

中国科学院大学
2026

数值代数 Numerical Algebra

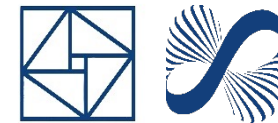
第一章、绪论与应用背景

Introduction and Applications

Chensong Zhang, AMSS

<https://lsec.cc.ac.cn/~zhanges>

1



第一章、绪论与应用背景

第二章、线性代数基础

第三章、矩阵计算基础

第四章、线性方程组的直接方法

第五章、线性方程组的迭代方法

第六章、最小二乘问题的数值方法

第七章、Krylov子空间方法

第八章、特征值问题的数值方法



第一章、绪论与应用背景

本章作为课程导论，介绍数值线性代数的核心问题及其在科学与工程中的实际背景。

内容涵盖线性空间、线性变换与矩阵的基本概念，并通过典型应用，如线性回归、主成分分析、PageRank等，展示数值线性代数的重要性。同时，讨论数值实现中的一些关键挑战，包括浮点运算、稀疏矩阵处理、数据移动与并行效率等，为学生建立课程的整体框架。

基础知识

Basics of numerical linear algebra

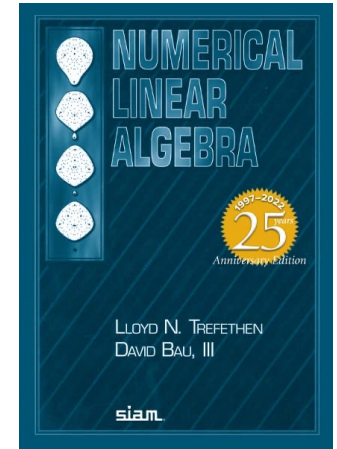
/1.1

Numerical Linear Algebra



- Real-world problems are too complex for pen-and-paper solutions.
- L. N. Trefethen and D. Bau, III on **NLA** (1997):

“If any other mathematical topic is as fundamental to the mathematical sciences as **calculus** and **differential equations**, it is **numerical linear algebra**.”



- **Some applications** (NLA or Applied Linear Algebra):
 - **Scientific/Engineering Computing:** Simulates physical systems governed by ODEs/PDEs/IEs
 - **Principal Component Analysis:** Uses SVD for dimensionality reduction and feature extraction
 - **Image Compression:** Uses the Discrete Cosine Transform, like in JPEG
 - **Recommendation Systems:** Uses matrix factorization techniques (a form of low-rank approximation)
 - **PageRank:** Models the web as a graph and finds the principal eigenvector of the link matrix
 - **Deep Learning:** Training DNNs involves massive matrix multiplications and solving optimization problems

Classical Problems in NLA



Numerical linear algebra (NLA) is the essential computational engine powering modern science and engineering, enabling scientific discoveries like **large-scale simulations, AI, quantum computing, and financial analytics**, via matrix decompositions, linear system solvers, eigenvalue solvers, etc.

- **Prob 1. Linear System solving (LS)** problems

$$\text{Find } \mathbf{x} \in \mathbb{C}^n : A\mathbf{x} = \mathbf{b}, \text{ where } A \in \mathbb{C}^{n \times n}, \mathbf{b} \in \mathbb{C}^n$$

- **Prob 2. Least Squares (LSQ)** problems

$$\arg \min_{\mathbf{x} \in \mathbb{C}^n} \|A\mathbf{x} - \mathbf{b}\|_2^2, \text{ where } A \in \mathbb{C}^{m \times n}, \mathbf{b} \in \mathbb{C}^m$$

- **Prob 3. EigenValue problems (EVP)**

$$\text{Find } \lambda \in \mathbb{C}, \mathbf{0} \neq \mathbf{x} \in \mathbb{C}^n \text{ s.t. } A\mathbf{x} = \lambda\mathbf{x}, \text{ where } A \in \mathbb{C}^{n \times n}$$

Other Problems in NLA



- **Prob 4. Singular Value Problems**

Find singular triplets $(\sigma_i, \mathbf{u}_i, \mathbf{v}_i) : A\mathbf{v}_i = \sigma_i\mathbf{u}_i, A^H\mathbf{u}_i = \sigma_i\mathbf{v}_i$
such that $\sigma_i \geq 0, \|\mathbf{u}_i\| = \|\mathbf{v}_i\| = 1$

- **Prob 5. Low-Rank Approximation Problems**

$$\min_{\text{rank}(X) \leq k} \|A - X\|_F$$

- **Prob 6. Matrix Function Action Problems**

Compute $\mathbf{y} = f(A)\mathbf{b}$, e.g. $f(t) = e^t, t^\alpha, \log(t), \dots$

AI and data science: matrix completion, sparse representation, spectral graph learning, kernel methods, ...

Core building blocks: BLAS, LU, Cholesky, QR, SVD, Schur, and other related matrix factorizations, ...

典型应用问题

Some practical applications

/1.2

Linear Regression



- Regression problem:

To predict a **continuous model** of a target variable based on one or more input features

$$\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)\}$$

where $\mathbf{x}_i \in \mathbb{R}^d$, $y_i \in \mathbb{R}$, $i = 1 : n$.

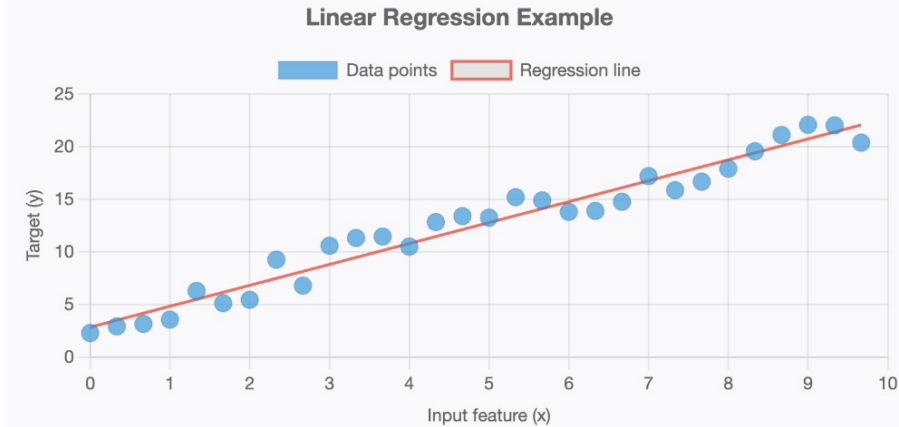
- Linear regression model:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_d x_d + \epsilon$$

$$\min_{\beta \in \mathbb{R}^{d+1}} \|\mathbf{y} - X\beta\|^2 \quad \longrightarrow \quad X^T X \beta = X^T \mathbf{y}$$
$$X := \begin{bmatrix} 1 & x_{11} & \cdots & x_{1d} \\ 1 & x_{21} & \cdots & x_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{nd} \end{bmatrix}$$

Provided that X has full column rank

Model: $y = 2x + 3 + \text{noise}$



Practical Applications:

- Predicting house prices based on size, location, and amenities
- Forecasting temperature based on historical weather data
- Estimating blood pressure based on age, weight, and lifestyle
- Predicting stock prices based on market indicators

Clustering and Eigen Decomposition

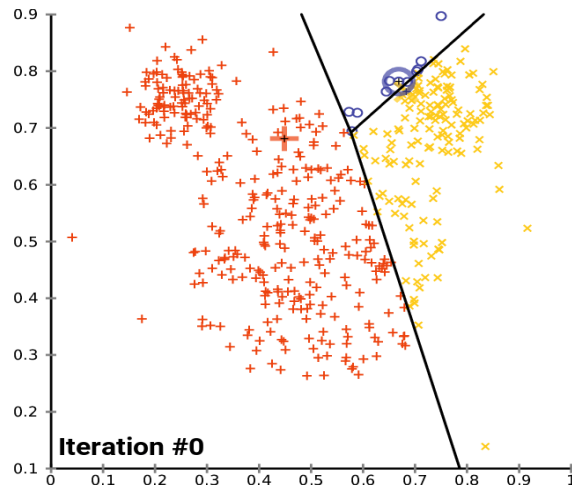


- Clustering:

Partition a set of data points into groups called clusters such that:

- Points in the same cluster are **similar**
- Points from different clusters are **dissimilar**

- k-Means clustering (MacQueen, J. 1967):



Limitations of k-means:

- ✗ Assumes clusters are convex and isotropic
- ✗ Sensitive to initialization
- ✗ Performs poorly with nonlinearly separable data

- Spectral Clustering:

1. Construct Similarity Graph:

Create affinity matrix W where $W_{ij} = \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{2\sigma^2}\right)$

2. Compute Graph Laplacian:

$$L = D - W$$

where D is the diagonal degree matrix: $D_{ii} = \sum_j W_{ij}$

3. Normalized Laplacian:

$$L_{\text{sym}} = D^{-1/2} L D^{-1/2} = I - D^{-1/2} W D^{-1/2}$$

4. Eigenvalue Decomposition:

Compute the first k eigenvectors $v_1, v_2, \dots, v_k$ of L_{sym}

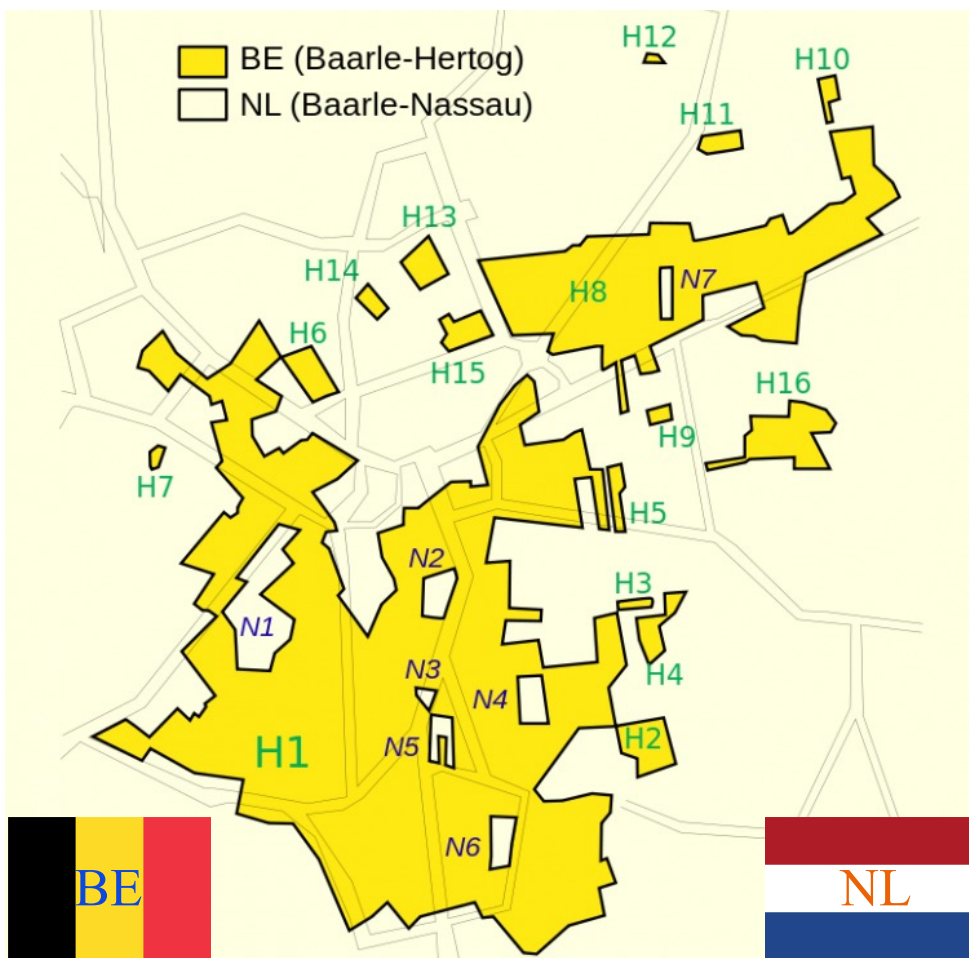
5. New Representation:

Form matrix $V \in \mathbb{R}^{n \times k}$ from the eigenvectors

6. Apply k-means:

Apply k-means to the rows of V to obtain clusters

A More Involved Example



Baarle-Hertog/Nassau:

The World's Most Confusing Border

The twin municipalities of Baarle-Hertog (Belgium) and Baarle-Nassau (Netherlands) together form what is widely considered the most complex international border in the world. Within and around the town of Baarle, there are 30 separate enclaves — 22 Belgian parcels within the Netherlands and 8 Dutch parcels within those Belgian enclaves, creating enclaves within enclaves that defy conventional cartography.

Q: How to design a method to label the BE / NL parts?

<https://blogs.transparent.com/dutch/baarle-one-of-the-strangest-municipalities-on-earth/>

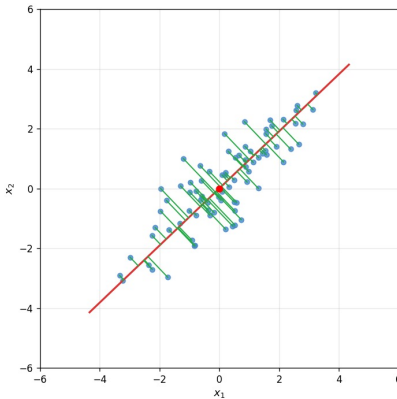
Principal Component Analysis



● Principal Component Analysis (PCA):

Given centered data $X \in \mathbb{R}^{n \times p}$ (n samples, p features), PCA finds orthogonal directions that maximize variance, i.e.

$$V_k = \arg \max_{V^T V = I_k} \|XV\|_F^2$$



$$\mathbf{v}_1 = \arg \max_{\|\mathbf{v}\|=1} \sum_{i=1}^n (\mathbf{v}^T \mathbf{x}_i)^2$$

$$p = 2, n = 80$$

● Maximum projected variance

$\Leftrightarrow$ Minimum orthogonal reconstruction error

$$\arg \max_{V^T V = I_k} \|XV\|_F^2 = \arg \min_{V^T V = I_k} \|X - XVV^T\|_F^2$$

EVD-Based
Method

1. Center data: $X_c = X - \bar{X}$
2. Compute covariance matrix: $C = \frac{1}{n-1} X_c^T X_c$
3. Eigen decomposition: $C = W\Lambda W^T$
4. Select top-k eigenvectors (columns of W)
5. Project: $T = X_c W_k$

SVD-Based
Method

1. Center data: $X_c = X - \bar{X}$
2. Compute SVD: $X_c = U\Sigma V^T$
3. Principal components: Columns of V
4. Scores: $T = U\Sigma$ or $T = X_c V$

Practical Applications:

- Data visualization (reducing to 2D/3D)
- Noise reduction
- Feature extraction
- Speeding up machine learning algorithms

PageRank and Eigenvectors



● PageRank:

PageRank is a link analysis algorithm developed by Google founders (1998). A random surfer follows an outgoing link with probability d , and jumps uniformly to any page with probability $1 - d$.

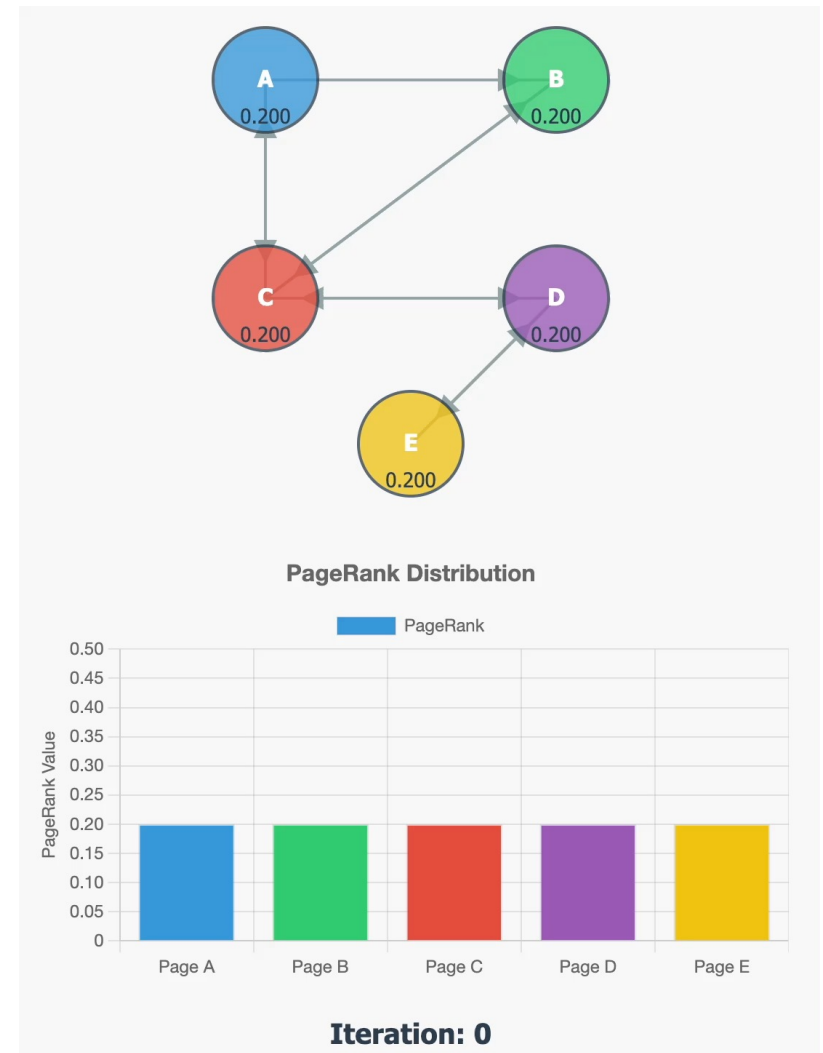
$$PR(A) = \frac{1 - d}{n} + d \sum_{B_i \rightarrow A} \frac{PR(B_i)}{C(B_i)} \quad \text{Q: How to solve it?}$$

- $PR(A)$ is the PageRank of page A
- $C(B_i)$ is the number of outbound links from B_i

L. Page & S. Brin, 1998: Find the principal eigenpair of the so-called Google matrix (need to handle dangling node j by setting $M_{:,j} = 1/n$):

$$G := dM + \frac{1 - d}{n} \mathbf{1} \mathbf{1}^T, \quad M_{ij} := \begin{cases} 1/C(j), & j \rightarrow i \\ 0, & \text{otherwise} \end{cases}$$

The spectral radius of G is 1 and its eigenvector (PageRank vector) is positive.



Landscape of Parallel Computing



The Landscape of Parallel Computing Research: A View from Berkeley

Berkeley
UNIVERSITY OF CALIFORNIA



Krste Asanovic
Ras Bodik
Bryan Christopher Catanzaro
Joseph James Gebis
Parry Husbands
Kurt Keutzer
David A. Patterson
William Lester Plishker
John Shalf
Samuel Webb Williams
Katherine A. Yelick

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-183
<http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html>

December 18, 2006

3.2.1 Machine Learning

One of the most promising areas for the future of computing is the use of statistical machine learning to make sense from the vast amounts of data now available due to faster computers, larger disks, and the use of the Internet to connect them all together.

