CS 594 — Scientific Computing for Engineers — Spring 2019

Dense Linear Algebra part 2

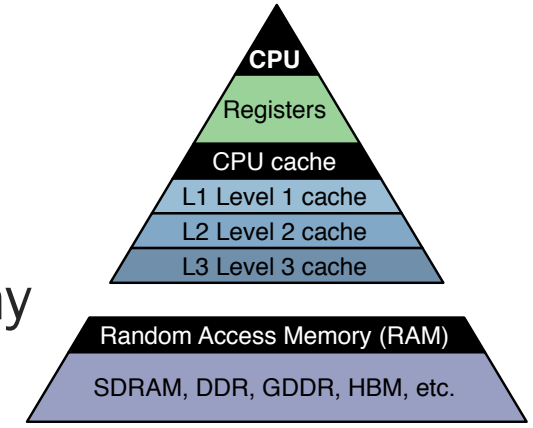
Mark Gates

mgates3@icl.utk.edu

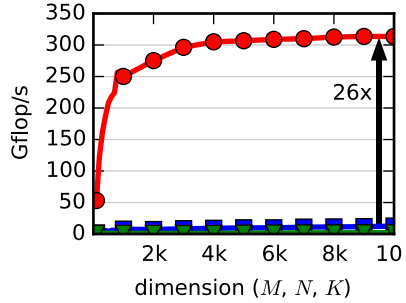
http://www.icl.utk.edu/~mgates3/

Revised March 31: fixed error on slide 37, SUMMA: in diagram, indices of B didn't match pseudo-code

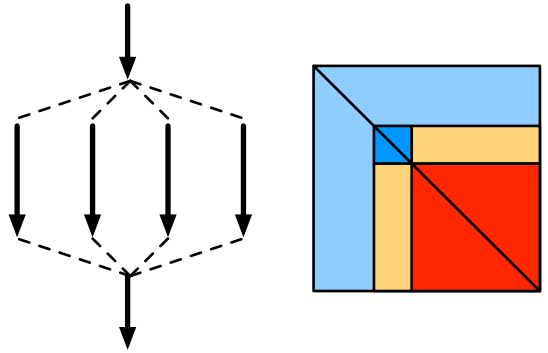
Review



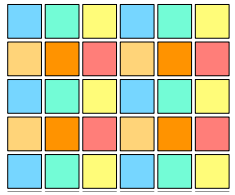
- **Motivation**
  - Work on data in fastest, smallest levels of memory hierarchy
  - Computation / communication speed ratio increasing
- **BLAS:**
  - Level 1: vector operations (dot product, norm, add vectors, ...)
  - Level 2: matrix-vector (multiply, rank update, triangular solve)
  - Level 3: matrix-matrix (multiply, rank-*k* update, triangular solve)



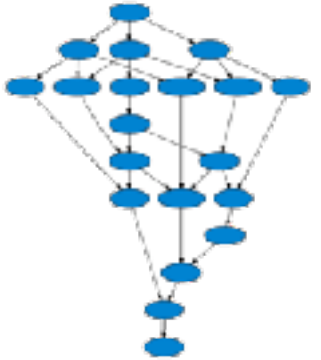
- **LAPACK**
  - Blocked operations (Level 3 BLAS)
  - Bulk-synchronous, fork-join parallelism via vendor BLAS



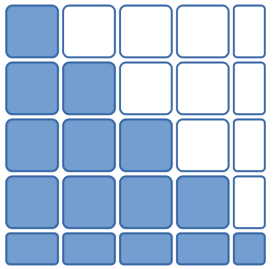
- **ScaLAPACK**
  - 2D block-cyclic distributed

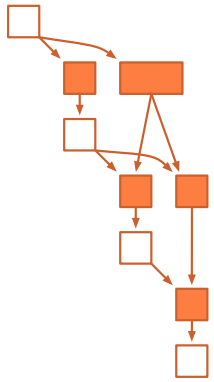


Review

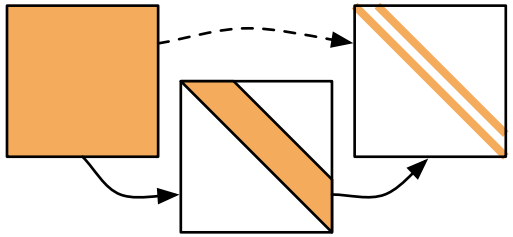


- **PLASMA and DPLASMA**
  - Task-based scheduling using directed acyclic graph (DAG)
  - Eliminates artificial synchronization overheads
- **SLATE**
  - Modern C++ replacement for ScaLAPACK
  - Map of tiles, global tile indexing
  - Matrix hierarchy





- **MAGMA**
  - Hybrid: panel on CPU, trailing matrix update (Level 3 BLAS) on GPU
  - Overlap CPU & GPU work, keep GPU busy all the time
- **SVD**
  - 2-stage reduction adds flops but uses Level 3 BLAS ⇒ faster

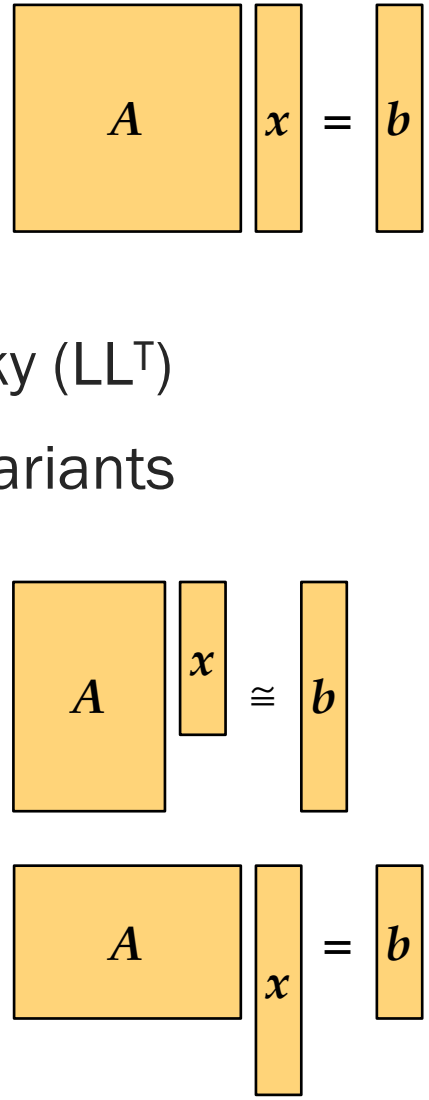


Outline

- Overview
- Linear algebra basics
- Matrix types
- Algorithms
  - Matrix multiply
  - Triangular solve
  - LU
  - Cholesky
  - LDL^T
  - QR
  - Eigenvalues / SVD

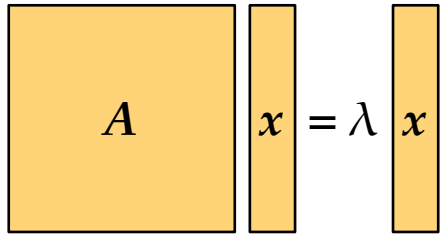
Linear algebra

- $Ax = b$
  - A is triangular: forward or back substitution (BLAS trsm)
  - A is general non-symmetric: LU, QR, …
  - A is Hermitian (symmetric) positive definite (all $\lambda > 0$): Cholesky (LL^T)
  - A is Hermitian (symmetric) indefinite ($\lambda < 0$ and $\lambda > 0$): LDL^T variants
- $Ax \cong b$
  - A is tall — least squares, minimize residual $\|b - Ax\|_2$
  - A is wide — find x with minimum norm satisfying $Ax = b$
  - Methods: QR/LQ, QR with pivoting, SVD

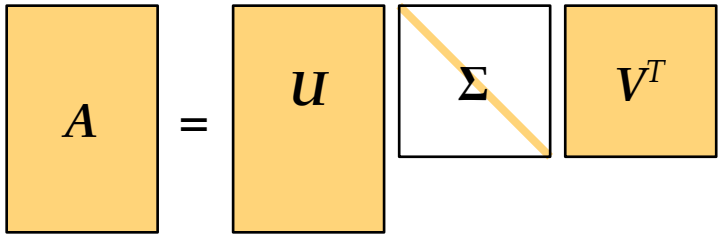


Linear algebra

- $Ax = \lambda x$ eigenvalue
  - $Ax = \lambda Mx$, etc. generalized eigenvalue
  - A is general non-symmetric: QR iteration
  - A is Hermitian (symmetric): QR iteration, divide & conquer, bisection, MRRR, Jacobi, QDWH, …



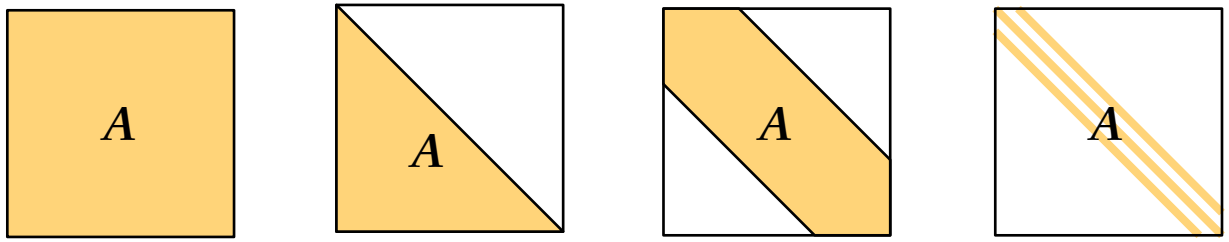
- $A = U\Sigma V^T$ singular value decomposition (SVD)
  - Methods: QR iteration, divide & conquer, bisection, MRRR, 1 & 2-sided Jacobi, QDWH, …



