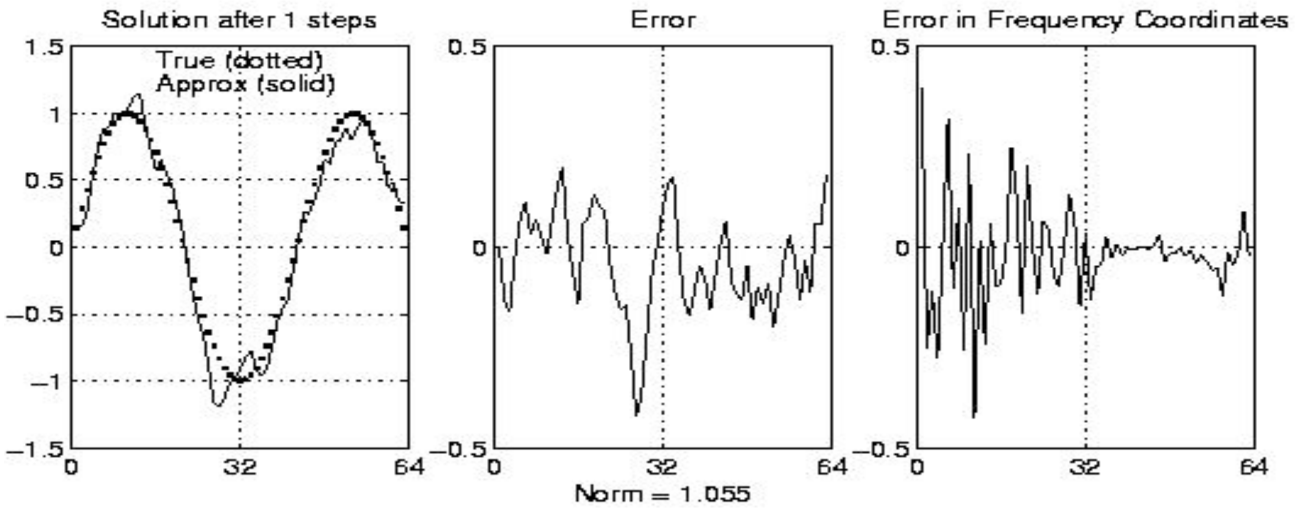


Initial error

"Rough"

Lots of high frequency components

Norm = 1.65

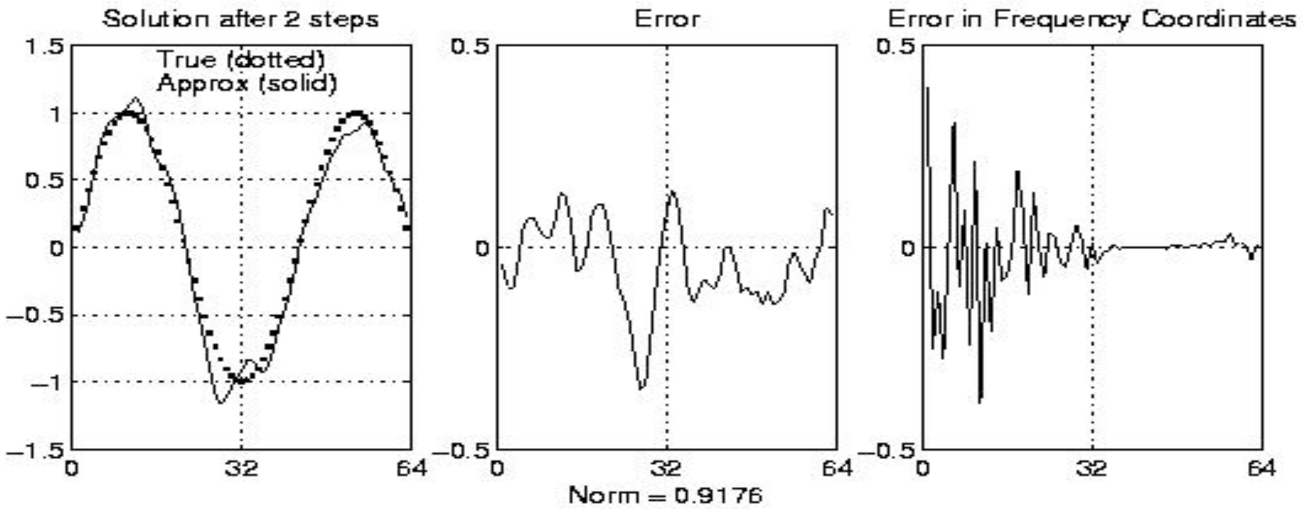


Error after 1 weighted Jacobi step

"Smoother"

Less high frequency component

Norm = 1.06



Error after 2 weighted Jacobi steps

"Smooth"

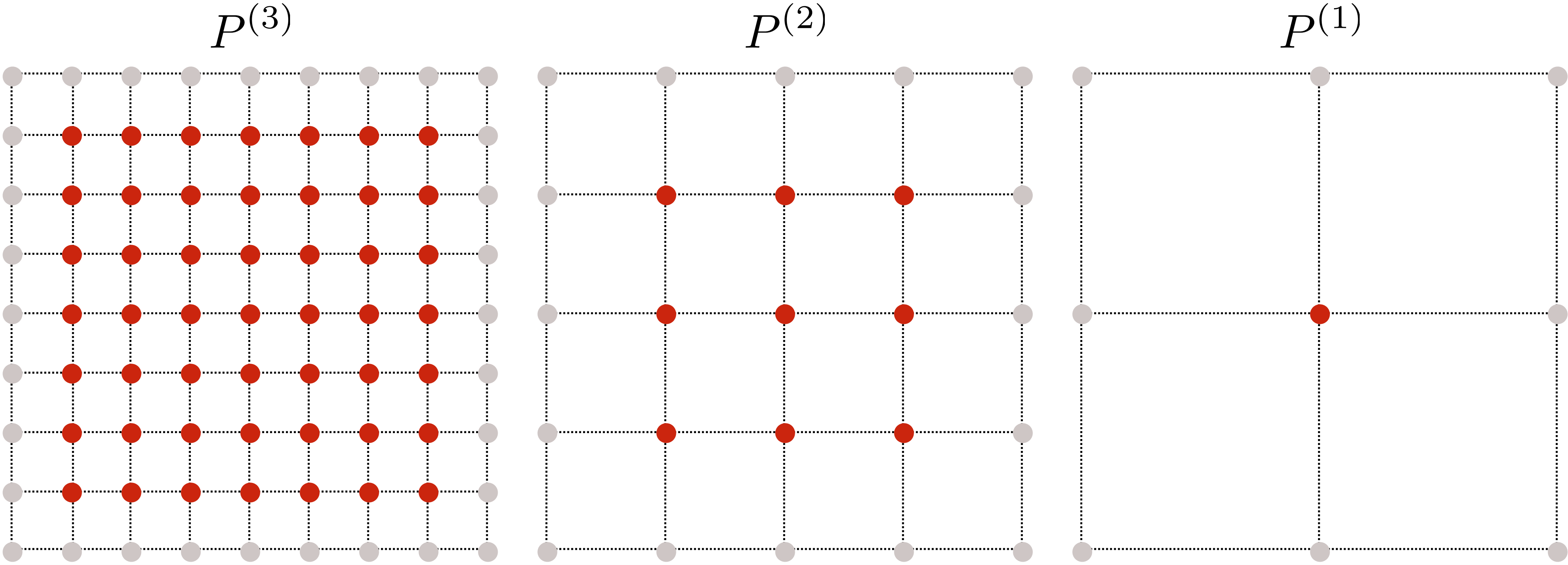
Little high frequency component

Norm = .92,

won't decrease much more

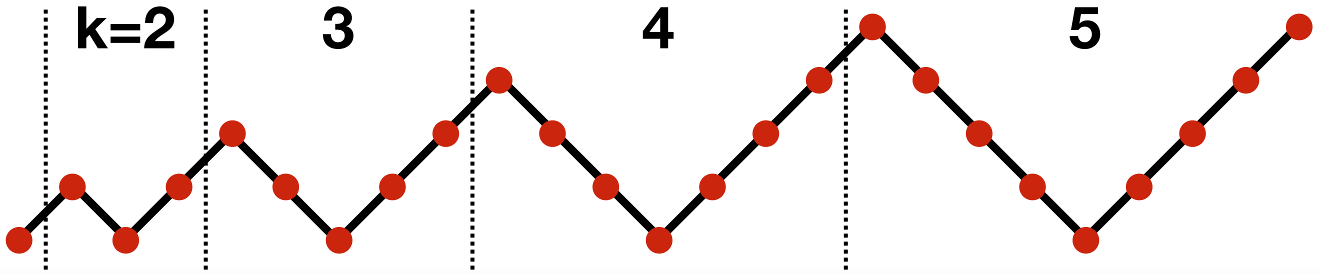
"Multigrids" in 2-D

$$
\begin{aligned}
P^{(i)} &= \text{Problem on } \left(2^i+1\right)\times\left(2^i+1\right) \text{ grid} \\
T^{(i)}x^{(i)} &= b^{(i)}
\end{aligned}
$$



Full multigrid algorithm

```
$\mathrm{FMG}\left(b^{(k)}, x^{(k)}\right)$
    $x^{(1)} \leftarrow$ Exact solution to $P^{(1)}$
    for $i = 2$ to $k$ do
        $x^{(i)} \leftarrow \mathrm{MGV}\left(b^{(i)}, L^{(i-1)}\left(x^{(i-1)}\right)\right)$
```



Interactions between parallelism and numerical stability, accuracy

Prof. Richard Vuduc

Georgia Institute of Technology

CSE/CS 8803 PNA: Parallel Numerical Algorithms

[L.13] Tuesday, February 19, 2008

Example 1: When single-precision is faster than double

- On STI Cell
  - **SPEED(single) = 14x SPEED(double)**: 204.8 Gflop/s vs. 14.6 Gflop/s
  - SPEs fully IEEE-compliant for double, but only support round-to-zero in single
- On regular CPUs with SIMD units
  - **SPEED(single) ~ 2x SPEED(double)**
  - SSE2: S(single) = 4 flops / cycle vs. S(double) = 2 flops/cycle
  - PowerPC Altivec: S(single) = 8 flops / cycle; no double (4 flops / cycle)
- On a GPU, **might not have double-precision** support

Example 2: Parallelism and floating-point semantics: Bisection on GPUs

- Bisection algorithm computes eigenvalues of symmetric tridiagonal matrix
- Inner-kernel is a routine, *Count(x)*, which counts the number of eigenvalues less than x
- Correctness 1: *Count(x)* must be "**monotonic**"
- Correctness 2: (Some approaches) IEEE FP-compliance
    - ATI Radeon X1900 XT GPU does not strictly adhere to IEEE floating-point standard, causing error in some cases
    - But workaround possible