Michael Jordan and Dan Klein, our local experts in machine learning, found two dwarfs that should be added to support machine learning:

- **Dynamic programming** is an algorithmic technique that computes solutions by solving simpler overlapping subproblems. It is particularly applicable for optimization problems where the optimal result for a problem is built up from the optimal result for the subproblems.
- **Backtrack and Branch-and-Bound:** These involve solving various search and global optimization problems for intractably large spaces. Some implicit method is required in order to rule out regions of the search space that contain no interesting solutions. Branch and bound algorithms work by the divide and conquer principle: the search space is subdivided into smaller subregions (“branching”), and bounds are found on all the solutions contained in each subregion under consideration.

“To maximize programmer productivity, future programming models must be more human-centric than the conventional focus on hardware or applications.”

Thirteen Dwarfs in Parallel Computing



Dwarf Name	Description	Examples	Performance Limitations (Current Understanding)
Dense Linear Algebra	Operations on dense matrices/vectors with regular memory access patterns	BLAS, ScaLAPACK	Computationally limited; modern GPUs and TPUs excel here
Sparse Linear Algebra	Operations on matrices with many zero values using compressed storage formats	SpMV, OSKI, SuperLU	50% computation, 50% memory bandwidth; hybrid approaches work best
Spectral Methods	Computations in frequency domain rather than time or spatial domains	FFT, Cooley-Tukey algorithm	Memory latency limited; can be optimized with cache-oblivious algorithms
N-Body Methods	Calculations based on interactions between discrete points	Barnes-Hut, Fast Multipole	Computationally limited; use spatial partitioning (octrees, k-d trees)
Structured Grids	Regular grid computations where points are updated together	Cactus, Lattice-Boltzmann	Memory bandwidth limited; excellent scaling with proper blocking
Unstructured Grids	Irregular grid where data locations and connectivity must be explicit	ABAQUS, FIDAP	Memory latency limited; partitioning and graph-based approaches
MapReduce	Parallel execution on independent data sets with combined results	Monte Carlo, Google MapReduce	Problem dependent; communication overhead becomes dominant at scale
Combinational Logic	Functions implemented with logical operations and stored state	Encryption, CRC computation	CRC problems bandwidth limited; excellent for SIMD/ASIC acceleration
Graph Traversal	Visiting nodes in a graph by following edges	Breadth-first search (BFS), DFS	Memory latency limited; significant gains from prefetching and locality
Dynamic Programming	Solving problems by combining solutions to overlapping subproblems	Optimization algorithms	Memory latency limited; can be parallelized with wavefront approaches, but suffers from synchronization overhead
Backtrack and Branch-and-Bound	Finding optimal solutions by recursive subdivision and pruning	Search, global optimization	Memory latency limited with synchronization overhead; irregular workloads and load imbalance issues
Graphical Models	Graphs representing random variables as nodes and dependencies as edges	Bayesian networks, Hidden Markov Models	Memory bandwidth limited with communication overhead
Finite State Machine	Systems defined by states, transitions based on inputs and current state	Parsing algorithms, protocol implementations	"Nothing helps!"; remains difficult to parallelize; modern approaches use speculative execution and state partitioning

Dwarf Popularity (Red Hot → Blue Cool)



	HPC	Embed	SPEC	ML	Games	DB
1 Dense Matrix	Red	Red	Red	Red	Red	Yellow
2 Sparse Matrix	Red	Yellow	Yellow	Red	Red	Light Blue
3 Spectral (FFT)	Red	Yellow	Light Blue	Yellow	Yellow	Light Blue
4 N-Body	Red	Light Blue	Yellow	Light Blue	Yellow	Light Blue
5 Structured Grid	Red	Red	Red	Light Blue	Yellow	Light Blue
6 Unstructured	Red	Light Blue	Light Blue	Yellow	Yellow	Light Blue
7 MapReduce	Red	Light Blue	Green	Red	Light Blue	Red
8 Combinational	Light Blue	Red	Light Blue	Green	Light Blue	Green
9 Graph Traversal	Light Blue	Red	Yellow	Red	Yellow	Yellow
10 Dynamic Prog	Light Blue	Yellow	Light Blue	Red	Light Blue	Red
11 Backtrack/ B&B	Light Blue	Light Blue	Light Blue	Red	Light Blue	Yellow
12 Graphical Models	Light Blue	Light Blue	Light Blue	Red	Light Blue	Yellow
13 FSM	Light Blue	Red	Red	Yellow	Yellow	Red

Source: David Patterson, <https://web.stanford.edu/class/ee380/Abstracts/070131-BerkeleyView1.7.pdf>

Foundation & Future of Scientific Computing



- Numerical Linear Algebra (NLA) is the cornerstone of scientific computing, more generally computational science: **numerical stability** and **fundamental understanding** matter more than **computational sophistication** alone.
- Today's Landscape:
 - **Scientific computing**: NLA kernels dominate runtime in climate modeling, quantum chemistry, and various other engineering simulations;
 - **AI/ML backbone**: Matrix operations power neural networks, transformers, optimization algorithms, etc.
- Pressing Imperatives:
 - **Stability at scale**: As systems become larger, numerical error accumulation requires new analysis;
 - **Hardware-aware algorithms**: NLA must evolve with novel architectures (quantum, neuromorphic);
 - **Cross-domain integration**: Bridging SC with AI demands → unified numerical frameworks.

数值挑战

Numerical challenges

/1.3

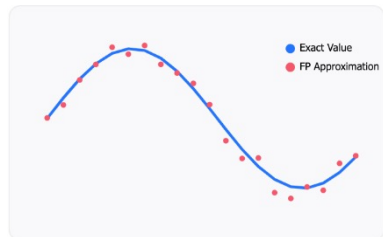
Considerations in Numerical LA



Floating-Point Precision & Error

Computational Imperfections

Digital computers represent real numbers with finite precision, leading to rounding errors and numerical instability.



- **Rounding Errors:** Accumulate during repeated calculations
- **Condition Number:** Measures sensitivity to input changes
- **Stability:** Algorithms must minimize error propagation
- **Mixed Precision:** Strategic use of different precision levels
- **Error Analysis:** Rigorous mathematical treatment of errors

计算存在误差



Scale & High-Dimensional Data

Big Data Challenges

Modern datasets often have very high dimensions that make traditional algorithms infeasible.



- **Sparse Matrices:** Specialized storage and algorithms
- **Streaming Algorithms:** Process data in single pass
- **Randomized Methods:** Probabilistic algorithms (RNLA)
- **Dimensionality Reduction:** PCA, random projections
- **Out-of-Core Computation:** Processing larger-than-memory data

矩阵稀疏特性



Data Movement & Energy Cost

The Memory Wall Problem

Moving data between memory hierarchy levels consumes more time and energy than computation itself.

Memory Hierarchy & Access Times	
Registers	~0.1ns
Cache	~1ns
RAM	~10ns
Disk/SSD	~100ns
Network	~1000ns

- **Memory Hierarchy:** Registers > Cache > RAM > Disk
- **Locality Optimization:** Maximizing data reuse in cache
- **Communication-Avoiding Algorithms:** Minimize data movement
- **Energy Efficiency:** Critical for large-scale systems

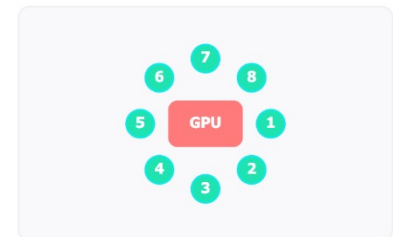
提高实现效率



Parallel Computing

Modern Architectures

Modern hardware requires algorithms designed for concurrency across multiple processing units.



- **Multicore CPUs:** Shared memory parallelism
- **GPUs:** Massively parallel architectures
- **Distributed Systems:** Cluster computing (MPI)
- **Load Balancing:** Even distribution of work
- **Synchronization:** Minimizing coordination overhead

现代计算硬件

数值挑战1：浮点运算精度

Numerical challenges

/1.3a

Some Historical Comments



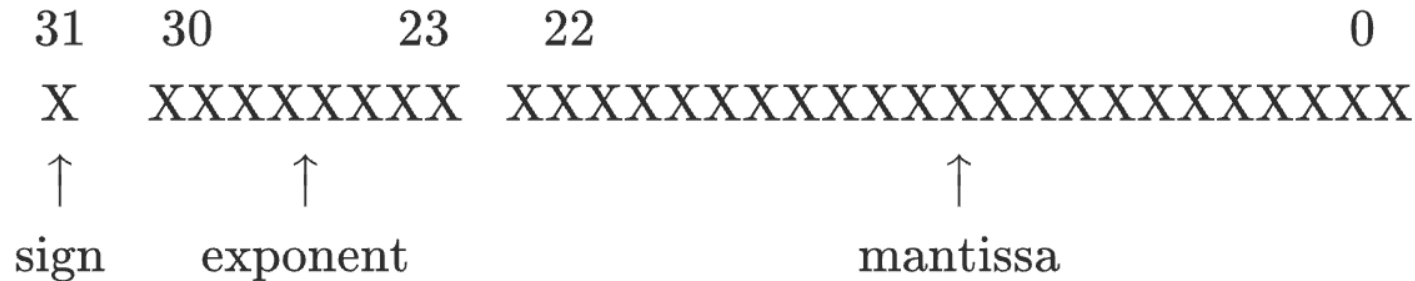
- Not so long ago ... computing machines are very different from now
 - Fixed-point arithmetic → Floating-point arithmetic: SP, DP, QP, ...
 - Floating-point arithmetic: Different standards for different machines
 - IEEE 754 Standard (1985, 2008, 2019), Intel 8087 coprocessor, ...
- Several multi-precision arithmetic simulators are introduced since 1980s:
 - Maple, Mathematica, vpa@Matlab ...
 - Julia, half, numpy, ...
- **NLA Example:** Solving an algebraic system to certain accuracy on a computer
 - The problem could be ill-conditioned (sensitive to perturbation) ...
 - If the precision is low, quality of solution might be very bad
 - The way to improve accuracy is using higher precision (costly) or **iterative refinement**



Floating-Point Number Storage



IEEE single-precision floating-point number



f: fraction field, with hidden
leading 1 (normalized)

$$\text{Number} = (-1)^s 2^{e-127} (1 \cdot f),$$

where $s = 0$ for positive numbers, 1 for negative numbers,
 $e = \text{exponent}$ (between 0 and 255), and
 $f = \text{mantissa}$.

- Fixed-size FP format
- Representation range
- Significand / Mantissa
- Unit round-off / precision
- Machine epsilon
- Little-endian
- Big-endian
- IEEE 754-1985 (FP32, FP64)
- IEEE 754-2008 (FP16, FP128)

Ref: Higham, N. J. and Mary, T. "Mixed Precision Algorithms in Numerical Linear Algebra", Acta Numerica, 2022;31:347-414.

Diagram illustrating the IEEE 754 floating-point format bit fields:

- 1** bit for **sign** (S)
- 11** bits for **exponent** (EEE.....EEE)
- 52** bits for **significand** (MMMM.....MMMM)

Note: **Significand** has a hidden leading bit “**1**” in front of the **mantissa**!

Exercise

- C.-S. Zhang, AMSS

Examples of FP Arithmetic (4 digits)



Example using decimal

$$A = 9.999 \times 10^{-1}, B = 1.610 \times 10^{-1}, A+B = ?$$

Step 1. Align the smaller exponent with the larger one.

$$B = 0.0161 \times 10^1 = 0.016 \times 10^1 \text{ (round off)}$$

Step 2. Add significands

$$9.999 + 0.016 = 10.015, \text{ so } A+B = 10.015 \times 10^1$$

Step 3. Normalize

$$A+B = 1.0015 \times 10^2$$

Step 4. Round off

$$A+B = 1.002 \times 10^2$$

Example using decimal

$$A = 1.110 \times 10^{10}, B = 9.200 \times 10^{-5} \quad A \times B = ?$$

Step 1. Exponent of $A \times B = 10 + (-5) = 5$

Step 2. Multiply significands

$$1.110 \times 9.200 = 10.212000$$

Step 3. Normalize the product

$$10.212 \times 10^5 = 1.0212 \times 10^6$$

Step 4. Round off

$$A \times B = 1.021 \times 10^6$$

Step 5. Decide the sign of $A \times B$ ($+ \times + = +$)

$$\text{So, } A \times B = + 1.021 \times 10^6$$

Source: <https://homepage.divms.uiowa.edu/~ghosh/>



Table 1. Parameters for floating-point arithmetics: number of bits in significand (mantissa), including the implicit most significant bit; number of bits in exponent; and, to three significant figures, unit roundoff u , smallest positive (subnormal) number $x_{\min}^{(s)}$, smallest normalized positive number $x_{\min}$ and largest finite number $x_{\max}$. In Intel's bfloat16 specification, subnormal numbers are not supported [10], so any number less than $x_{\min}$ in magnitude is flushed to zero; the value shown holds if subnormal numbers are supported.

	signif.	exp.	u	$x_{\min}^{(s)}$	$x_{\min}$	$x_{\max}$
bfloat16	8	8	3.91×10^{-3}	9.18×10^{-41}	1.18×10^{-38}	3.39×10^{38}
fp16	11	5	4.88×10^{-4}	5.96×10^{-8}	6.10×10^{-5}	6.55×10^4
fp32	24	8	5.96×10^{-8}	1.40×10^{-45}	1.18×10^{-38}	3.40×10^{38}
fp64	53	11	1.11×10^{-16}	4.94×10^{-324}	2.22×10^{-308}	1.80×10^{308}
fp128	113	15	9.63×10^{-35}	6.48×10^{-4966}	3.36×10^{-4932}	1.19×10^{4932}

Source: Dongarra J, Grigori L, Higham NJ. “Numerical algorithms for high-performance computational science”. Phil. Trans. R. Soc. A 378: 20190066, 2020

Why Mixed-Precision Is Important



- One of the main **driving forces** is DL: It is tolerant of reduced-precision computations
- Faster hardware with mixed-precision capability
 - Reduce amount of data movement
 - Fit more data in the cache
- Traditional algorithms need to be modified in order to take advantage of mixed-precision computation

Tensor Cores since Volta: 4x faster than regular FP16

V100 PCIe	DP (FP64)	SP (FP32)	Tensor (FP16)
Peak Performance	7 TFLOPs	14 TFLOPs	112 TFLOPs
Mem Bandwidth	900 GB/sec		

Technical Specifications			
	H100 SXM	H100 PCIe	H100 NVL ¹
FP64	34 teraFLOPS	26 teraFLOPS	68 teraFLOPS
FP64 Tensor Core	67 teraFLOPS	51 teraFLOPS	134 teraFLOPS
FP32	67 teraFLOPS	51 teraFLOPS	134 teraFLOPS
TF32 Tensor Core	989 teraFLOPS ²	756 teraFLOPS ²	1,979 teraFLOPS ²
BFLOAT16 Tensor Core	1,979 teraFLOPS ²	1,513 teraFLOPS ²	3,958 teraFLOPS ²
FP16 Tensor Core	1,979 teraFLOPS ²	1,513 teraFLOPS ²	3,958 teraFLOPS ²
FP8 Tensor Core	3,958 teraFLOPS ²	3,026 teraFLOPS ²	7,916 teraFLOPS ²
INT8 Tensor Core	3,958 TOPS ²	3,026 TOPS ²	7,916 TOPS ²
GPU memory	80GB	80GB	188GB
GPU memory bandwidth	3.35TB/s	2TB/s	7.8TB/s ³
Decoders	7 NVDEC 7 JPEG	7 NVDEC 7 JPEG	14 NVDEC 14 JPEG
Max thermal design power (TDP)	Up to 700W (configurable)	300-350W (configurable)	2x 350-400W (configurable)
Multi-instance GPUs	Up to 7 MIGs @ 10GB each	Up to 7 MIGs @ 10GB each	Up to 14 MIGs @ 12GB each
Form factor	SXM	PCIe ➤ dual-slot ➤ air-cooled	2x PCIe ➤ dual-slot ➤ air-cooled
Interconnect	NVLink: ➤ 900GB/s PCIe ➤ Gen5: 128GB/s	NVLink: ➤ 600GB/s PCIe ➤ Gen5: 128GB/s	NVLink: ➤ 600GB/s PCIe ➤ Gen5: 128GB/s
Server options	NVIDIA HGX™ H100 partner and NVIDIA-Certified Systems™ with 4 or 8 GPUs NVIDIA DGX™ H100 with 8 GPUs	Partner and NVIDIA-Certified Systems with 1-8 GPUs	Partner and NVIDIA-Certified Systems with 2-4 pairs
NVIDIA Enterprise	Add-on	Included	Included

¹ Preliminary specifications. May be subject to change. Specifications shown for 2x H100 NVL PCIe cards paired with NVLink Bridge.

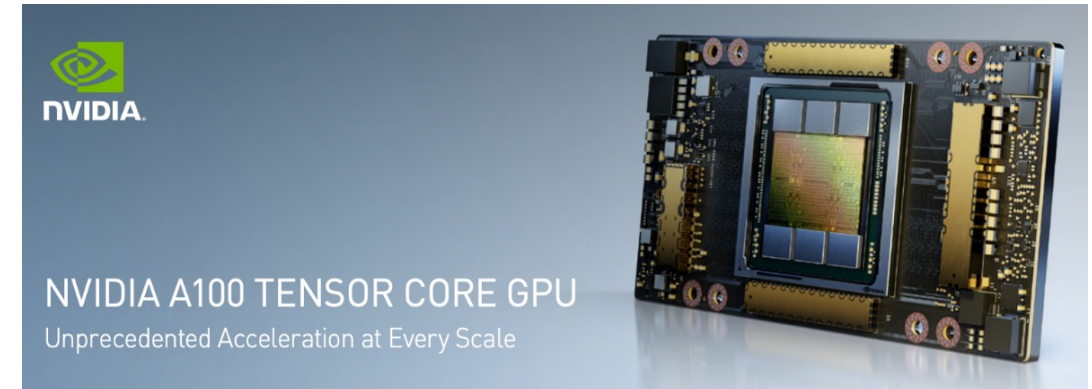
² With sparsity.

³ Aggregate HBM bandwidth.

Driving Force: Deep Learning



- Most deep learning (DL) applications usually do **NOT need to use double precision**
- The computations performed by most models are compositions of a relatively small handful of dense linear algebra calculations . . .
- Many of the mechanisms developed over the past 40 years to **enable general-purpose programs** to run with high performance on modern CPUs are not necessary for deep learning computations



$$\begin{array}{c} D \\ \left[\begin{array}{cccc} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{array} \right] \\ \underbrace{\hspace{1.5cm}}_{\text{fp16 or fp32}} \end{array} = \begin{array}{c} C \\ \left[\begin{array}{cccc} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{array} \right] \\ \underbrace{\hspace{1.5cm}}_{\text{fp16 or fp32}} \end{array} + \begin{array}{c} A \\ \left[\begin{array}{cccc} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{array} \right] \\ \underbrace{\hspace{1.5cm}}_{\text{fp16}} \end{array} \begin{array}{c} B \\ \left[\begin{array}{cccc} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{array} \right] \\ \underbrace{\hspace{1.5cm}}_{\text{fp16}} \end{array}$$

Fused multiply-add (FMA) used in Tensor Core

Ref: J. Dean, "The Deep Learning Revolution and Its Implications for Computer Architecture and Chip Design," 2020 IEEE International Solid-State Circuits Conference (ISSCC), 2020, pp. 8-14.



