SIAM Conference on Computational Science and Engineering (CSE19)

February 25 - March 1, 2019

Spokane Convention Center | Spokane, Washington, USA

SIAM Review
Vol. 60, No. 4, pp. 808–865

The Singular Value Decomposition: Anatomy of Optimizing an Algorithm for Extreme Scale*

Jack Dongarra[†]
Mark Gates[‡]
Azzam Haidar[‡]
Jakub Kurzak[‡]
Piotr Luszczek[‡]
Stanimire Tomov[‡]
Ichitaro Yamazaki[‡]

SIAM/ACM CS&E Awardees

- 2017 - Thomas J. R. Hughes

- 2015 - PETSc Core Development Group:
  - Satish Balay, Jed Brown, William Gropp, Matthew Knepley, Lois Curfman McInnes, Barry Smith, and Hong Zhang

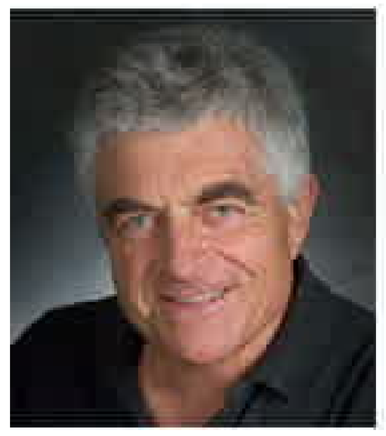
- 2013 - Linda R. Petzold

- 2011 - J. Tinsley Oden

- 2009 - Cleve Moler

- 2007 - Chi-Wang Shu

- 2005 - Achi Brandt



- 2003 - John B. Bell and Phillip Colella

Singular Value Decomposition

For an *mxn* matrix **A** of rank *r* there exists a factorization (Singular Value Decomposition = **SVD**) as follows:

$$A = \underbrace{U}_{mxm}\underbrace{\Sigma}_{mxn}\underbrace{V^T}_{nxn}$$

The columns of ***U*** are orthogonal eigenvectors of $\boldsymbol{AA^T}$.

The columns of ***V*** are orthogonal eigenvectors of $\boldsymbol{A^TA}$.

Eigenvalues $\lambda_1 \ldots \lambda_r$ of $\boldsymbol{AA^T}$ are the eigenvalues of $\boldsymbol{A^TA}$.

$$\sigma_i = \sqrt{\lambda_i}$$

$$\Sigma = diag(\sigma_1 \ldots \sigma_r)$$

Singular values.

SVD Background

- **Singular Value Decomposition**

E. Beltrami
(1835 - 1900)

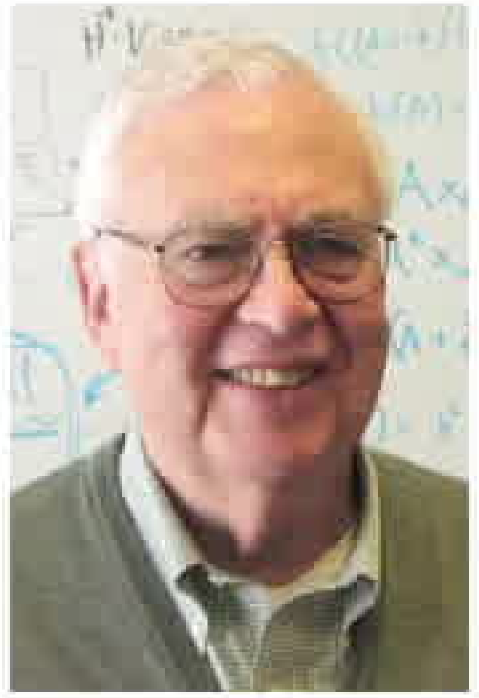
M. Jordan
(1838 - 1922)

J. Sylvester
(1814 - 1897)

E. Schmidt
(1814 - 1897)

H. Weyl
(1885 - 1955)

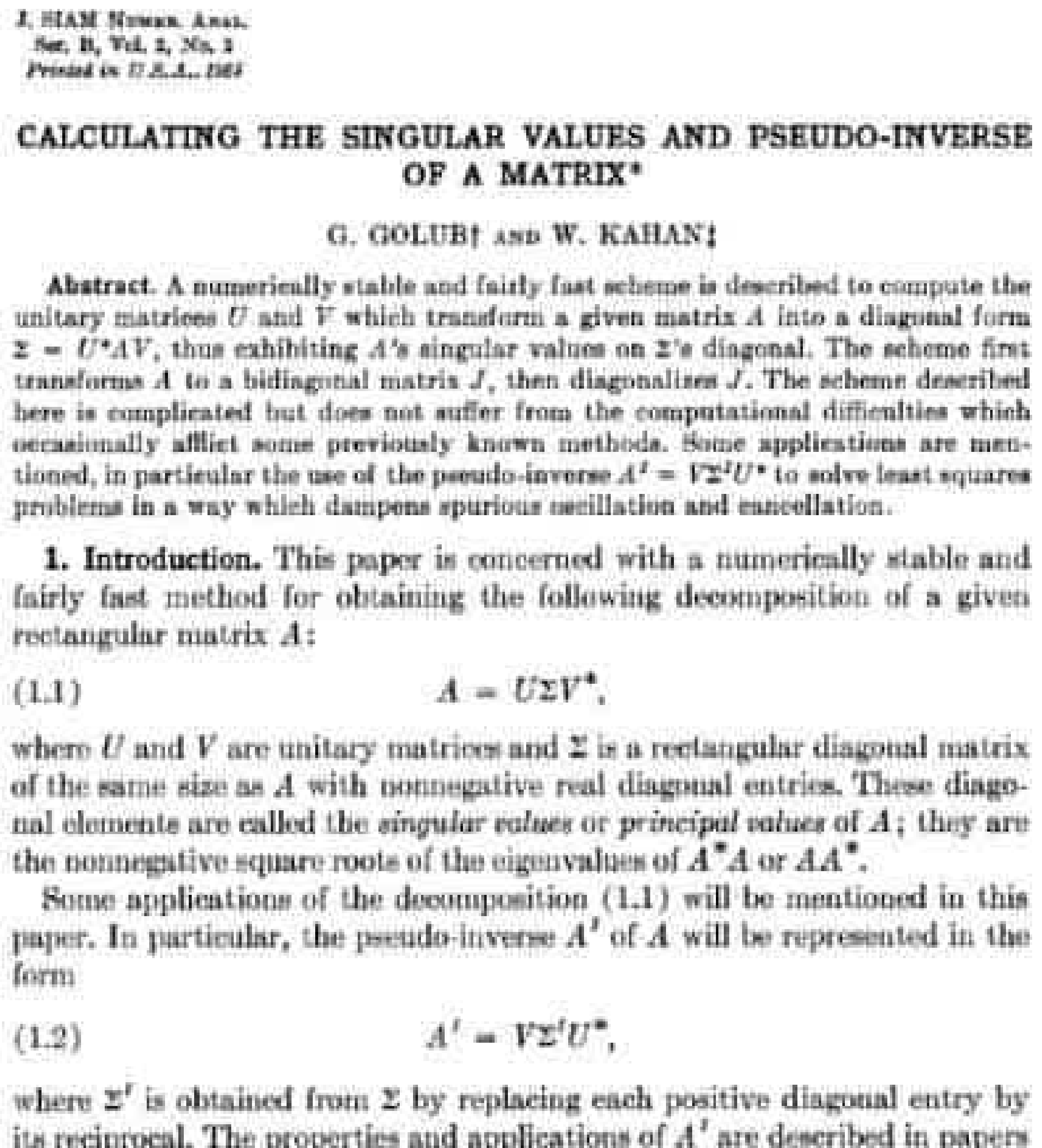
C. Eckart
(1902 – 1973)



Gene Golub
(1932 – 2007)

Velvel Kahan
(1933 –)

Efficient and Stable Algorithm (1965)



J. SIAM Numer. Anal.
Ser. B, Vol. 2, No. 2
Printed in U.S.A., 1965

CALCULATING THE SINGULAR VALUES AND PSEUDO-INVERSE OF A MATRIX*

G. GOLUB† AND W. KAHAN‡

Abstract. A numerically stable and fairly fast scheme is described to compute the unitary matrices U and V which transform a given matrix A into a diagonal form $\Sigma = U^*AV$, thus exhibiting A's singular values on Σ's diagonal. The scheme first transforms A to a bidiagonal matrix J, then diagonalizes J. The scheme described here is complicated but does not suffer from the computational difficulties which occasionally afflict some previously known methods. Some applications are mentioned, in particular the use of the pseudo-inverse $A^I = V\Sigma^I U^*$ to solve least squares problems in a way which dampens spurious oscillation and cancellation.

1. Introduction. This paper is concerned with a numerically stable and fairly fast method for obtaining the following decomposition of a given rectangular matrix A:

$$A = U\Sigma V^*, \tag{1.1}$$

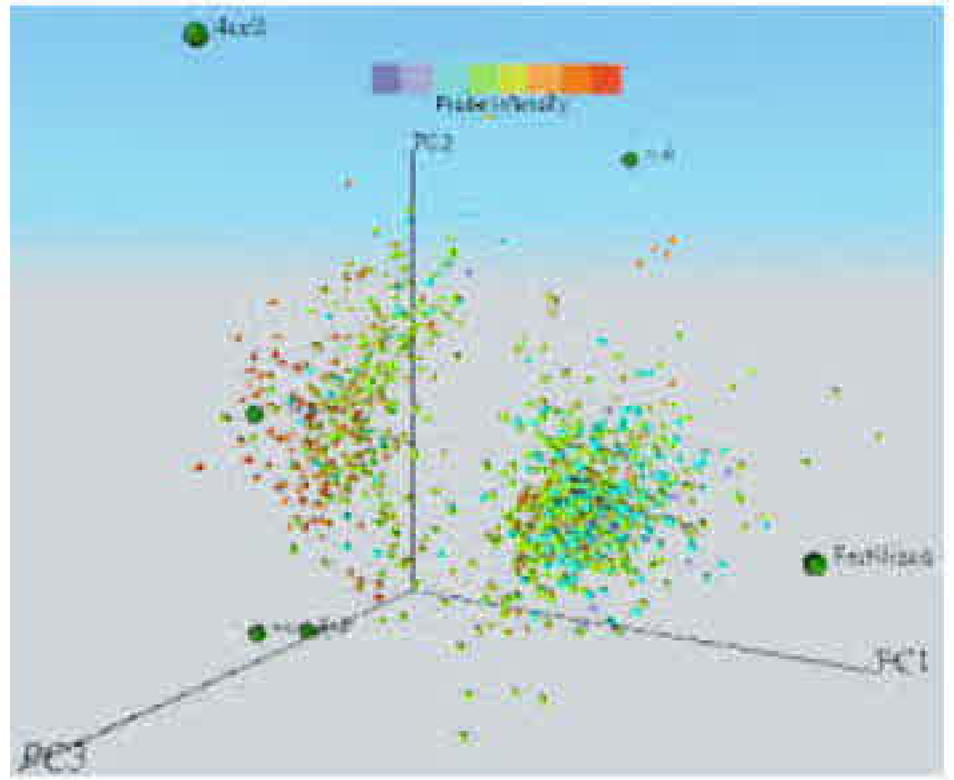
where U and V are unitary matrices and Σ is a rectangular diagonal matrix of the same size as A with nonnegative real diagonal entries. These diagonal elements are called the *singular values* or *principal values* of A; they are the nonnegative square roots of the eigenvalues of A^*A or AA^*.

Some applications of the decomposition (1.1) will be mentioned in this paper. In particular, the pseudo-inverse A^I of A will be represented in the form