The impact of parallelism on numerical algorithms

- **Larger problems** magnify errors: Round-off, ill-conditioning, instabilities
- **Reproducibility**: a + (b + c) ≠ (a + b) + c
- Fast **parallel algorithm** may be much **less stable** than fast serial algorithm
- **Flops cheaper** than communication
- **Speeds at different precisions** may vary significantly [*e.g.*, SSE_k, Cell]
- Perils of **arithmetic heterogenity**, *e.g.*, CPU vs. GPU support of IEEE

A computational paradigm

- Use fast algorithm that may be unstable (or “less” stable)
- Check result at the end
- If needed, re-run or fix-up using slow-but-safe algorithm

Sources for today’s material

- *Applied Numerical Linear Algebra*, by Demmel
- *Accuracy and stability of numerical algorithms*, by Higham
- “Trading off parallelism and numerical stability,” by Demmel (1992)
- “Exploiting the performance of 32 bit arithmetic in obtaining 64 bit accuracy,” by Langou, *et al*. (2006)
- “Using GPUs to accelerate the bisection algorithm for finding eigenvalues of symmetric tridiagonal matrices,” by Volkov and Demmel (2007)
- CS 267 (Demmel, UCB)
- Plamen Koev (NCSU)

Reasoning about accuracy and stability

Sources of error in a computation

- **Uncertainty** in input data, due to measurements, earlier computations, ...
- **Truncation** errors, due to algorithmic approximations
- **Rounding** errors, due to finite-precision arithmetic

- Today’s focus: Rounding error analysis

Accuracy vs. precision

- **Accuracy**: Absolute or relative error in computed quantity
- **Precision**: Accuracy of basic operations (+, -, *, /, …)

- **Accuracy not limited by precision!**
  - Can simulate arbitrarily higher precision with a given precision
  - Cost: Speed

A model of floating-point arithmetic

- Basic operations (+, -, *, /, …) satisfy:

$$\mathrm{fl}(a \text{ op } b) = (a \text{ op } b)(1+\delta), \qquad |\delta| \le \epsilon$$

- ε = “Unit round-off” or “machine/format precision”
  - IEEE 754 single-precision (32-bit) ~ 10^{-8}
  - Double (64-bit) ~ 10^{-16}
  - Extended (80-bit on Intel) ~ 10^{-20}
- Fused multiply-add: Two ops with only one error

Error analysis notation

- **Desired** computation:

$$y = f(x)$$

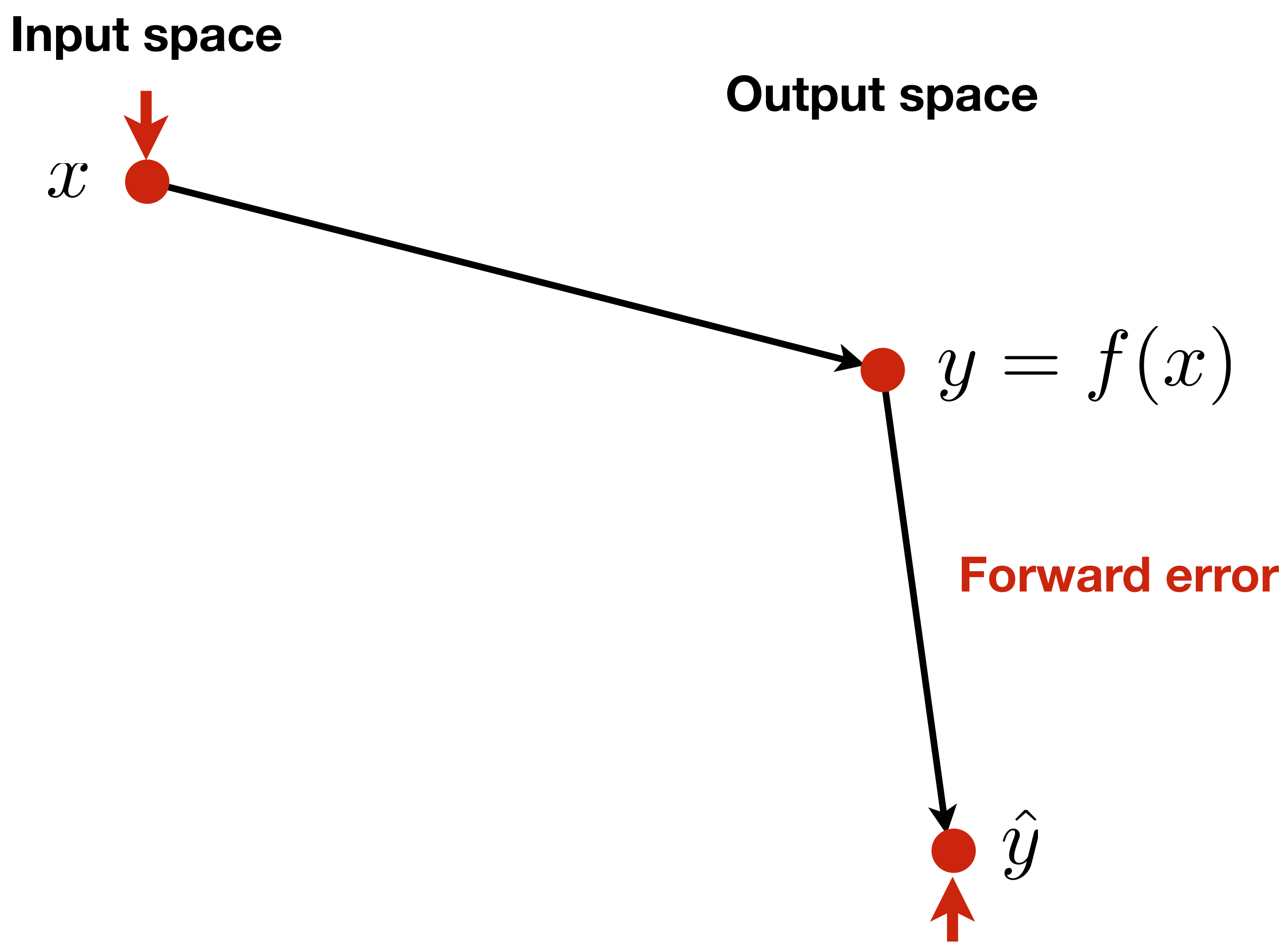
- What our **algorithm** actually computes:

$$\hat{y} = \hat{f}(x)$$

Measuring errors

- **Absolute** error $$|\hat{x} - x|$$
- **Relative** error $$\frac{|\hat{x} - x|}{|x|}$$
- **(Forward) Stability**: “Small” bounds on error
- For vectors and matrices, use “norms”

$$||x||_2 \equiv \sqrt{\sum_i x_i^2} \qquad ||x||_1 \equiv \sum_i |x_i| \qquad ||x||_\infty \equiv \max_i |x_i|$$



Input space

Output space

x

$y = f(x)$

Forward error

$\hat{y}$

Forward vs. backward errors

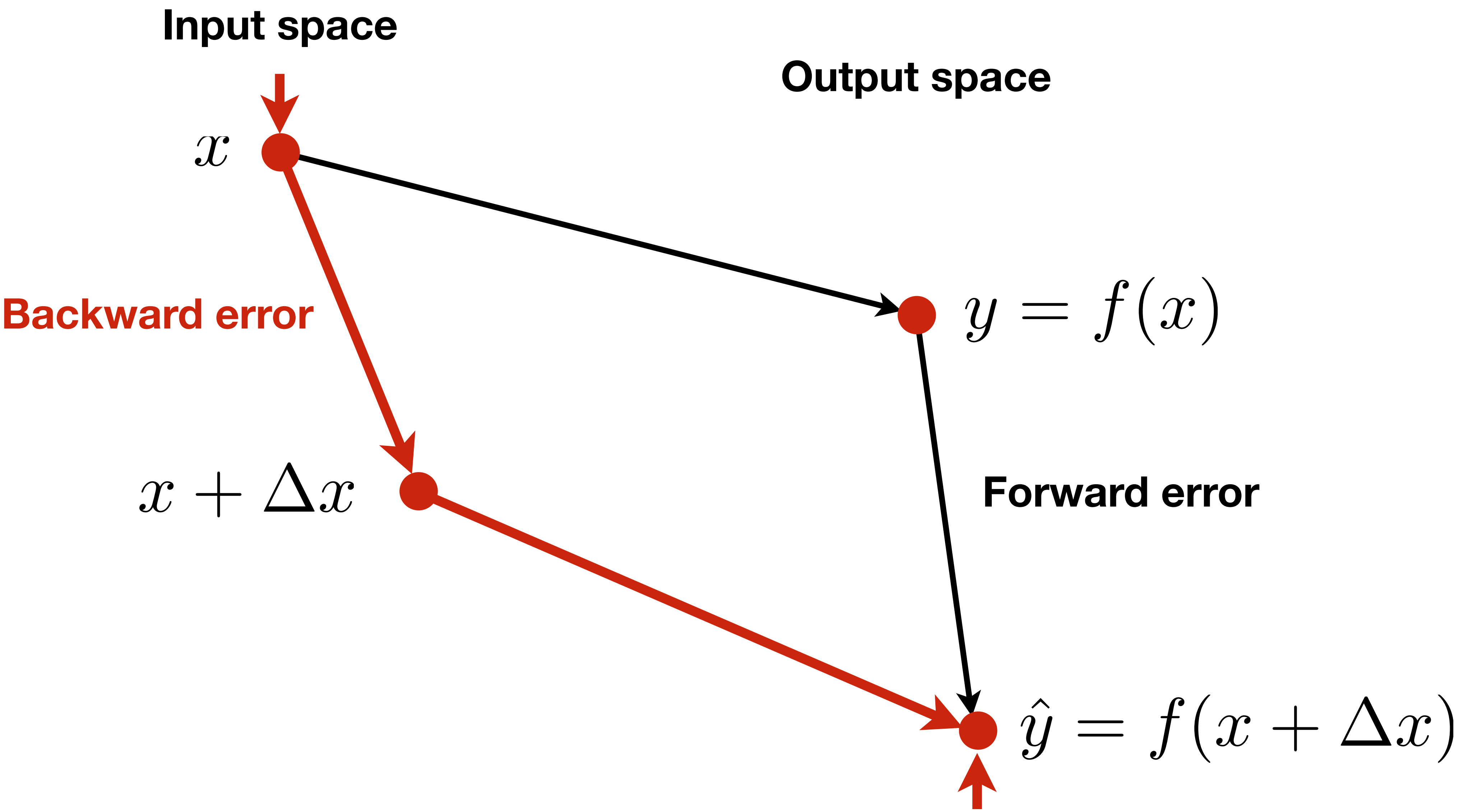
- **Forward** error analysis bounds

$$|\hat{y} - y| \qquad \text{or} \qquad \frac{|\hat{y} - y|}{|y|}$$

- **Backward** error analysis bounds

$$\Delta x : \qquad \hat{y} = f(x + \Delta x)$$

- **Numerically stable**: Bounds are “small”