HPL-MxP Benchmark (Nov, 2025)



Rank	Site	Computer	Cores	HPL-MxP (Eflop/s)	TOP500 Rank	HPL Rmax (Eflop/s)	Speedup
1	DOE/SC/LLNL	El Capitan	11,340,000	16.680	1	1.8090	9.2
2	DOE/SC/ANL	Aurora	9,264,128	11.643	3	1.0120	11.5
3	DOE/SC/ORNL	Frontier	9,066,176	11.390	2	1.3530	8.4
4	EuroHPC/FZJ	JUPITER Booster	4,801,344	6.255	4	1.0000	6.3
5	SoftBank	CHIE-4	662,256	3.304	17	0.1354	24.4
6	AIST	ABCI 3.0	479,232	2.363 ^a	16	0.1451	16.3
7	EuroHPC/CSC	LUMI	2,752,704	2.350	9	0.3797	6.2
8	RIKEN	Fugaku	7,630,848	2.000	7	0.4420	4.5
9	EuroHPC/CINECA	Leonardo	1,824,768	1.842	10	0.2412	7.6
10	KAUST	Shaheen III - GPU	574,464	0.704	18	0.1228	5.7

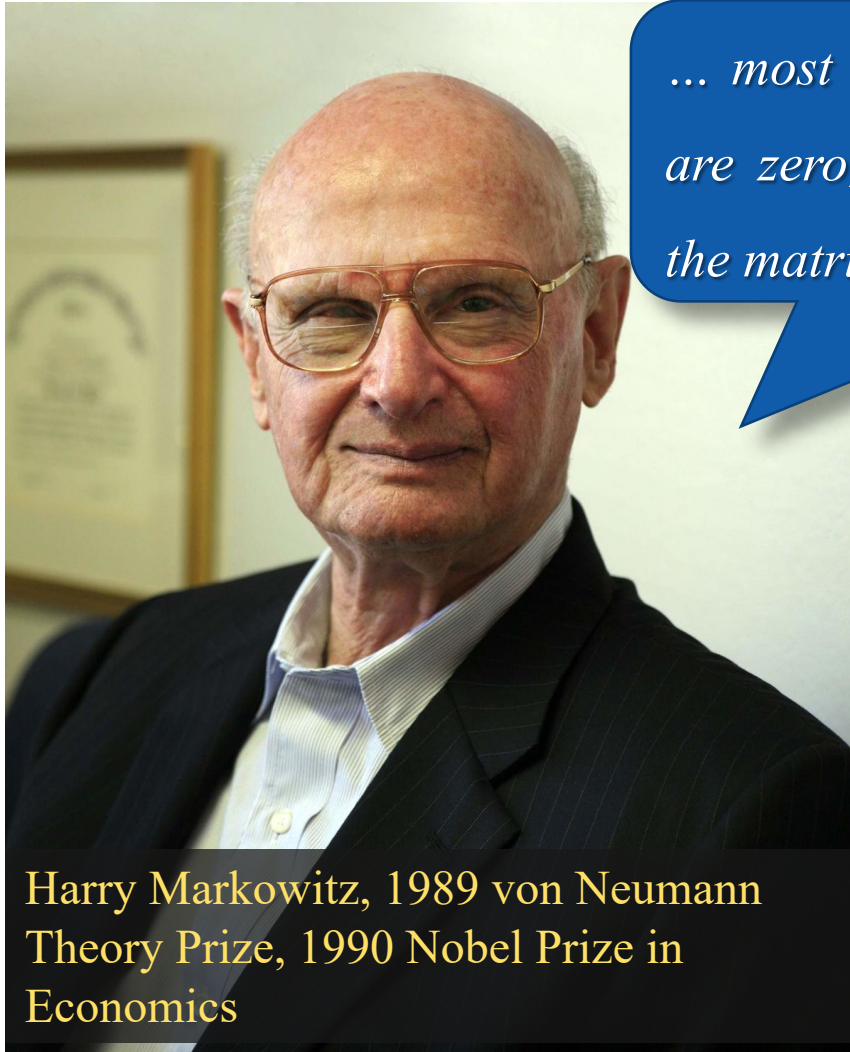
HPL-MxP uses GMRES-based iterative refinement with low-precision LU factors as a preconditioner. Under appropriate conditioning and rounding-error assumptions, the iteration can recover the target accuracy. (Source: <https://hpl-mxp.org/>)

数值挑战2：稀疏矩阵运算

Numerical challenges

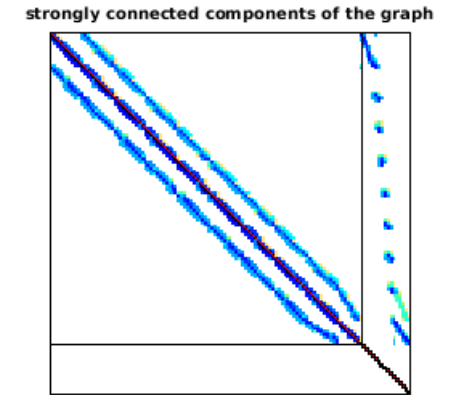
/1.3b

Sparse Matrices



Harry Markowitz, 1989 von Neumann Theory Prize, 1990 Nobel Prize in Economics

*... most of the coefficients in our matrices are zero; i.e., the nonzeros are **sparse** in the matrix ... (1950's)*



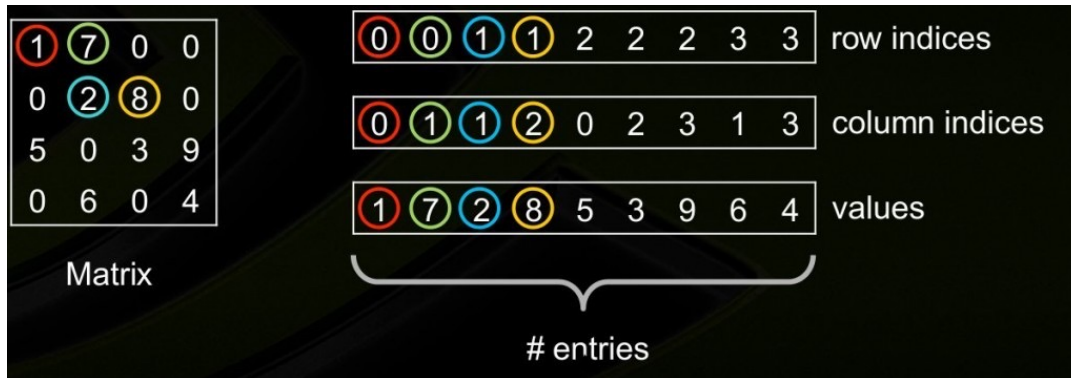
https://sparse.tamu.edu/Goodwin/Goodwin_010

- Sparse matrix: number of nonzeros (nnz's) $\sim O(n)$
- Typical in numerical PDE, graph, network, economy, ...
- **Q: How to store sparse matrices?**
CSR, CSRx, CSC, DIA, ELL, COO, HYB, CSR5, ...
- **Q: How to perform operations on sparse matrices?**
SpMV, SpMM (Sparse*Dense), SpGEMM

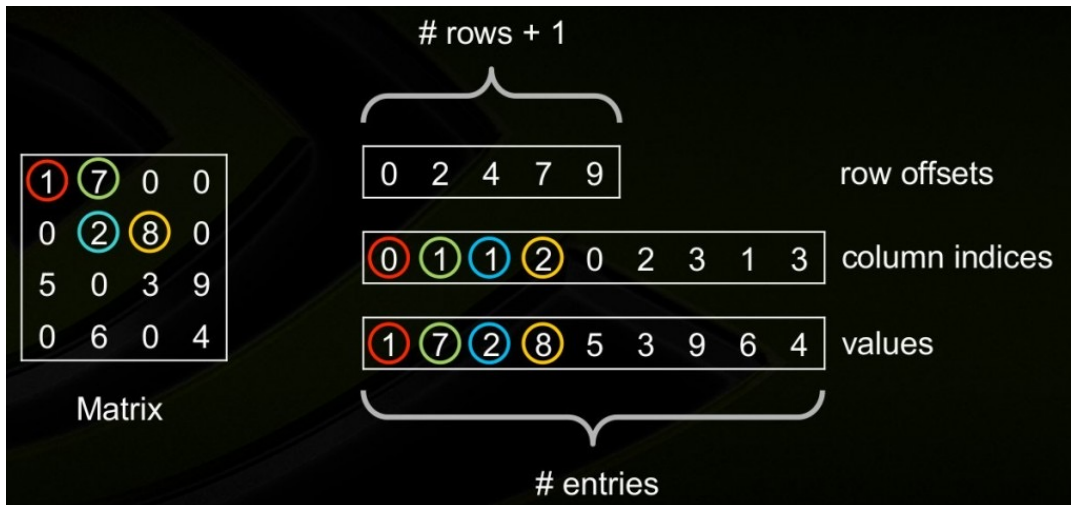
Storing Sparse Matrices



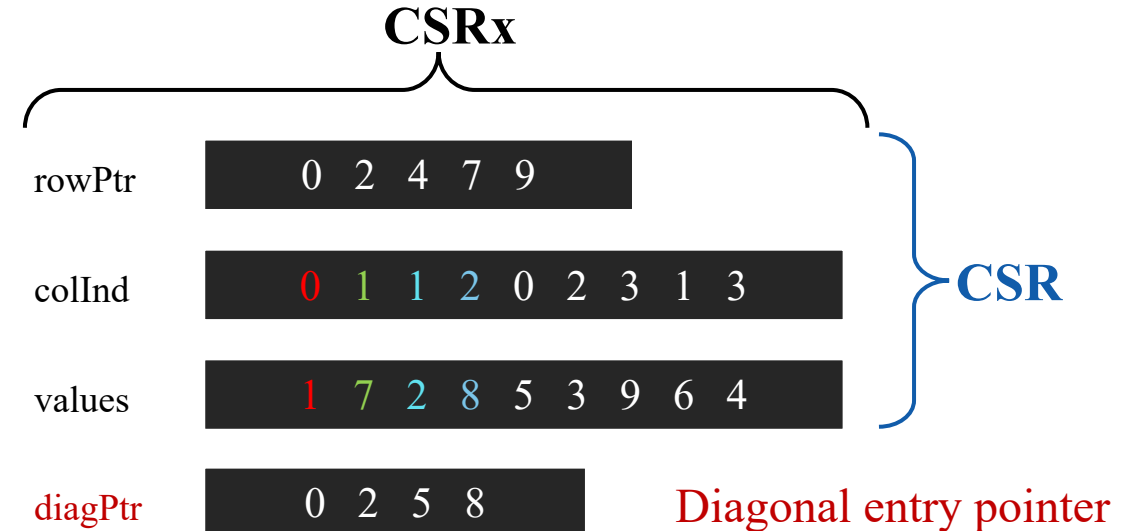
COO



CSR



- Compatible with CSR-based subroutines
 - Easy to access diagonal entries
 - Easy to find lower triangular entries
 - Easy to find upper triangular entries
 - Do not increase asymptotic storage requirement
- Use Implicit Ordering Info**



Ref: FASP developing team, <https://gitee.com/faspdevteam/faspxx.git>

Sparse Matrix Formats

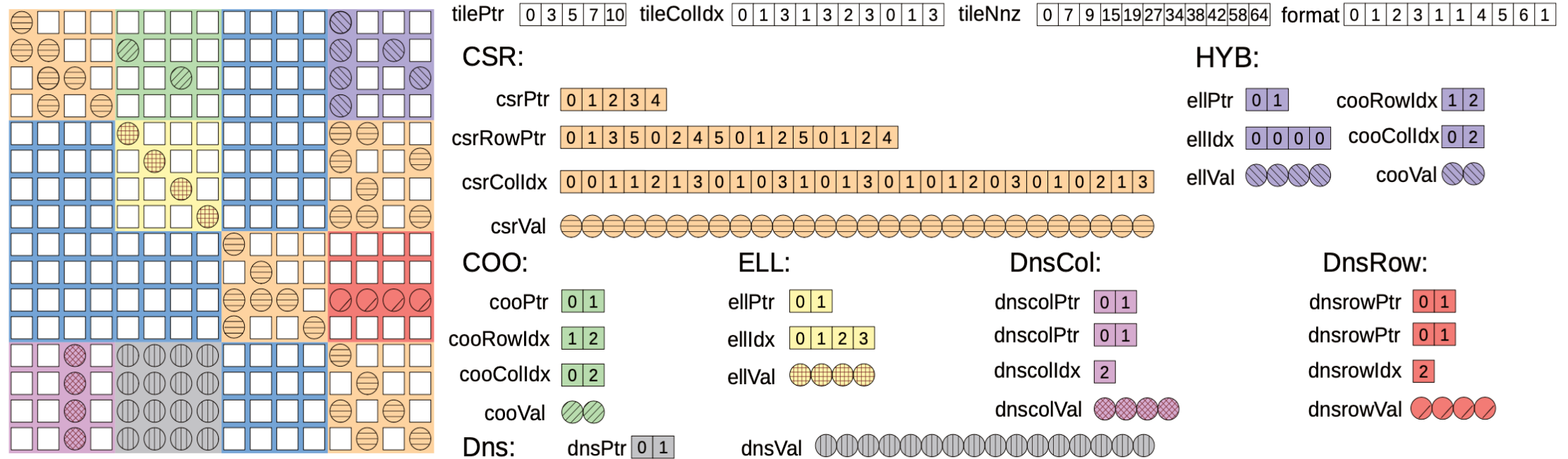


Fig. 3. An example matrix A of size 16-by-16 stored in 10 sparse tiles of size 4-by-4. The tile structure includes three arrays tilePtr, tileColIdx and tileNnz representing the memory offsets of tiles, tile column indices and the offsets for the number of nonzeros in sparse tiles. According to the format selection method, four of the 10 tiles keep the CSR format unchanged and use three arrays to store their information. The remaining six tiles are transformed to different formats, including COO, ELL, HYB, Dns, DnsRow and DnsCol. Each format has several corresponding arrays to store the nonzeros and their indices.

Source: Y. Niu, Z. Lu, M. Dong, Z. Jin, W. Liu and G. Tan, "TileSpMV: A Tiled Algorithm for Sparse Matrix-Vector Multiplication on GPUs," 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2021, pp. 68-78.

数值挑战3：计算、存储与移动

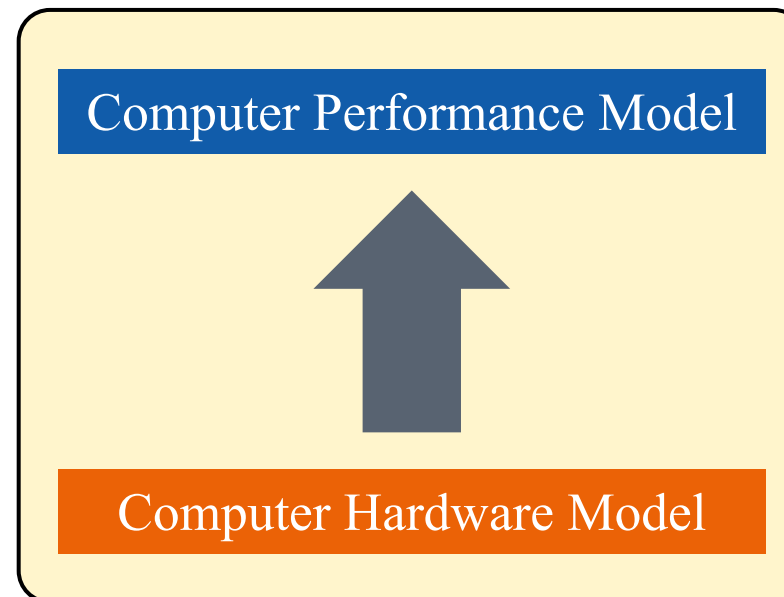
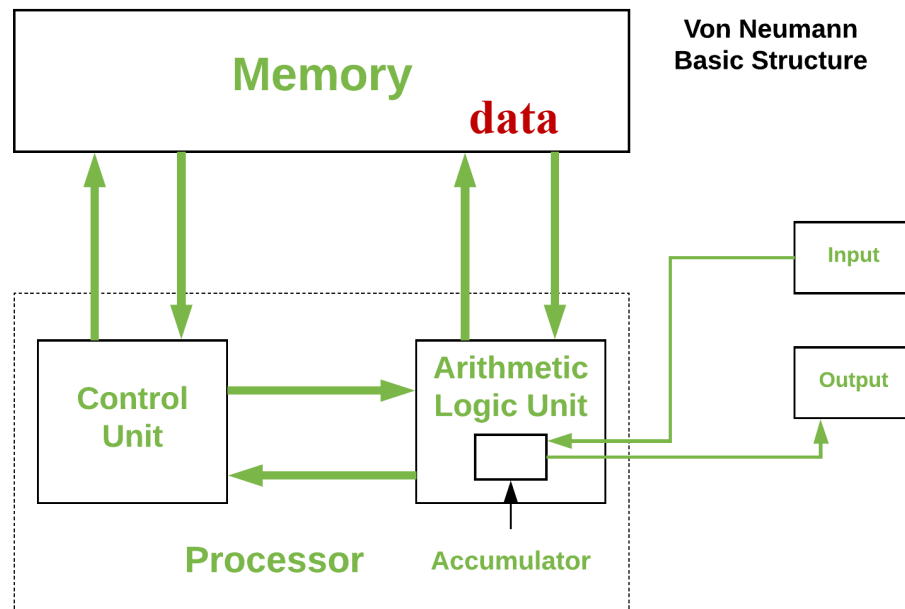
Numerical challenges

/1.3c

Performance Models



- **Q: How to measure performance? What is a fair measurement of performance?**
- We usually compare **wall-time** (actual computing time)
- Solve different **problems** with different **solvers** to different **accuracy** on different **hardware** ...