U, V orthogonal

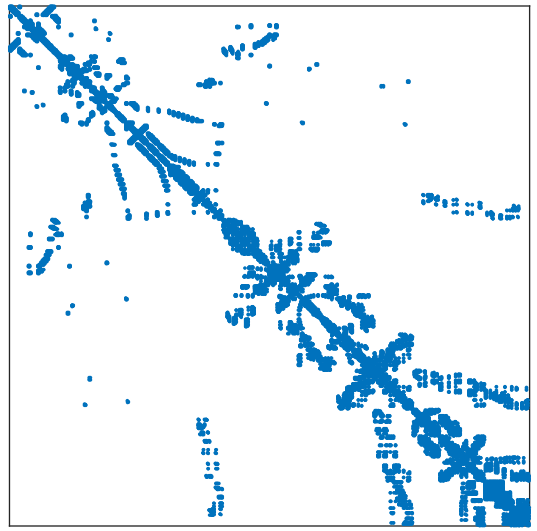
Dense vs. Sparse

- Sparse matrix has most entries are zero ($nnz \ll n^2$)
  - Sparse direct: SuperLU, CHOLMOD, …
  - Sparse iterative
    - Solve $Ax = b$: CG, GMRES, multigrid, …
    - Eigenvalue: Arnoldi (ARPACK), Lanczos, …
- Dense operates on $O(n^2)$ entries,
  or "sparsity" is highly structured: triangular or band

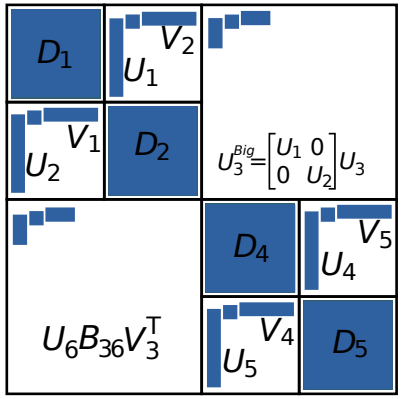


- Dense but low rank, numerically compressible
  - Hierarchical $\mathcal{H}$ matrices

SuiteSparse: bcsstk13



2003 x 2003
4 million entries
83,883 nonzeros (~ 42 per row)



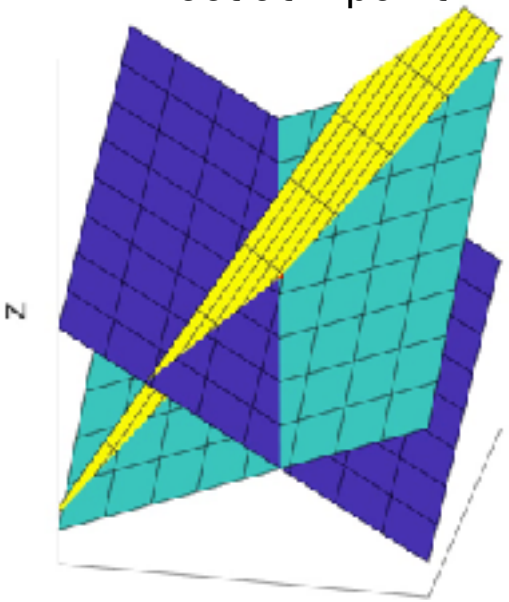
STRUMPACK
Hierarchically Semi-Separable (HSS)

Linear algebra basics

Non-singular square matrix

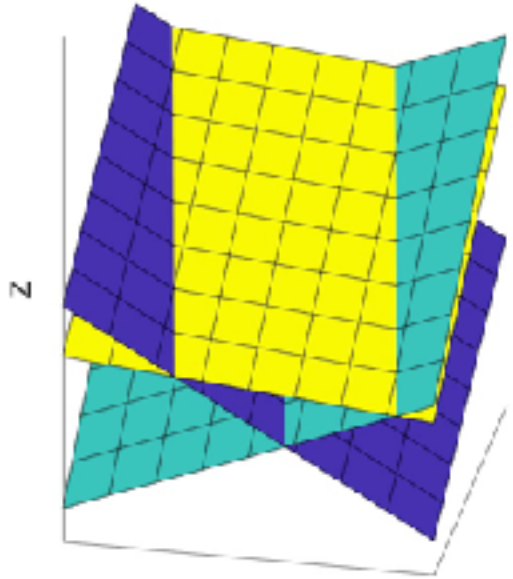
- **Non-singular**
  - $Ax = b$ has unique solution
- **Singular**
  - $Ax = b$ has either no solution or infinitely many solutions

unique solution:
meet at 1 point



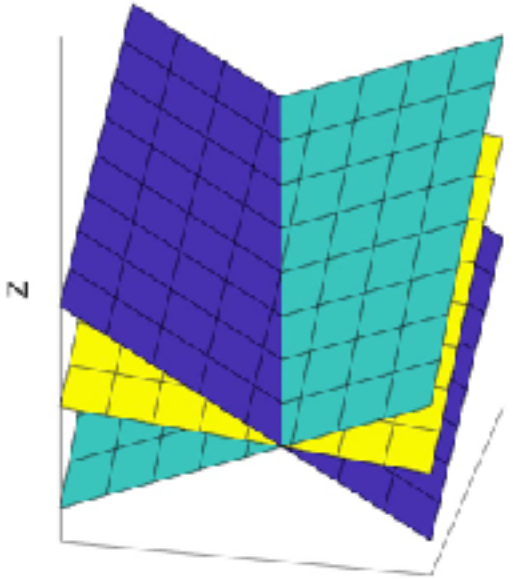
$$\begin{bmatrix} 2 & 3 & 5 \\ 3 & -2 & 5 \\ 2 & 5 & -3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -4 \\ -4 \\ 2 \end{bmatrix}$$

no solution:
meet in 3 parallel lines



$$\begin{bmatrix} 2 & 3 & 5 \\ 3 & -2 & 5 \\ -5 & -1 & -10 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -4 \\ -4 \\ 5 \end{bmatrix}$$

infinitely many solutions:
meet in 1 line



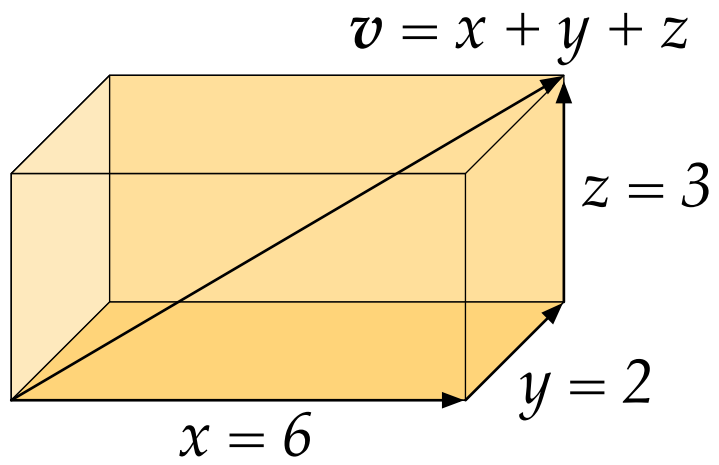
$$\begin{bmatrix} 2 & 3 & 5 \\ 3 & -2 & 5 \\ -5 & -1 & -10 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -4 \\ -4 \\ 8 \end{bmatrix}$$

Non-singular square matrix

- **Non-singular**
  - $Ax = b$ has unique solution
  - $Az = 0$ if and only if $z = 0$
  - All $\lambda \neq 0$
  - All $\sigma \neq 0$
  - $\text{rank}(A) = n$
  - A^{-1} exists,
    but usually don't compute it!
  - $\det(A) \neq 0$,
    but almost never compute it!
- **Singular**
  - $Ax = b$ has either no solution or infinitely many solutions
  - $Az = 0$ for some $z \neq 0$
    (null space of A)
  - Some $\lambda = 0$
  - Some $\sigma = 0$
  - $\text{rank}(A) < n$
  - A^{-1} doesn't exist
  - $\det(A) = 0$

Vector norms

- p-norms: $\|x\|_p = \left(\sum_i |x_i|^p\right)^{1/p}$
- 2-norm: $\|x\|_2 = \sqrt{\sum_i |x_i|^2}$
  - geometric distance
- 1-norm: $\|x\|_1 = \sum_i |x_i|$
  - USPS: length + width + height
- inf-norm: $\|x\|_\infty = \max_i(|x_i|)$

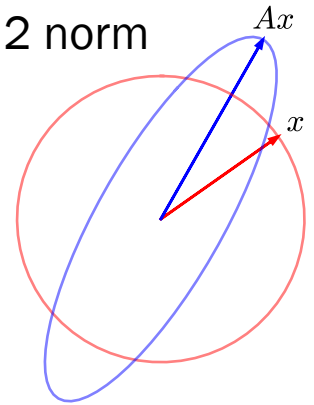


$$\|v\|_2 = \sqrt{|x|^2 + |y|^2 + |z|^2} = \sqrt{6^2 + 2^2 + 3^2} = 7$$
$$\|v\|_1 = |x| + |y| + |z| = 6 + 2 + 3 = 11$$
$$\|v\|_\infty = \max(|x|, |y|, |z|) = \max(6,\ 2,\ 3) = 6$$