Input space

Output space

x

$y = f(x)$

Backward error

$x + \Delta x$

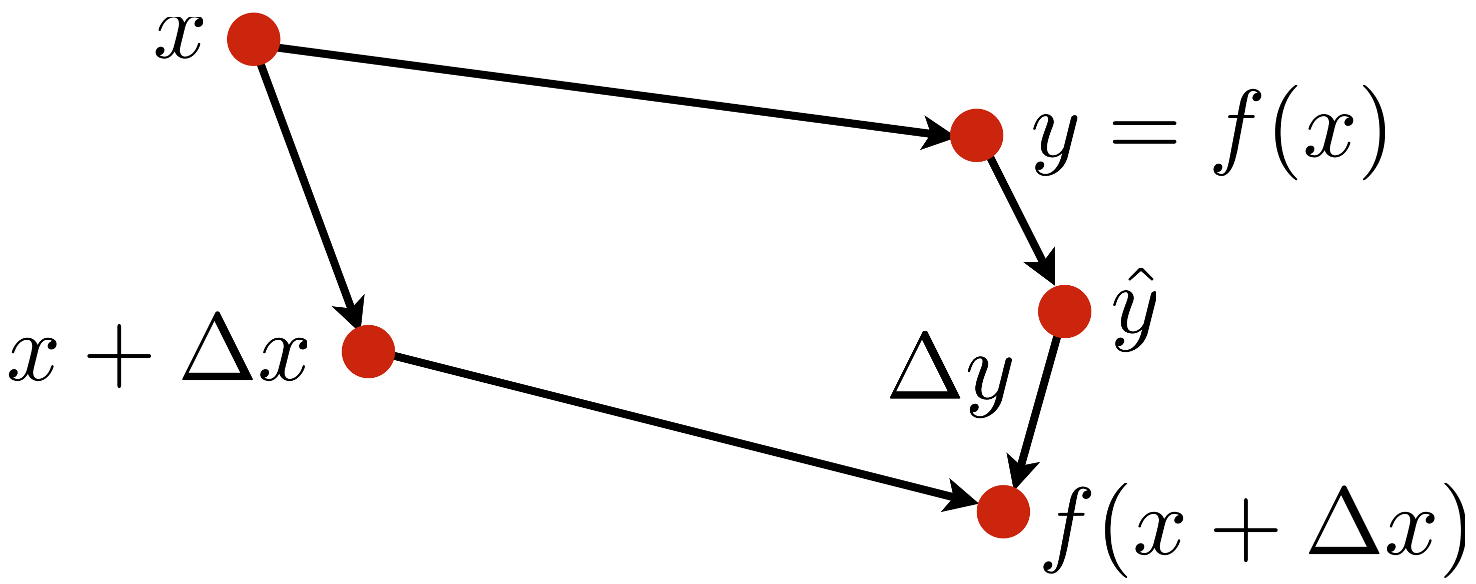
Forward error

$\hat{y} = f(x + \Delta x)$

Mixed (forward-backward) stability

- Computed answer “near” exact solution of a nearby problem

$$\Delta x, \Delta y : \qquad \hat{y} + \Delta y = f(x + \Delta x)$$



Conditioning: Relating forward and backward error

$$
\begin{aligned}
\hat{y} = f(x + \Delta x) &\approx f(x) + f'(x)\Delta x = y + f'(x)\Delta x \\
\Longrightarrow \hat{y} - y &\approx f'(x)\Delta x \\
\frac{\hat{y} - y}{y} &\approx \frac{f'(x)\Delta x}{f(x)} \cdot \frac{x}{x} \\
&\Downarrow \\
\left| \frac{\hat{y} - y}{y} \right| &\lesssim \left| \frac{x f'(x)}{f(x)} \right| \cdot \left| \frac{\Delta x}{x} \right|
\end{aligned}
$$

Conditioning: Relating forward and backward error

$$\left|\frac{\hat{y}-y}{y}\right| \quad \lesssim \quad \left|\frac{x f'(x)}{f(x)}\right| \cdot \left|\frac{\Delta x}{x}\right|$$

- Define **(relative) condition number**:

$$c(x) \quad = \quad \left|\frac{x f'(x)}{f(x)}\right|$$

- Roughly: **(Forward error) ≤ (Condition number) * (Backward error)**

Comments on conditioning

- Rule-of-thumb: **(Forward error) ≤ (Condition number) * (Backward error)**

$$c(x) = \left| \frac{x f'(x)}{f(x)} \right|$$

- Condition number is **problem dependent**
- **Backward stability ⇒ Forward stability**, but not vice-versa
- **Ill-conditioned** problem can have large forward error

Example: Condition number for solving linear systems

$$\begin{aligned} Ax &= b \\ (A+\Delta A)\cdot\hat{x} &= b+\Delta b \\ &\Downarrow \\ \frac{||\Delta x||}{||\hat{x}||} &\le \underbrace{||A^{-1}||\cdot||A||}_{\equiv\kappa(A)}\cdot\left(\frac{||\Delta A||}{||A||}+\frac{||\Delta b||}{||A||\cdot||\hat{x}||}\right) \end{aligned}$$

Condition number

Example: Condition number for solving linear systems

$$
\begin{array}{rcl}
-A \cdot x & = & -b \\
(A + \Delta A) \cdot \hat{x} & = & b + \Delta b \\
\hline
A \cdot (\hat{x} - x) + \Delta A \cdot \hat{x} & = & \Delta b \\
 & \Downarrow & \\
\Delta x & = & A^{-1} \cdot (\Delta b - \Delta A \cdot \hat{x}) \\
||\Delta x|| & \leq & ||A^{-1}|| \cdot (||\Delta A|| \cdot ||\hat{x}|| + ||\Delta b||) \\
\frac{||\Delta x||}{||\hat{x}||} & \leq & ||A^{-1}|| \cdot \left(||\Delta A|| + \frac{||\Delta b||}{||\hat{x}||} \right) \\
 & \Downarrow & \\
\frac{||\Delta x||}{||\hat{x}||} & \leq & ||A^{-1}|| \cdot ||A|| \cdot \left(\frac{||\Delta A||}{||A||} + \frac{||\Delta b||}{||A|| \cdot ||\hat{x}||} \right)
\end{array}
$$

Subtractive cancellation

$$\hat{a} \approx a > 0,\ \hat{b} \approx b > 0 \quad \Longrightarrow \quad \begin{array}{rcl} \hat{a} + \hat{b} & \approx & a + b \\ \hat{a} \cdot \hat{b} & \approx & a \cdot b \\ \hat{a}/\hat{b} & \approx & a/b \end{array}$$

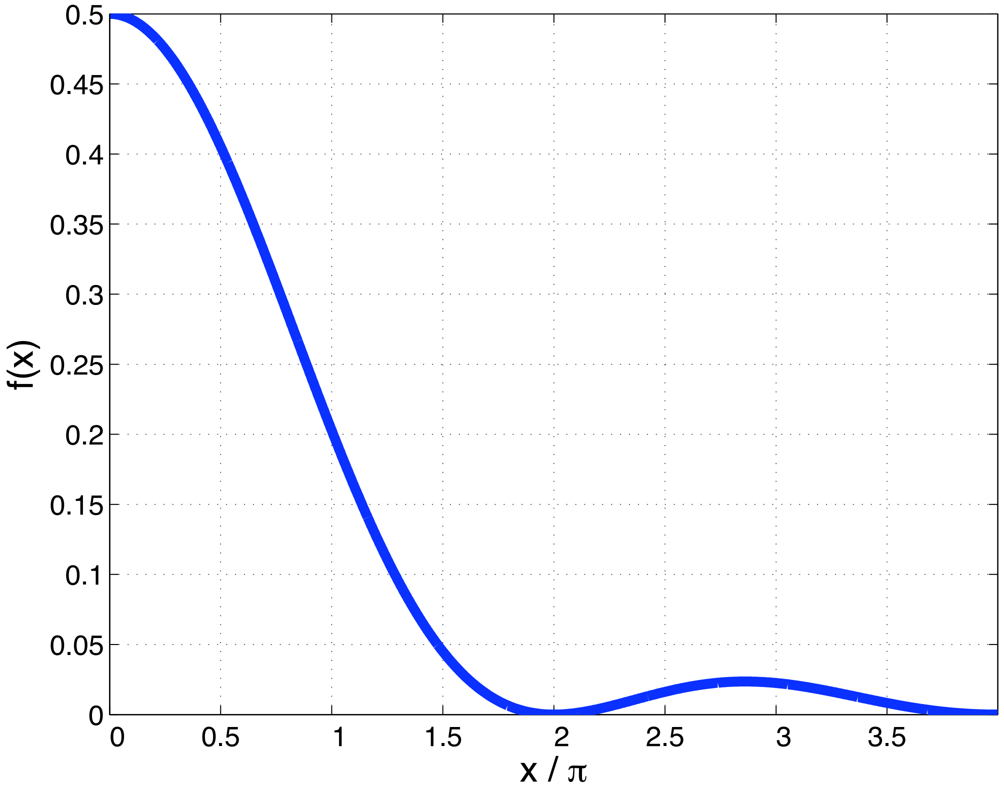
- **May lose accuracy when subtracting nearly equal values**

Example:
12 ⇒ 3 significant digits

$$\begin{array}{rr} & .123456789\color{blue}{xxx} \\ - & .123456789\color{blue}{yyy} \\ \hline & .000000000\color{red}{zzz} \end{array}$$

Example: Subtractive cancellation

$$f(x) \equiv \frac{1 - \cos x}{x^2} < \frac{1}{2} \qquad \forall x \neq 0$$



Example: Subtractive cancellation

$$f(x) \quad \equiv \quad \frac{1 - \cos x}{x^2} < \frac{1}{2} \qquad \forall x \neq 0$$

- Suppose *x* = 1.2 × 10^{-5} and precision = 10 digits:

$$\begin{aligned} c \equiv \cos(1.2 \times 10^{-5}) \quad &\approx \quad 0.999\ 999\ 999\ 9 \\ 1 - c \quad &= \quad 0.000\ 000\ 000\ {\color{red}1} = 10^{-10} \\ \frac{1-c}{x^2} \quad &= \quad \frac{10^{-10}}{1.44 \times 10^{-10}} = {\color{red}0.6944\ldots} \end{aligned}$$