Consider the actual performance of a computer program

- ✓ hardware
- ✓ algorithm
- ✓ problem

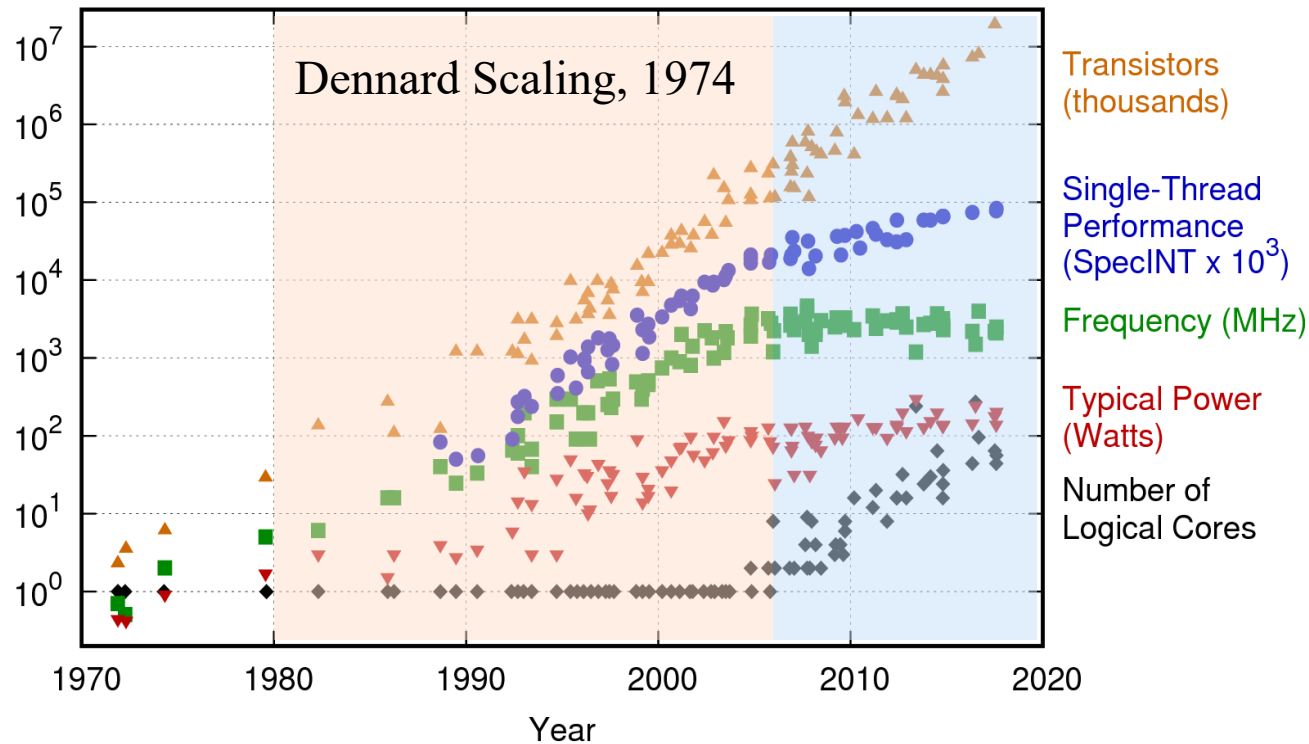
Source: <https://www.geeksforgeeks.org/computer-organization-von-neumann-architecture/>

Moore's Law

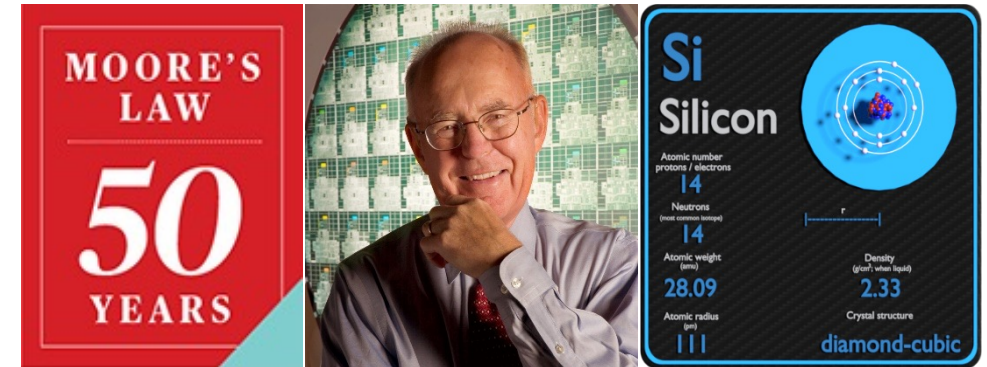


- “There’s plenty of room at the bottom”, R. Feynman, 1959
- Moore’s law, Gordon Moore, Intel® Co-founder, 1975

42 Years of Microprocessor Trend Data

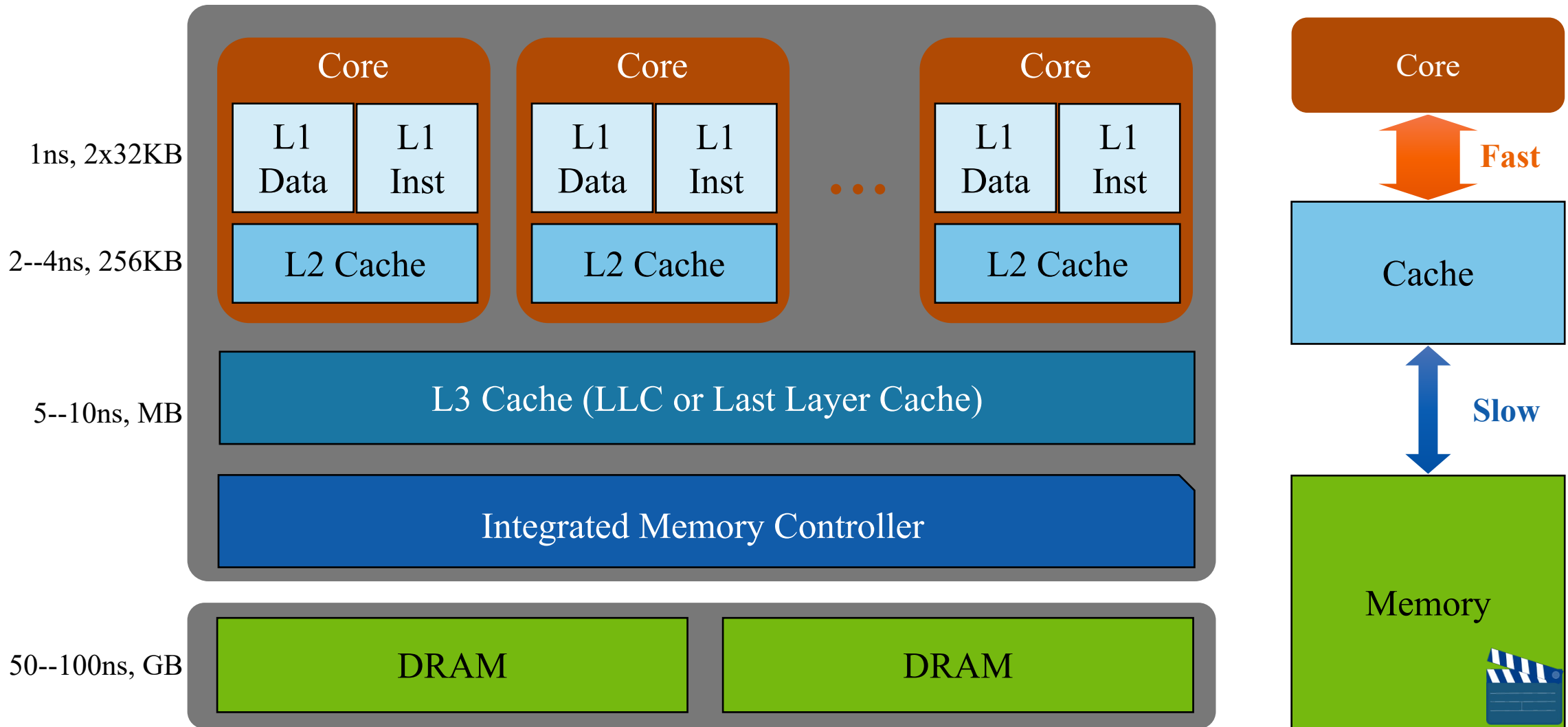


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp



- Due to the limitations of semiconductor technology, the traditional CPU roadmap is falling short of demands.
- Bottlenecks: power, latency, instruction-level parallelism (ILP), ...
- The sub-nano era? Roadmap: 20Å by 2024, 2Å by the 2040s.

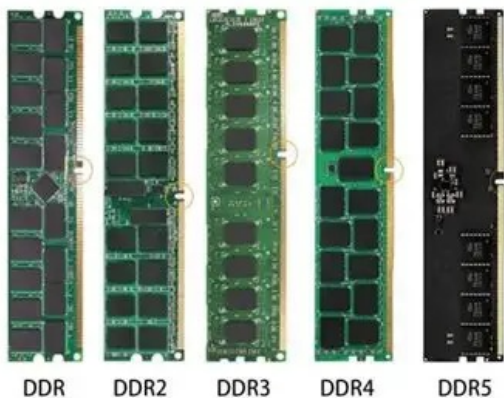
Memory Hierarchy



Improving Memory Performance

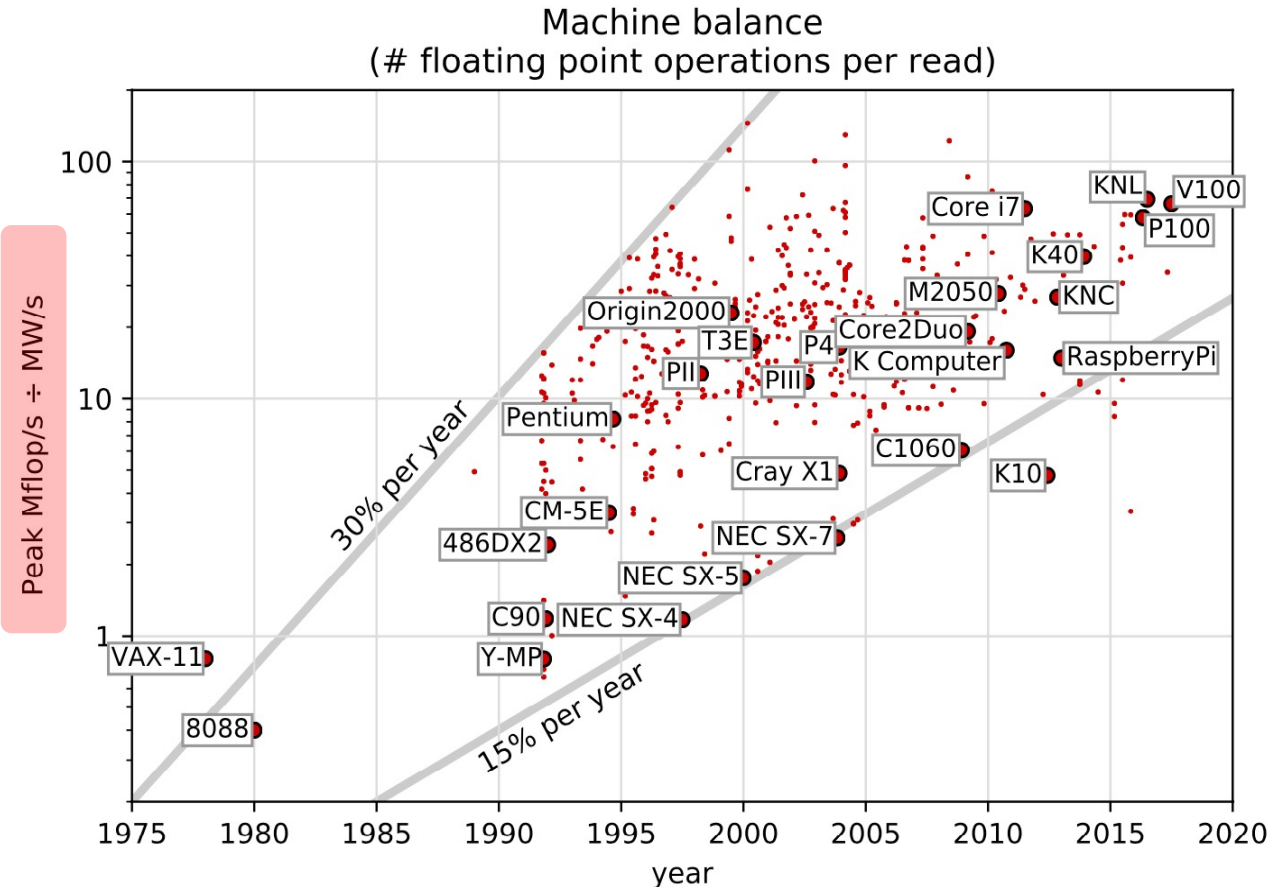


Metric	Prediction (2009)	El Capitan (Nov. 2024)	Notes
System Peak	1 exaflop/s	2.746 exaflops	Ranked #1 on TOP500 from Nov. 2024 to Nov. 2025
Node Memory Bandwidth	0.4-4 TB/s	5.3 TB/s	AMD Instinct MI300A APUs with HBM3 memory
Total Node Interconnect Bandwidth	100-400 GB/s	~1.2 TB/s	AMD's Infinity Fabric 3.0 between CPUs and GPUs
Memory Latency	50 ns	~100 ns	Has not reached predictions due to physical constraints
Interconnect Latency	0.5 μs	~0.4 μs	Using Slingshot-12 interconnect



	DDR	DDR2	DDR3	DDR4	DDR5
Market Introduction	1998-2000	2003	2007	2014	2020
Working Voltage	2.5V	1.8V	1.5V	1.2V	1.1V
Peak Bandwidth	1.6-3.2 GB/s	3.2-8.5 GB/s	6.4-17 GB/s	12.8-25.6 GB/s	25.6-67.2+ GB/s
CAS Latency (Nanoseconds)	~10-15ns	~10-15ns	~10-15ns	~10-15ns	~12-15ns
Power Consumption	High (baseline)	~20% lower than DDR1	~40% lower than DDR2	~25% lower than DDR3	~20% lower than DDR4
Prefetch Architecture	2n	4n	8n	8n	16n

Machine Balance Over Time



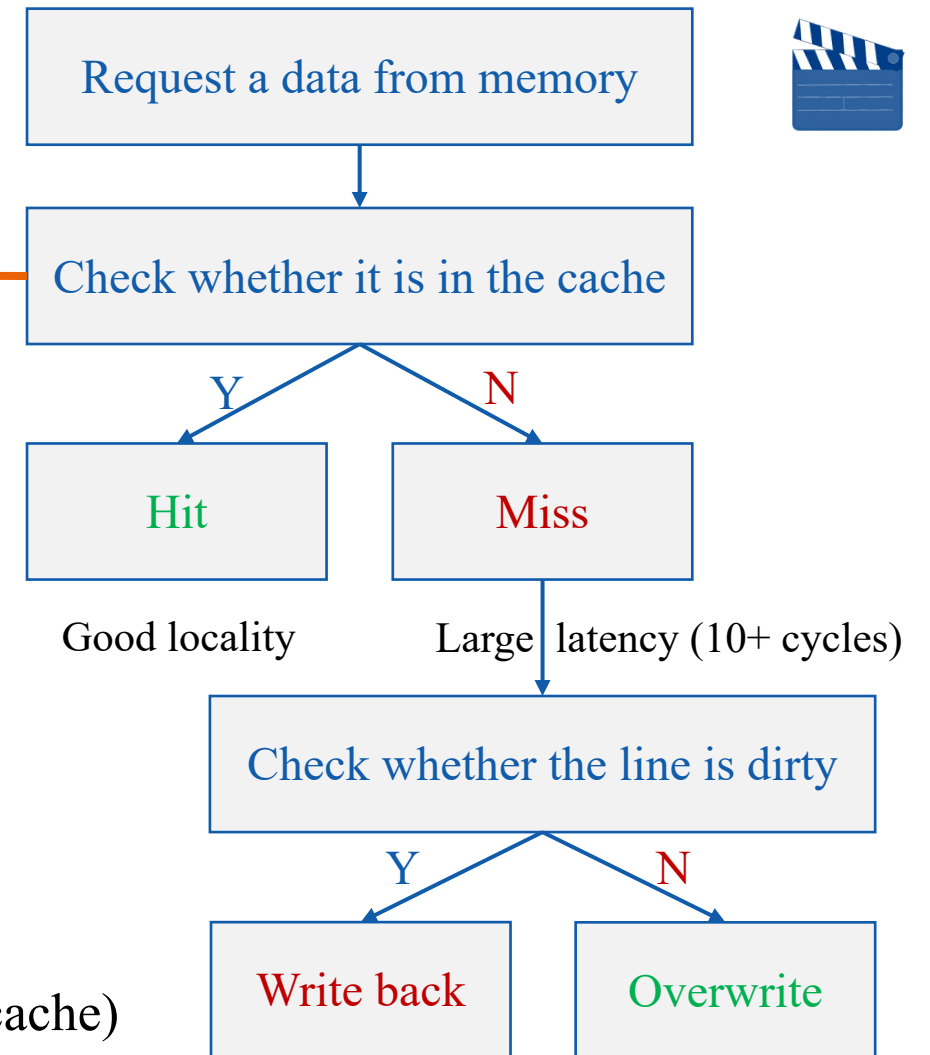
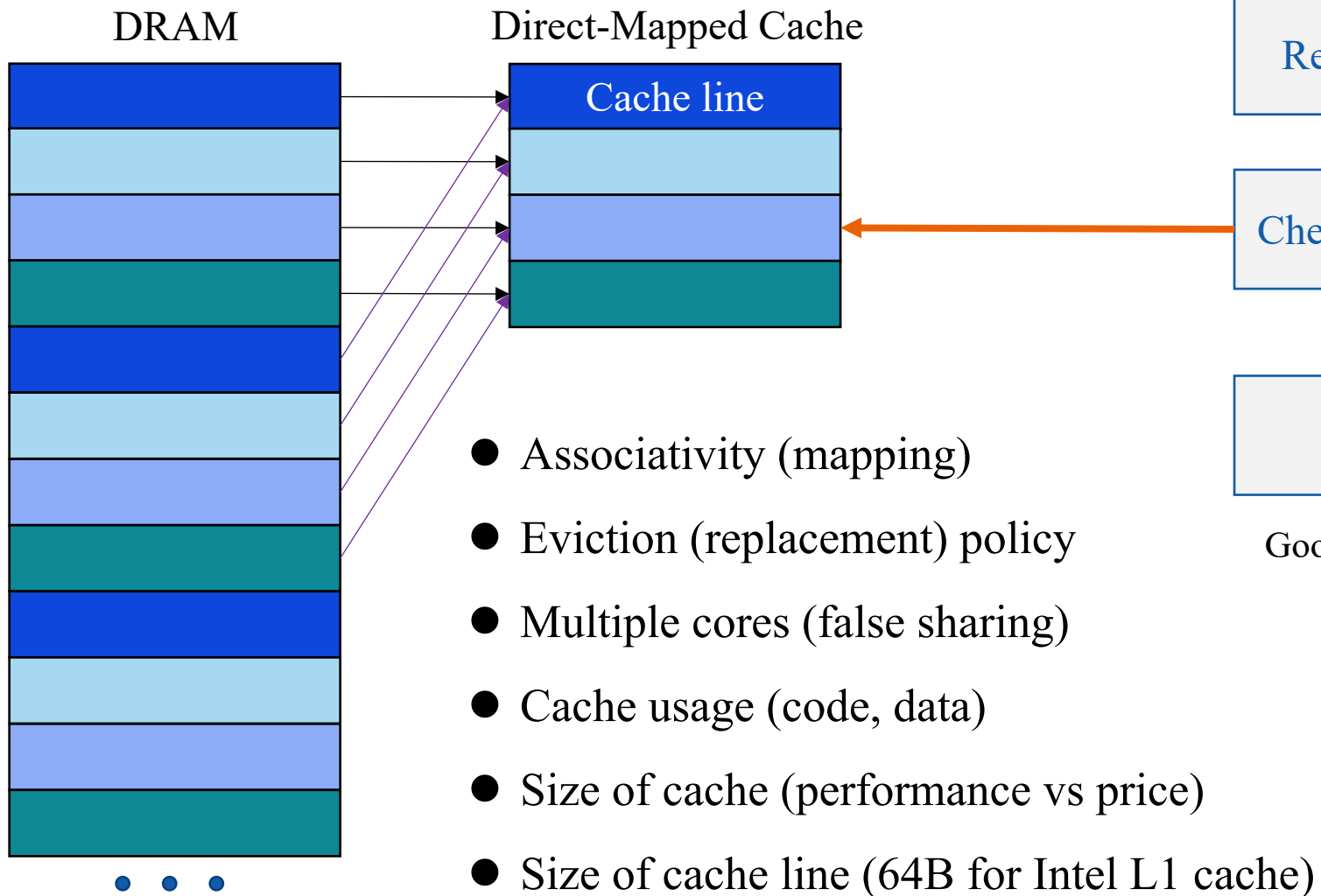
The gap between the computing power and the memory bandwidth **keeps increasing** in the last 45 years