Matrix norms

- Induced by vector norm:
  - $\|A\|_p = \max_{\|x\|_p=1} \|Ax\|_p$
    - Maximum stretching of a vector
  - Instances
    - $\|A\|_2 = \sigma_{\max}$ max singular value
    - $\|A\|_1 = \max_j \|A_{:,j}\|_1$ max column sum
    - $\|A\|_\infty = \max_i \|A^T_{i,:}\|_1$ max row sum
- Other norms
  - $\|A\|_{\text{fro}} = \sqrt{\sum_{i,j} |A_{i,j}|^2}$ Frobenius norm
  - $\|A\|_{\max} = \max_{i,j} |A_{i,j}|$

2 norm



$$A = \begin{bmatrix} 0.8 & 0.1 \\ 0.9 & 0.9 \end{bmatrix}$$

$\|A\|_2 = 1.44$

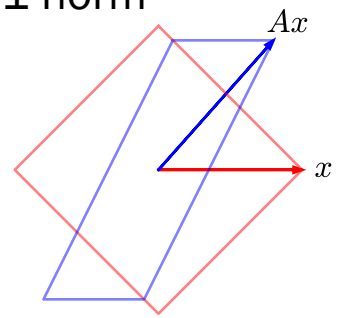
$\|A\|_1 = 1.7$

$\|A\|_\infty = 1.8$

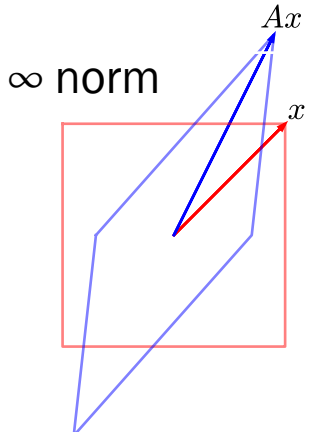
$\|A\|_{\text{fro}} = 1.51$

$\|A\|_{\max} = 0.9$

1 norm

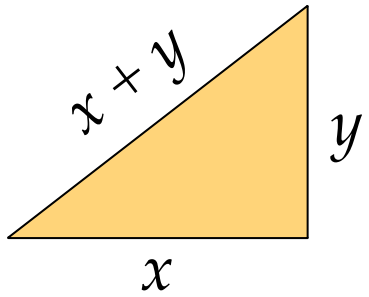


∞ norm



Norms

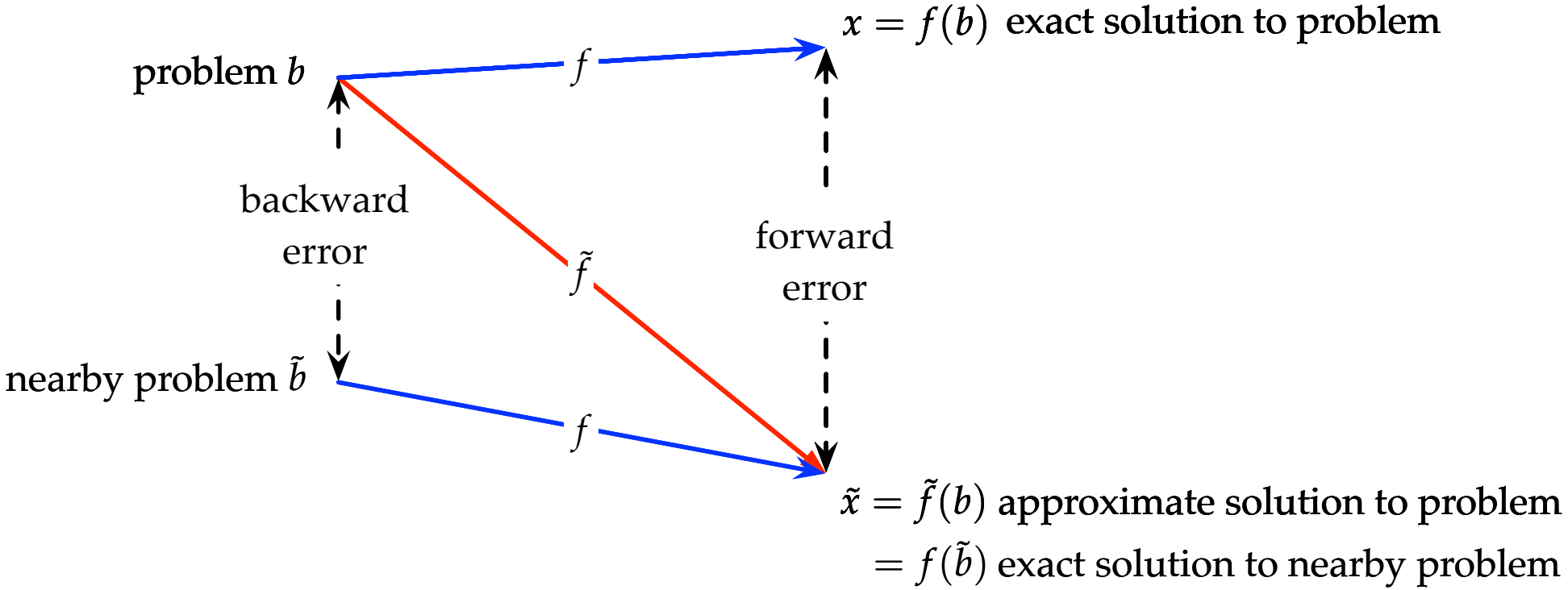
- Properties
  - $\|\boldsymbol{x}\| > 0$ if $\boldsymbol{x} \neq \boldsymbol{0}$
  - $\|\gamma \boldsymbol{x}\| = |\gamma| \cdot \|\boldsymbol{x}\|$ for any scalar γ
  - $\|\boldsymbol{x} + \boldsymbol{y}\| \leq \|\boldsymbol{x}\| + \|\boldsymbol{y}\|$ triangle inequality



- Some matrix norms also satisfy
  - $\|\boldsymbol{AB}\| \leq \|\boldsymbol{A}\| \cdot \|\boldsymbol{B}\|$ submultiplicative
  - Induced norms, Frobenius norm
  - Not max norm

Error

- Solve $Ax = b$ as $x = A^{-1}\, b = f(\, b\,)$
- Forward error is unknown
- Instead, ask did we **exactly** solve a **nearby** problem?
- Backward error analysis ascribes all error to input



Error

- Forward error is error in solution x. Usually not known!

$$\frac{\|\Delta x\|}{\|x\|} = \frac{\|x - \tilde{x}\|}{\|x\|}$$

  - x is exact solution, $\tilde{x}$ is computed solution
- Instead, ask did we **exactly** solve a **nearby** problem?
- Backward error is error in input b to account for error in solution

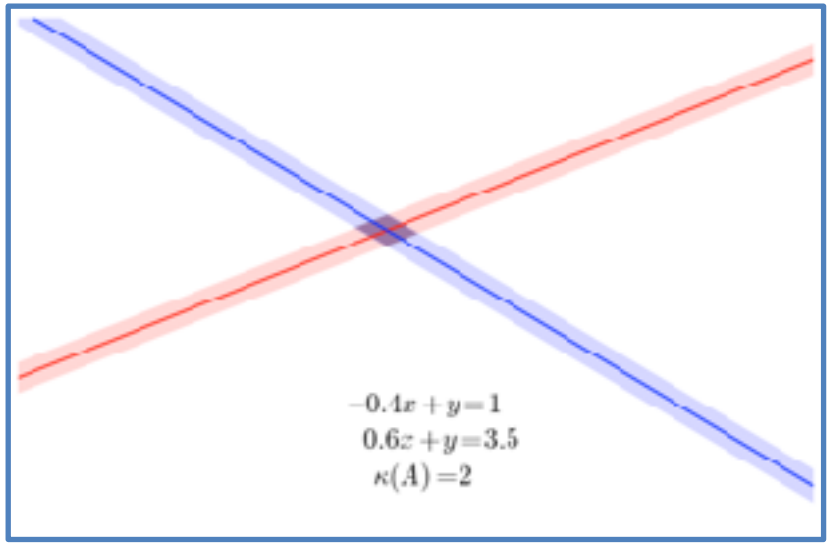
$$\frac{\|\Delta b\|}{\|b\|} = \frac{\|b - \tilde{b}\|}{\|b\|}$$

  - b is exact input, $\tilde{b}$ is nearby problem that we actually solved

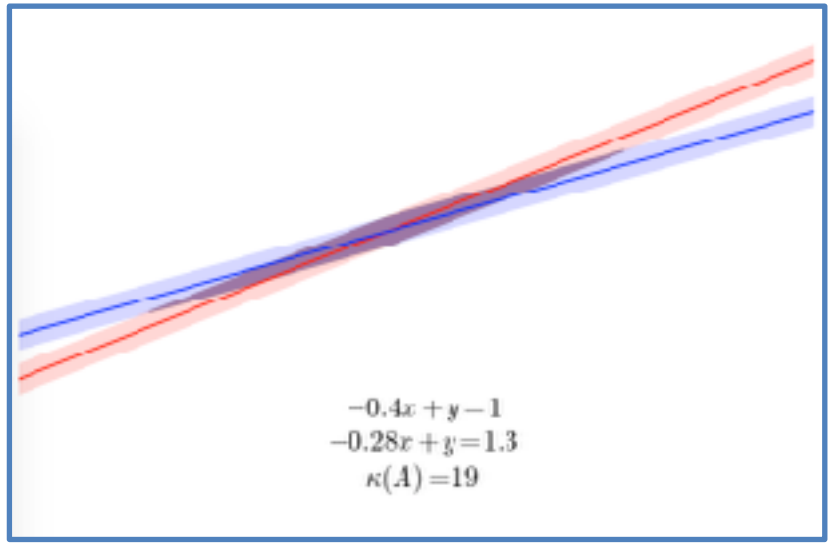
Condition number

- Property of your problem (matrix or function)
- Measures sensitivity of problem's solution to changes in input
  - If adjusting the faucet a tiny amount changes water from cold to hot, it is ***sensitive*** or ***ill-conditioned***. If adjusting it a large amount causes little temperature change, it is ***insensitive*** or ***well-conditioned***.