- "*1 - c*" is exact, but result is same size as error in *c*

Cancellation magnifies errors

$$
\begin{aligned}
\hat{a} &\equiv a \cdot (1 + \Delta a) \\
\hat{b} &\equiv b \cdot (1 + \Delta b) \\
\hat{y} &\equiv \hat{a} - \hat{b} \\
\left| \frac{\hat{y} - y}{y} \right| &= \left| \frac{-a\Delta a - b\Delta b}{a - b} \right| \\
&\leq \max(|\Delta a|, |\Delta b|) \cdot \frac{|a| + |b|}{|a - b|}
\end{aligned}
$$

Cancellation **not always bad!**

- Operands may be exact (*e.g.*, initial data)
- Cancellation may be symptomatic of ill-conditioning
- Effect may be irrelevant, *e.g.*, ***x* + *(y - z)*** is safe if

$$0 < y \approx z \ll x$$

A few bad errors can ruin a computation

- Instability can often be traced to just a few insidious errors, not just accumulation of lots of error

$$\begin{aligned} e &\equiv \lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n \\ &= 2.71828\ldots \end{aligned}$$

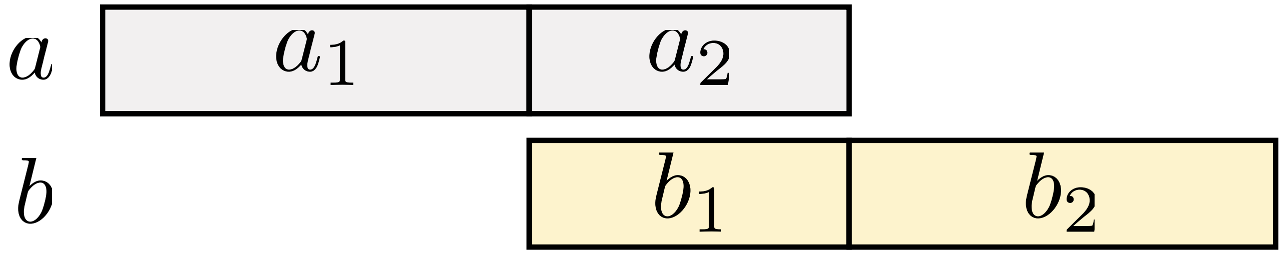
n	$\hat{f}_n$	$\lvert\hat{f}_n - e\rvert$
10	2.593743	1.25×10^{-1}
1,000	2.717051	1.23×10^{-3}
10,000	2.718597	3.15×10^{-4}
1,000,000	2.595227	1.23×10^{-1}

Rounding errors can be beneficial

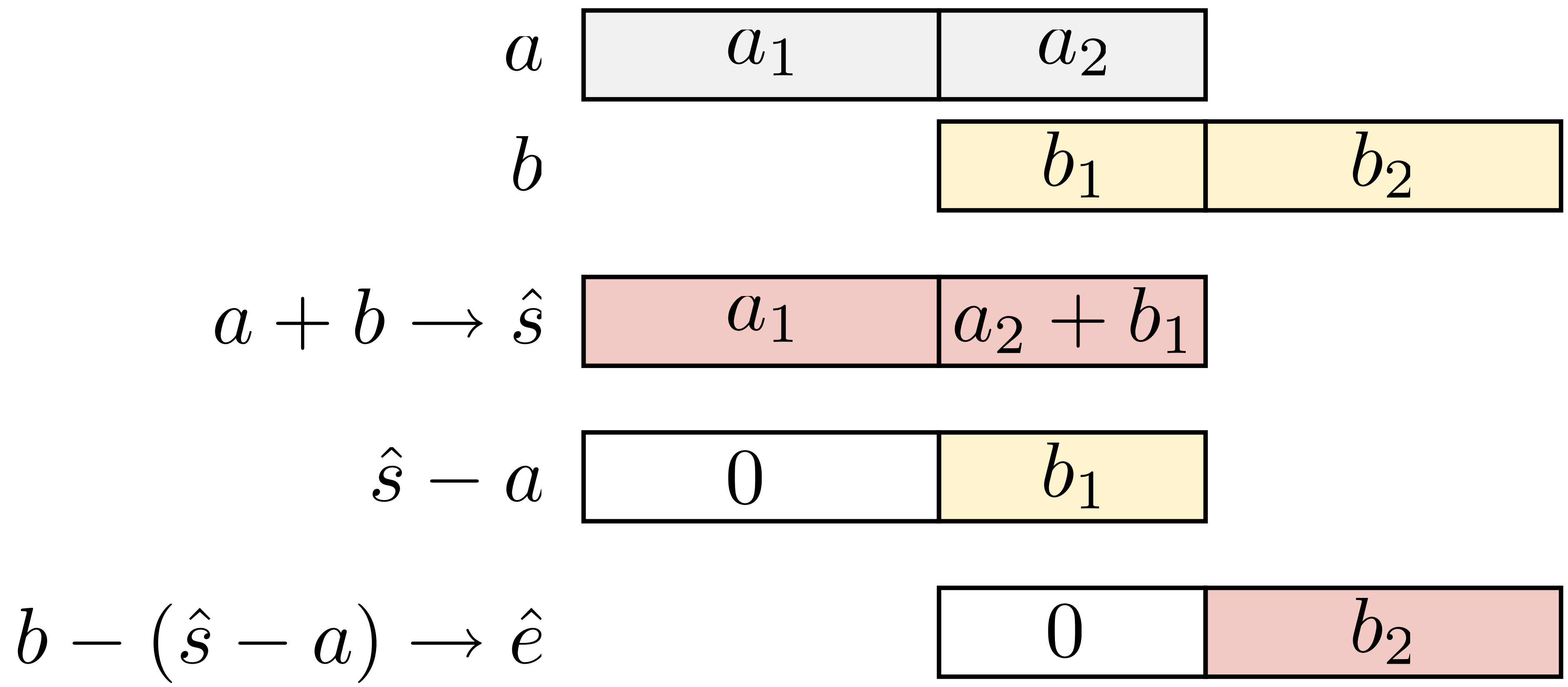
$$A = \begin{pmatrix} 0.4 & -0.6 & 0.2 \\ -0.3 & 0.7 & -0.4 \\ -0.1 & -0.4 & 0.5 \end{pmatrix}$$

- A has 3 eigenvalues: 0, 0.4394…, 1.161…
- Eigenvalue 0 has eigenvector $[1, 1, 1]^T$
- Consider power method with initial guess $[1, 1, 1]^T$
  - **1-step in exact arithmetic ⇒ 0, so no eigenpair information**
  - With rounding, get principal eigenvector in ~ 40 steps

Non-overlapping expansion



Non-overlapping expansion



$(\hat{s}, \hat{e}) : \;\; \hat{s} + \hat{e} = a + b$ ⇐ **Higher accuracy in fixed precision**

Misconceptions [Higham]

- Cancellation is always bad
- Rounding errors can overwhelm only for many accumulations
- Rounding errors cannot be beneficial
- Accuracy always limited by precision

- Final computed answer cannot be more accurate than intermediate values
- Short computation w/o cancellation, underflow, and overflow is accurate
- Increasing precision always increases accuracy

Designing stable algorithms

- Avoid subtracting quantities contaminated by error if possible
- Minimize size of intermediate quantities
- Look for formulations that are mathematically but not numerically equivalent
- Update paradigm: new = old + correction
- Use well-conditioned transformations, *e.g.*, multiply by orthogonal matrices
- Avoid unnecessary overflow and underflow

Example 1: Solve $Ax = b$ using mixed-precision iterative refinement

When single-precision is faster than double

- On Cell
  - **SPEED(single) = 14x SPEED(double)**: 204.8 Gflop/s vs. 14.6 Gflop/s
  - SPEs fully IEEE-compliant for double, but only support round-to-zero in single
- On regular CPUs with SIMD units
  - **SPEED(single) ~ 2x SPEED(double)**
  - SSE2: S(single) = 4 flops / cycle vs. S(double) = 2 flops/cycle
  - PowerPC Altivec: S(single) = 8 flops / cycle; no double (4 flops / cycle)
- On a GPU, **might not have double-precision** support

Improving an estimate using Newton's method

$$\begin{aligned}
f(x) &= 0 \\
x^{(t+1)} &\leftarrow x^{(t)} - \frac{f(x^{(t)})}{f'(x^{(t)})} \\
&\Downarrow \\
f(x) &= Ax - b \\
x^{(t+1)} &\leftarrow x^{(t)} - A^{-1}(A \cdot x^{(t)} - b) \\
&\Downarrow \\
d^{(t)} &\equiv x^{(t+1)} - x^{(t)} = A^{-1} \cdot r^{(t)}
\end{aligned}$$

One step of “iterative refinement”

$$d^{(t)} \quad \equiv \quad x^{(t+1)} - x^{(t)} = A^{-1} \cdot r^{(t)}$$

- Inner loop of iterative refinement algorithm

$$\hat{x} = \text{Estimated solution to } Ax = b$$

$$\hat{r} \leftarrow b - A \cdot \hat{x}$$

$$\text{Solve } A \cdot \hat{d} = \hat{r}$$

$$\hat{x}^{(\text{improved})} \leftarrow \hat{x} + \hat{d}$$

Mixed-precision iterative refinement

- **Theorem**: Given a computed LU factorization of A, and

$$\begin{aligned} \hat{x} &= \text{Estimate, in precision } \epsilon \\ \hat{r} &= \text{Residual, in precision } \epsilon^2 \\ \eta &= \epsilon \cdot \|\, |A^{-1}| \cdot |\hat{L}| \cdot |\hat{U}| \,\|_\infty < 1 \end{aligned}$$