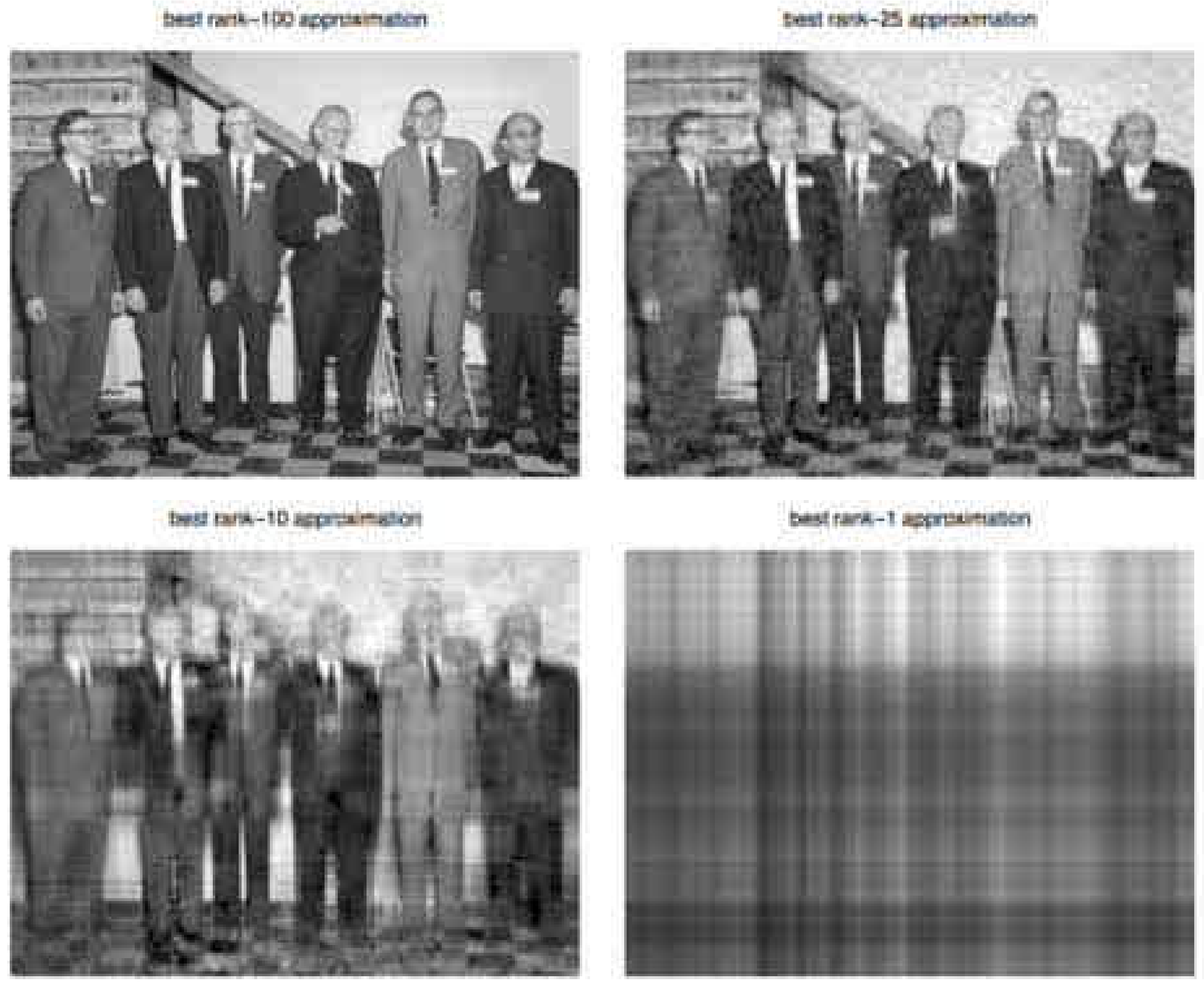
$$A^I = V\Sigma^I U^*, \tag{1.2}$$

where Σ^I is obtained from Σ by replacing each positive diagonal entry by its reciprocal. The properties and applications of A^I are described in papers

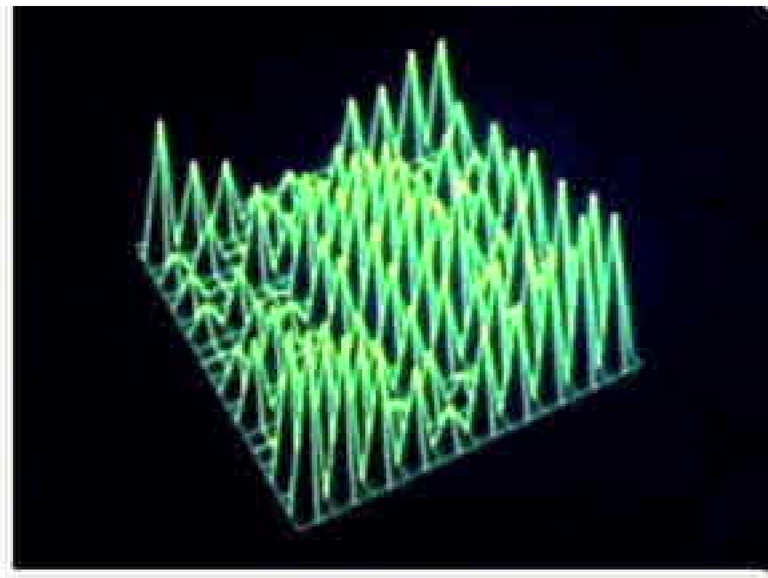
SVD is the "Working Horse" of Linear Algebra

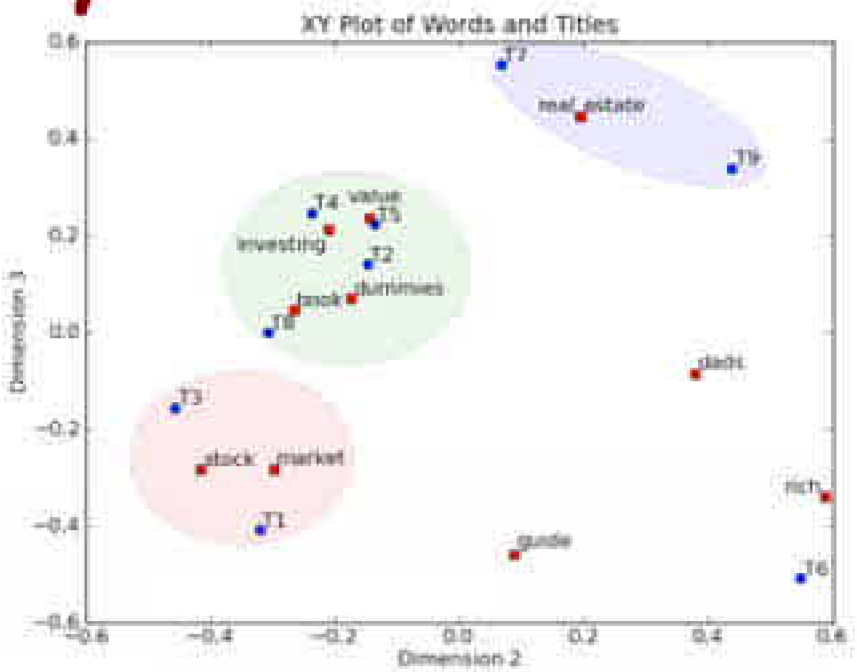


- Once you have it, you have many things:
  - Numerical rank of a matrix
  - Low rank approximation
  - Solve least-squares problems
  - Data fitting
  - Principal Component Analysis
  - Digital Signal Processing
  - Image processing
  - Information retrieval
  - Latent Semantic Indexing
  - Cryptography
  - Many more...




```
a =

     1     2     3
     4     5     6
     7     8     9
    10    11    12

>> [u,s,v]=svd(a)

u =

   -0.1409   -0.8247    0.5447   -0.0571
   -0.3439   -0.4263   -0.6838    0.4822
   -0.5470   -0.0278   -0.2667   -0.7930
   -0.7501    0.3706    0.4057    0.3680

s =

   25.4624         0         0
         0    1.2907         0
         0         0    0.0000
         0         0         0

v =

   -0.5045    0.7608   -0.4082
   -0.5745    0.0571    0.8165
   -0.6445   -0.6465   -0.4082
```



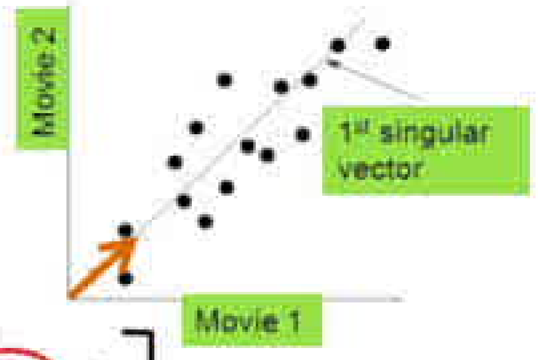


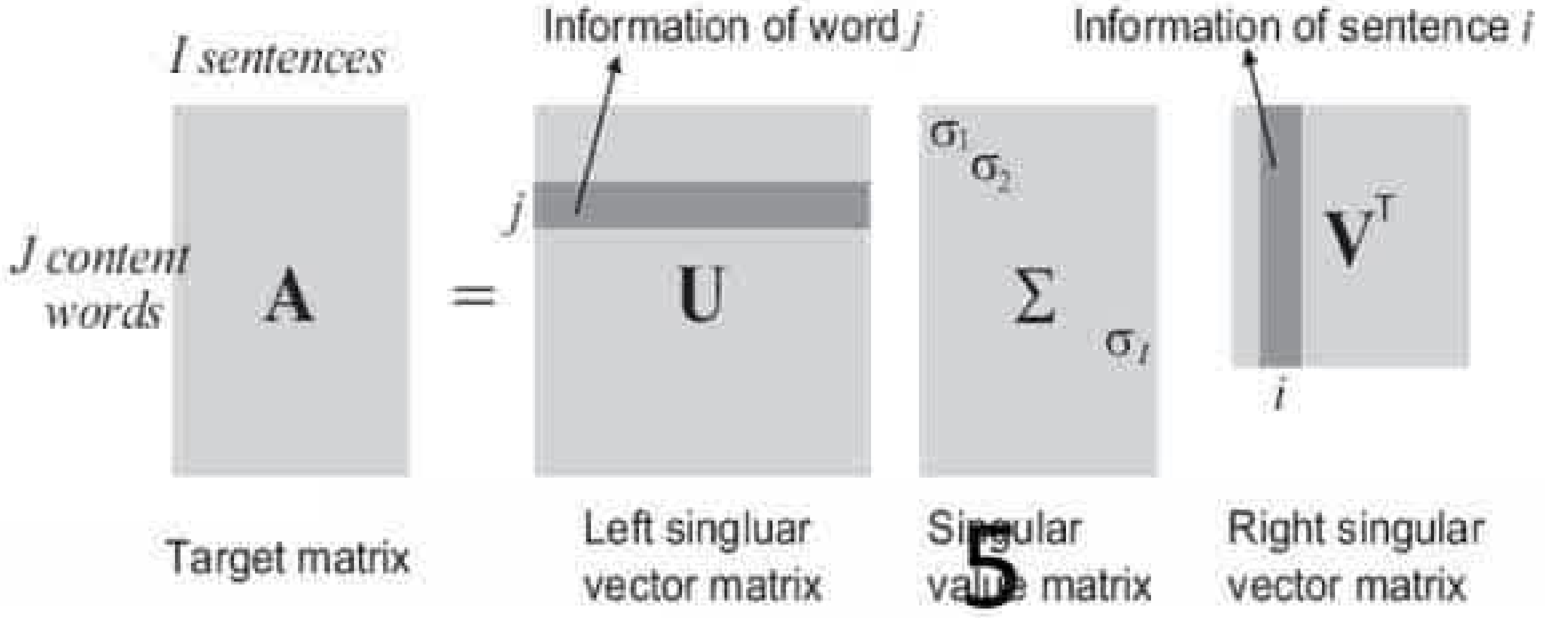
SVD – Example: Users-to-Movies

- $A = U \Sigma V^T$ - example: **Users to Movies**

$$
\begin{array}{c} \\ \text{SciFi} \\ \\ \\ \\ \text{Romance} \\ \\ \end{array}
\begin{array}{c}
\begin{array}{ccccc} \text{Matrix} & \text{Alien} & \text{Serenity} & \text{Casablanca} & \text{Amelie} \end{array} \\
\begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 3 & 3 & 3 & 0 & 0 \\ 4 & 4 & 4 & 0 & 0 \\ 5 & 5 & 5 & 0 & 0 \\ 0 & 0 & 0 & 4 & 4 \\ 0 & 0 & 0 & 5 & 5 \\ 0 & 0 & 0 & 2 & 2 \end{bmatrix}
\end{array}
= \begin{bmatrix} 0.14 & 0.00 \\ 0.42 & 0.00 \\ 0.56 & 0.00 \\ 0.70 & 0.00 \\ 0.00 & 0.60 \\ 0.00 & 0.75 \\ 0.00 & 0.30 \end{bmatrix}
\times \begin{bmatrix} 12.4 & 0 \\ 0 & 9.5 \end{bmatrix}
\times \begin{bmatrix} 0.58 & 0.58 & 0.58 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.71 & 0.71 \end{bmatrix}
$$

Σ is the "spread (variance)" matrix





The "Classical" Algorithm

Derivation of the SVD can be broken down into two major computational steps :

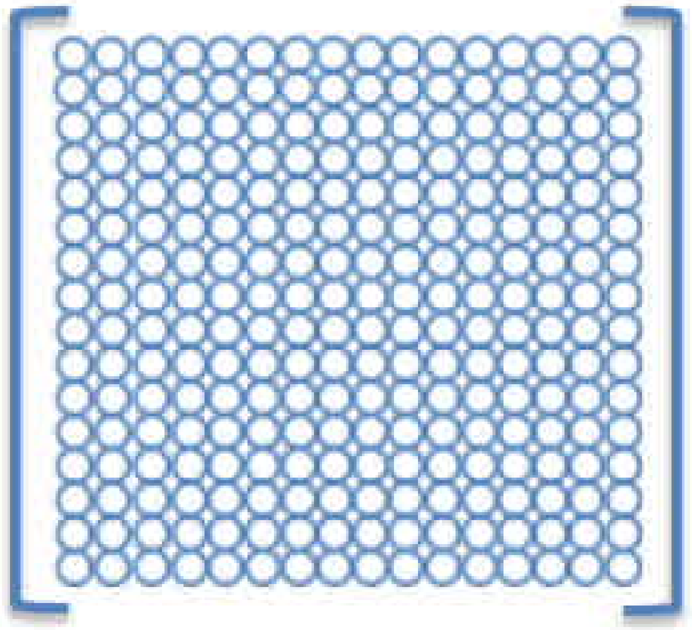
1. Reduce the initial matrix to bidiagonal form using Householder transformations (*a direct and stable process*)
2. Diagonalize the resulting matrix using QR algorithm (*an iterative process*)
3. If the singular vectors desired, back transform U and V.

$$\begin{bmatrix} x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \end{bmatrix} \rightarrow \begin{bmatrix} x & x & & & \\ & x & x & & \\ & & x & x & \\ & & & x & x \\ & & & & x \end{bmatrix} \rightarrow \begin{bmatrix} x & & & & \\ & x & & & \\ & & x & & \\ & & & x & \\ & & & & x \end{bmatrix}$$