intersection is well-conditioned



intersection is ill-conditioned



Condition number

- For solving $Ax = b$,

$$\text{cond}(A) = \|A\| \cdot \|A^{-1}\| = \frac{\sigma_{\max}}{\sigma_{\min}} \geq 1$$

- Condition number is **amplification factor** of backward error (in input) to forward error (in output)

$$|\text{relative forward error}| \leq \text{cond} \cdot |\text{relative backward error}|$$

$$\frac{\|\Delta x\|}{\|x\|} \leq \text{cond}(A) \frac{\|\Delta b\|}{\|b\|}$$

  - Lose $\log_{10}(\text{cond}(A))$ digits of accuracy
- Expensive to compute via SVD, but can be estimated via LU, Cholesky, …

Stability

- Property of the algorithm (LU, QR, Newton, etc.)
- Algorithm is stable if result is relatively insensitive to perturbations ***during*** computation
- Algorithm is stable if backward error is small, i.e., we ***exactly*** solved a ***nearby*** problem
  - However, if problem is ill-conditioned, result can be inaccurate

Stable algorithm + well-conditioned problem = accurate result

- Large backward error (large residual $r = b - Ax$) implies an unstable algorithm

Floating point

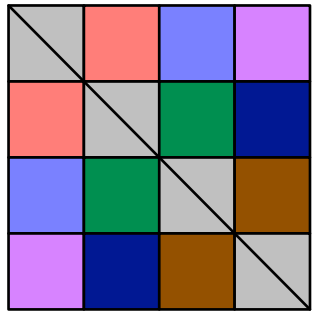
	bits	precision		epsilon (2 × unit roundoff)	underflow (min normal)	overflow (max)
bfloat16	16	8 bits	≈ 2 digits	7.81×10^{-3}	1.18×10^{-38}	3.40×10^{38}
half	16	11 bits	≈ 3 digits	9.77×10^{-4}	6.10×10^{-5}	65504
single (float)	**32**	**24 bits**	**≈ 7 digits**	$\mathbf{1.19 \times 10^{-7}}$	$\mathbf{1.18 \times 10^{-38}}$	$\mathbf{3.40 \times 10^{38}}$
double	**64**	**53 bits**	**≈ 16 digits**	$\mathbf{2.22 \times 10^{-16}}$	$\mathbf{2.23 \times 10^{-308}}$	$\mathbf{1.79 \times 10^{308}}$
double-double	64x2	107 bits	≈ 32 digits	1.23×10^{-32}	2.23×10^{-308}	1.79×10^{308}
quad	128	113 bits	≈ 34 digits	1.93×10^{-34}	3.36×10^{-4932}	1.19×10^{4932}

- Lose $\log_{10}(\text{cond}(A))$ digits of accuracy
- NVIDIA GPU implements half
- Google TPU implements bfloat16
- QD library implements double-double in software

Matrix Types

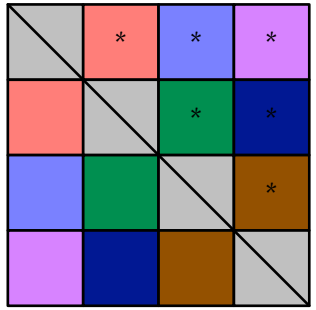
Matrix types: symmetric / Hermitian

- **Symmetric**
  - $A = A^T$ (transpose)
  - Entries $A_{ij} = A_{ji}$
  - λ are real, eigenvectors are orthogonal (if real-symmetric)



$$A = \begin{bmatrix} 3 & 1 & -2 \\ 1 & 4 & 2 \\ -2 & 2 & 6 \end{bmatrix}$$

- **Hermitian**
  - $A = A^H$ (conjugate-transpose)
  - Entries $A_{ij} = \text{conj}(\ A_{ji}\ )$
  - Implies diagonal is real
  - λ are real (even if A is complex), eigenvectors are unitary

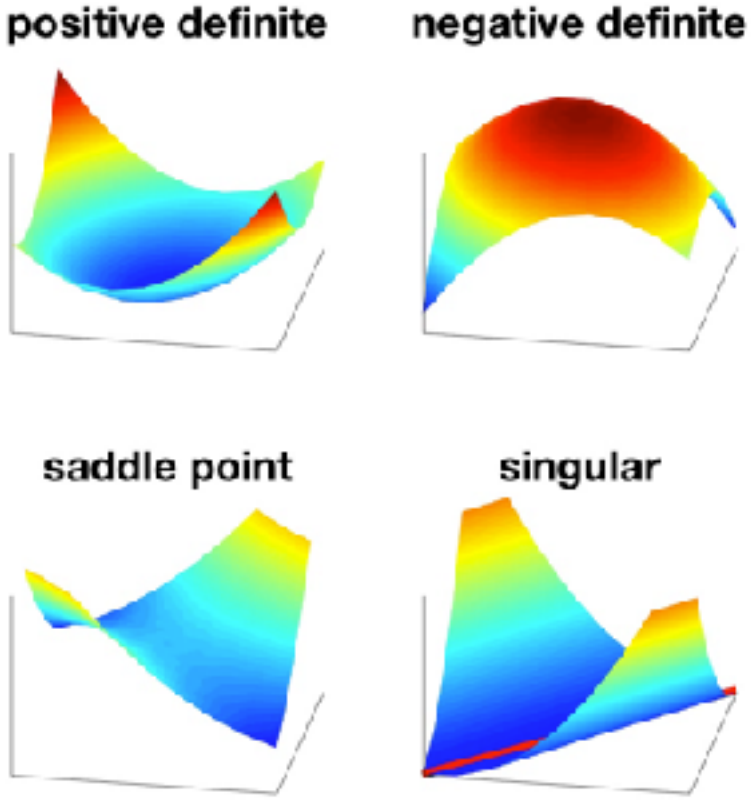


$$A = \begin{bmatrix} 3 & 1+2i & -2-3i \\ 1-2i & 4 & 2+4i \\ -2+3i & 2-4i & 6 \end{bmatrix}$$

Matrix types: symmetric / Hermitian

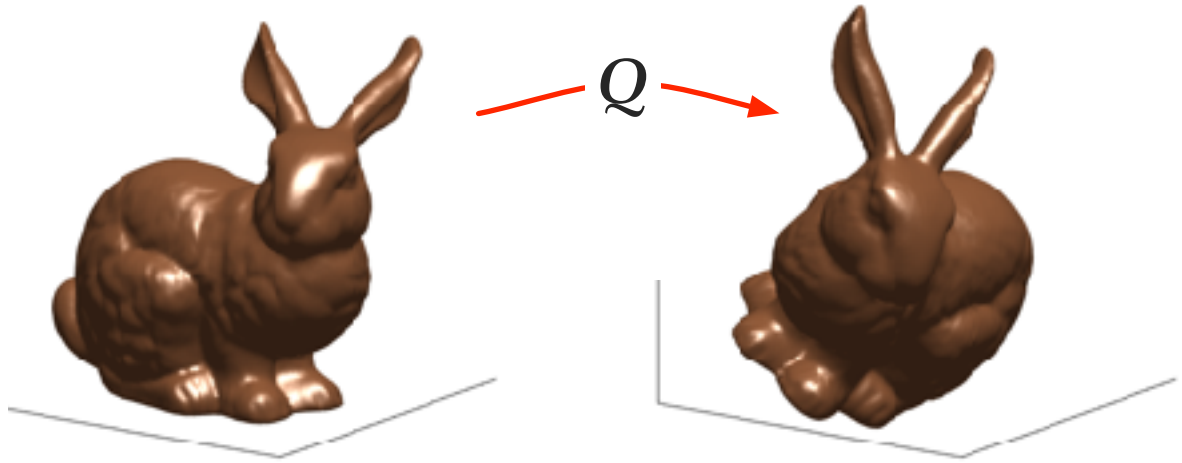
- **Symmetric (Hermitian) positive definite (SPD/HPD)**
  - all $\lambda > 0$
  - $\boldsymbol{x}^T \boldsymbol{A} \boldsymbol{x} > 0$ for all $\boldsymbol{x} \neq \boldsymbol{0}$
  - Cholesky factorization exists (if and only if)
- In optimization, Jacobian $\boldsymbol{J}$ of $f(\boldsymbol{x})$ being:
  - positive definite (all $\lambda > 0$), $f(\boldsymbol{x})$ is concave up
  - negative definite (all $\lambda < 0$), $f(\boldsymbol{x})$ is concave down
  - indefinite ($\lambda < 0$ and $\lambda > 0$), $f(\boldsymbol{x})$ is saddle point
  - singular (some $\lambda = 0$), $f(\boldsymbol{x})$ is flat in some direction

$$\boldsymbol{J} = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix}$$



Matrix types: orthogonal / unitary

- **Orthogonal (unitary if complex)**
  - Rotation, reflection, or combination



  - Columns are orthogonal to each other, and unit length

$$Q_{:,i}^T Q_{:,j} = \begin{cases} 0 \text{ if } i \neq j \\ 1 \text{ if } i = j \end{cases} \quad \text{or} \quad Q^T Q = I \;\; (Q^H Q = I \;\; \text{if complex})$$