- Computing power is (relatively) cheap
- Data movement is slow and will be slower relatively (**another reason to use low-precision now**)
- Key takeaway:

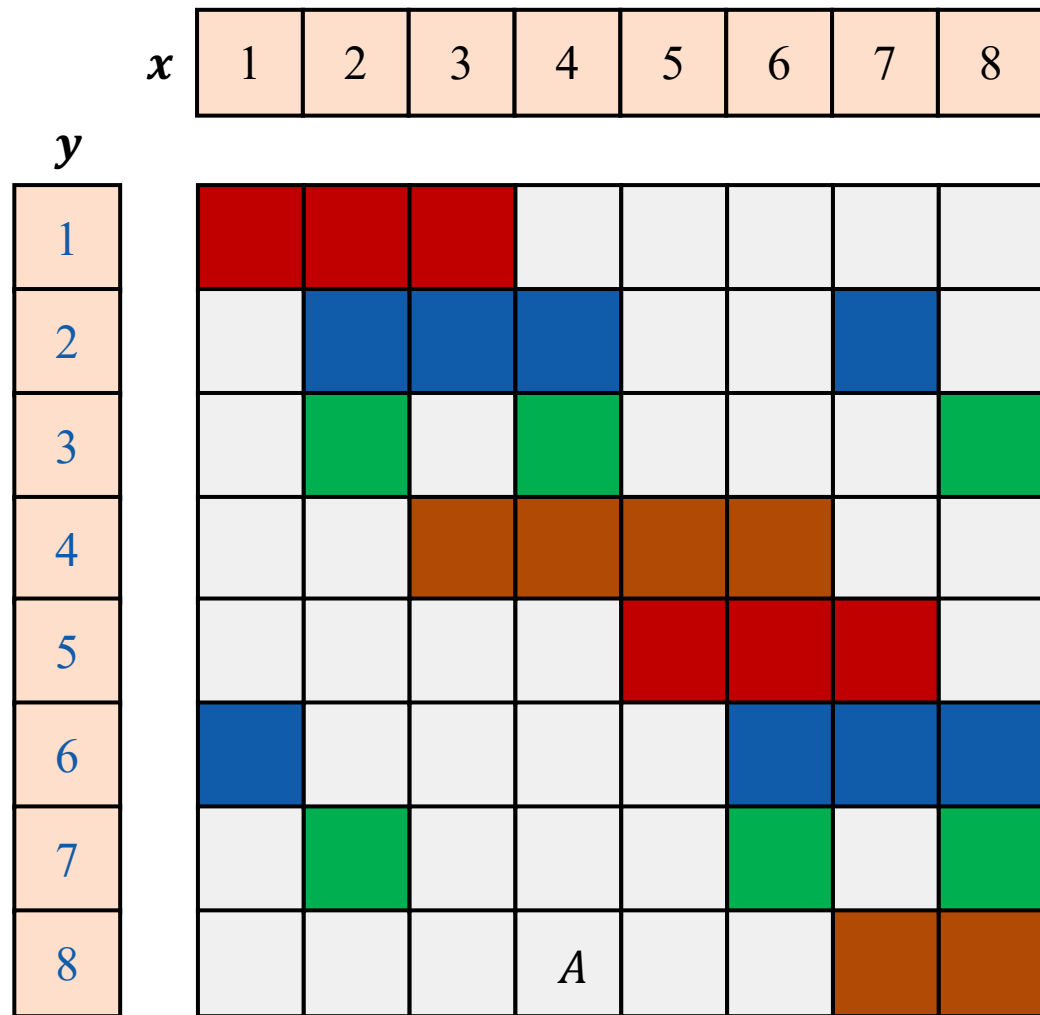
Make less data movement!

Source: A. Abdelfattah et al. “A survey of numerical linear algebra methods utilizing mixed-precision arithmetic”. The International Journal of High Performance Computing Applications. 2021;35(4):344-369.

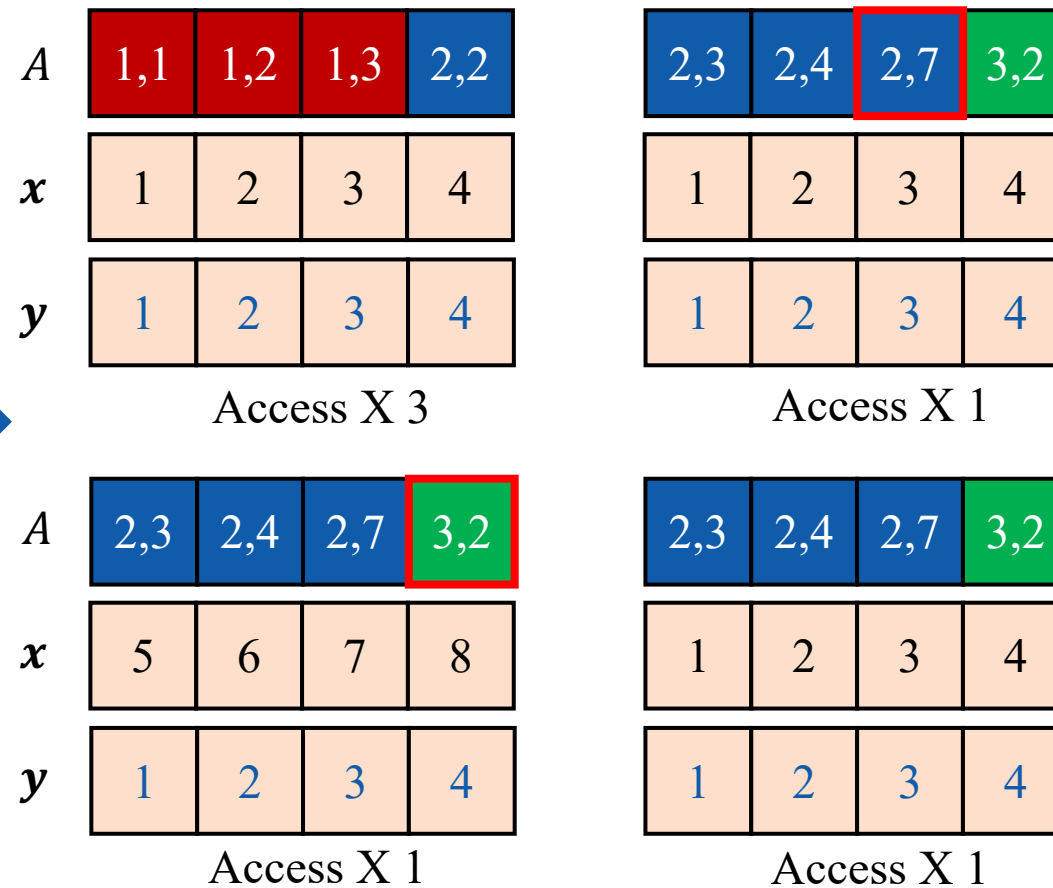
A Simplified Cache Mechanism



SpMV, A Simplified Example



$$y = Ax + y \quad \text{Sparse GEMV}$$



Measuring Performance

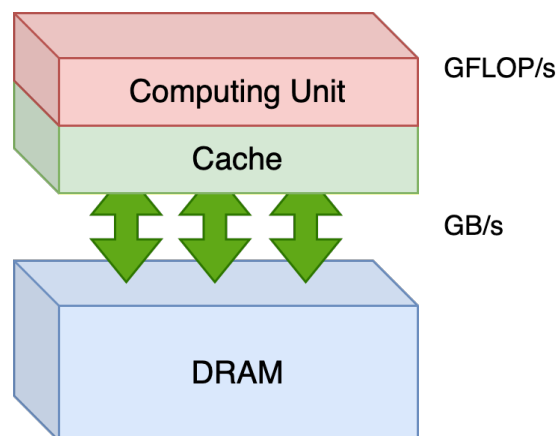


Q: How much time should a program cost?
Is my code fast ENOUGH?

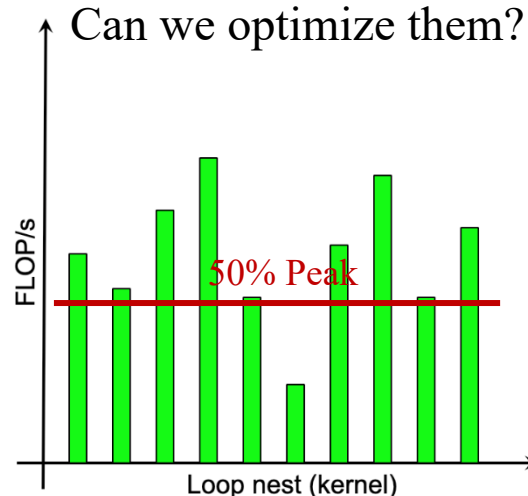
$$\text{WallTime} = \max\left(\frac{\#FP \text{ GFLOP}}{\text{Peak GFLOP/s}}, \frac{\text{DataMove GB}}{\text{Bandwidth GB/s}}\right)$$

$$\frac{\#FP}{\text{WallTime}} = \min\left(\text{Peak GFLOP/s}, \frac{\#FP \text{ FLOP}}{\text{DataMove Byte}} \times \text{Bandwidth GB/s}\right) \longrightarrow \text{Arithmetic Intensity (AI or CI)}$$

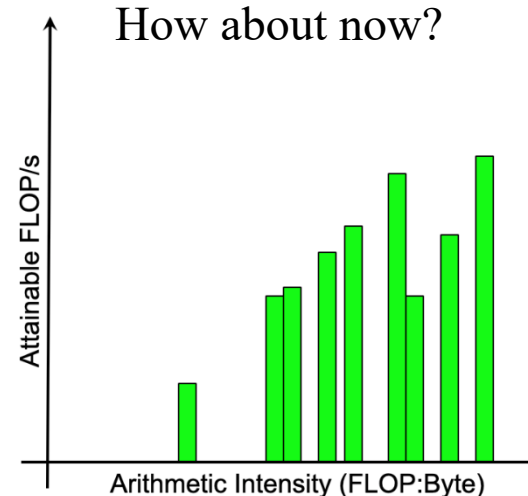
A Simple Architecture



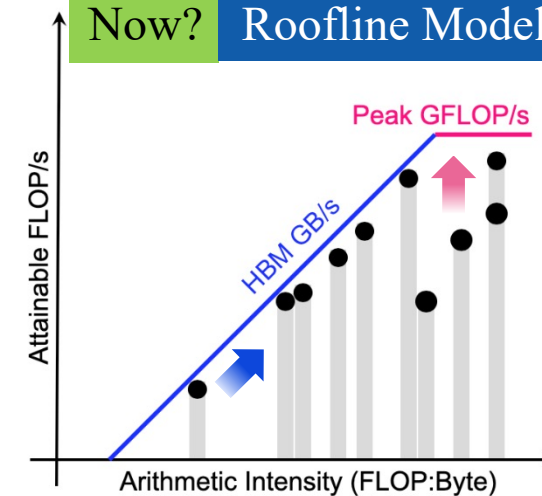
Can we optimize them?



How about now?

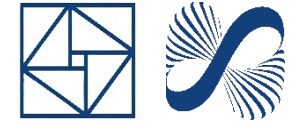


Now? Roofline Model



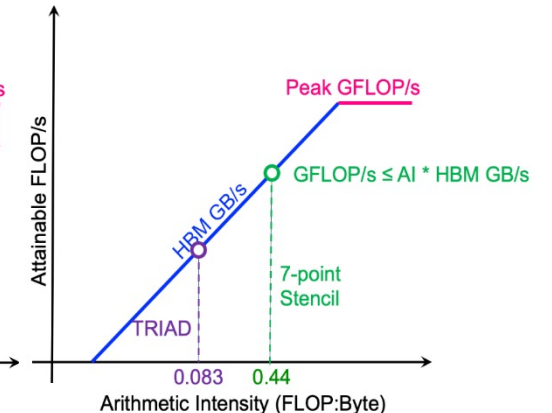
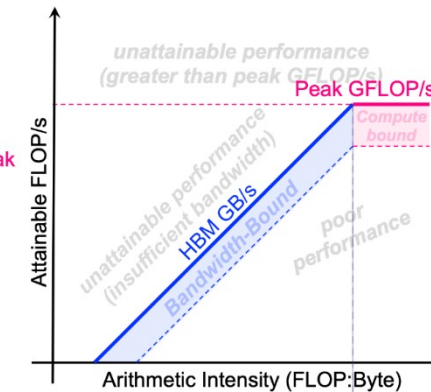
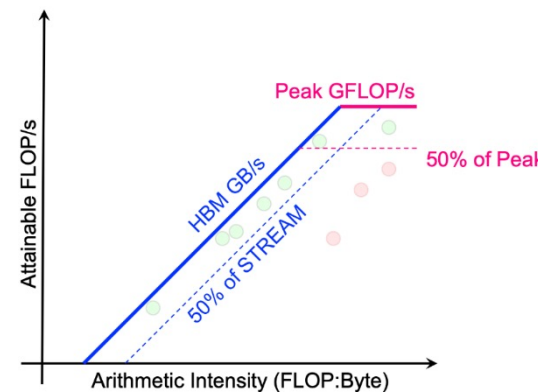
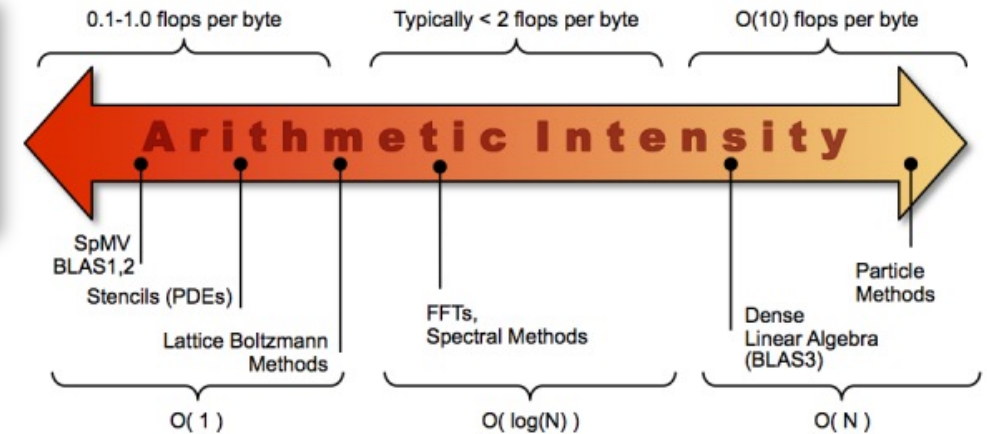
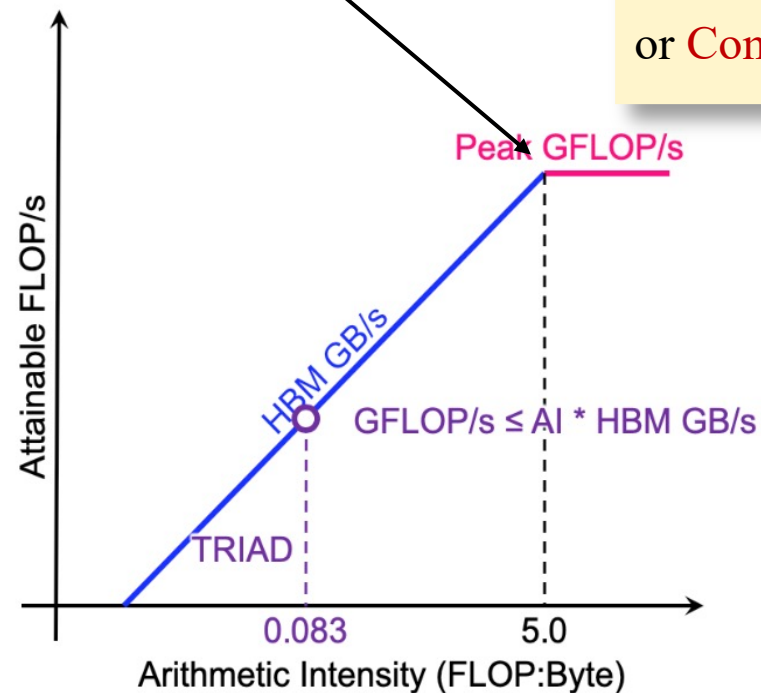
Source: Introduction to the Roofline Model, by Samuel Williams, Lawrence Berkeley National Lab (2020)

Compute-Bound vs Memory-Bound



Machine balance is typically 5~10 FLOP/Byte

Does it go anywhere near the roofline? Is it **CPU** or **Memory** or **Comm** bound?