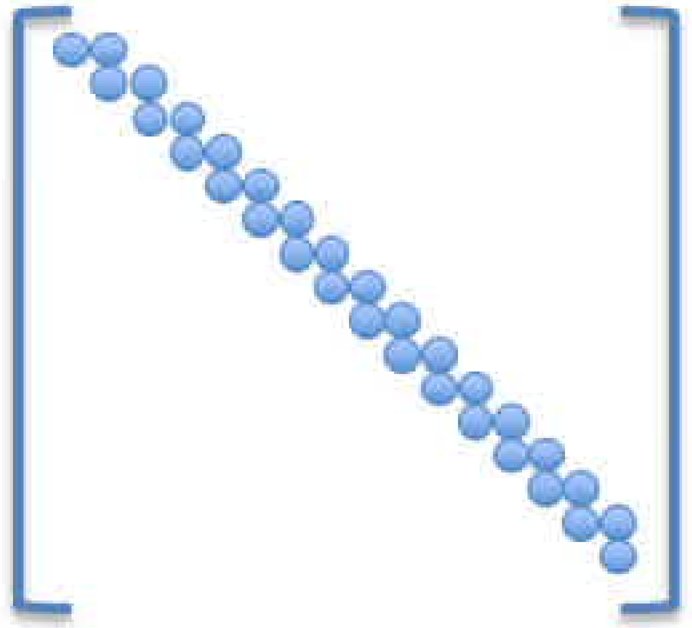
Initial Matrix | Bidiagonal Form | Diagonal Form

Cost for Computing all Singular Values and Vectors

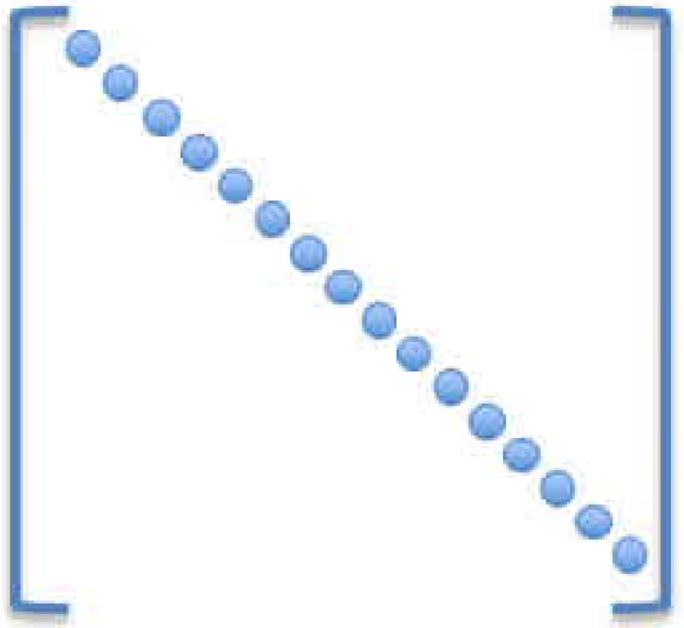
(This discussion we will be using square matrices)



Values: 8/3 n^3 flops
V+Vectors: 5n^3 flops

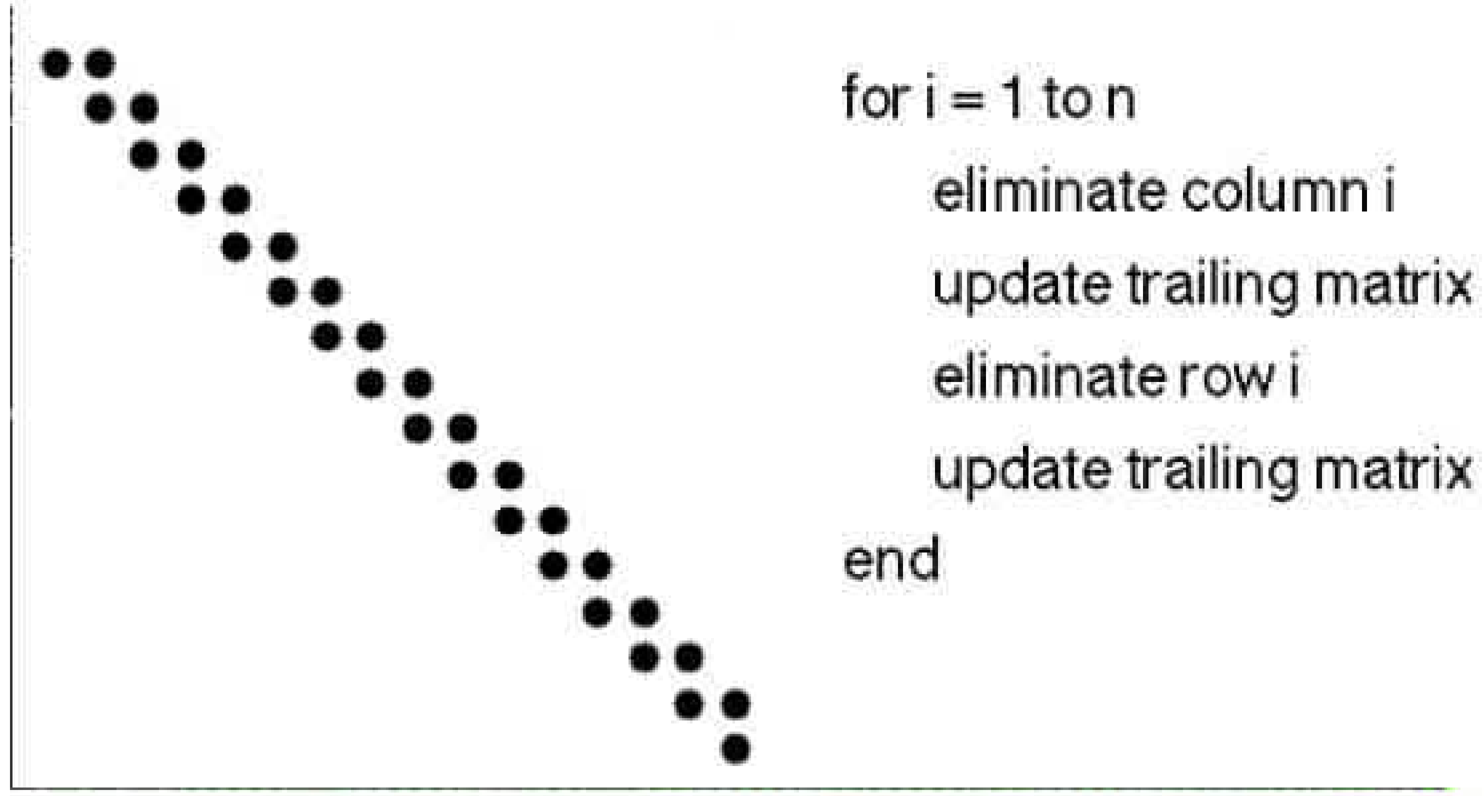


Values: O(n^2) flops
V+Vectors: 12n^3 flops



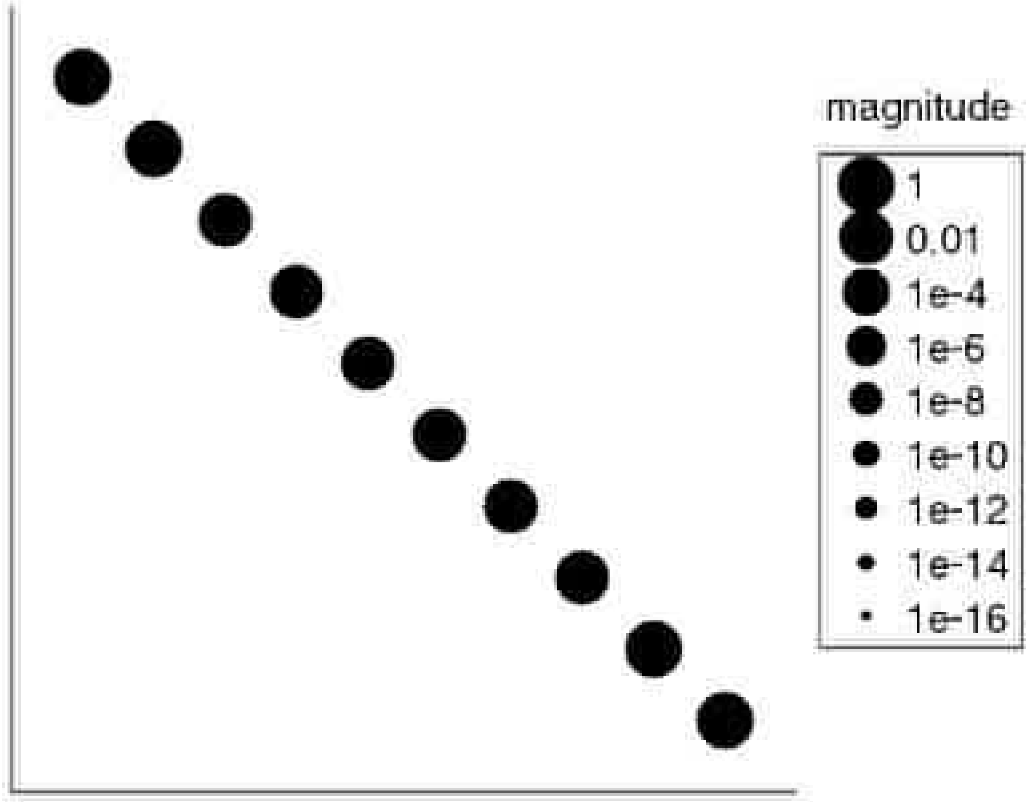
Total Cost:
Values: 8/3 n^3 flops
V + Vectors: 17n^3 flops

Full to Bidiagonal
(Sequence of Householder Transformations)



```
for i = 1 to n
    eliminate column i
    update trailing matrix
    eliminate row i
    update trailing matrix
end
```

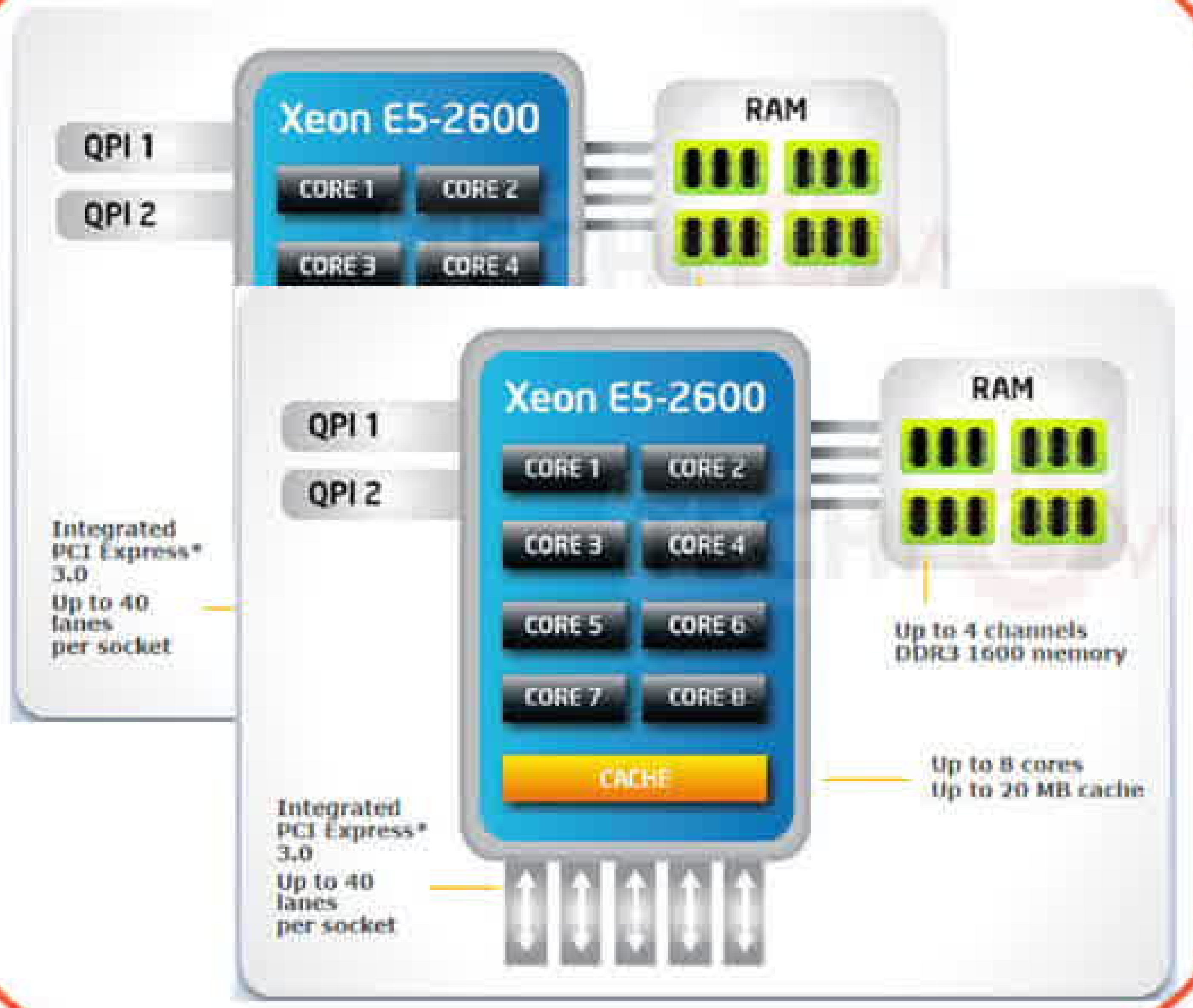
Bidiagonal to Diagonal
(Uses QR Algorithm)