- Easily inverted: $Q^{-1} = Q^T$ ($Q^{-1} = Q^H$ if complex)
- Matrix multiplication preserves length: $\|Qx\|_2 = \|x\|_2$
- Perfectly conditioned: $\text{cond}(Q) = 1$

Matrix types: orthogonal / unitary

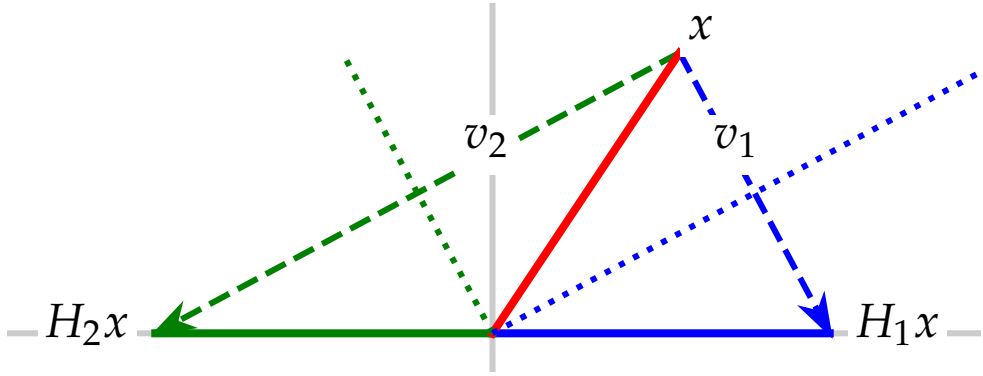
- **Givens rotation**
  - Rotation with θ chosen to eliminate x_2

$$Q = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}, \qquad Q \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \alpha \\ 0 \end{bmatrix}$$

- **Householder reflection**
  - Reflect vector x onto axis

$$H = I - \tau v v^T,$$
$$v = x \pm \|x\|_2 e_1,$$
$$\tau = \frac{2}{v^T v}$$

$$H \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} \alpha \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

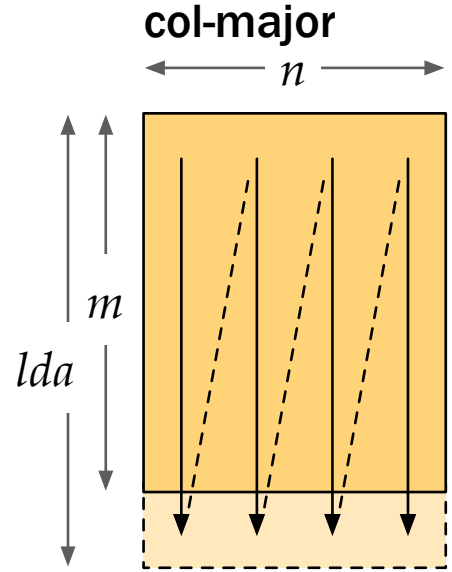


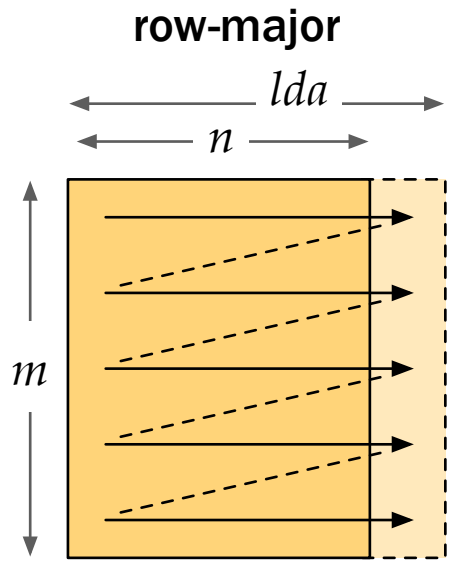
  - Block reflector

$$H = H_k \cdots H_1 = I - V T V^T$$

Column-major vs. row-major

- Col-major: A(i, j) = A[i + j*lda], where lda ≥ m
- Row-major: A(i, j) = A[i*lda + j], where lda ≥ n
  - Leading dimension lda
  - lda being multiple of cache line (e.g., 64 bytes) or GPU warp size (e.g., 32) is helpful (and **not** multiple of the page size, e.g., 4 KiB)
- Fortran uses col-major, hence BLAS and LAPACK do
- C/C++ doesn't have dynamically allocatable 2D array type
  - You can define matrix as either col-major or row-major
  - From a math perspective, I suggest col-major: we work with columns more than rows
  - For its limited statically sized 2D arrays, C uses row-major e.g., A[4][4] is row-major





Column-major vs. row-major

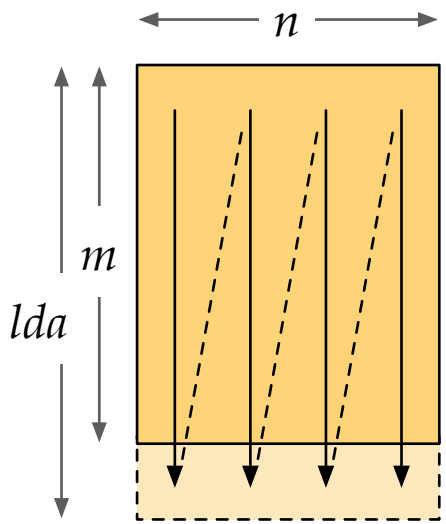
```
// Typical loop order for col-major
// access in C. lda >= m.
for (j = 0; j < n; ++j)
    for (i = 0; i < m; ++i)
        A[ i + j*lda ] = ...
```

```
// Typical loop order for row-major
// access in C. lda >= n.
for (i = 0; i < m; ++i)
    for (j = 0; j < n; ++j)
       A[ i*lda + j ] = ...
```

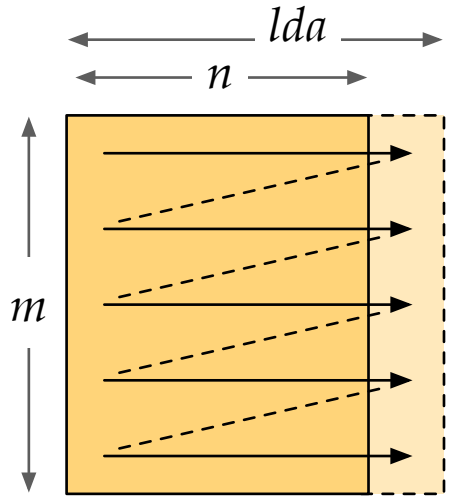
```
// can use macro in C,
// or operator overloading in C++
#define A(i_, j_)  A[ (i_) + (j_)*lda ]

    A(i, j) = ...

#undef A
```

```
// Typical loop order for col-major
// access in Fortran. lda >= m
do j = 1, n
    do i = 1, m
        A(i,j) = ...
    continue
continue
```

col-major



row-major



Algorithms

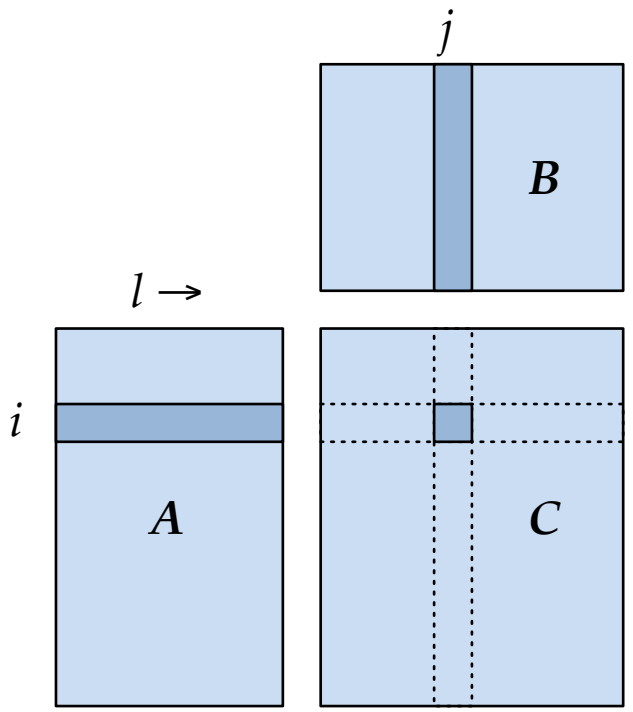
Matrix multiply (gemm)

- $C = \alpha AB + \beta C$
  A is $m \times k$, B is $k \times n$, C is $m \times n$
- Inner products of rows of A and columns of B

$$C_{i,j} = A_{i,:} B_{:,j} = \sum_{l=1}^{k} A_{i,l} B_{l,j}$$

- Cost: $2mnk$ floating point operations (flops), counting multiplies and adds, ignoring lower order terms

```
// basic 3-loop inner-product version
for j = 0 to n-1
    for i = 0 to m-1
        tmp = 0
        for l = 0 to k-1
            tmp += A( i, l ) * B( l, j )
        C( i, j ) = alpha*tmp + beta*C( i, j )
```



For flop counts, see
LAPACK Working Note (LAWN) 41: Installation Guide for LAPACK