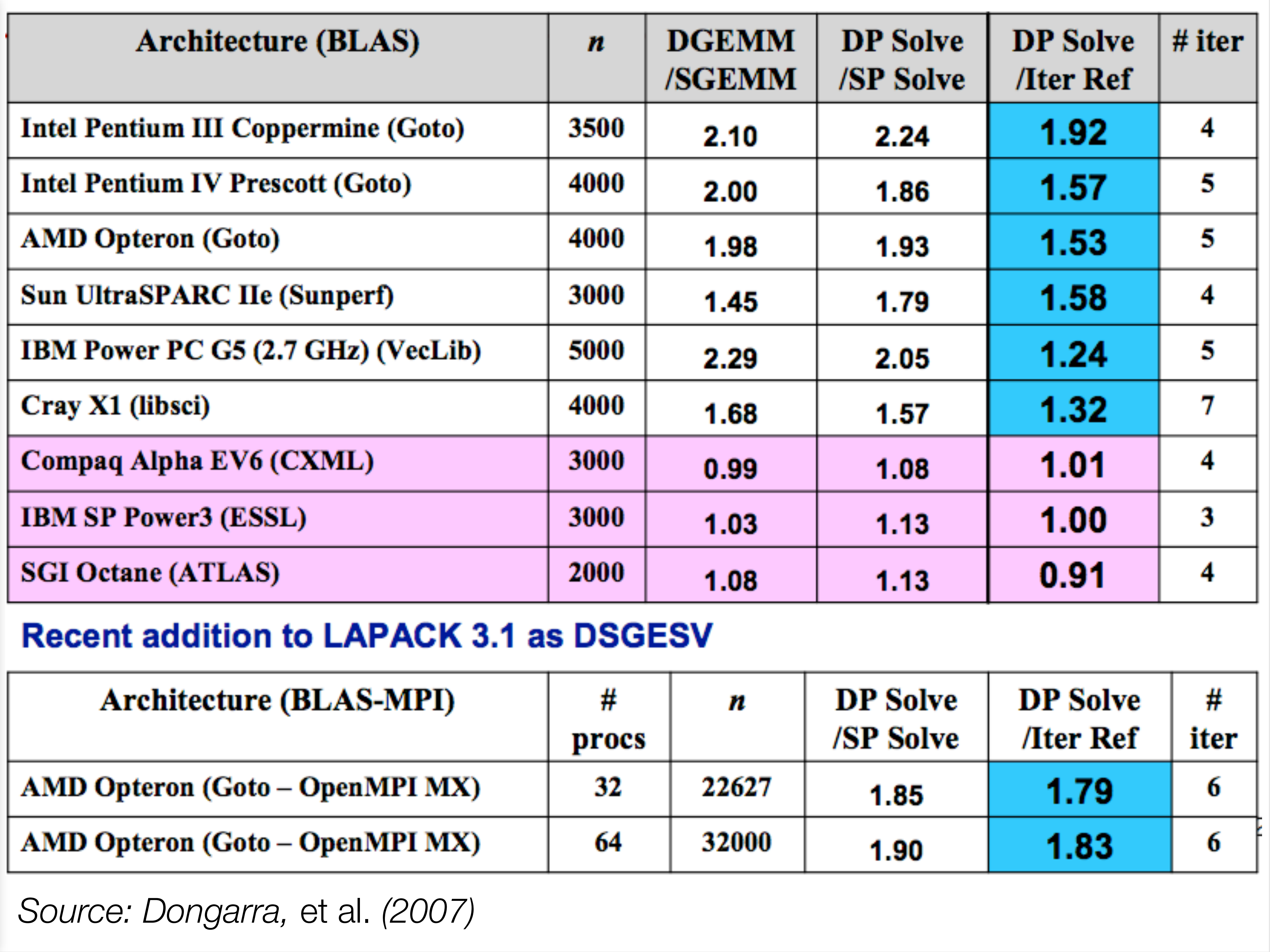
- Then repeated iterative refinement converges by η at each stage, and

$$\frac{\|x^{(t)} - x\|_\infty}{\|x\|_\infty} \rightarrow O(\epsilon)$$

Independent of κ*(A)*!

When single-precision is much faster than double

- Compute a solution in single-precision, *e.g.*, LU ⇒ **O(n^3) single-flops**
- Apply one-step of iterative refinement
  - Compute residual in double ⇒ **O(n^2) double-flops**
  - Solve in single, *e.g.*, reuse LU factors ⇒ **O(n^2) single-flops**
  - Correct in double, round to single ⇒ **O(n) double-flops**
- Matrix needs to be not-too-ill-conditioned

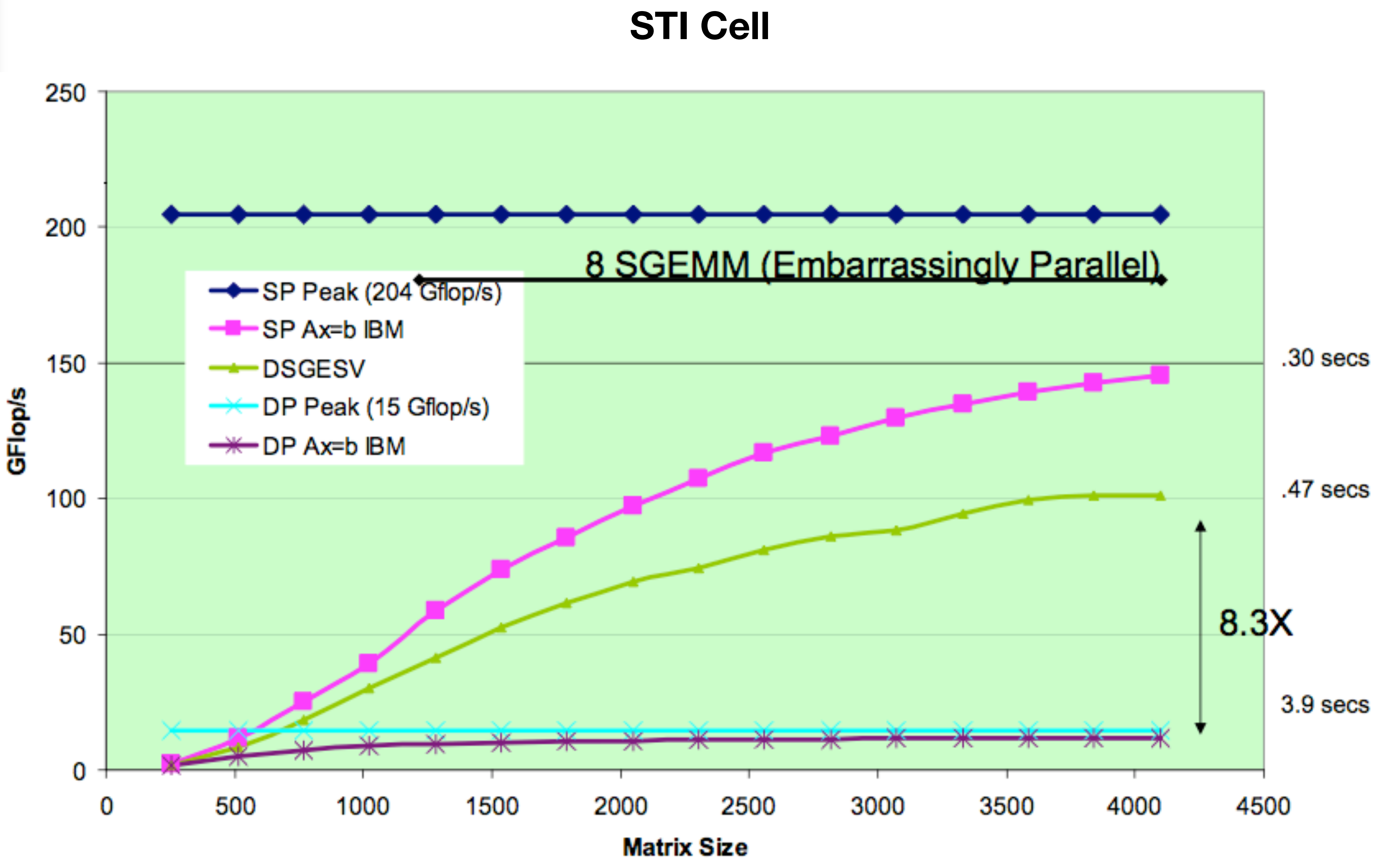


Architecture (BLAS)	*n*	DGEMM /SGEMM	DP Solve /SP Solve	DP Solve /Iter Ref	# iter
Intel Pentium III Coppermine (Goto)	3500	2.10	2.24	1.92	4
Intel Pentium IV Prescott (Goto)	4000	2.00	1.86	1.57	5
AMD Opteron (Goto)	4000	1.98	1.93	1.53	5
Sun UltraSPARC IIe (Sunperf)	3000	1.45	1.79	1.58	4
IBM Power PC G5 (2.7 GHz) (VecLib)	5000	2.29	2.05	1.24	5
Cray X1 (libsci)	4000	1.68	1.57	1.32	7
Compaq Alpha EV6 (CXML)	3000	0.99	1.08	1.01	4
IBM SP Power3 (ESSL)	3000	1.03	1.13	1.00	3
SGI Octane (ATLAS)	2000	1.08	1.13	0.91	4

Recent addition to LAPACK 3.1 as DSGESV

Architecture (BLAS-MPI)	# procs	*n*	DP Solve /SP Solve	DP Solve /Iter Ref	# iter
AMD Opteron (Goto – OpenMPI MX)	32	22627	1.85	1.79	6
AMD Opteron (Goto – OpenMPI MX)	64	32000	1.90	1.83	6

Source: Dongarra, et al. *(2007)*



Source: Dongarra, et al. *(2007)*

Fixed-precision iterative refinement

- **Theorem**: If instead *r* computed in same precision **ε**, then

$$\frac{||\hat{x} - x||_\infty}{||x||_\infty} \lesssim 2n \cdot \kappa(A) \cdot \epsilon$$

- Compare to bound for the original computed solution using LU:

$$\frac{||\hat{x} - x||_\infty}{||x||_\infty} \lesssim 3n \cdot \frac{||\ |A^{-1}| \cdot |\hat{L}| \cdot |\hat{U}|\ ||_\infty}{||x||_\infty} \cdot \epsilon$$

Administrivia

Two joint classes with CS 8803 SC

- **Tues 2/19**: Floating-point issues in parallel computing by me
- **Tues 2/26**: GPGPUs by Prof. Hyesoon Kim
- **Both classes meet in Klaus 1116E**

Administrative stuff

- **New room** (dumpier, but cozier?): College of Computing Building **(CCB) 101**
- **Accounts**: Apparently, you already have them
- Front-end login node: **ccil.cc.gatech.edu** (CoC Unix account)
  - We “own” **warp43—warp56**
  - Some docs (**MPI**): http://www-static.cc.gatech.edu/projects/ihpcl/mpi.html
  - **Sign-up** for mailing list: https://mailman.cc.gatech.edu/mailman/listinfo/ihpc-lab

Homework 1:
Parallel conjugate gradients

- Implement a parallel solver for Ax = b (serial C version provided)
  - Evaluate on three matrices: 27-pt stencil, and two application matrices
  - “Simplified:” No preconditioning
  - **Bonus**: Reorder, precondition
- Performance models to understand scalability of your implementation
  - Make measurements
  - Build predictive models
- Collaboration encouraged: Compare programming models or platforms

Parallelism and stability trade-offs

Obstacles to fast and stable parallel numerical algorithms

- **Algorithms that work on small problems may fail at large sizes**
    - Round-off accumulates
    - Condition number increases
    - Probability of “random instability” increases