Experiments Using Different SVD Implementations

- Look at various SVD algorithms and software implementations on a specific hardware platform

- **Fix the hardware platform**
  - Intel Sandy Bridge 2.6 GHz, 8 cores per socket
    - Each Core: 8 Flops per core per cycle
      - 2.6 Gcycles/sec * 8 flops / cycle = 20.8 Gflop/s per core
    - Socket Peak 166.4 Gflop/s
  - Dual Socket – total of 16 cores
    - Dual Socket Peak DP 333 Gflop/s



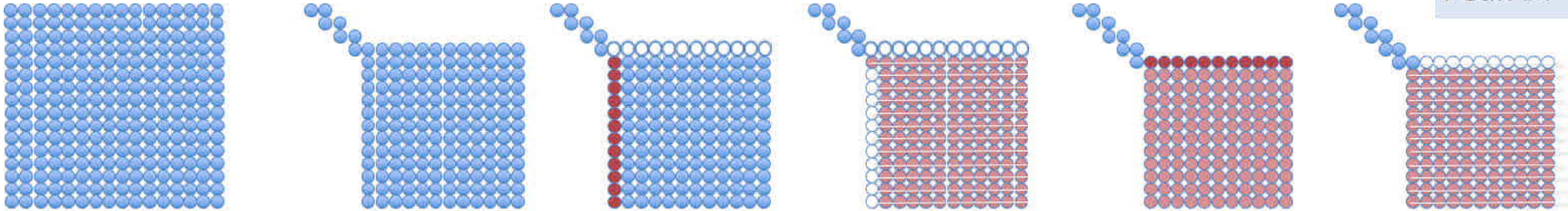
- Use maximum compiler optimization and the best implementation of the BLAS available, Intel's MKL
  - Compiled with icc and using MKL 2015.3.187

EISPACK Version

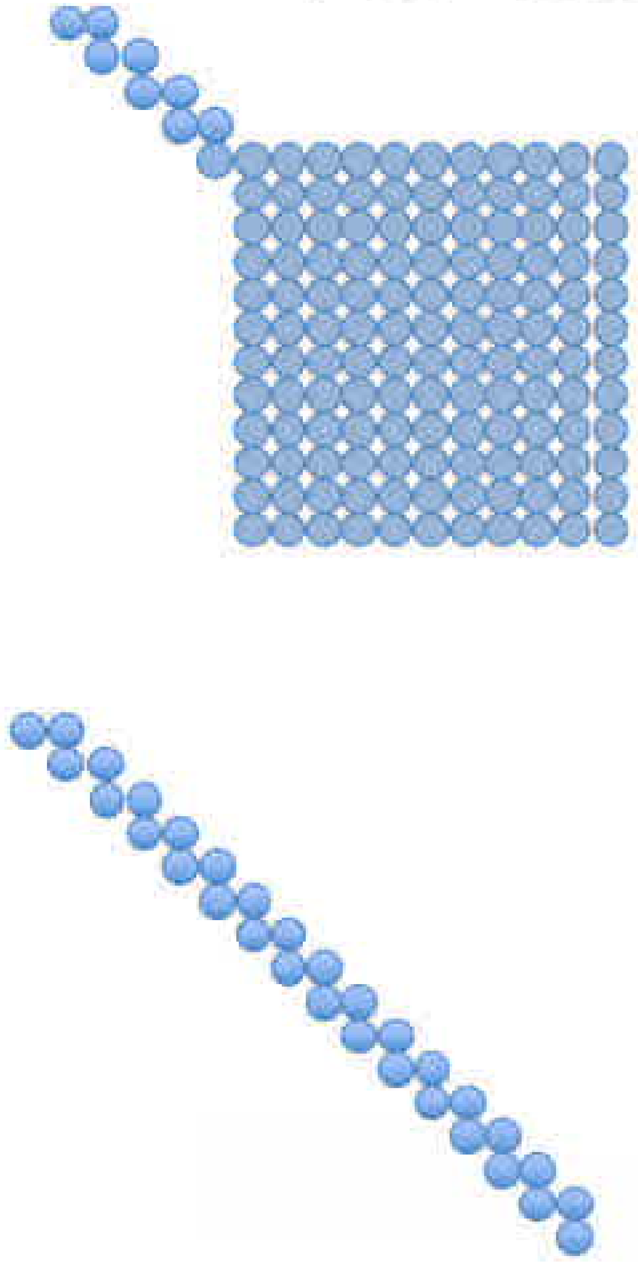
1970 Golub & Reinsch Algol Translation into Fortran

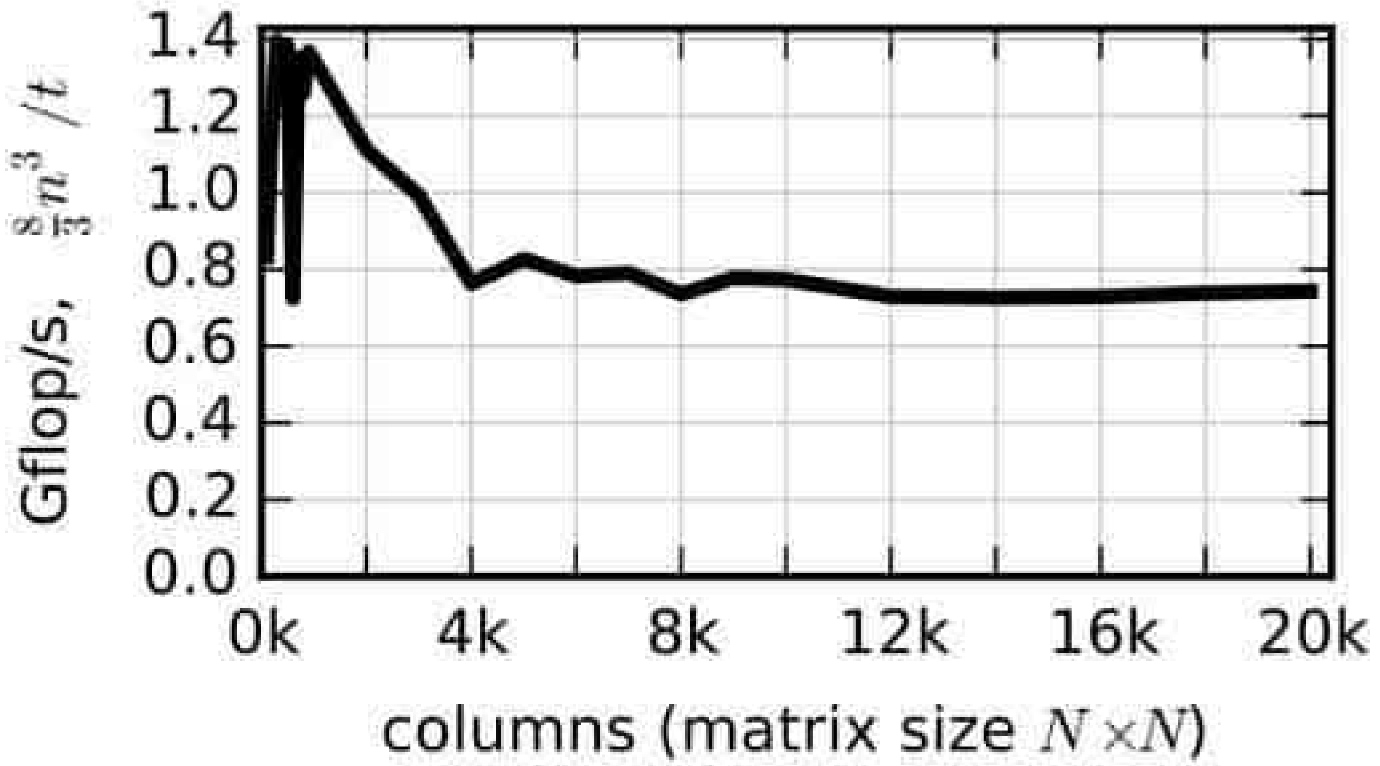
All experiments performed dual socket - 16 core
Intel Sandy Bridge 2.6 GHz
Peak DP 333 Gflop/s

Row oriented; No BLAS

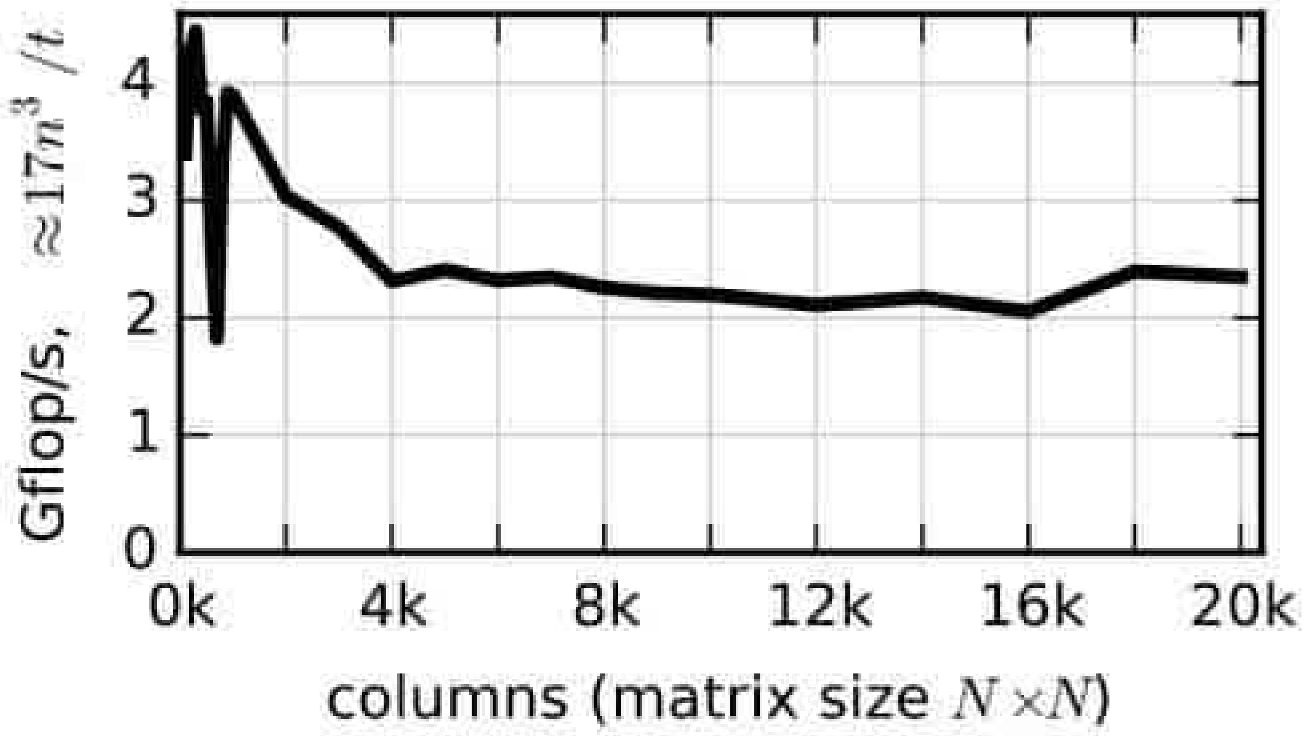


No BLAS, compiler optimization only, Only single core used
Row oriented; Very little data reuse.