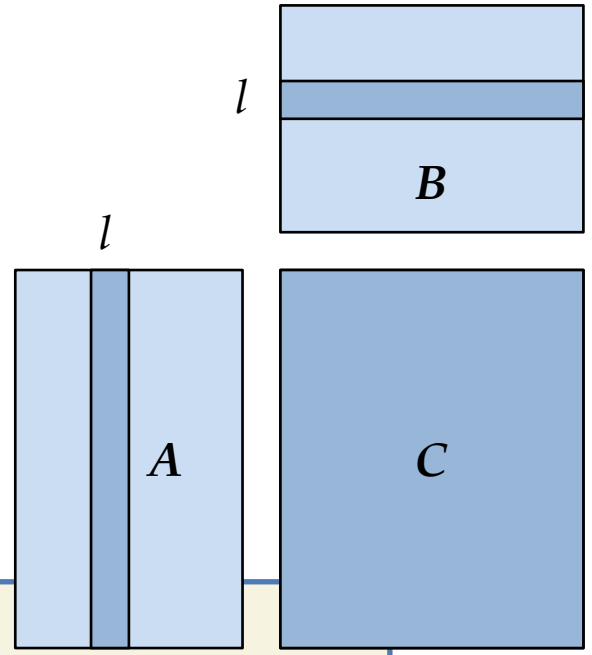
Matrix multiply (gemm)

- $C = \alpha AB + \beta C$

  A is $m \times k$, B is $k \times n$, C is $m \times n$

- Sum of outer products of columns of A and rows of B

$$C = \sum_{l=1}^{k} A_{:,l} B_{l,:}$$

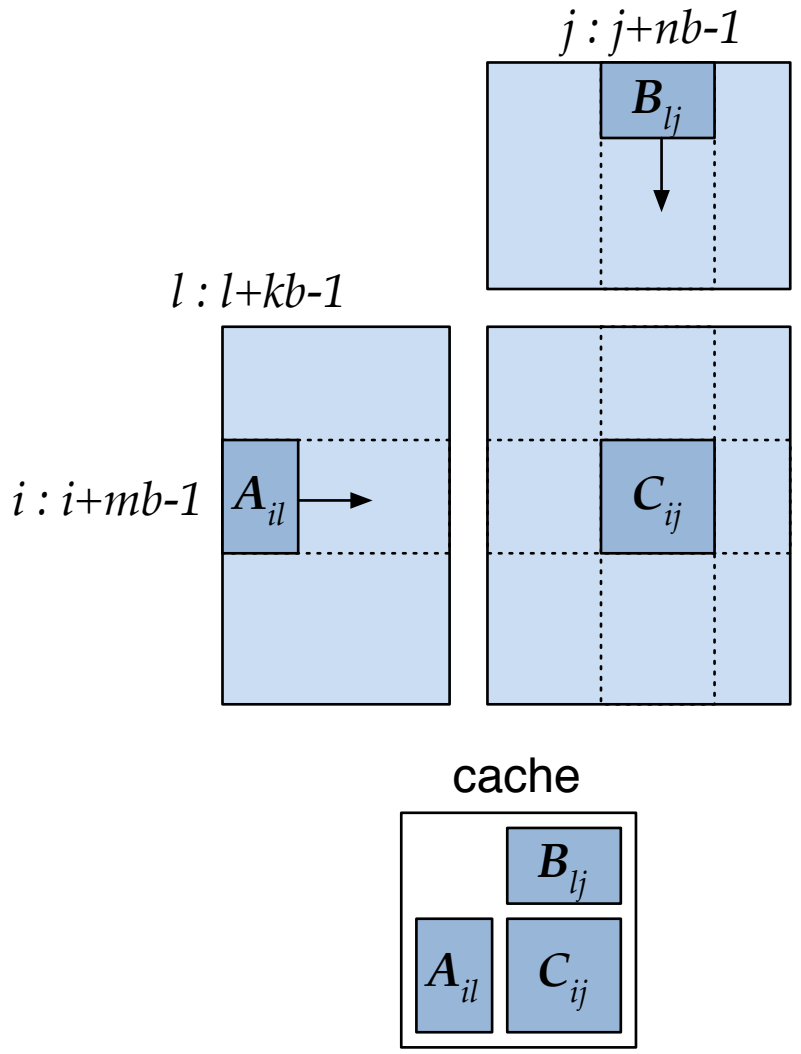



```
// basic 3-loop outer-product version
beta_ = beta
for l = 0 to k-1
    for j = 0 to n-1
        for i = 0 to m-1
            C( i, j ) = alpha*A( i, l )*B( l, j ) + beta_*C( i, j )
    beta_ = 1
```

Matrix multiply (gemm)

- Block at multiple levels for vector instructions, registers, and caches (L1, L2, L3)

```
// simple, single level blocked version
// block sizes mb, nb, kb
for j = 0 to n-1 by nb
    jb = min( nb, n - j )
    for i = 0 to m-1 by mb
        ib = min( mb, m - i )
        tmp = zeros( ib, jb )
        for l = 0 to k-1 by kb
            kb = min( nb, k - l )
            tmp += A( i : i+ib-1, l : l+kb-1 )
                 * B( l : l+kb-1, j : j+jb-1 )

        C( i : i+ib-1, j : j+jb-1 )
            = alpha*tmp
            + beta*C( i : i+ib-1, j : j+jb-1 )
```

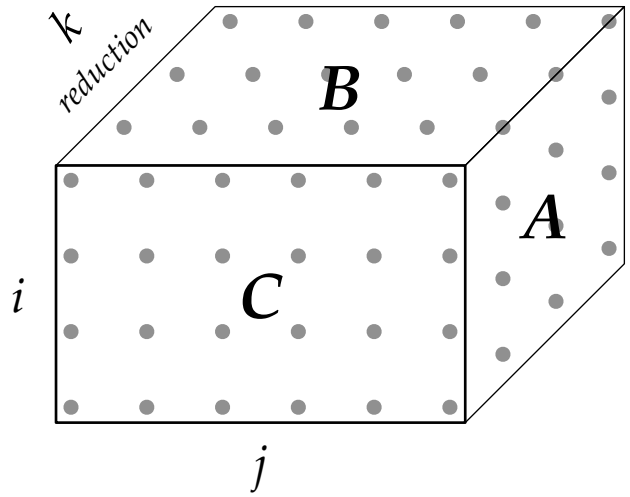


Parallelization strategy

- Split into fine-grained tasks
- Group fine-grained tasks into larger coarse-grained tasks

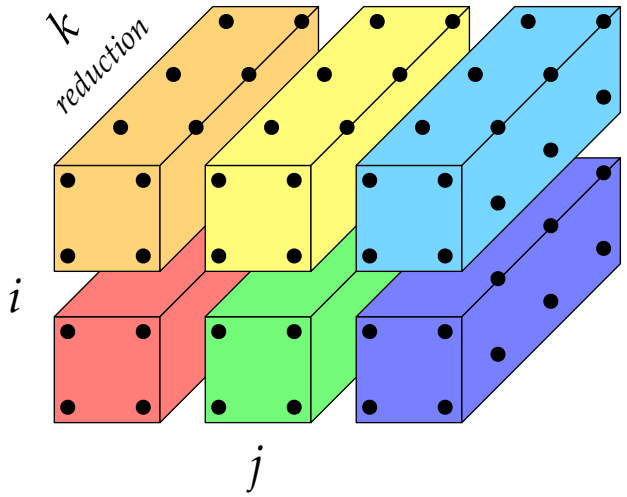
Each dot represents one multiply, $A_{il}B_{lj}$

C is 4×6, A is 4×4, B is 4×6



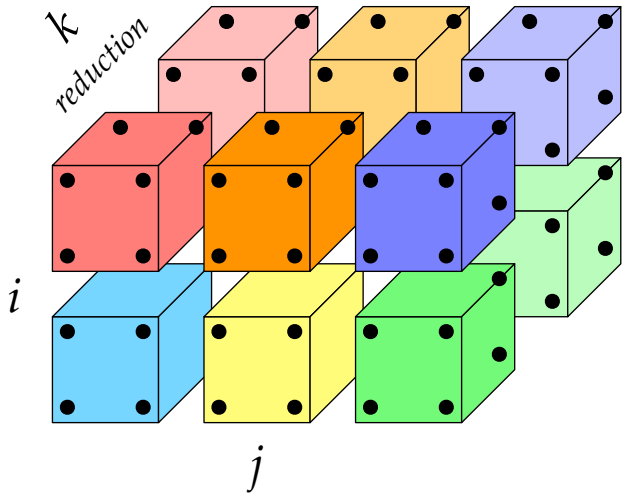
2D grouping

2 × 2 × k blocks



3D grouping

2 × 2 × 2 blocks



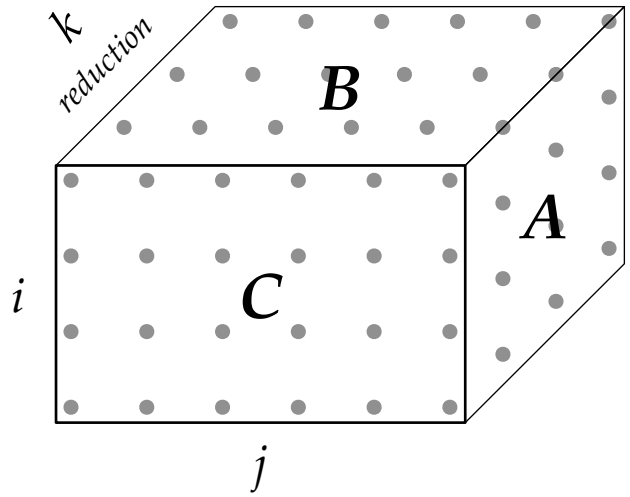
ICL THE UNIVERSITY OF TENNESSEE KNOXVILLE

Parallelization strategy

- Split into fine-grained tasks
- Group fine-grained tasks into larger coarse-grained tasks