Source: Samuel Williams, LBNL
<https://crd.lbl.gov/divisions/amcr/computer-science-amcr/par/research/roofline/introduction/>

$a[i] = b[i] + \text{scalar} * c[i]$: 2 FLOP, 2 Read, 1 Write, $\text{CI} = 2/(3*8) = 0.083$
 7-point stencil: 7 FLOP, 1 Read (cache-hit), 1 Write, $\text{CI} = 7/(2*8) = 0.438$

Communications Are Costly



Communication → Data Movement

- Computing ability improves faster than communication ability
- **Q: Why?**
- **Physics:** Longer the distance, higher the cost
- **Technology:** Increase speed with different methods
- **Economy:** Manufacturing cost is also important

	Petascale Systems (2009)	Predicted Exascale Systems	Factor Improvement
System Peak	$2 \cdot 10^{15}$ flops/s	10^{18} flops/s	~1000
Node Memory Bandwidth	25 GB/s	0.4-4 TB/s	~10-100
Total Node Interconnect Bandwidth	3.5 GB/s	100-400 GB/s	~100
Memory Latency	100 ns	50 ns	~1
Interconnect Latency	$1 \mu\text{s}$	$0.5 \mu\text{s}$	~1

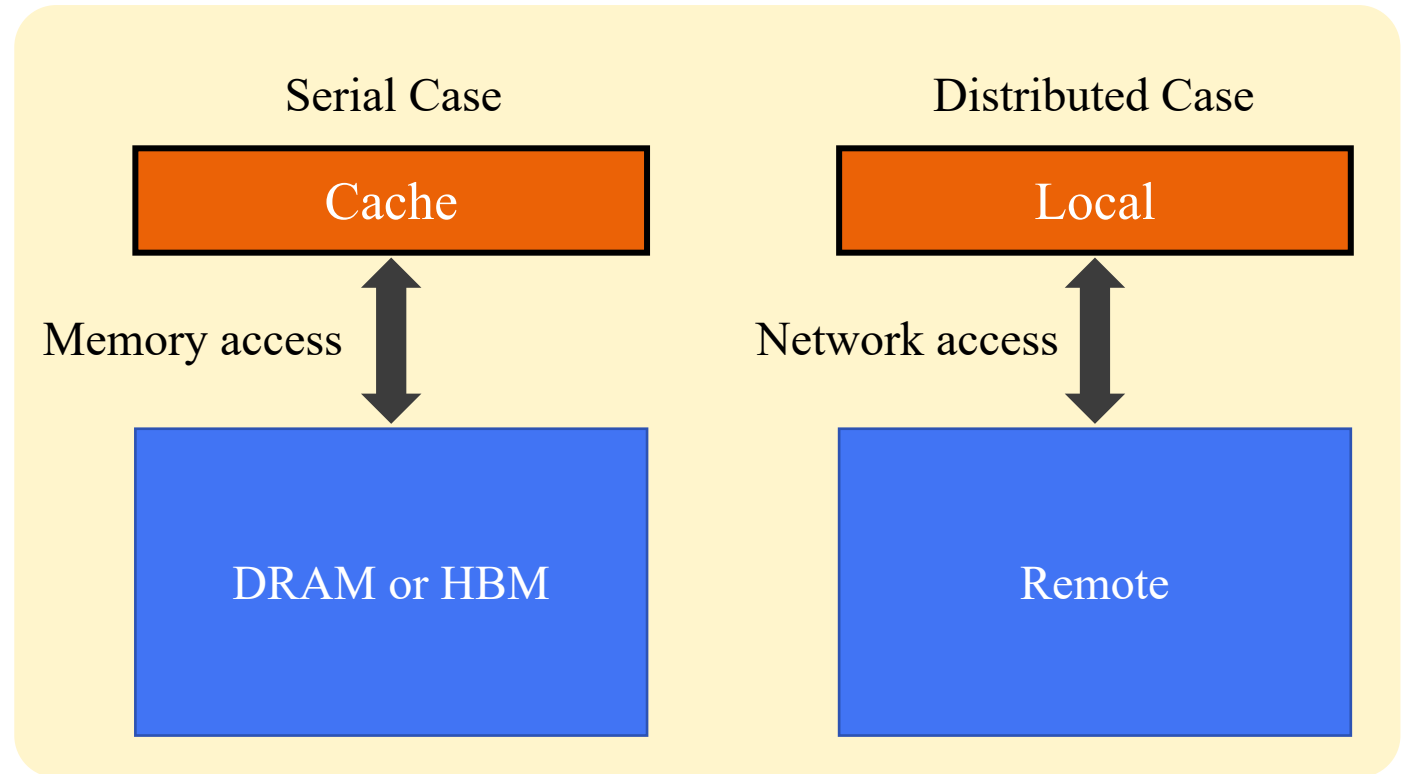
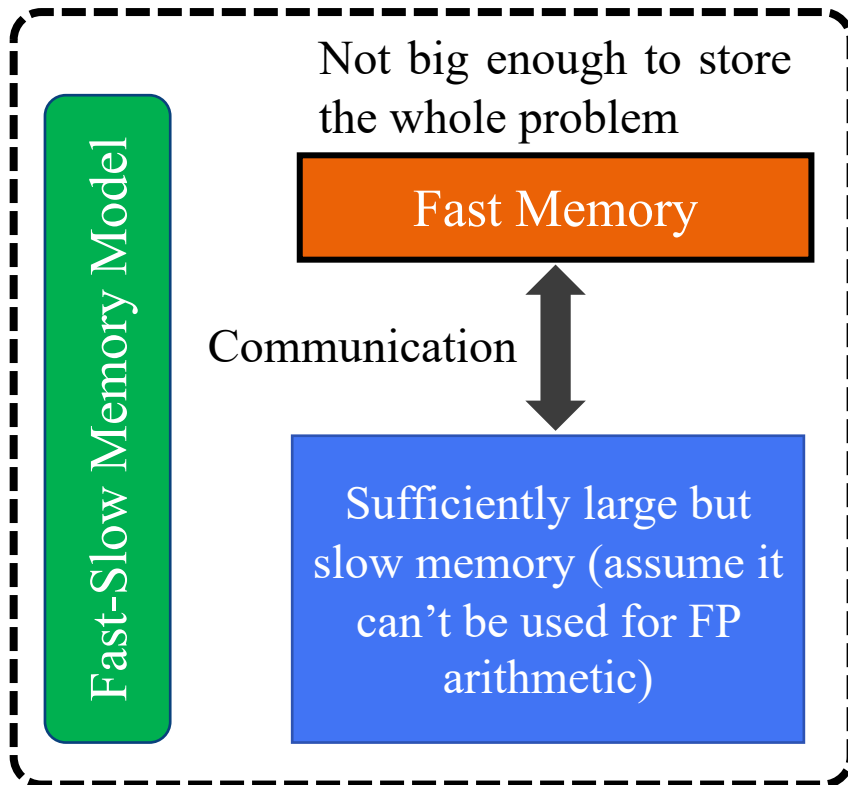
*Sources: from P. Beckman (ANL), J. Shalf (LBL), and D. Unat (LBL)

Annual improvements			
Time/flop		Bandwidth	Latency
59%	Network	26%	15%
	DRAM	23%	5%

Modelling Communication Costs



$$\text{Communication Time} = \text{Latency} + \text{Num of Bytes Moved} \div \text{Bandwidth}$$



Ref: CS267 lecture notes on J. Demmel's webpage: <https://people.eecs.berkeley.edu/~demmel/>

Example: Matrix Multiplication



Considering data movement

- Matrix multiplication is not just about arithmetic operations
- Data movement (between levels of memory) often dominates performance on modern hardware
- Maximize data reuse, minimize transfers between slow and fast memory, organize computations to match hardware characteristics

Algorithm 19: Naïve matrix multiplication

```
1 %% Given two matrices  $A, B \in \mathbb{R}^{n \times n}$ ;  
2 for  $i = 1 : n$   
3     for  $j = 1 : n$   
4         for  $k = 1 : n$   
5              $C(i, j) \leftarrow C(i, j) + A(i, k) * B(k, j)$ ;  
6         end  
7     end  
8 end
```

Algorithm 20: Naïve matrix multiplication with data movement

```
1 %% Given two matrices  $A, B \in \mathbb{R}^{n \times n}$ ;  
2 for  $i = 1 : n$   
3     %% Read  $A(i, :)$  into fast memory,  $n \times n$  times in total  
4     for  $j = 1 : n$   
5         %% Read  $B(:, j)$  into fast memory,  $n^2 \times n$  times in total  
6         %% Read  $C(i, j)$  into fast memory,  $n^2 \times 1$  times in total  
7         for  $k = 1 : n$   
8              $C(i, j) \leftarrow C(i, j) + A(i, k) * B(k, j)$ ;  
9         end  
10        %% Write  $C(i, j)$  back to slow memory,  $n^2 \times 1$  times in total  
11    end  
12 end
```

数值挑战4：并行计算效率

Numerical challenges

/1.3d

Richardson's Fantastic Forecast Factory



In 1922, Lewis Fry Richardson published his fantasy weather forecasting factory:

- A 20-story building with a spherical central chamber where human “computers” would calculate weather patterns
- A **director of operations** at the center who coordinated calculations like an orchestra conductor, using red (teams ahead)/blue (teams behind) lights for scheduling
- 64,000 **human computers** working on different regions of a global weather map simultaneously

Source: <https://www.emetsoc.org/resources/rff/>



FIGURE 1: “Weather Forecasting Factory” by Stephen Conlin, 1986. Based on the description in Weather Prediction by Numerical Process, by L.F. Richardson, Cambridge University Press, 1922, and on advice from Prof. John Byrne, Trinity College Dublin. Image: ink and water colour, c. 50 x 38.5 cm. © Stephen Conlin 1986. All Rights Reserved. (Courtesy: Hendrik Hoffmann, School of Mathematics & Statistics, University College Dublin)

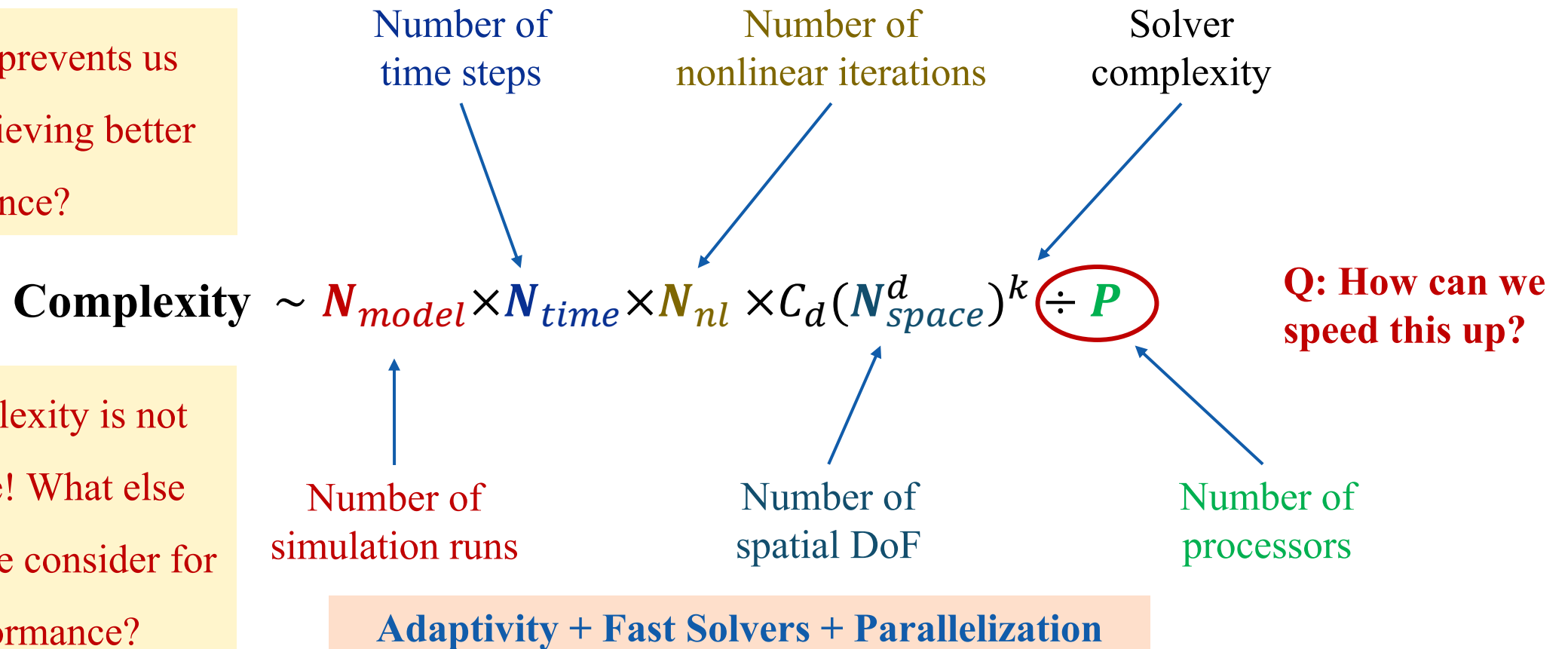
Computational Complexity



Time Marching → Linearization → Spatial Discretization → Linear Solver → Parallelization

Q: What prevents us from achieving better performance?

Q: Complexity is not wall-time! What else should we consider for real performance?



An Illustration of Parallel Computing



What's the problem?

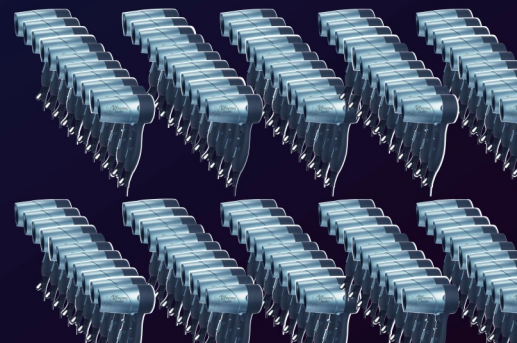
replacing 4 strong jet engines



Would you want to propel
a Super Jumbo



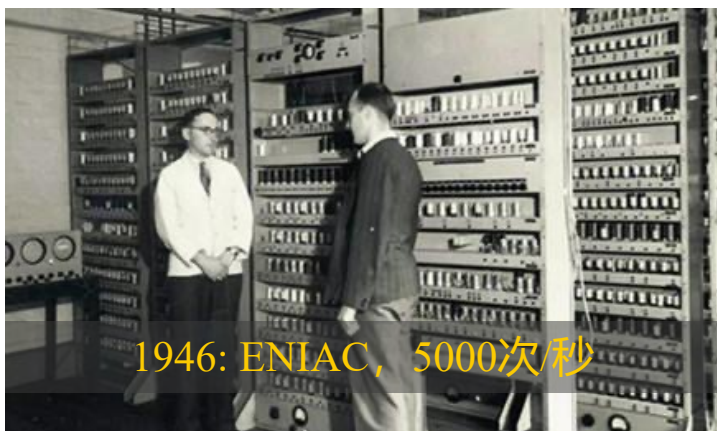
with 300,000
blow dryer fans?



An Airbus A380 engine generates approximately 90MW power output. This is roughly equivalent to the power of 300K ordinary household hair dryers. However, could you power an A380 with these hair dryers? Now, imagine you have 10 million hair dryers of varying power levels!

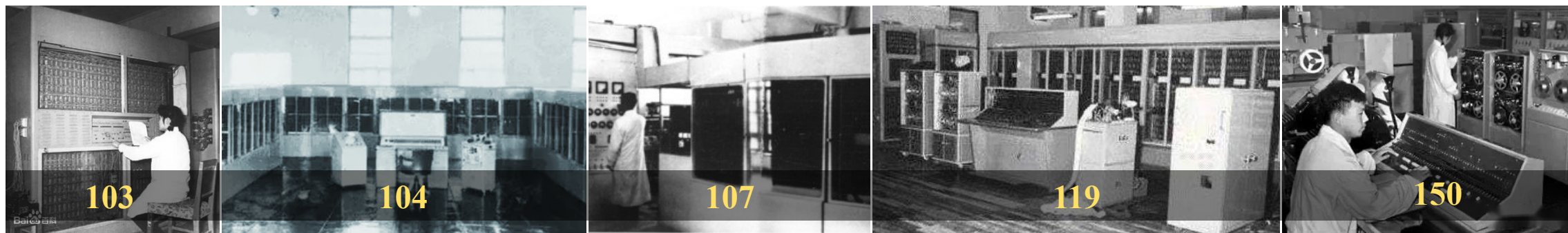
Ulrich Rüde, [Friedrich-Alexander-University of Erlangen-Nürnberg](#), Erlangen, Germany

75 Years, 14 Orders of Magnitude



Source: 中国工程物理研究院徐小文2021年报告

中国早期电子计算机发展史



- 1952年：中科院数学所成立，华罗庚提出计算数学的基础地位，为中国计算机事业奠定了思想基础
- 1958年：首台电子管计算机103机诞生（仿制苏联M-3计算机），运算速度每秒30次，后提升至1800次
- 1959年：首台大型电子管计算机104机研制成功（仿制苏联БЭСМ-II计算机），运算速度每秒1万次
- 1960年：夏培肃主持研制首台自行设计的107机，安装于中科大，成为高校首台计算机
- 1964年：首台自行设计的大型电子管计算机119机问世，运算速度每秒5万次，居世界电子管计算机之首
- 1973年：首台百万次集成电路计算机150机诞生，配备国内操作系统，实现石油勘探第一次数字化革命
- 1983年：成功研制亿次巨型机银河-1，中国成为美、日之后第三个掌握巨型机技术的国家
- 1989年：美限制对华出口高性能计算机，要求设备置于透明玻璃房中监控使用，倒逼中国自主创新

HPC Top500 List 2026.06



Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	LineShine - LingKun, LX2 304C 1.55GHz, LingQi, Kylin OS, Shenzhen Cloud Computing Center Co., Ltd. National Supercomputing Centre in Shenzhen (NSCS) China	13,789,440	2,198.40	2,735.82	42,220
2	El Capitan - HPE Cray EX255a, AMD 4th Gen EPYC 24C 1.8GHz, AMD Instinct MI300A, Slingshot-11, TOSS, HPE DOE/NNSA/LLNL United States	11,340,000	1,809.00	2,821.10	29,685
3	Frontier - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Cray OS, HPE DOE/SC/Oak Ridge National Laboratory United States	9,066,176	1,353.00	2,055.72	24,607

Remarks:

- A lot of cores!!
- Cost a lot of money!!!
- Can this trend continue?
- Do we need HPC?
- Can we use HPC well?
- How to use HPC well?
- HPL (dense/direct)
- HPCG (sparse/iterative)
- HPL-AI (low precision)
- Green 500 (energy)



FIND OUT MORE AT
top500.org

Source: <https://top500.org/lists/top500/2026/06/>

How to Measure Parallel Efficiency



● 强可扩展性

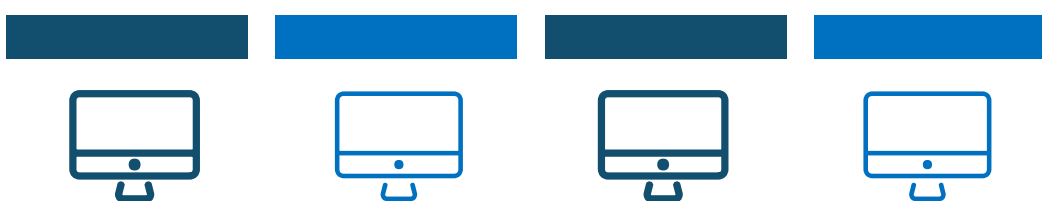


$$\frac{W_1}{pC_1} \quad S = \frac{T_1}{T_p} \approx p?$$

若需要解决一个任务，把计算资源增大到原来的 p 倍，能加速 p 倍吗？

定义**并行效率** $E = S/p$ ，我们希望能是100%！

● 弱可扩展性



$$\frac{W_p}{pC_1} \quad T_p \approx T_1?$$

若需要解决 p 倍大的任务，计算资源同比例增加，能保持相同耗时吗？

为了**效率**保持不变，需要多大的问题规模？

Three Mountains on Parallel Simulation



Amdahl's Law 1967



如果串行部分占总时间的10%，那并行加速比不可能超过10倍

对于大规模系统来说，很多应用程序的**强可扩展性**是难以实现的！

对于很多应用来说，更需要的是计算越来越大规模的问题！需要的是**弱可扩展性**，而不是强可扩展性

Gustafson-Barsis's Law 1988



当问题规模与计算资源同比例增大 S 倍时，加速比最高就可达 $0.9p + 0.1$

Gabriel Wittum: HPC Paradox



当购买了大10倍的硬件系统，希望能更快地求解大10倍的问题；但现实很残酷，必须有最优算法才可以！

最优算法对于充分发挥HPC效率及实现弱可扩展性至关重要！

硬件投资



并行效率



最优算法



软件投资

Implementation of Algorithms



Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices. Each version represents a successive refinement of the original Python code. “Running time” is the running time of the version. “GFLOPS” is the billions of 64-bit floating-point operations per second that the version executes. “Absolute speedup” is time relative to Python, and “relative speedup,” which we show with an additional digit of precision, is time relative to the preceding line. “Fraction of peak” is GFLOPS relative to the computer’s peak 835 GFLOPS. See Methods for more details.