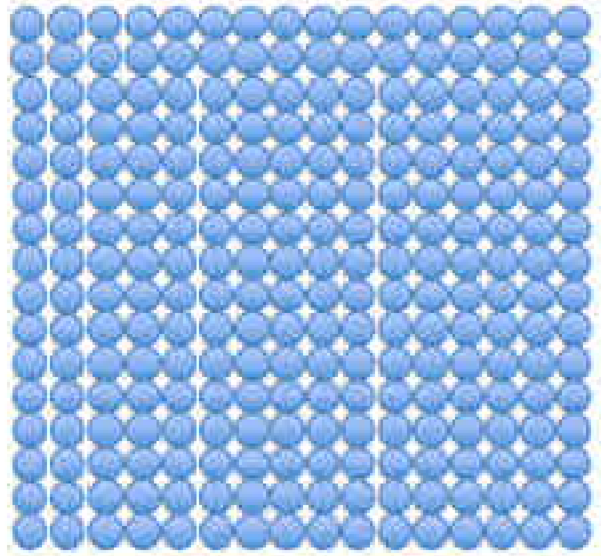


Singular values only

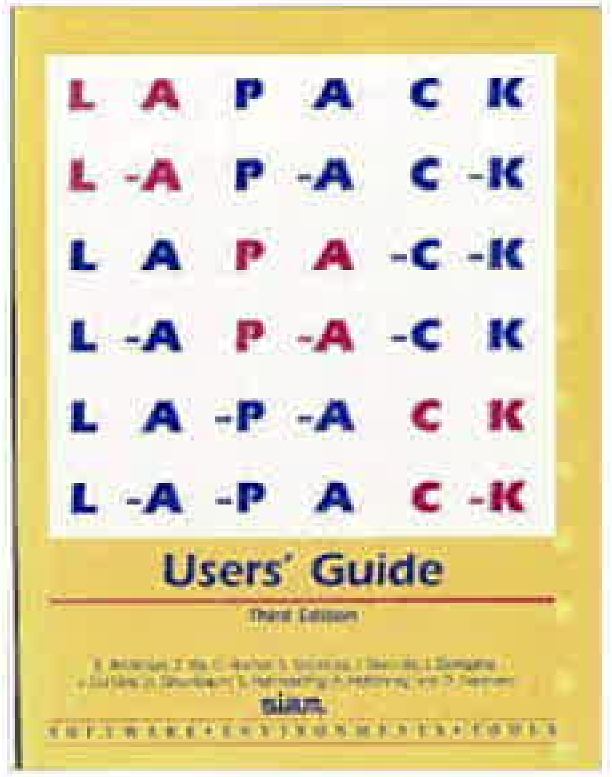


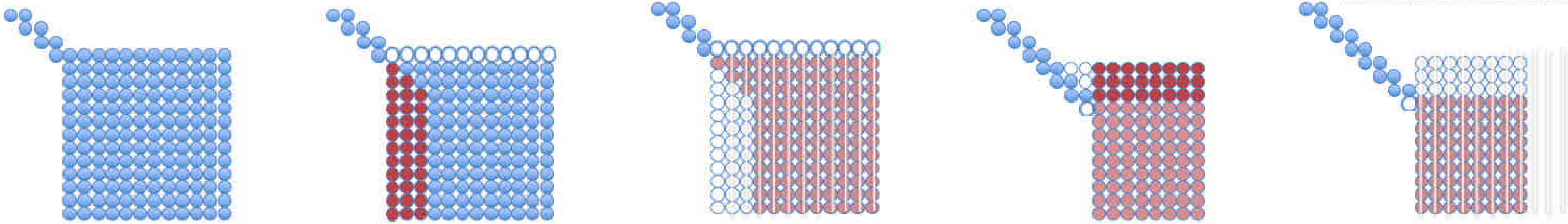
Singular values & vectors

21 Gflop/s peak per core

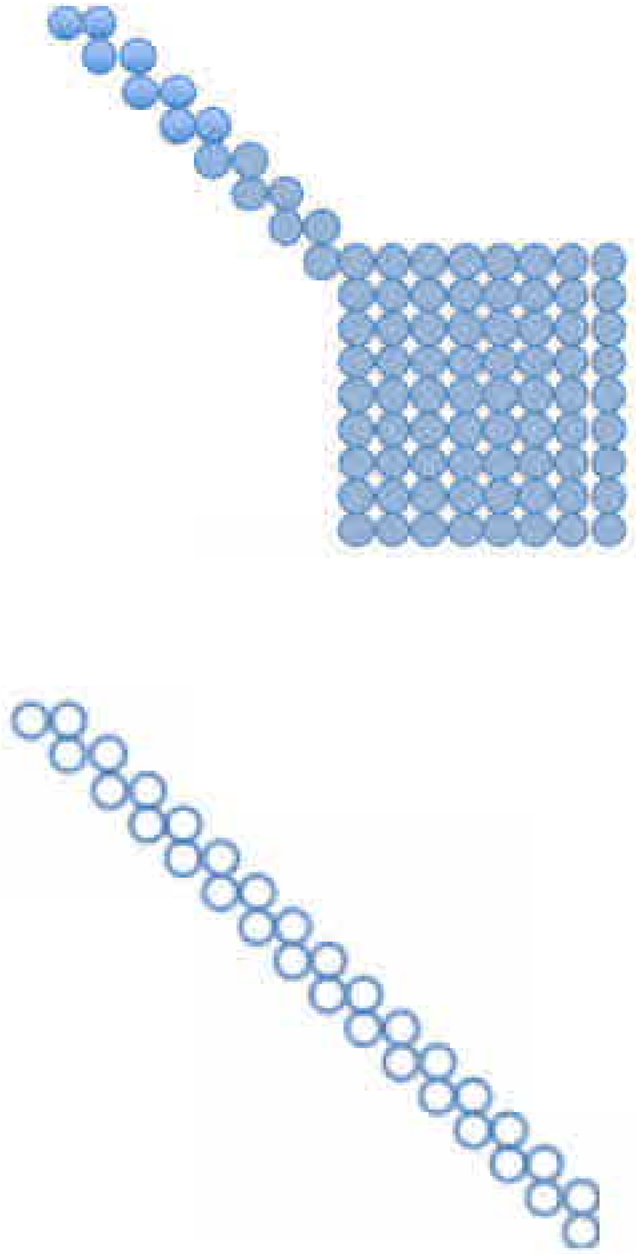


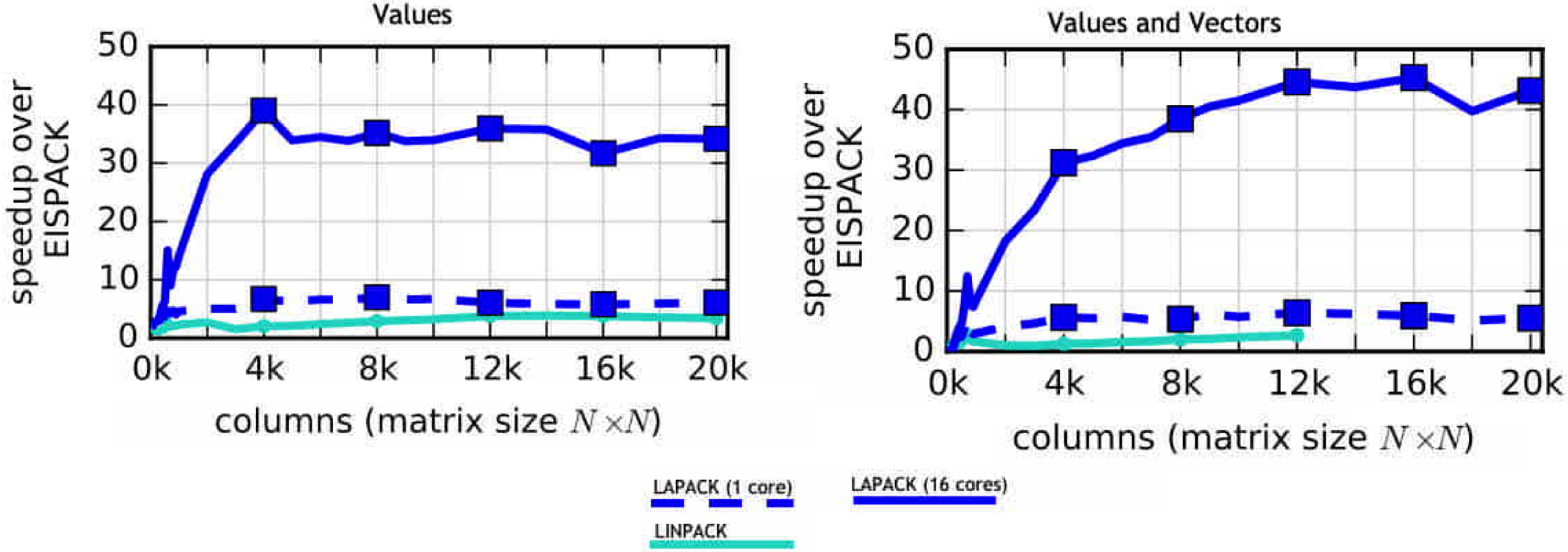
LAPACK Version
Blocked Partitioned Reduction
Level 1, 2, & 3 BLAS
1991 Architectures: Cache based, SMP