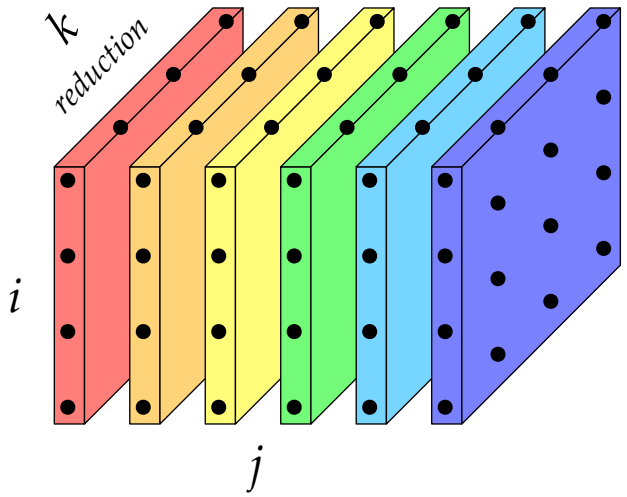
Each dot represents one multiply, $A_{il}B_{lj}$

C is 4×6, A is 4×4, B is 4×6



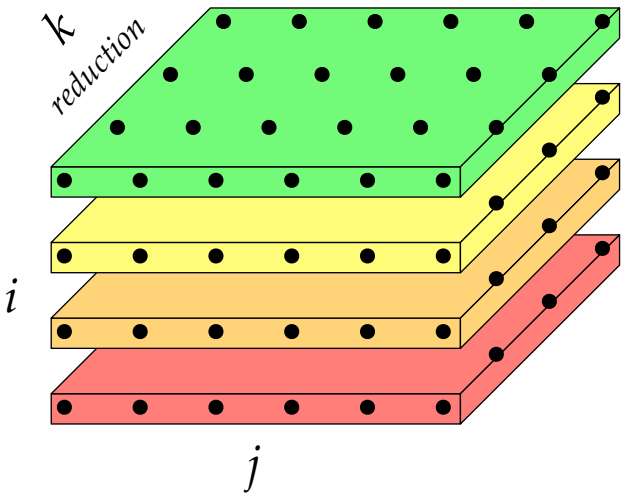
1D column grouping

$m \times 1 \times k$ blocks



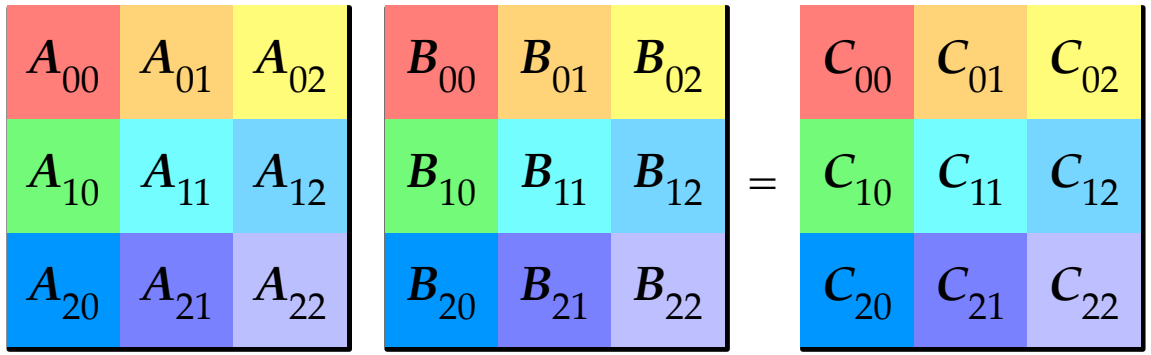
1D row grouping

$1 \times n \times k$ blocks



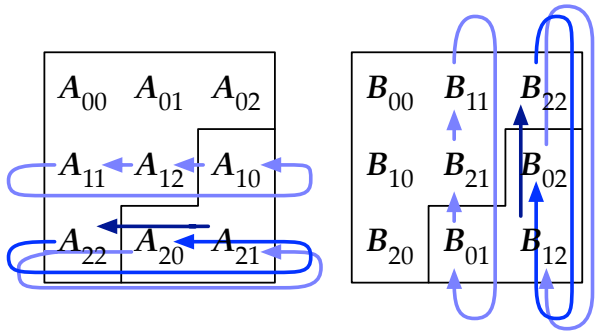
Parallel matrix multiply

- **Cannon's Algorithm (1969)**
  - Square matrices A, B, C
  - Simple block distribution
  - Cycles A and B through nodes
  - Would need post-processing step to return A and B to original nodes



Parallel matrix multiply

```
// A, B, C are N x N blocks
// setup step
for k = 0 to N-1
    shift row A(:,k) left k cols, mod N
    shift col B(k,:) up   k rows, mod N
```




```
// multiplication step
for k = 0 to N-1
   parallel for each i, j
      // by current position of blocks!
      C(i, j) += A(i, j) * B(i, j)
   shift blocks of A right 1 col, mod N
   shift blocks of B down  1 row, mod N
```

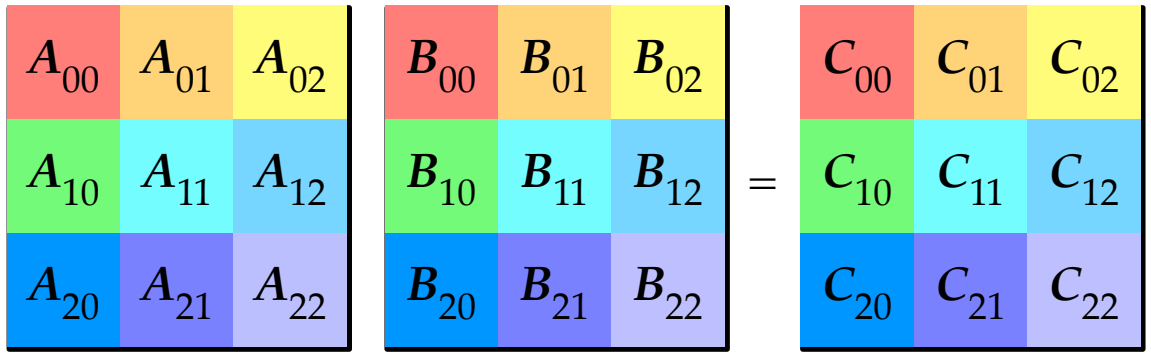
$$\begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{11} & A_{12} & A_{10} \\ A_{22} & A_{20} & A_{21} \end{bmatrix} .* \begin{bmatrix} B_{00} & B_{11} & B_{22} \\ B_{10} & B_{21} & B_{02} \\ B_{20} & B_{01} & B_{12} \end{bmatrix} = \begin{bmatrix} A_{00}B_{00} & A_{01}B_{11} & A_{02}B_{22} \\ A_{11}B_{10} & A_{12}B_{21} & A_{10}B_{02} \\ A_{22}B_{20} & A_{20}B_{01} & A_{21}B_{12} \end{bmatrix} +$$

$$\begin{bmatrix} A_{02} & A_{00} & A_{01} \\ A_{10} & A_{11} & A_{12} \\ A_{21} & A_{22} & A_{20} \end{bmatrix} .* \begin{bmatrix} B_{20} & B_{01} & B_{12} \\ B_{00} & B_{11} & B_{22} \\ B_{10} & B_{21} & B_{02} \end{bmatrix} = \begin{bmatrix} A_{02}B_{20} & A_{00}B_{01} & A_{01}B_{12} \\ A_{10}B_{00} & A_{11}B_{11} & A_{12}B_{22} \\ A_{21}B_{10} & A_{22}B_{21} & A_{20}B_{02} \end{bmatrix} +$$

$$\begin{bmatrix} A_{01} & A_{02} & A_{00} \\ A_{12} & A_{10} & A_{11} \\ A_{20} & A_{21} & A_{22} \end{bmatrix} .* \begin{bmatrix} B_{10} & B_{21} & B_{02} \\ B_{20} & B_{01} & B_{12} \\ B_{00} & B_{11} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{01}B_{10} & A_{02}B_{21} & A_{00}B_{02} \\ A_{12}B_{20} & A_{10}B_{01} & A_{11}B_{12} \\ A_{20}B_{00} & A_{21}B_{11} & A_{22}B_{22} \end{bmatrix}$$