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	—	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03
4	Parallel loops	69.80	1.969	366	7.8	0.24
5	Parallel divide and conquer	3.80	36.180	6,727	18.4	4.33
6	plus vectorization	1.10	124.914	23,224	3.5	14.96
7	plus AVX intrinsics	0.41	337.812	62,806	2.7	40.45

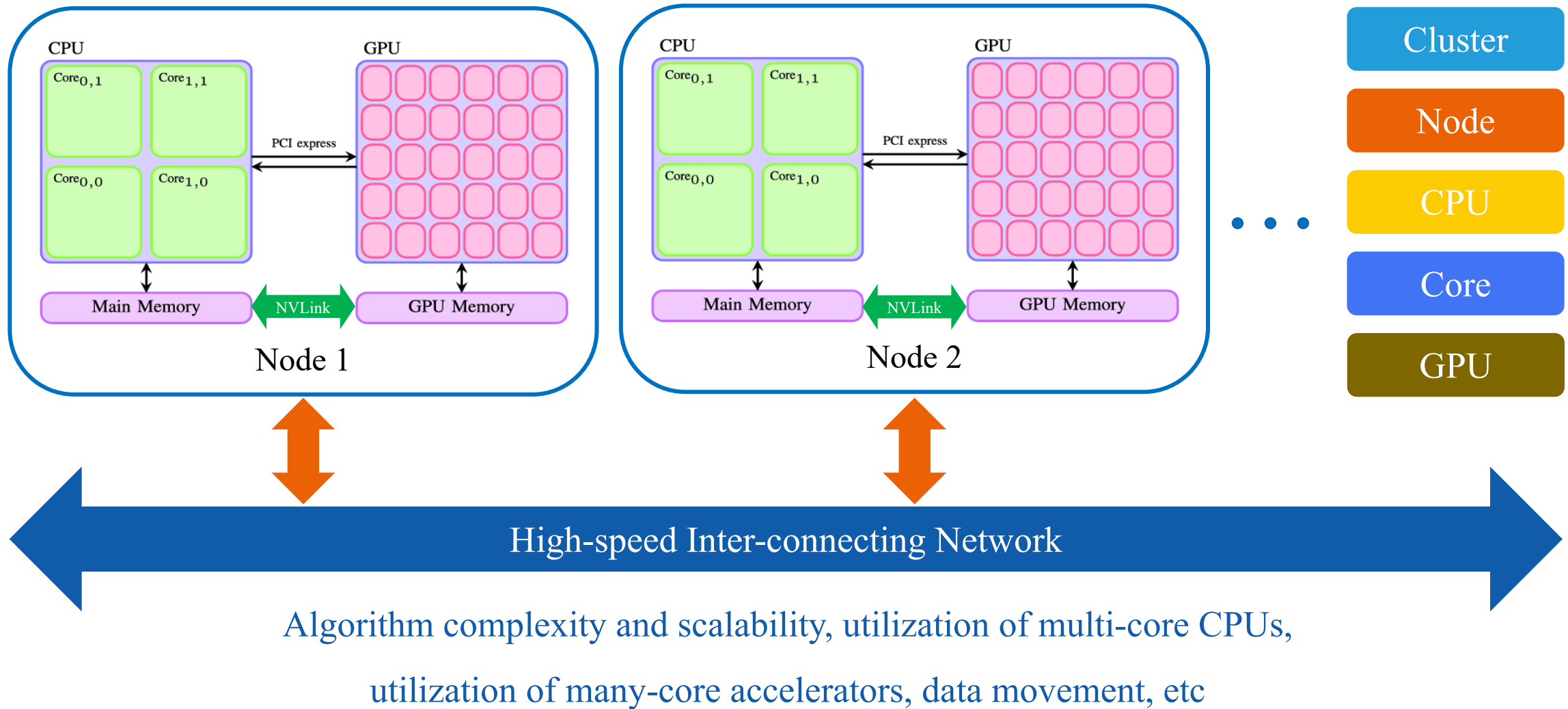
- Compiler optimization; loop ordering; parallel loops
- Vectorization; AVX intrinsic
- Tiling; cache-oblivious; divide-and-conquer
- Source: Leiserson & Shun, MIT Open Course 6.172

There’s plenty of room at the Top: What will drive computer performance after Moore’s law?

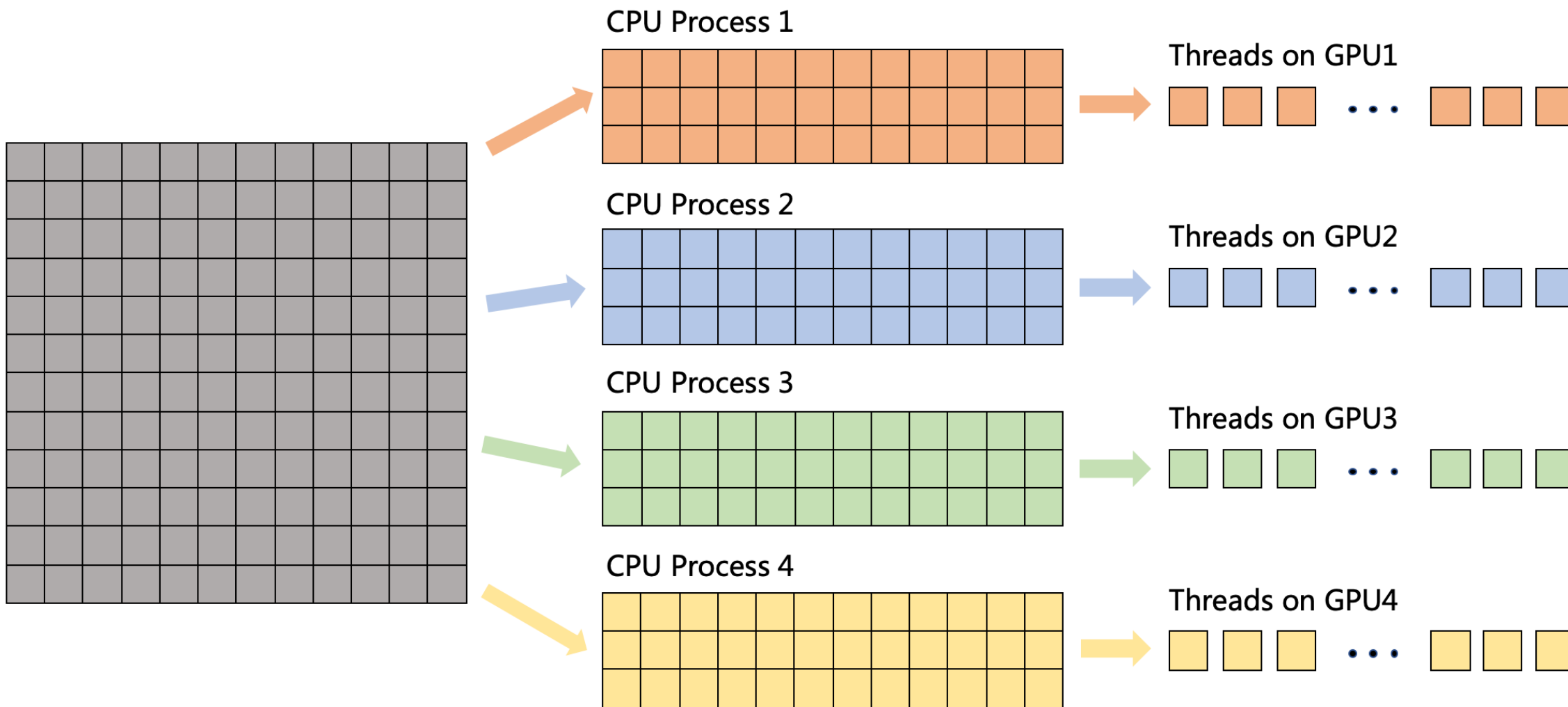
Science **368** (6495), June, 2020
<http://science.sciencemag.org/content/368/6495/eaam9744>

MKL

Heterogeneous Cluster Architecture



Parallel Matrix Data Layout



软件实现

Engineering numerical software

/1.4

Why Results Differ from Expectations?



A mismatch may originate in the problem, the numerics, or the implementation.

01

Problem & Model Complexity

- Modeling assumptions, parameters, and boundary/initial conditions
- Nonlinearity, multiphysics coupling, or multiple scales
- Chaotic dynamics and sensitive dependence on initial conditions

02

Numerical Error & Stability

- Discretization and truncation errors
- Iterative and round-off errors
- Mesh, time-step, tolerance, and stability effects

03

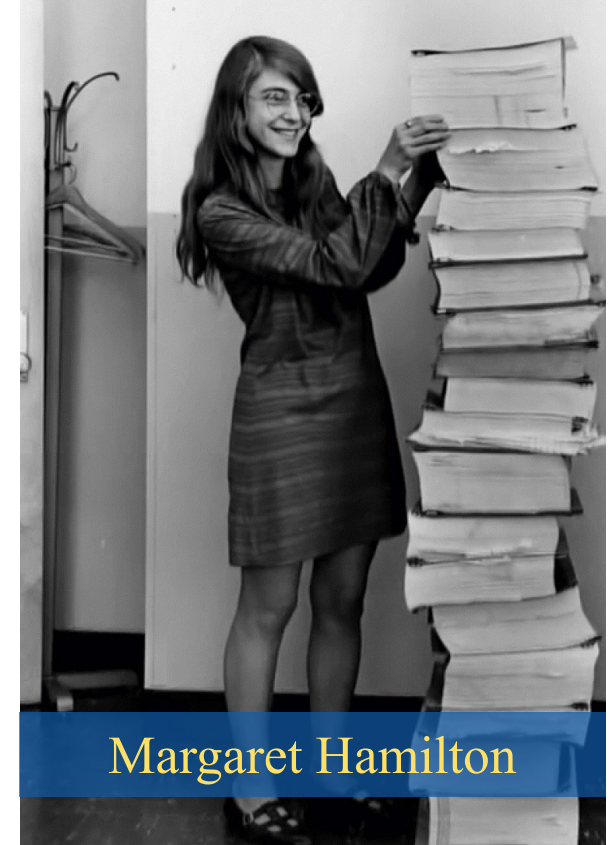
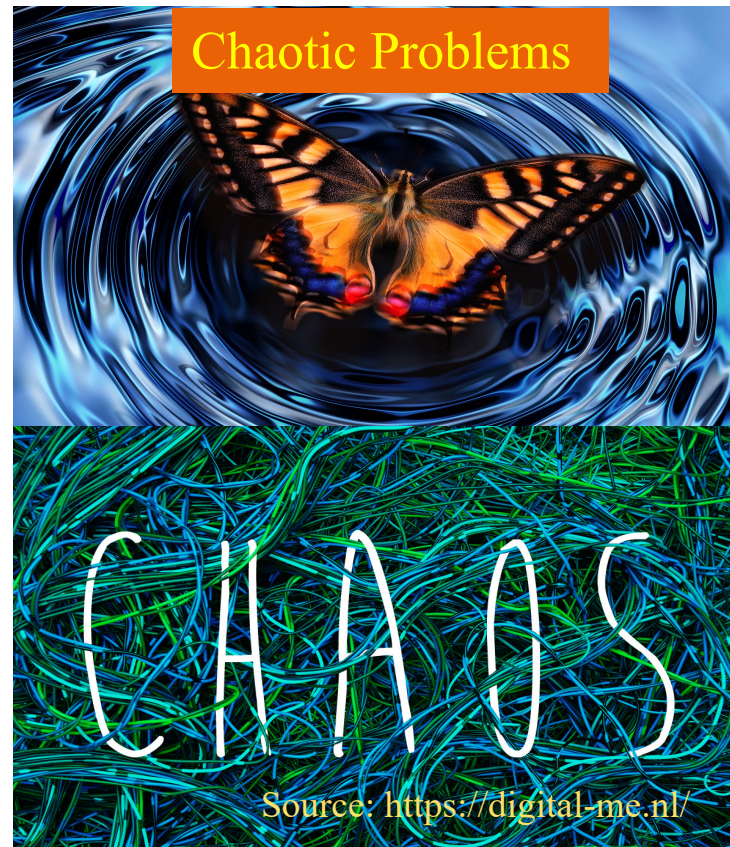
Implementation Issues

- Incorrect indexing or data handling
- Faulty control flow or state updates
- Parallel communication, synchronization, or nondeterminism

Chaotic Dynamics



Edward N. Lorenz



Margaret Hamilton

In 1961, E. N. Lorenz was using a Royal McBee LGP-30 computer with 6-digit precision to predict weather. The printout rounded variables off to a 3-digit number, so a value (0.506127) was printed as 0.506. He tried to input a rounded number, but ...

Pay Attention to Numerical Error



What Every Computer Scientist Should Know About Floating-Point Arithmetic

DAVID GOLDBERG

Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304

Floating-point arithmetic is considered an esoteric subject by many people. This is rather surprising, because floating-point is ubiquitous in computer systems: Almost every language has a floating-point datatype; computers from PCs to supercomputers have floating-point accelerators; most compilers will be called upon to compile floating-point algorithms from time to time; and virtually every operating system must respond to floating-point exceptions such as overflow. This paper presents a tutorial on the aspects of floating-point that have a direct impact on designers of computer systems. It begins with background on floating-point representation and rounding error, continues with a discussion of the IEEE floating-point standard, and concludes with examples of how computer system builders can better support floating point.

Categories and Subject Descriptors: (Primary) C.0 [Computer Systems Organization]: General—*instruction set design*; D.3.4 [Programming Languages]: Processors—*compilers, optimization*; G.1.0 [Numerical Analysis]: General—*computer arithmetic, error analysis, numerical algorithms* (Secondary) D.2.1 [Software Engineering]: Requirements/Specifications—*languages*; D.3.1 [Programming Languages]: Formal Definitions and Theory—*semantics* D.4.1 [Operating Systems]: Process Management—*synchronization*

General Terms: Algorithms, Design, Languages

Additional Key Words and Phrases: denormalized number, exception, floating-point, floating-point standard, gradual underflow, guard digit, NaN, overflow, relative error, rounding error, rounding mode, ulp, underflow

Oracle® Solaris Studio 12.4: Numerical Computation Guide

ORACLE®

Part No: E39019
January 2015

ACM Computing Surveys, 1991

Sun/Oracle Numerical Computation Guide, 2015

Hardware Failures



Mean Time To Failure

MTTF: measures the average time a non-repairable asset operates before it fails

2 days

Ancient but stable

The mean time to failure of ENIAC (1946) [[Alexander Randall 2006](#)]

26.1

CPU failures per week

ASCI Q at LANL has 26.1 CPU failures per week [[Michalak et al. 2005](#)]

4-6 hours

Soft error in L1 cache

BlueGene/L has one soft error in L1 cache every 4 to 6 hours [[Bronevetsky, Supinski 2008](#)]

4.2 hours

Requires repair actions

Lessons learned from the analysis of system failures at petascale; see [DSN 2014](#)

30 mins

Run without problems

Researchers have predicted that large jobs may fail once every 30 minutes on **exascale** platforms

The Patriot Missile Failure



- On the news:

During the Gulf War, a U.S. Patriot Missile battery at Dhahran, Saudi Arabia failed to intercept an incoming Iraqi Scud missile on Feb. 25 1991, resulting in 28 deaths and 100+ injuries.

- Failure Reasons: **0.343 seconds time-discrepancy**

- The system represented **0.1** by a truncated 24-bit fixed-point binary value, which has no exact binary representation.
- After about 100 hours, the clock count was approximately 3,600,000 tenths of a second.
- Multiplying this large count by the truncated value introduced an error of about 0.343 sec, large enough to shift the tracking gate.



The Loss of the Sleipner A Platform

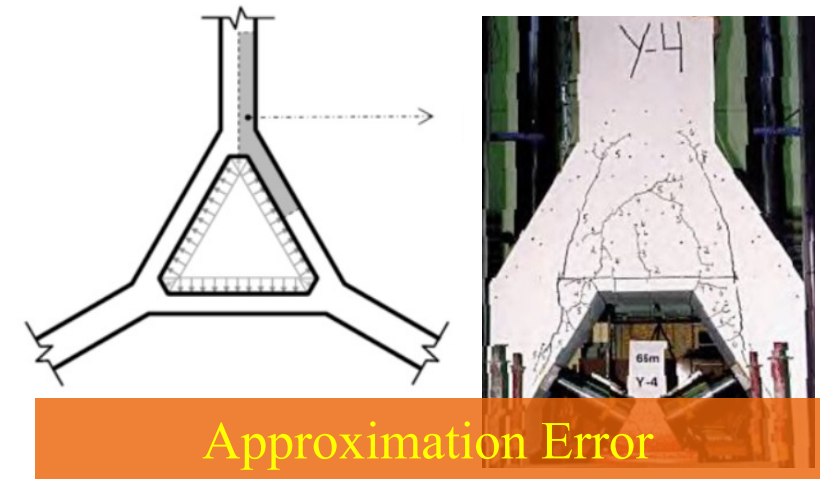


- On the news:

“On August 23, 1991, the Sleipner A platform suddenly sank near Stavanger, resulting in approximately \$700M in losses.”

- Failure Reasons: **47% underestimation of shear forces**

- Engineers used NASA's NASTRAN software, but the model contained distorted elements with skewed faces.
- When analyzing stress results, engineers fitted a second-order polynomial curve to their data and used extrapolation to determine edge stresses (over-fitting).
- The analysis relied on the 1977 Norwegian concrete code, which predicted failure at nearly twice the actual pressure.



Approximation Error

The Legendary “Nine Calculations”



- Chinese scientists attempted to replicate calculations from a Soviet teaching model. But the calculation of pressure values during explosive implosion consistently fell short of the Soviet figures.
- Over **nine intense rounds** of recalculations and cross-disciplinary debates involving physicists, mechanics experts, and mathematicians, the discrepancy persisted, stalling the entire atomic bomb project.

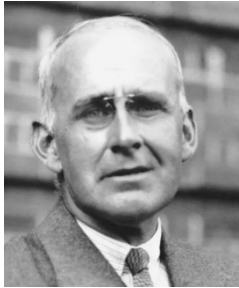
九次计算：原子弹原理突破

- 一天三班倒不间断计算
- 九次计算肯定流体假定
- 最大功原理放倒拦路虎

Theoretical Breakthrough: Zhou Guangzhao constructed an ideal model and proved that the maximum achievable pressure **could never reach** the Soviet expert's claimed value.



Reliability of Numerical Simulator



Never believe an experimental result until it has been confirmed by theory

Sir Arthur Stanley Eddington

01



Model Reliability

Mathematical model and its data describe the real physics with sufficient accuracy

02



Algorithm Reliability

Underlying numerical algorithms should be stable, consistent, and convergent

03



Software Reliability

The ability to perform its required functions under static conditions for a specific period

Legacy Implementation



In 1979, the authors of the LINPACK software package posed by J. Dongarra's car, which had a LINPACK license plate, in the suburbs of Chicago near Argonne National Laboratory.

The LINPACK project began in 1974, initiated by Jim Pool, to address the need for a portable, high-quality library for solving linear systems.