Data reuse, Level 2 and 3 BLAS (Level 2.5 Algorithm)
Parallelism from BLAS

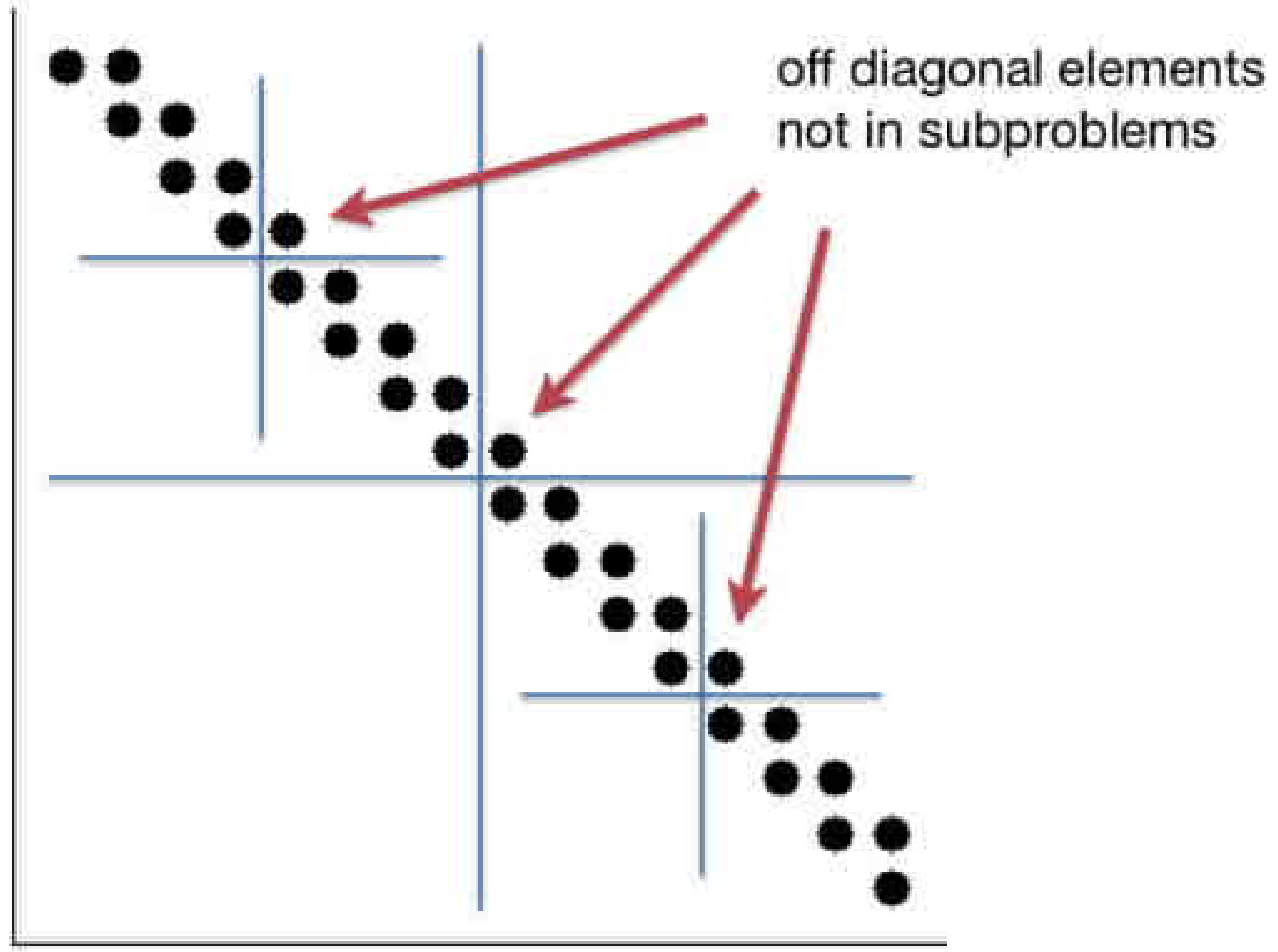


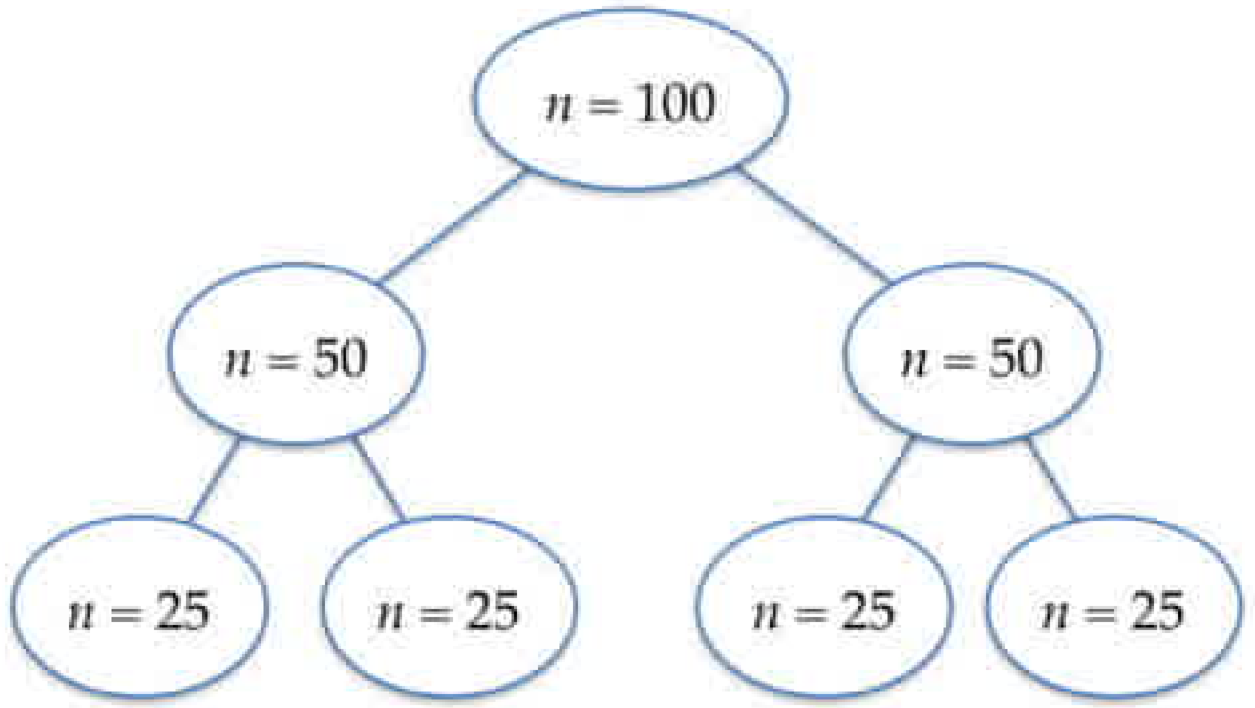


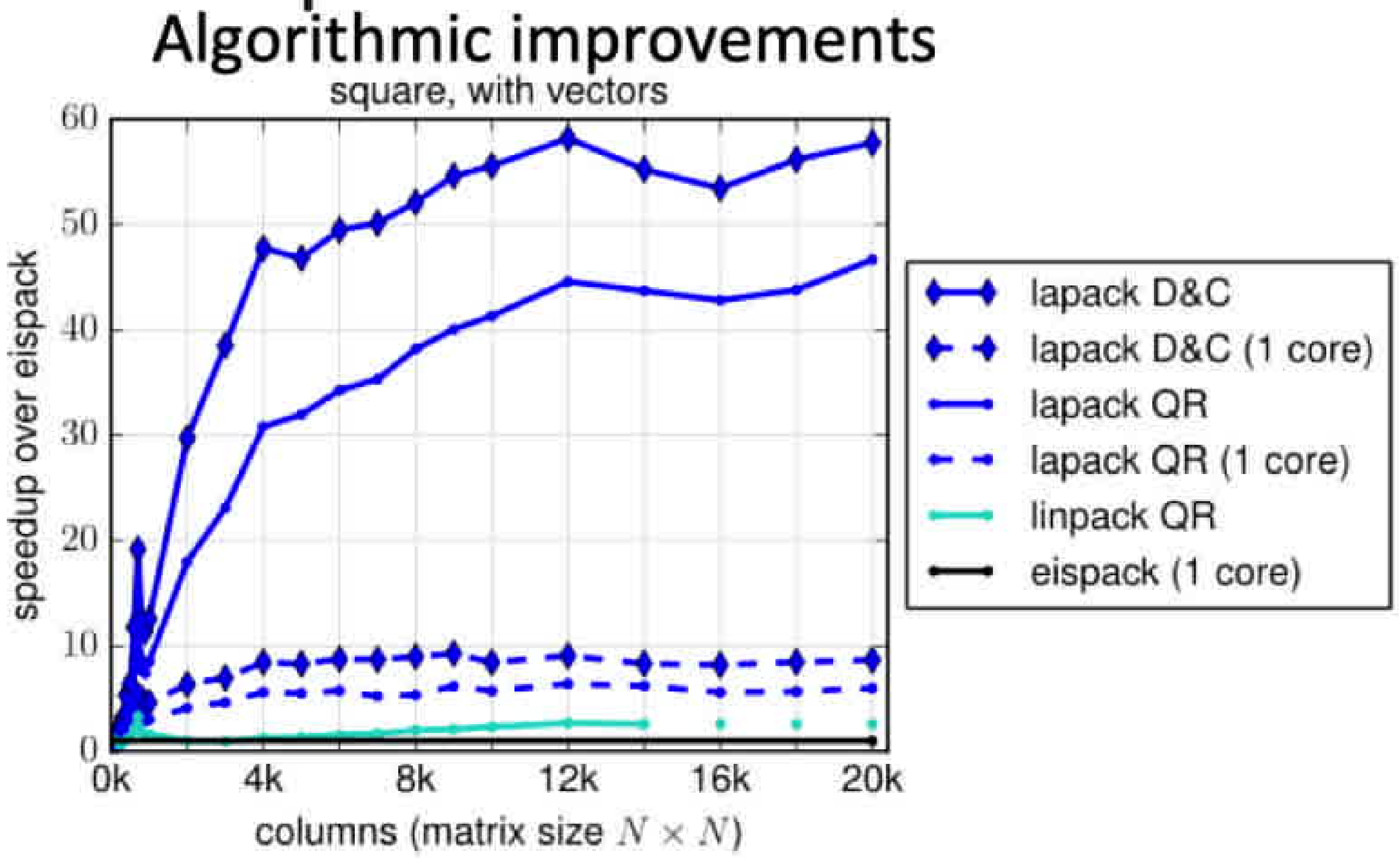
Using Divide & Conquer Algorithm
LAPACK Version

J. Cuppen (Num Math 1980), Gu & Eisenstat (*SIAM J. Matrix Anal. Appl.*, 1993)
(1993: Algorithmic Improvement)

- Reduction to bidiagonal form (using the previous approach), then
- Divide bidiagonal in half (rank-1 change), and do it recursively
  Recurse down to $n \approx 25$, then use QR algorithm to fine singular values
- Combine subproblems by solving secular equation
- Reduces flops from $\sim 17n^3$ to $\sim 9n^3$ and uses Level 3 BLAS
  Much less work in accumulation of *U & V* from subproblems.

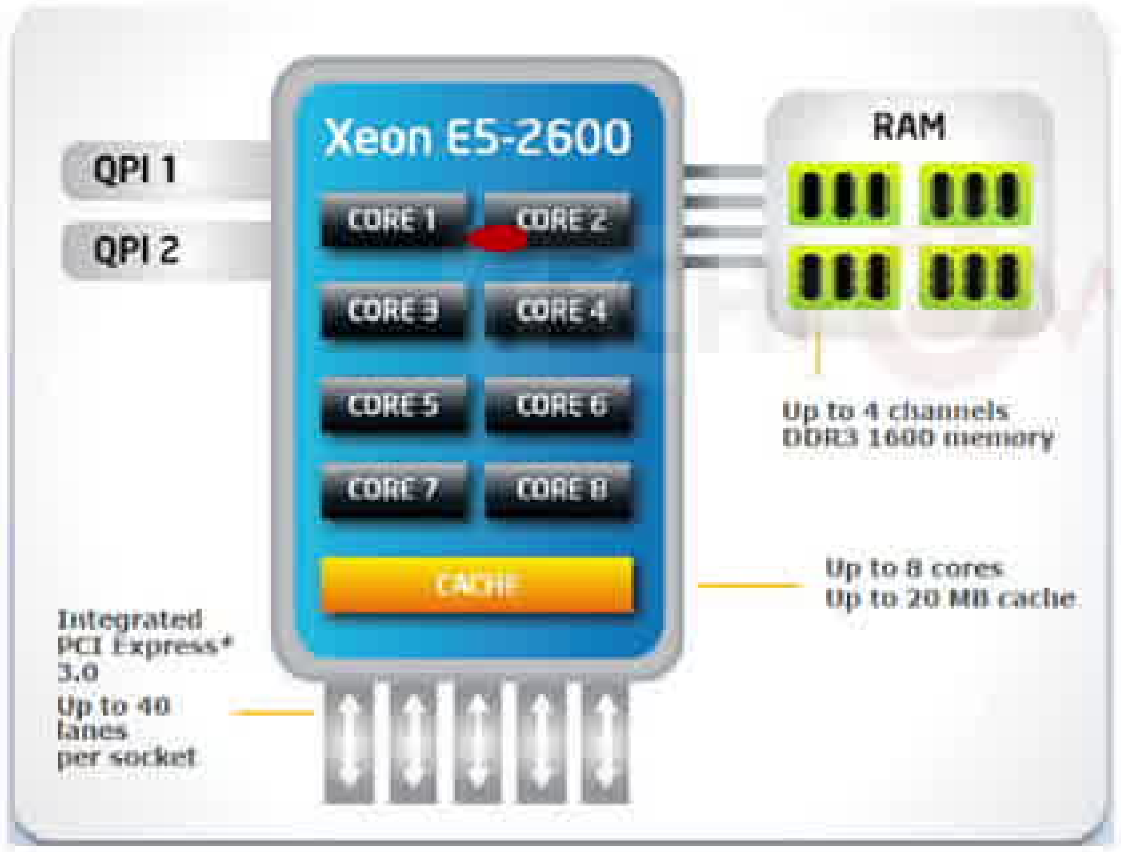






Commodity Processors …

Over provisioned for floating point operations
Today it's all about data movement

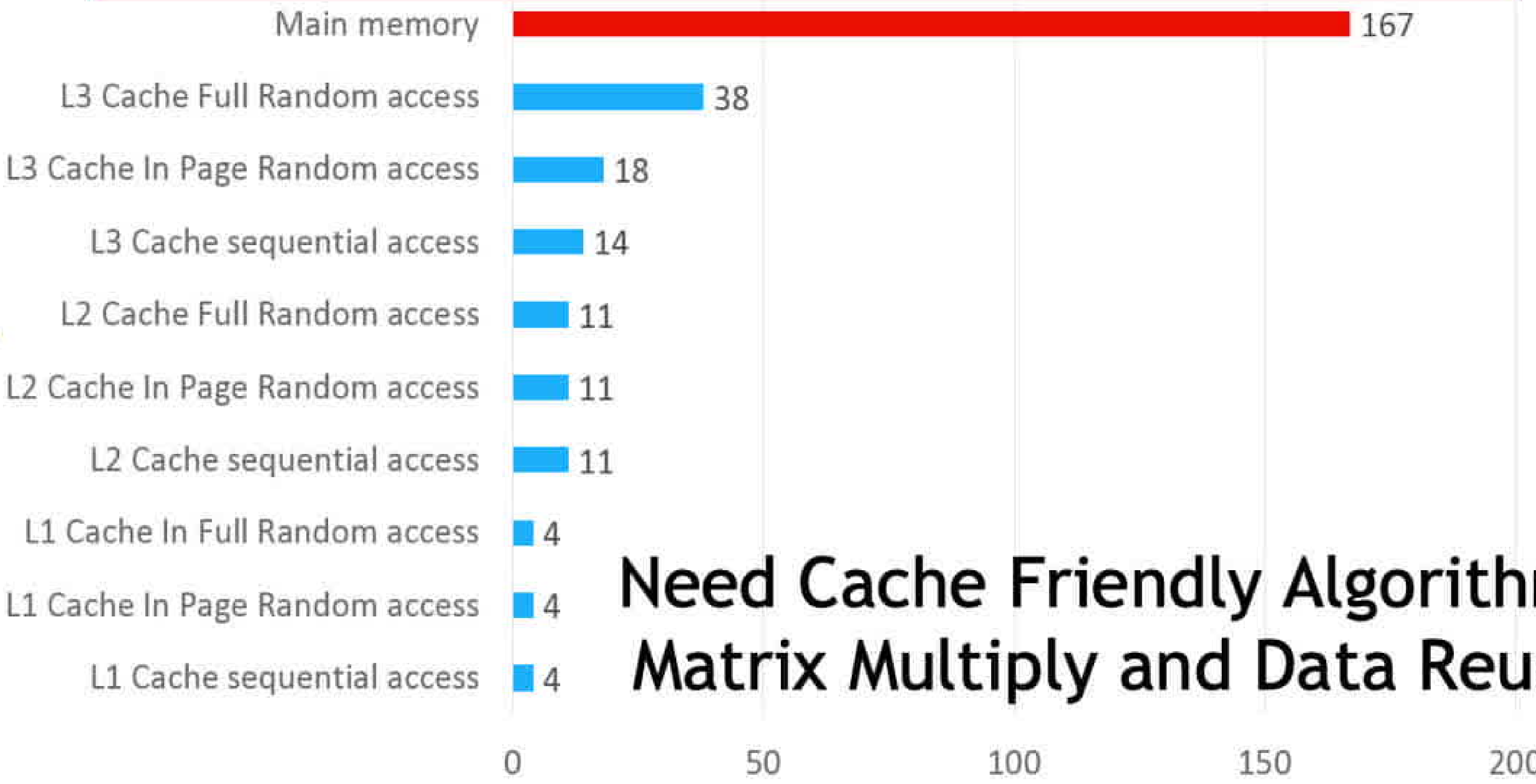


Each Core: 8 Flops per core / cycle
(Old processor, newer 32 f/c)

Each Core Peak DP 20.8 Gflop/s
Each Socket Peak 166.4 Gflop/s

Memory Access Latencies in Clock Cycles
167 cycles to move a word from memory to a register