- **Fast (parallel) algorithm may be less stable ⇒ trade-off**

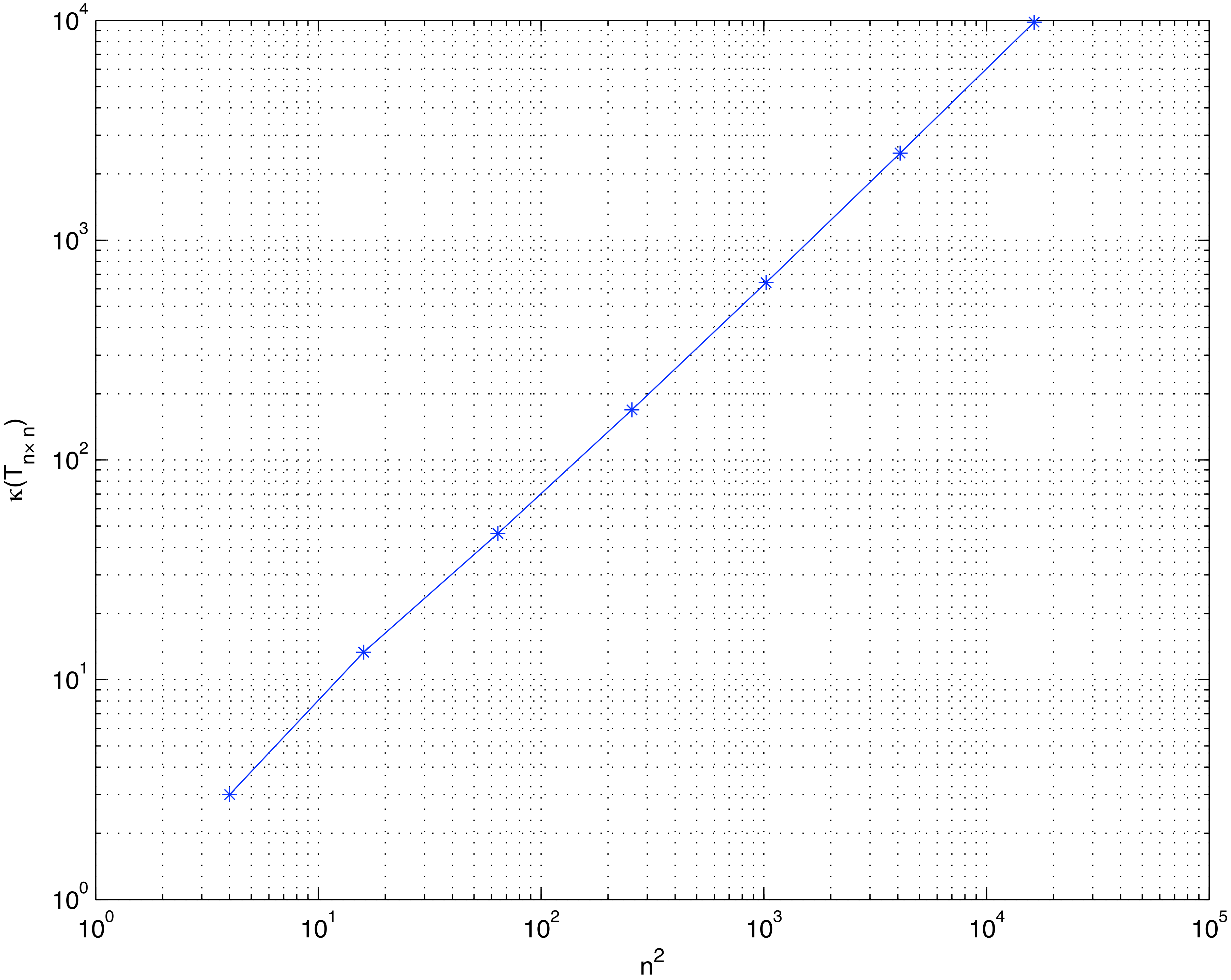
Round-off accumulates

$$\begin{aligned}\bar{x} &= \frac{1}{n}\sum_{i=1}^{n} x_i \\ \sigma(x) &= \frac{1}{n-1}\sum_{i=1}^{n}(x_i-\bar{x})^2\end{aligned}$$

$$\begin{aligned}\text{Let } \hat{\sigma}(x) &= \textit{computed } \sigma(x), \\ \text{and } \epsilon &= \text{machine precision.} \\ \text{then:} & \end{aligned}$$

$$\frac{\hat{\sigma}(x)-\sigma(x)}{\sigma(x)} \le (n+3)\epsilon + O\left(\epsilon^2\right)$$

Condition number of T$_{nxn}$ increases



Random stabilities increase

- If A is an $n \times n$ matrix selected at random [Edelman '92]:

$$Pr\left(\kappa(A) > \frac{1}{\eta}\right) = O(n^{\frac{3}{2}} \cdot \eta)$$

- Let **η** = $10^d \cdot$**ε**. Then if p processors all do plain LU on i.i.d. A matrices:

$$\text{Prob. per sec. that instability occurs}$$
$$\sim p \cdot \frac{\text{(speed in flop/s)}}{\frac{2}{3}n^3} \cdot n^{\frac{3}{2}} \cdot 10^d \cdot \epsilon$$

Trading-off speed and stability: Serial example

- Conventional error bound for naïve matrix multiply

$$|\mathrm{fl}_{\text{naïve}}(A \cdot B) - A \cdot B| \leq n \cdot \epsilon \cdot |A| \cdot |B|$$

- Bound for Strassen’s, O($n^{\text{log\_2 7}}$) ≈ O($n^{2.81}$)

$$||\mathrm{fl}_{\text{Strassen}}(A \cdot B) - A \cdot B||_M \leq O(n^{3.6}) \cdot \epsilon \cdot ||A||_M \cdot ||B||_M$$

Trading-off speed and stability: Parallel example

- Consider *A* to be a dense symmetric positive definite matrix
- Suppose triangular solve is **slow**
- Conventional algorithm:

$$A = R^T \cdot R = \begin{bmatrix} R_{11}^T & 0 & 0 \\ R_{12}^T & R_{22}^T & 0 \\ R_{13}^T & R_{23}^T & R_{33}^T \end{bmatrix} \cdot \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ 0 & R_{22} & R_{23} \\ 0 & 0 & R_{33} \end{bmatrix} \Rightarrow \boxed{||\Delta A|| = O(\epsilon) \cdot \kappa(A)}$$

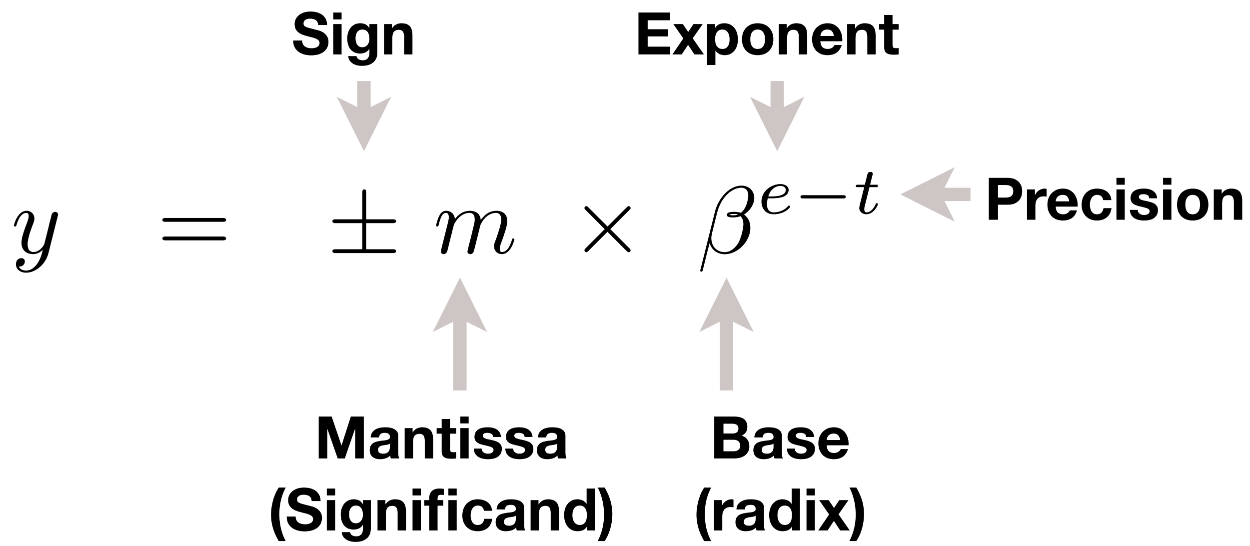
- Fast “block LU” algorithm (no triangular solves)

$$A = L \cdot U = \begin{bmatrix} I & 0 & 0 \\ L_{21} & I & 0 \\ L_{31} & L_{32} & I \end{bmatrix} \cdot \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix} \Rightarrow \boxed{O(\epsilon) \cdot (\kappa(A))^{\frac{3}{2}}}$$

IEEE floating-point arithmetic

Floating-point number systems

- Subset of reals of the form:



Sign

Exponent

$$y = \pm\, m \times \beta^{e-t}$$

Precision

Mantissa (Significand)

Base (radix)

$$y \in F \subset \mathbb{R}$$

$$0 \leq m < \beta^t$$

- Representation = Sign + 2 integers (*m*, *e*); *t*, β implicit

Normalization

$$\begin{aligned} y &= \pm\, m \times \beta^{e-t} \\ m &\geq 2^{t-1} \end{aligned}$$

$$y \in F \subset \mathbb{R}$$

$$0 \leq m < \beta^{t}$$

Normalization

- Set leading digit of *m* implicitly
  - Guarantees unique representation of each value
  - Avoids storage of leading zeros
  - Get extra digit (bit) of precision

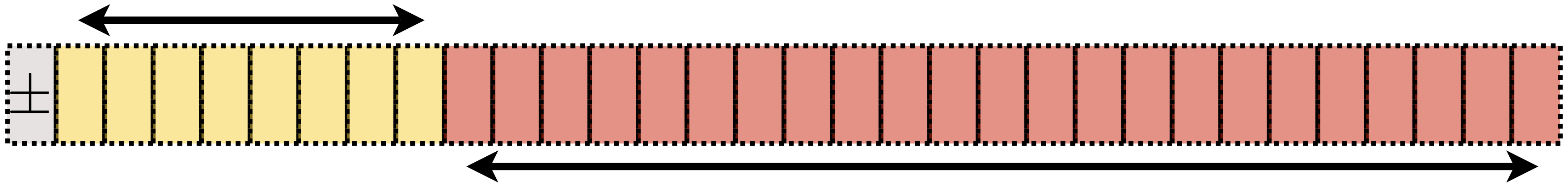
IEEE 754 Standard [Kahan]

- Base-2 representation with *m* “normalized”

$$
\begin{aligned}
y &= \pm\, m \times 2^{e-t} \\
y \neq 0 &\implies m \geq 2^{t-1} \quad \longleftarrow \text{“Normalized”}
\end{aligned}
$$