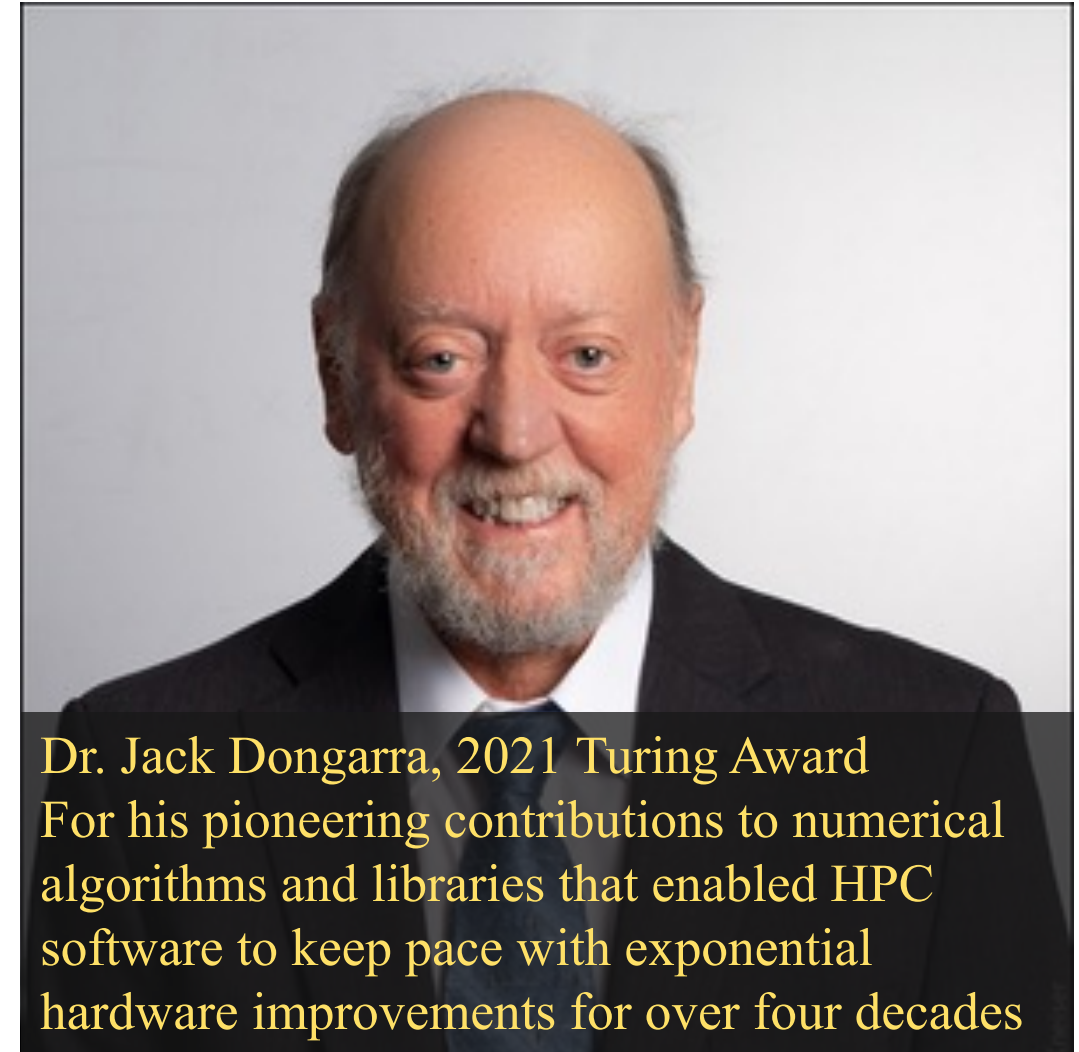
From left to right, the people in this photo are the core developers of the **LINPACK** linear algebra library:

- Jack Dongarra
- Jim Bunch
- Cleve Moler, who created MATLAB
- Gilbert Stewart

Modern Implementation



- **Software Stack:** **BLAS** provides computational kernels; **LAPACK** builds dense factorizations and solvers on BLAS; **ScaLAPACK** extends these methods to distributed memory architectures.
- NumPy and SciPy provide high-level interfaces for Python users.
- **Main Considerations:**
 - reliability
 - user-friendliness
 - efficiency



Dr. Jack Dongarra, 2021 Turing Award
For his pioneering contributions to numerical algorithms and libraries that enabled HPC software to keep pace with exponential hardware improvements for over four decades

Why Bother Using Numerical Libraries



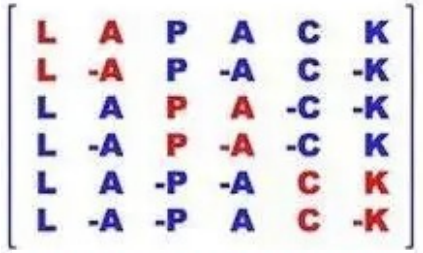
Q: The subroutines look simple; why not write them by ourselves?

Performance and Reliability

- There's a huge gap between "academically correct" and "industrial-grade robust"
- Optimized: Built on efficient **BLAS** (Basic Linear Algebra Subprograms) implementations
- Numerically stable: Careful error analysis and robust algorithms
- Portable: Works across different computer architectures
- Modern libraries provide built-in multi-threading, MPI, and GPU acceleration

Algorithmic Sophistication

- Blocked algorithms: Efficient cache usage for modern processors
- Divide-and-conquer methods: For eigenvalue problems
- Iterative refinement: For improving solution accuracy



LAPACK

LAPACK provides dense matrix factorizations, linear-system solvers, least-squares solvers, and eigenvalue/SVD routines. It relies on BLAS kernels, including GEMM, for matrix multiplication and other basic operations.

<https://netlib.org/lapack/>; <https://github.com/Reference-LAPACK/lapack/>

1. Solving Systems of Linear Equations

- Dense matrices: General, symmetric, positive definite, triangular systems
- Band matrices: Tridiagonal, general banded systems
- Solvers: LU with partial pivoting; Cholesky for SPD systems; pivoted LDL^T for symmetric systems

2. Linear Least Squares Problems

- Overdetermined and underdetermined systems
- Using QR and LQ factorizations
- With or without constraints

TBC



3. Eigenvalue Problems

- Standard eigenvalue problems: $Ax = \lambda x$
- Generalized eigenvalue problems: $Ax = \lambda Bx$
- For different types of matrices: symmetric, nonsymmetric, Hermitian matrices
- All eigenvalues or selected subsets

4. Singular Value Decomposition (SVD)

- Full and reduced SVD
- Computing singular values/vectors
- Applications to pseudoinverses, rank determination

5. Matrix Factorizations/Decompositions

- LU (with partial pivoting)
- Cholesky (for symmetric positive definite)
- QR (and QL, RQ, LQ variants)
- SVD
- Schur decomposition
- Spectral decomposition for symmetric matrices

LAPACK XYYZZZ Names



X - Data type:

- S - REAL (Single precision)
- D - DOUBLE PRECISION (Double precision)
- C - COMPLEX (Single precision complex)
- Z - COMPLEX*16 (Double precision complex)

YY - Matrix type:

- GE - General matrix
- GB - General band matrix
- SY - Symmetric matrix
- SP - Symmetric packed storage
- HE - Hermitian matrix
- TR - Triangular matrix
- PO - Symmetric positive definite

ZZZ - Operation:

- SV - Solve system of equations
- SVX - Solve with error bounds
- TRF - Triangular factorization
- TRI - Triangular inverse
- EV - Eigenvalues/vectors
- EVD - Eigenvalue decomposition
- SVD - Singular value decomposition
- QR - QR factorization

Some examples:

xGESV - Solves $AX = B$ for general matrices

xGELS - Solves least squares problems

xGESVD - Computes Singular Value Decomposition

xGEQRF - QR factorization

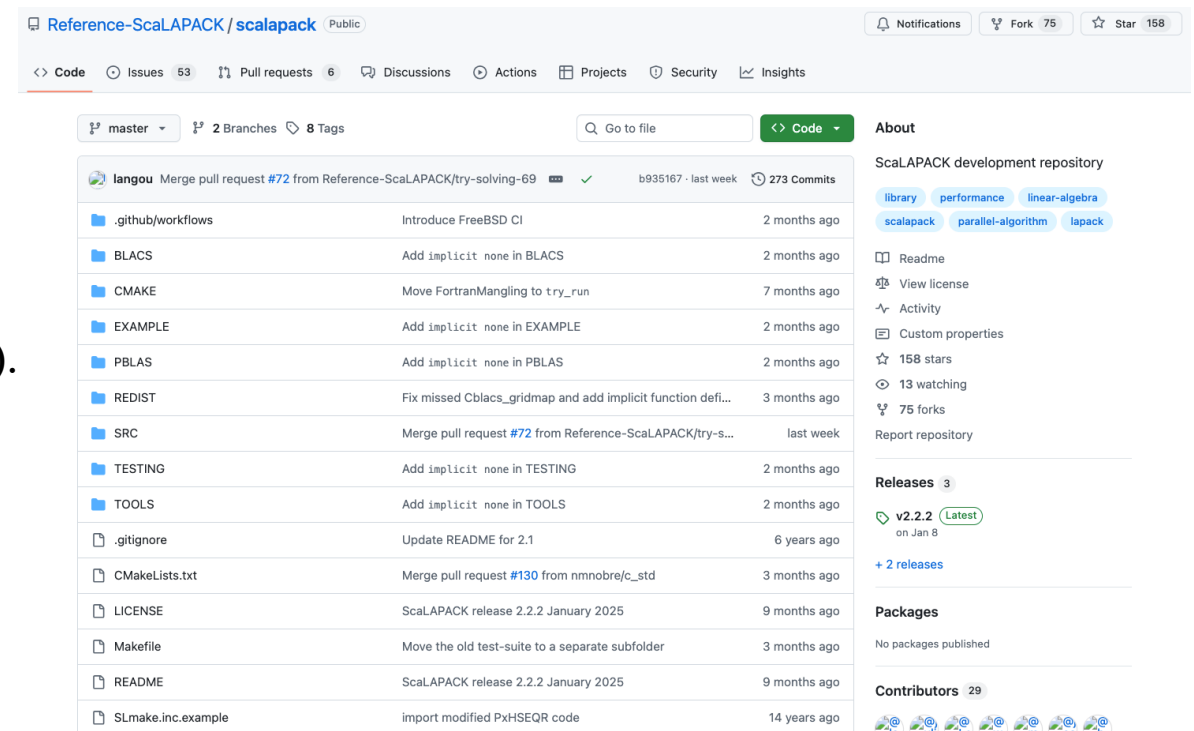
ScaLAPACK



ScaLAPACK is a high-performance library designed for solving linear algebra problems on distributed memory systems. It extends the functionality of LAPACK to support parallel computations.

Key components:

- **BLACS** (Basic Linear Algebra Communication Subprograms): Handles communication tasks like broadcasting and reductions in parallel.
- **PBLAS** (Parallel Basic Linear Algebra Subprograms).
- It uses a 2D block cyclic data distribution to ensure load balancing and minimize communication overhead. This layout is critical for achieving scalability and performance on distributed systems.



NumPy and SciPy



NumPy



The fundamental package for scientific computing with Python

LATEST RELEASE: NUMPY 2.4. VIEW ALL RELEASES

SciPy



Fundamental algorithms for scientific computing in Python

GET STARTED

NumPy 2.4.0 released! 2025-12-20

```
1  """Short NumPy/SciPy BLAS and LAPACK demo."""
2
3  import numpy as np
4  from scipy.linalg import blas, lapack
5
6
7  np.set_printoptions(precision=4, suppress=True)
8
9  A = np.array([[3.0, 1.0, 2.0],
10              [1.0, 4.0, 0.0],
11              [0.0, 2.0, 5.0]])
12  B = np.array([[1.0, 0.0, 2.0],
13              [0.0, 1.0, 1.0],
14              [2.0, 1.0, 0.0]])
15  b = np.array([1.0, 2.0, 3.0])
16  n = A.shape[0]
17
18  # NumPy high-level linear algebra (LAPACK-backed, not a BLAS level)
19  x_solve = np.linalg.solve(A, b)
20  A_inv = np.linalg.inv(A) # For demonstration; prefer solve(A, b) for a system.
21  print("solve residual :", np.linalg.norm(A @ x_solve - b))
22  print("inverse residual:", np.linalg.norm(A @ A_inv - np.eye(n)))
```

SciPy 1.17.0 released! 2026-01-10

```
24  # BLAS Level 1: vector-vector operations
25  u = np.array([1.0, 2.0, 3.0])
26  v = np.array([4.0, 5.0, 6.0])
27  v_axpy = blas.daxpy(u, v.copy(), a=2.0)      # v + 2u
28  u_copy = blas.dcopy(u, y=np.empty_like(u))   # explicit destination
29
30  # BLAS Level 2: matrix-vector and rank-1 operations
31  Au = blas.dgemv(1.0, A, u)                   # Au
32  A0 = np.eye(n)
33  A_rank1 = blas.dger(1.0, u, v, a=A0.copy())   # A0 + uv^T
34
35  # BLAS Level 3: matrix-matrix operation
36  AB = blas.dgemm(1.0, A, B)                   # AB
37
38  # LAPACK: Cholesky factorization of an explicitly SPD matrix
39  M = np.array([[1.0, 2.0, 0.0],
40              [0.0, 1.0, 1.0],
41              [2.0, 0.0, 1.0]])
42  A_spd = M.T @ M + 0.5 * np.eye(n)
43  chol, info_chol = lapack.dpotrf(A_spd.co
44  L = np.tril(chol)
```

blas_lapack_demo.py

Set Up A Python Environment



Here I am using a MacBook Pro 2019 and macOS Tahoe as an example.

Bash



Install Python3 via Homebrew

```
brew install python
```

Bash



Set up a virtual environment for isolation and to avoid conflicts

```
python3 -m venv .venv
```

Bash



Activate virtual environment; after you finish, deactivate

```
source .venv/bin/activate
```

Bash



Install NumPy and SciPy via pip

```
pip install numpy scipy
```

Examples of SciPy



```
40 # BLAS Level 1: vector-vector operations.
41 dot_xy = blas.ddot(x, y)
42 y_axpy = blas.daxpy(x, y.copy(), a=2.0) # y + 2*x
43 y_copy = blas.dcopy(x, y=np.empty_like(x))
44 x_swap, y_swap = blas.dswap(x.copy(), y.copy())
45 norm_x = blas.dnrm2(x)
46
47 # BLAS Level 2: matrix-vector and rank-1 operations.
48 y_mv = blas.dgemv(1.0, A, x, beta=0.5, y=y.copy())
49 x_tri = blas.dtrmv(T, x.copy(), lower=False, trans=0, diag=0)
50 A0 = np.eye(n)
51 A_rank1 = blas.dger(1.0, x, y, a=A0.copy()) # A0 + x*y.T
52
53 # BLAS Level 3: matrix-matrix and symmetric rank-k operations.
54 C = blas.dgemm(1.0, A, B) # beta defaults to zero
55 D = blas.dtrmm(1.0, T, np.eye(n), side=0, lower=False, trans_a=0, diag=0)
56 syrk_upper = blas.dsyrk(1.0, A, lower=False) # upper triangle of A*A.T
57 syrk = np.triu(syrk_upper) + np.triu(syrk_upper, 1).T
```

TBC

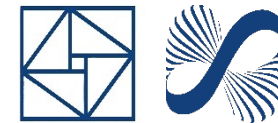
Examples of SciPy, Continued



```
72 # LAPACK LU factorization and solve. Keep pivots from the same factor.
73 lu, piv, info = lapack.dgetrf(A.copy())
74 require_info("dgetrf", info)
75 b = np.array([[1.0], [2.0], [3.0]])
76 solution_lu, info = lapack.dgetrs(lu, piv, b.copy())
77 require_info("dgetrs", info)
78 assert_close(A @ solution_lu, b)
79
80 # LAPACK Cholesky factorization and solve of an explicitly SPD matrix.
81 M = np.array(
82     [[1.0, 2.0, 0.0],
83      [0.0, 1.0, 1.0],
84      [2.0, 0.0, 1.0]]
85 )
86 A_spd = M.T @ M + 0.5 * np.eye(n)
87 chol, info = lapack.dpotrf(A_spd.copy(), lower=True)
88 require_info("dpotrf", info)
89 solution_chol, info = lapack.dpotrs(chol, b.copy(), lower=True)
90 require_info("dpotrs", info)
91 assert_close(A_spd @ solution_chol, b)
```

scipy_blas_lapack_demo.py

Finite Element Circus Poem

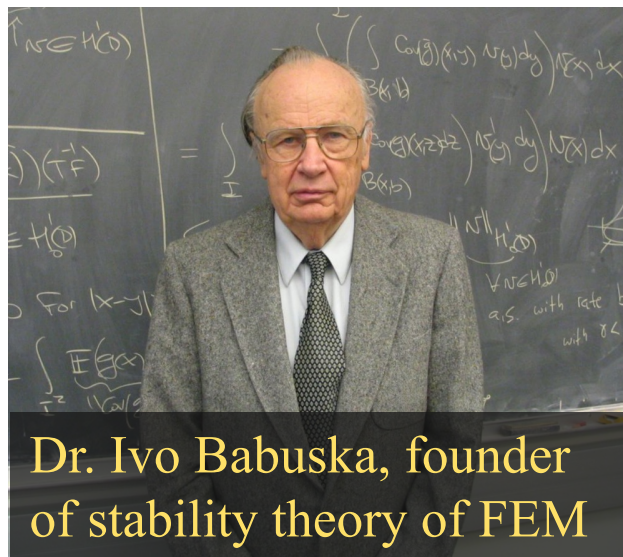


Ivo's Lament

by R. Falk, Fall 1996

*We used to determine the order of speakers
By drawing numbers from a hat.
Now we use a laptop computer
And random numbers to do that.*

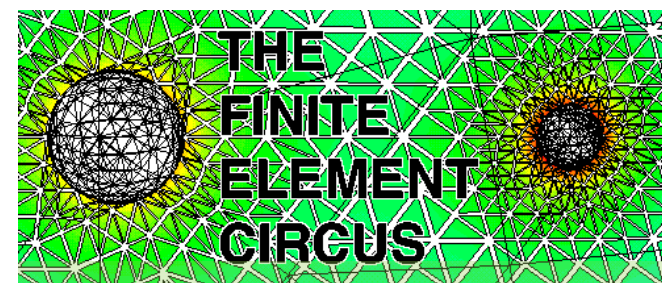
*Someday, we may only have a virtual circus,
Connecting only by computer and phone.
I knew when I let those new guys run the circus
They would not leave well enough alone.*



**Dr. Ivo Babuska, founder
of stability theory of FEM**



**Dr. Rick Falk, long-time
organizer of the Circus**



伊沃叹

豆包译, 2026 年秋

昔抽签筹定序章,
今凭微机列次行。
异日云筵成幻场,
电讯遥连隔参商。
早知后辈司坛掌,
厌守恒规改旧章。

中国科学院大学
2026

数值线性代数
Numerical Linear Algebra

THANKS

Chensong Zhang, AMSS

<https://lsec.cc.ac.cn/~zhanges>

Release version 2026.09.22