In 167 cycles single core: 1336 DP Flops, socket: >10K Flops

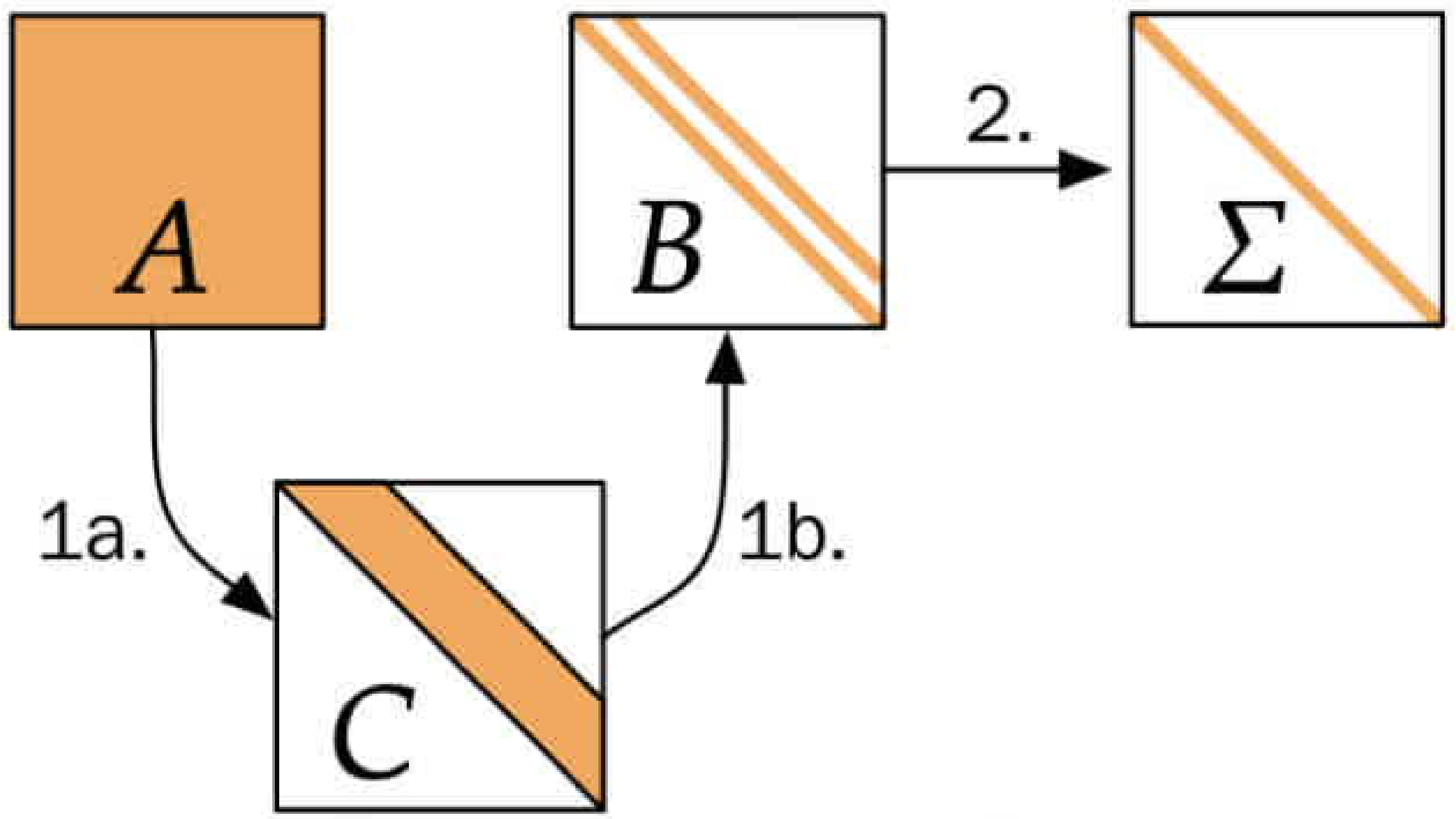


Need Cache Friendly Algorithms
Matrix Multiply and Data Reuse

2-stage Bidiagonalization

(Großer & Lang, *Parallel Computing*, 1999)

$B = U_2 \Sigma V_2^T$



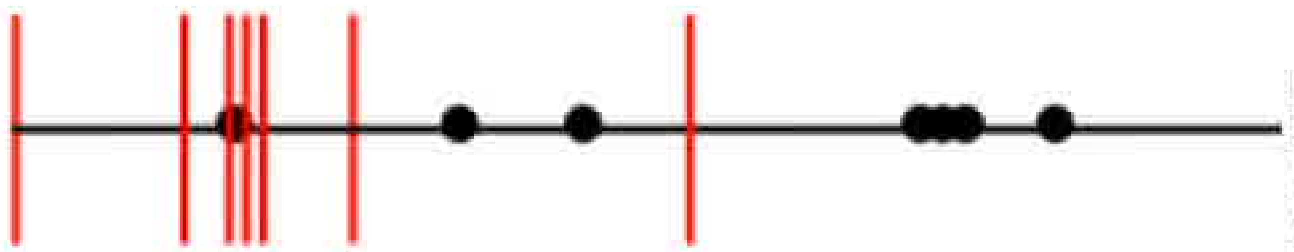
$A = U_a C V_a^T$ $\quad C = U_b B V_b^T$

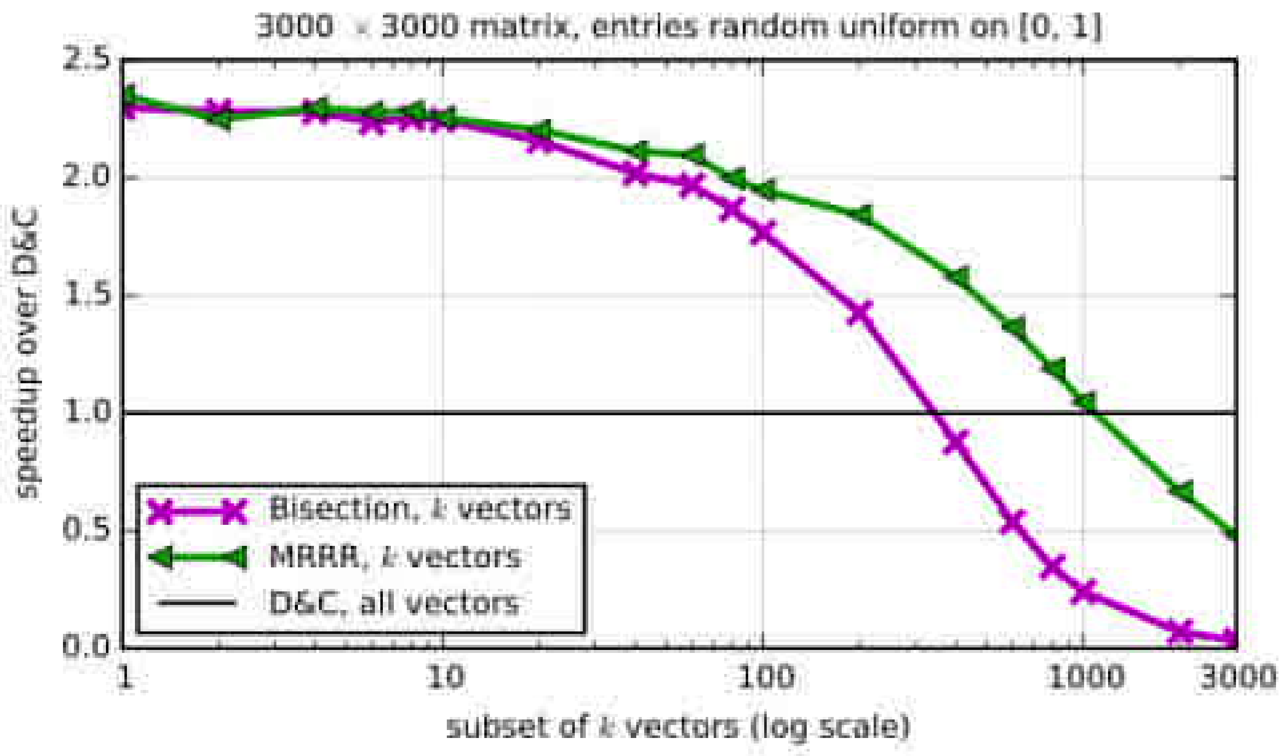
1. Reduction to bidiagonal form (2-stage)
   a. Reduce to band (matrix multiple driven)
   b. Band to bidiagonal (bulge chasing, "cache friendly")
2. But adds additional transformations: U_b, V_b, doubling work in computing singular vectors

3. $U = U_a \, U_b \, U_2$ $\qquad V = V_a \, V_b \, V_2$

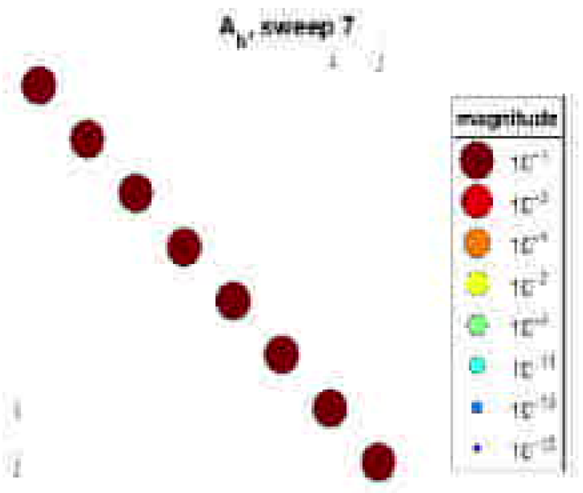
And Many More Algorithms …

- Bisection
  - Strength when computing subset of singular vectors





- Jacobi & Blocked Jacobi
  - Simplicity, Easy parallelization and Potentially better accuracy for certain classes of matrices



- Polar Decomposition, *QDWH*
  - QR-based Dynamically Weighted Halley's iteration
    - 2x to 7x more flops but all parallel
    - 2 iterations maximum
    - Can be up to 3x faster

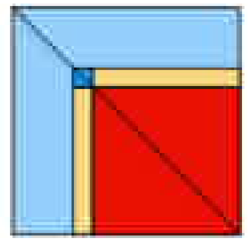
QDWH-SVD is a competitive alternative

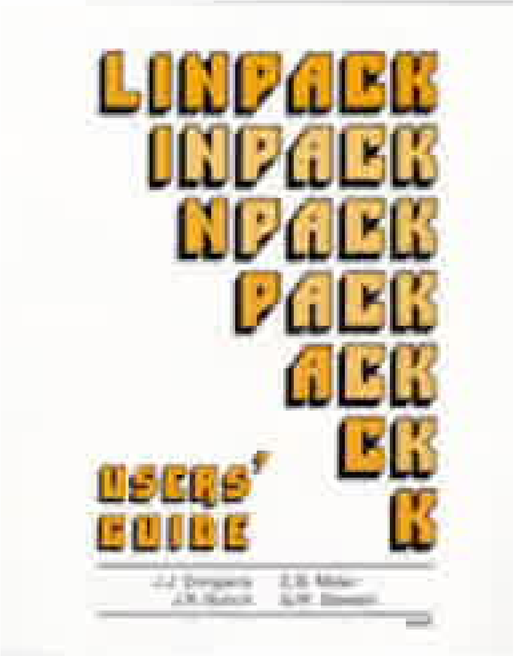
WARNING

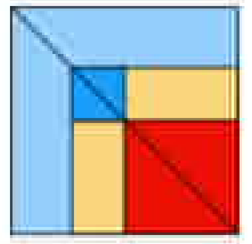
On very large problems	:	**n > 50,000**
With low condition number	:	**$K_2(A) < 10^2$**
On a large # of nodes	:	**N > 32** nodes
On many-core architectures	:	Intel **KNL**
On accelerators	:	NVIDIA **GPU**
In low precision arithmetic	:	**single**
If fast *SYEV* available	:	**2-stage** (MKL-18.2+)
If vectorization available	:	**AVX**-512 (Skylake)

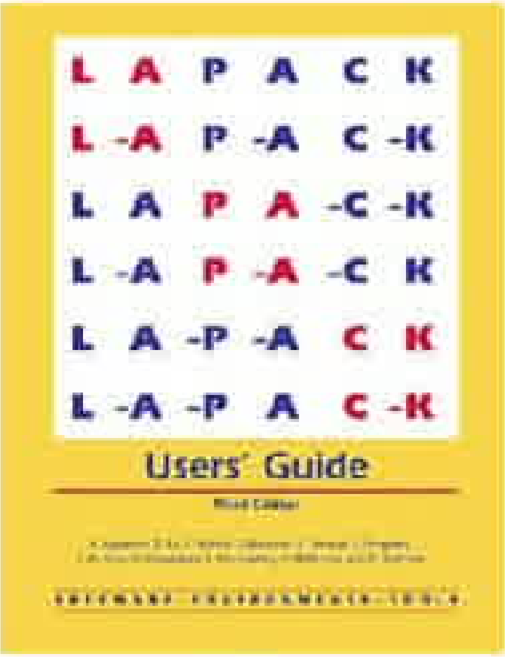
40 Years Evolving Software and Algorithms Tracking Hardware Developments (Past, Present, & Future)

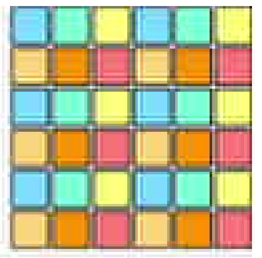
EISPACK (1970's)
Fortran
element-wise operations