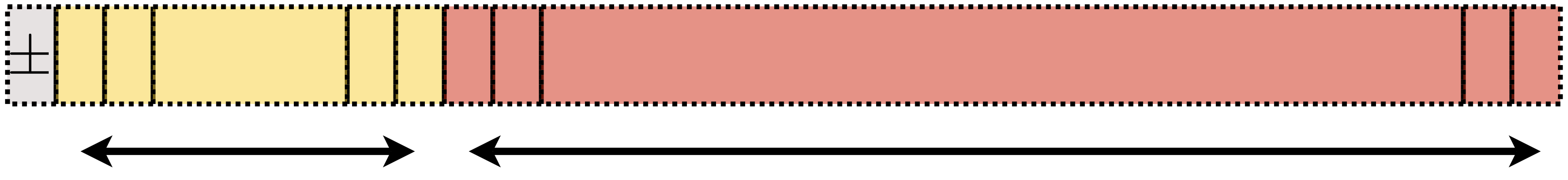
- IEEE **single precision**: **32 bits,** ε ≈ **6×10^{-8}** (`REAL / float`)

$$-125 = e_{\min} \leq e \leq e_{\max} = 128$$



$$0 \leq m < 2^{24} \approx 16 \text{ million}$$

IEEE formats



$$e_{\min} \leq e \leq e_{\max} \qquad 0 \leq m < 2^t$$

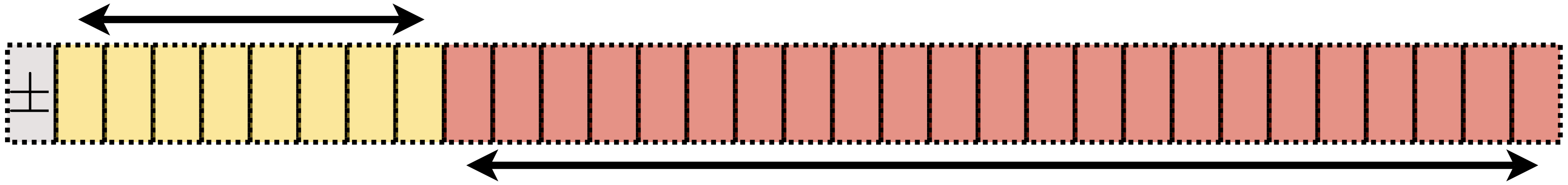
Format	Total bits	Exp. bits ($e_{\min}$, $e_{\max}$)	t-1	ε	Fortran / C
Single	32	8 (-125, 128)	23	6×10^{-8}	`REAL*4` `float`
Double	64	11 (-1021, 1024)	52	10^{-16}	`REAL*8` `double`
Extended (Intel)	80	15 (-16381, 16384)	64	5×10^{-20}	`REAL*10` `long double`

$$y \quad = \quad \pm\, m \times 2^{e-t}$$

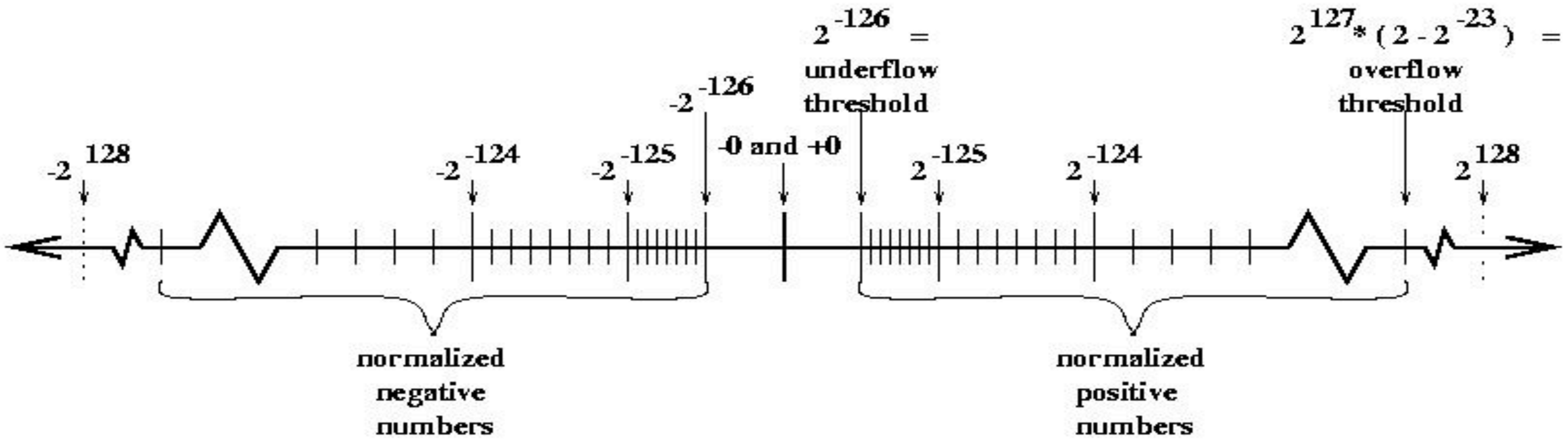
$$y \neq 0 \quad \Longrightarrow \quad m \geq 2^{t-1}$$

← **“Normalized”**

$$-125 = e_{\min} \leq e \leq e_{\max} = 128$$



$$0 \leq m < 2^{24} \approx 16 \text{ million}$$



Rules of arithmetic

- Philosophy: As simple as possible
  - **Correct rounding**: Round exact value to nearest floating-point number
  - **Round to nearest even**, to break ties
  - Other modes: up, down, toward 0
  - Don't actually need exact value to round correctly (!)
- Applies to +, -, *, /, sqrt, conversion between formats ⇒ model holds

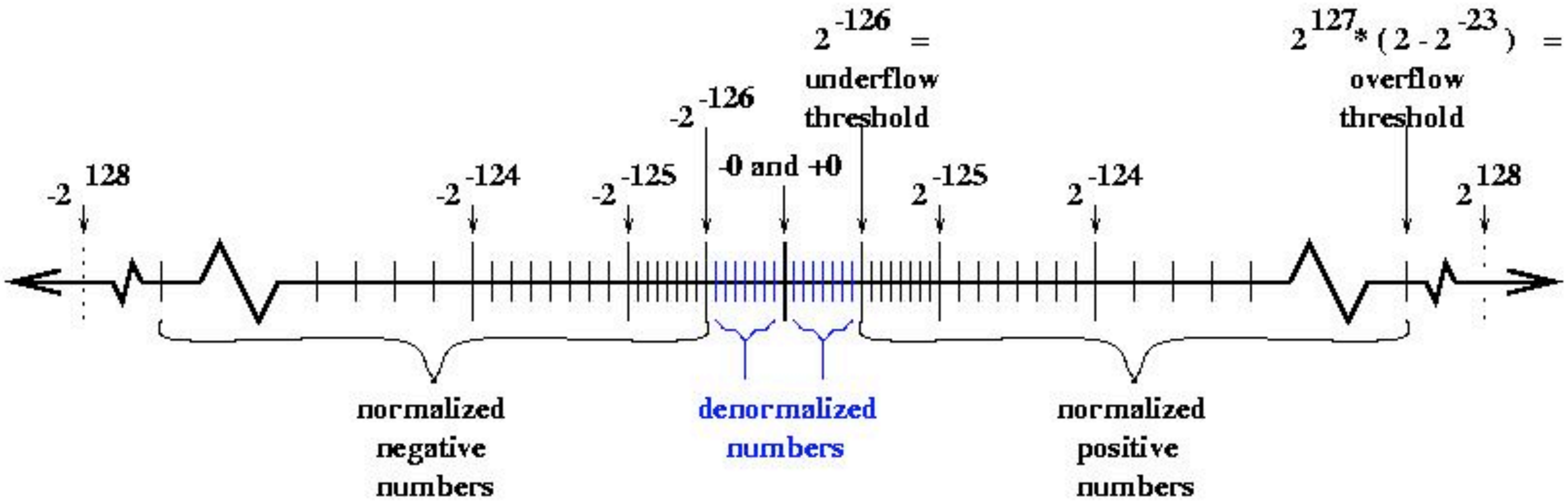
$$\mathrm{fl}(a \text{ op } b) = (a \text{ op } b)(1+\delta), \qquad |\delta| < \epsilon$$

Exception handling

- What happens when exact value is not a real number? Too large or small?
  - Overflow/underflow
  - Invalid, *e.g.*, 0 / 0
  - Divide by zero
  - “Inexact”Inexact
- Answer: Exception generated
  - Set flag and continue (default)
  - Trap to custom handler

Denormalized numbers (“denorms”)

- Value exceeds overflow threshold or falls below underflow threshold



- Underflow permits safely executing: if (a != b) then x = a / (a-b)

Other special values

- Infinity (INF): Divide by zero
- Not-a-number (NaN): 0 / 0; 0 * INF; INF - INF; INF / INF; sqrt(-1)
  - Operations involving NaNs generate NaNs (except “max”/”min”)
  - Can use to represent uninitialized or missing data
  - Quiet vs. signaling

Example 2: Fast and accurate bisection on GPUs

Dense symmetric eigensolvers

- Tridiagonal reduction — Transform *A* to *T* using, *e.g.*, Householder: O(n^3)

$$T = \begin{pmatrix} a_1 & b_1 & & & \\ b_1 & a_2 & b_2 & & \\ & b_2 & \cdot & \cdot & \\ & \cdot & \cdot & \cdot & \\ & & \cdot & a_{n-1} & b_{n-1} \\ & & & b_{n-1} & a_n \end{pmatrix}$$

- Solve $Tv = \lambda v$
- λ is eigenvalue for A; back-transform *v* to get corresponding eigenvector

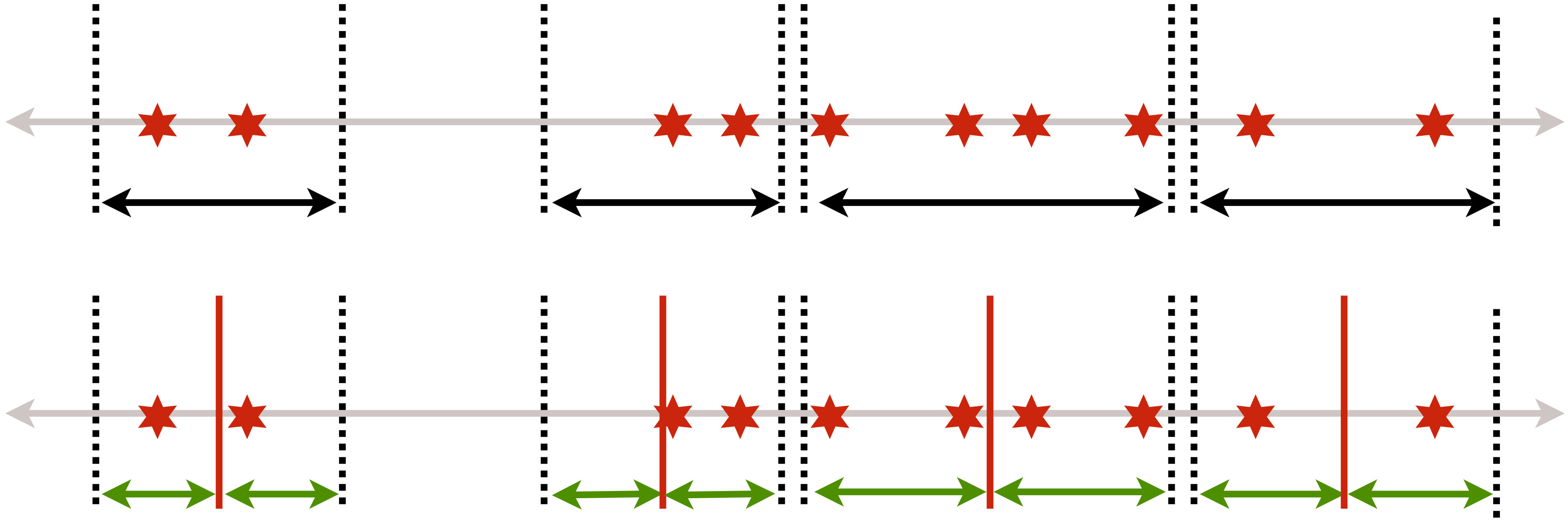
Bisection kernel: *Count(x)*

- **Bisection**: Finds eigenvalues in a given interval [a, b) by “search”
- Inner-loop of one algorithm for solving $Tv = \lambda v$

```
Count(x)        ⇐ Counts no. of eigenvalues of T less than x
  count ← 0
  $d \leftarrow 1$
  for $i = 1$ to $n$ do
    $d \leftarrow a_i - x - \frac{b_{i-1}^2}{d}$
    if $d < 0$ then
      count ← count + 1
  return count
```

Bisection algorithm

★ = An eigenvalue



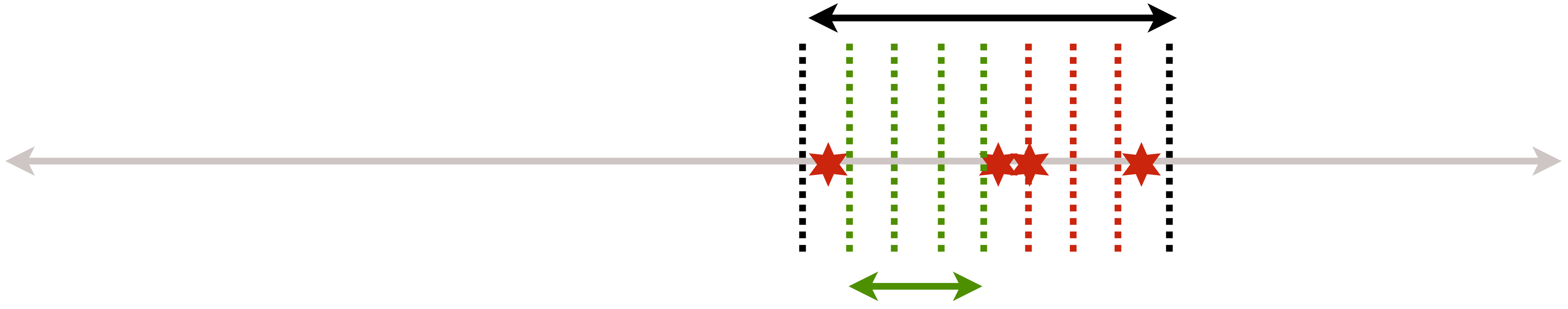
$[l_i, u_i)$

$C(u_i) - C(l_i)$

Repeatedly subdivide intervals until each one contains 1 eigenvalue.

Multisection: Increase parallelism

Easiest parallelization:
Evaluate Count(x) on multiple intervals simultaneously, at cost of redundancy.

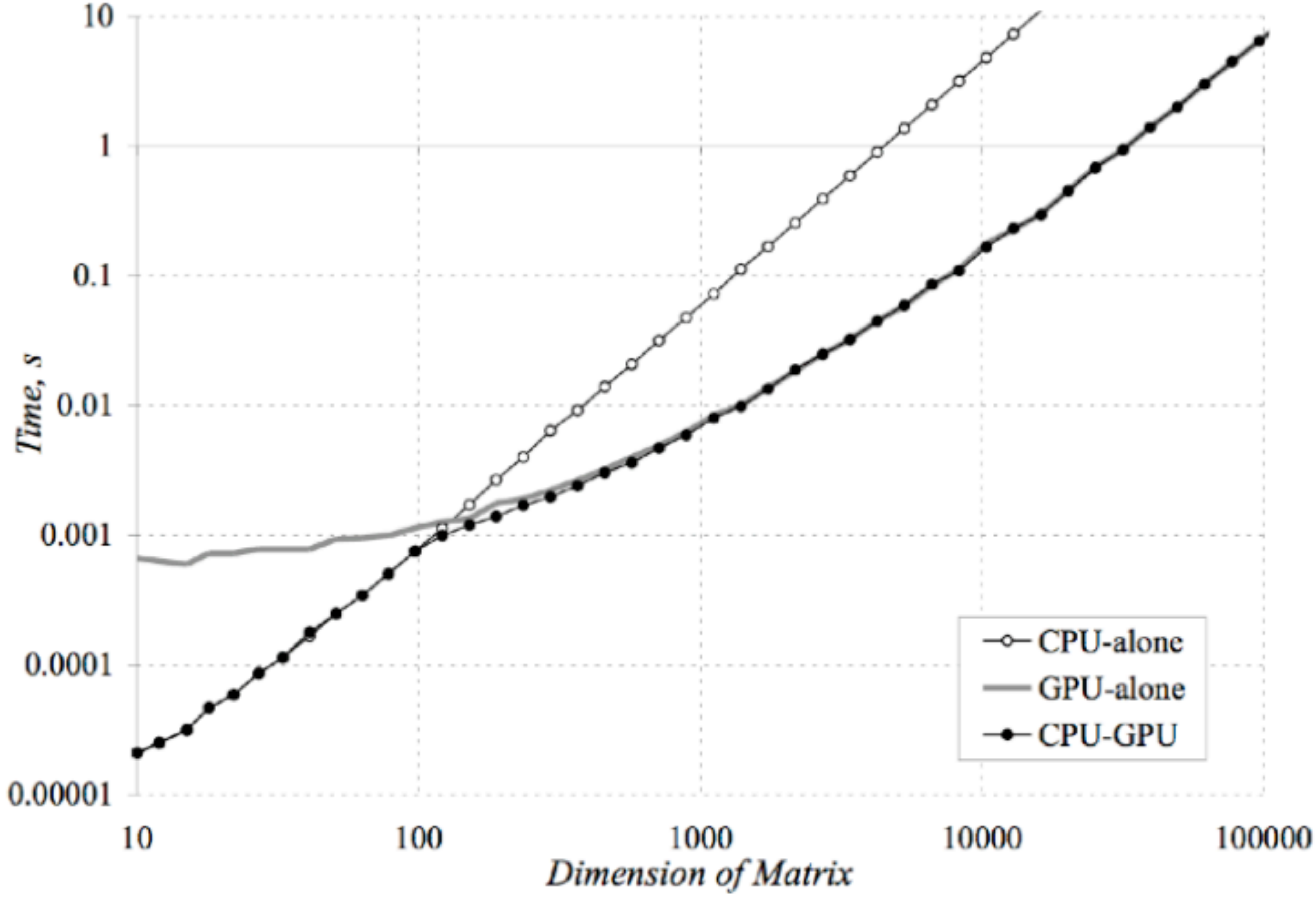


Same Count(.)

Correctness requires monotonicity

- Count(x) must be monotonic for overall algorithm to work
  - Can modify Count(x) slightly to guarantee it is monotonic iff basic operations on scalars (+, -, *, /) are monotonic
  - IEEE floating-point semantics guarantee monotonicity

```
$\vdots$
    $d \leftarrow a_i - x - \frac{b_{i-1}^2}{d}$
    if $d < 0$ then count $\leftarrow$ count $+1$
$\vdots$
```


10
1
0.1
0.01
0.001
0.0001
0.00001
Time, s
10
100
1000
10000
100000
Dimension of Matrix
CPU-alone
GPU-alone
CPU-GPU

“In conclusion…”

The impact of parallelism on numerical algorithms

- **Larger problems** magnify errors: Round-off, ill-conditioning, instabilities
- **Reproducibility**: a + (b + c) ≠ (a + b) + c
- Fast **parallel algorithm** may be much **less stable** than fast serial algorithm
- **Flops cheaper** than communication
- **Speeds at different precisions** may vary significantly [*e.g.*, SSE$_k$, Cell]
- Perils of **arithmetic heterogenity**, *e.g.*, CPU vs. GPU support of IEEE

Backup slides

A brief history of floating-point

[Slide from Demmel]

- von Neumann and Goldstine (1947): “Can’t expect to solve most big **[n>15]** systems without carrying many decimal digits **[d>8]**, otherwise the computed answer would be completely inaccurate.”
- Turing (1949): Backward error analysis
- Wilkinson (1961): Rediscovers and publicizes idea — Turing Award 1970
- Kahan (late 1970s): IEEE 754 floating-point standard — Turing Award 1989
  - Motivated by many years of machines with slightly differing arithmetics
  - First implementation in Intel 8087
  - Nearly universally implemented

Recall: Condition number for $Ax = b$

$$\frac{||\Delta x||}{||\hat{x}||} \leq \underbrace{||A^{-1}|| \cdot ||A||}_{\equiv \kappa(A)} \cdot \left(\frac{||\Delta A||}{||A||} + \frac{||\Delta b||}{||A|| \cdot ||\hat{x}||} \right)$$

Condition number

Alternative view of conditioning for *Ax = b*

- Recall conditioning relationship for Ax = b based on perturbation theory

$$\frac{||\Delta x||}{||\hat{x}||} \leq \underbrace{||A^{-1}|| \cdot ||A||}_{\equiv \kappa(A)} \cdot \left(\frac{||\Delta A||}{||A||} + \frac{||\Delta b||}{||A|| \cdot ||\hat{x}||} \right)$$

- Consider bound on forward error based on residual, *r = b - A·x_computed*

$$\begin{aligned} r = b - A\hat{x} \quad &\Longrightarrow \quad \hat{x} = A^{-1} \cdot (b - r) = A^{-1}(Ax - r) = x - A^{-1}r \\ &\Longrightarrow \quad \Delta x = A^{-1}r \\ &\Longrightarrow \quad \underline{||\Delta x|| \leq ||A^{-1}|| \cdot ||r||} \end{aligned}$$