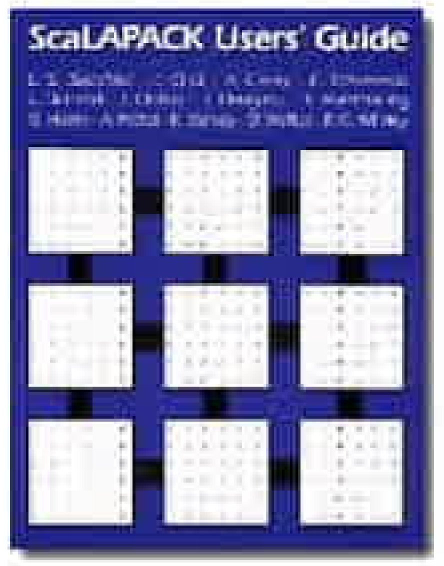
LINPACK (1980's)
vector machines

Level 1 BLAS
vector operations

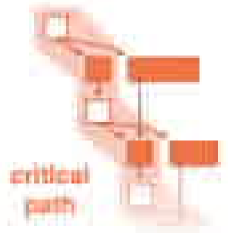


LAPACK (1990's)
cache hierarchies

Level 3 BLAS
blocked operations

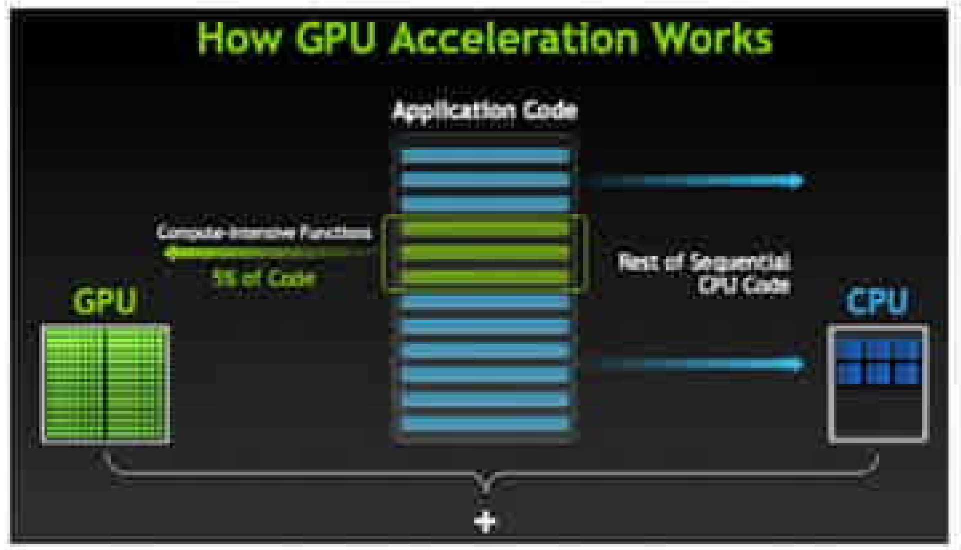


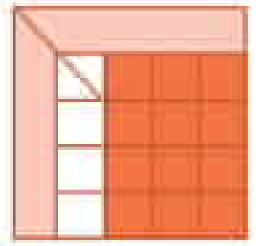
ScaLAPACK (1990's)
distributed memory

PBLAS
message passing

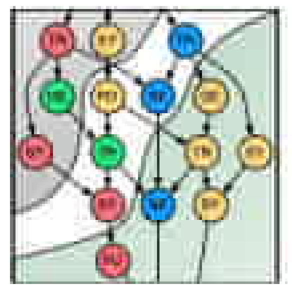


MAGMA (2010's)
accelerators


Hybrid algorithms



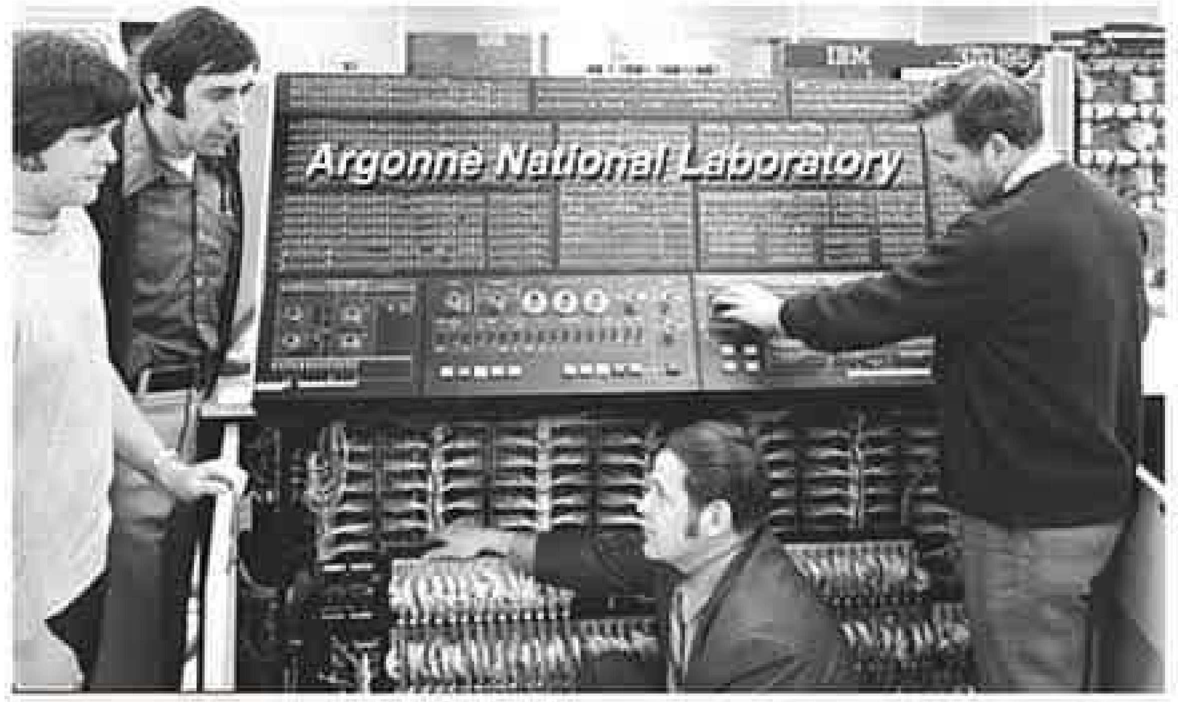
PLASMA (2010's)
multicore

Tiled algorithms +
runtime scheduler

DPLASMA (2010's)
distributed multicore

Implicit DAG +
ParSEC distributed scheduler

SLATE (2020's)
hybrid
Hybrid distributed

Historical Perspective, 1977 to Today

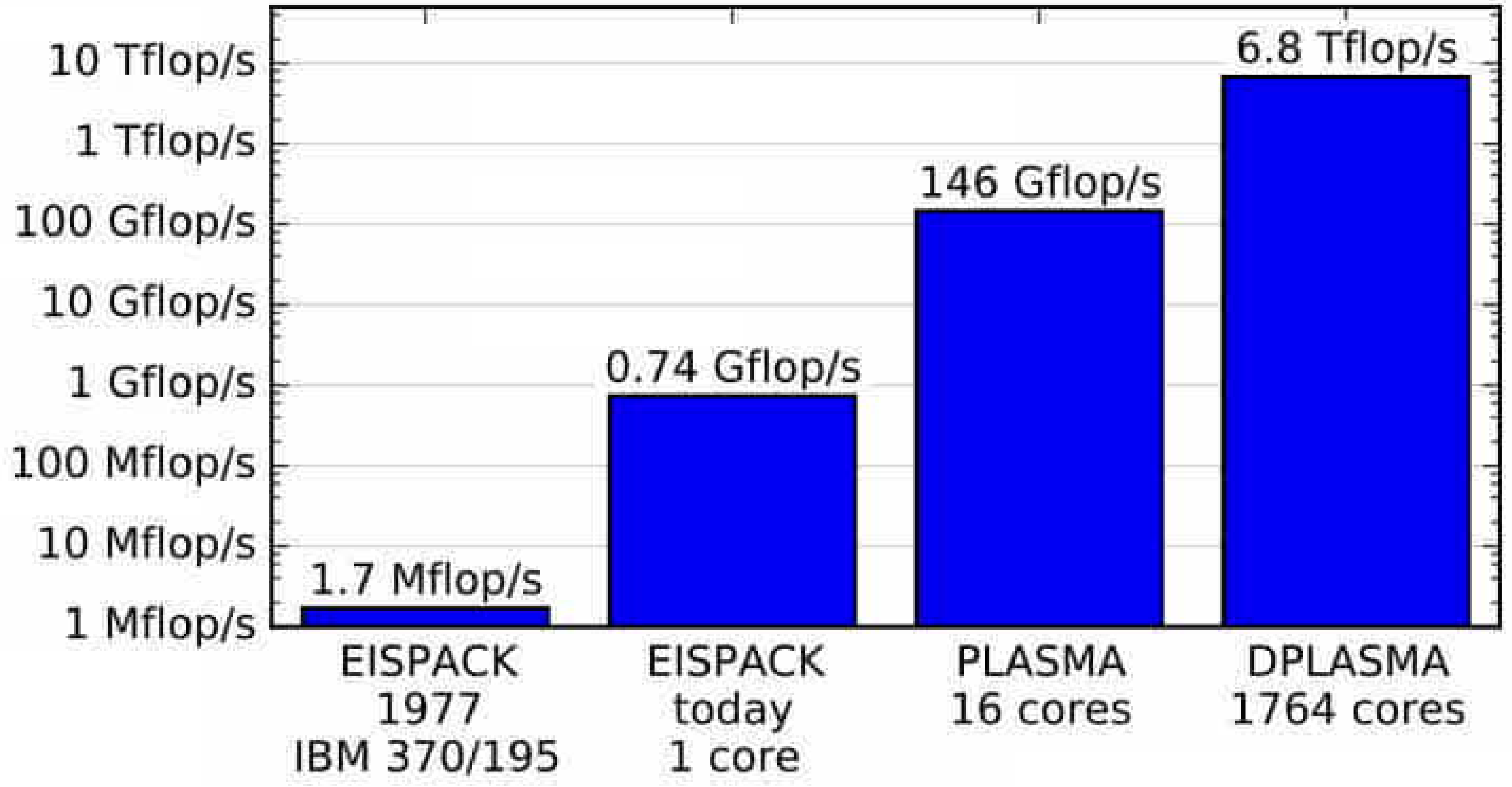
- Hardware improvement
  - EISPACK on Intel SB (1 core) is **435** x faster them EISPACK on IBM 370/195
  - Same software, 40 year improvement in hardware



IBM 370/195 (circa 1977)

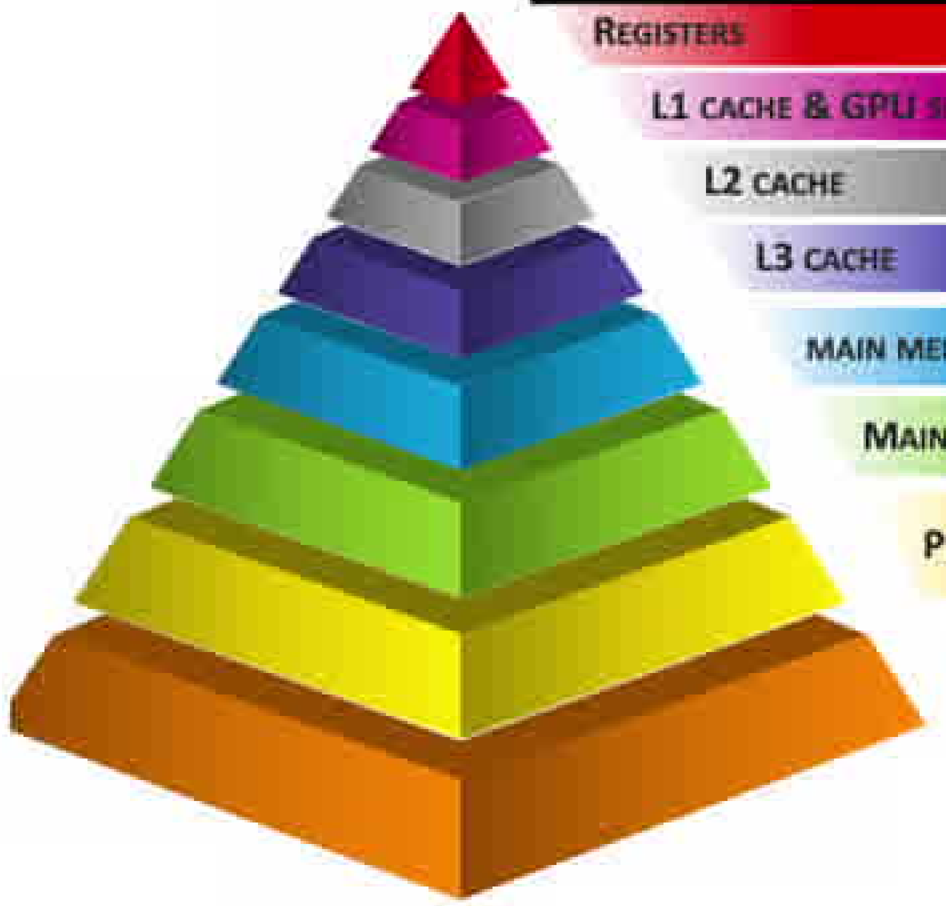
- Algorithm improvement
  - PLASMA **2-stage reduction** and **D&C implementation** is another 197 x faster than EISPACK on Intel SB
    - 2-stage is driven by matrix multiply performance
    - D&C is a parallel algorithm uses 16 cores
    - 85,000 faster then IBM 370/195

- DPLASMA is 47x faster on 1764 processors Haswell cores compared to PLASMA on 16 cores



The Take Away

- Algorithm and software follow architectural changes are critical for performance
  - "Communication avoiding" approach
    - Level 3 BLAS, keep data in upper levels of memory hierarchy
    - Blocked algorithm
    - 2-stage reduction
  - Divide & conquer algorithm
  - QDHW Algorithm
    - More operations can be faster!
    - Operation count ≠ time to solution
- Jacobi
  - Basic version is easy, parallel, accurate, but slow (L1 BLAS)
  - Block Jacobi can be competitive



Memory hierarchies	Haswell E5-2650 v3	P100
	10 cores	56 SM x 64 cores
Registers	16/core AVX2	256 KB/SM
L1 cache & GPU shared memory	32 KB/core	64 KB/SM
L2 cache	256 KB/core	4 MB
L3 cache	25 MB	N/A
Main memory	64 GB	16 GB
Main memory bandwidth	68 GB/s	720 GB/s
PCI express gen3 x16	16 GB/s	16 GB/s
Interconnect Cray Gemini	6 GB/s	6 GB/s