Parallel matrix multiply

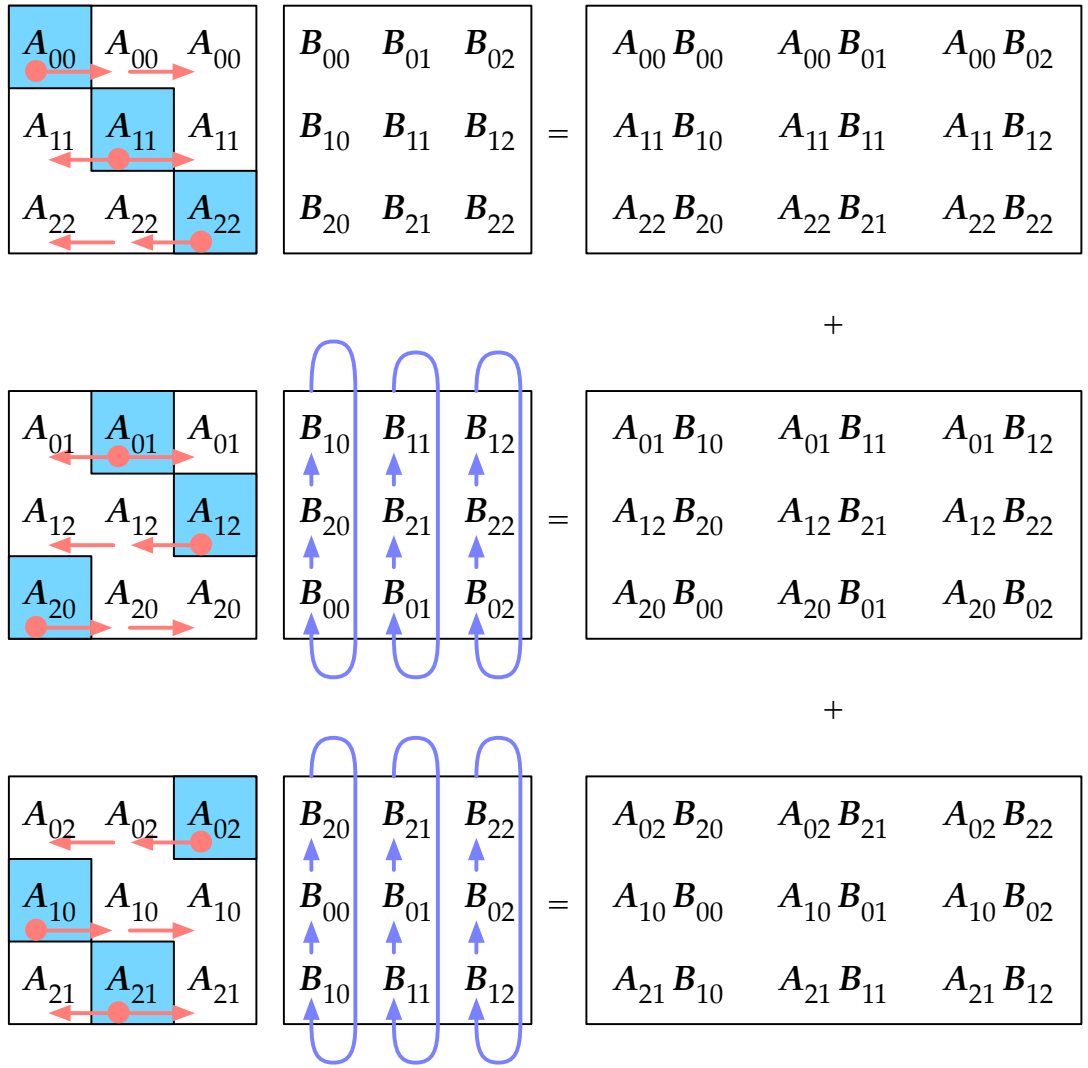
- **Fox's Algorithm (1987)**
  - Square matrices A, B, C
  - Simple block distribution
  - Broadcasts blocks of A
  - Cycles B through nodes
  - No setup phase
  - After N steps, B is returned to its original nodes



Parallel matrix multiply

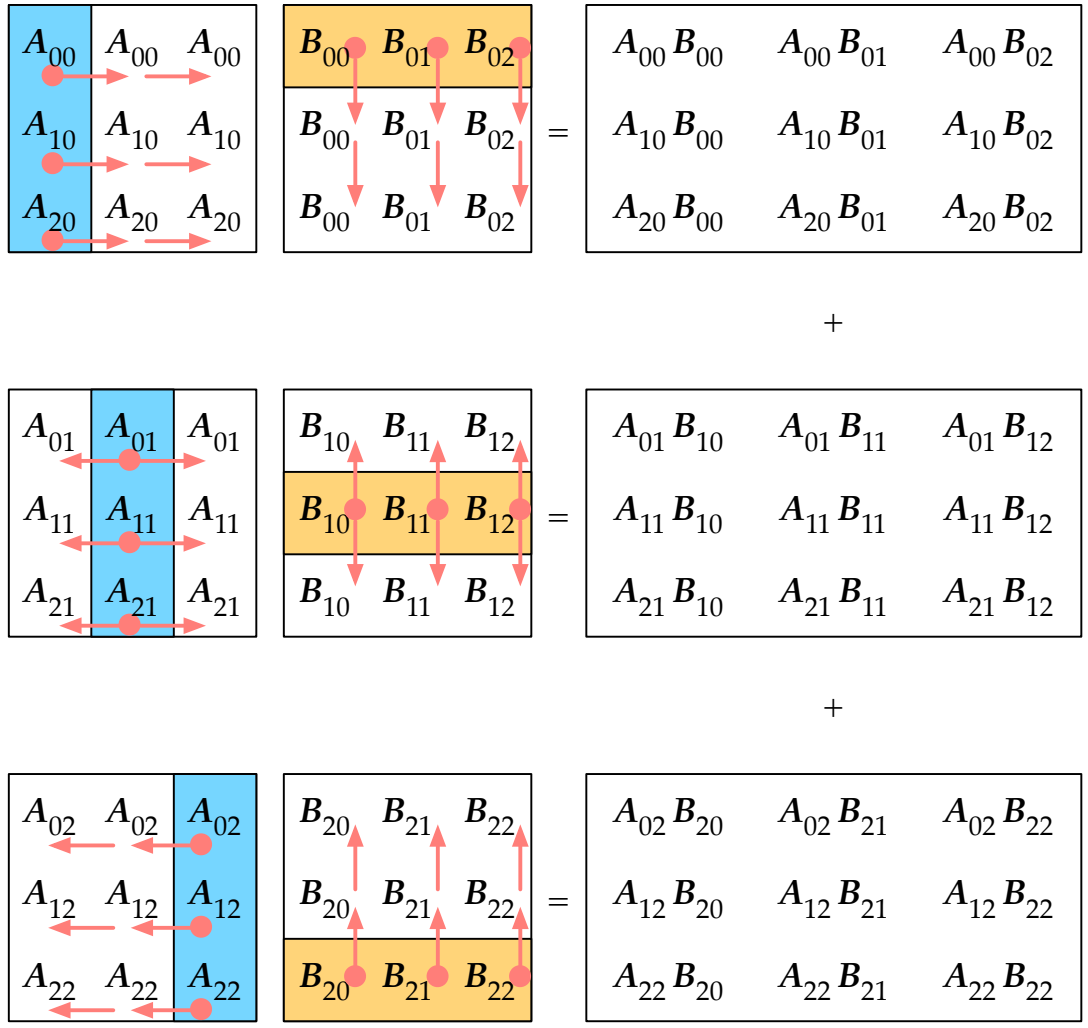
- Fox's Algorithm (1987)

```
// A, B, C are N x N blocks
for k = 0 to N-1
    for i = 0 to N-1
        Bcast A(i, (i+k) % n) to row i
    parallel for all i, j
        // by current position of blocks!
        C(i, j) += A(i, j) * B(i, j)
    shift blocks of B up 1 row, mod N
```



Parallel matrix multiply

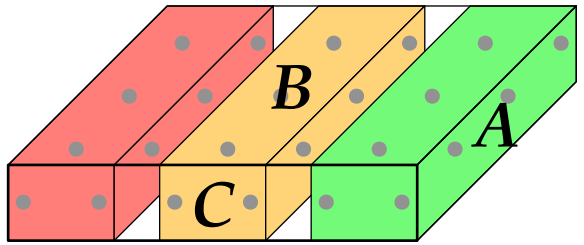
- **SUMMA (Agarwal et al 1994; van de Geijn et al 1995; PBLAS 1995)**
  - Series of block outer products
  - Broadcasts block-cols of A
  - Broadcasts block-rows of B
  - Arbitrary dimensions
  - Arbitrary distribution

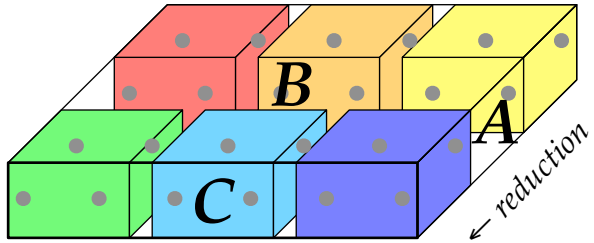


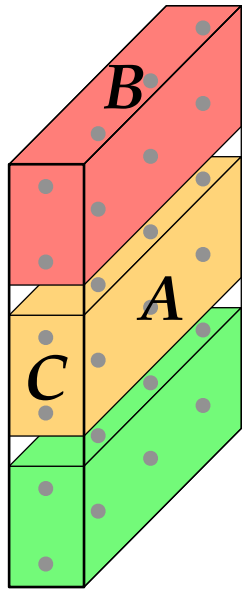

```
// C is M x N, A is M x K blocks,
// B is K x N blocks
for k = 0 to K-1
    for i = 0 to M-1
        Bcast A(i, k) to row i
    for j = 0 to N-1
        Bcast B(k, j) to col j
    parallel for all i, j
        // based on global indices!
        C(i, j) += A(i, k) * B(k, j)
```

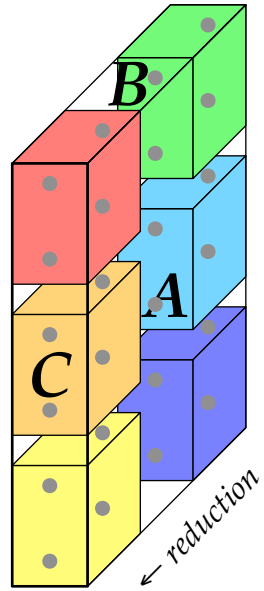
Parallel matrix multiply

- What if C is small — one block row or one block column?
  - Then SUMMA has limited parallelism
  - If A is large: send B to A, multiply, parallel reduce to C
  - If B is large: send A to B, multiply, parallel reduce to C









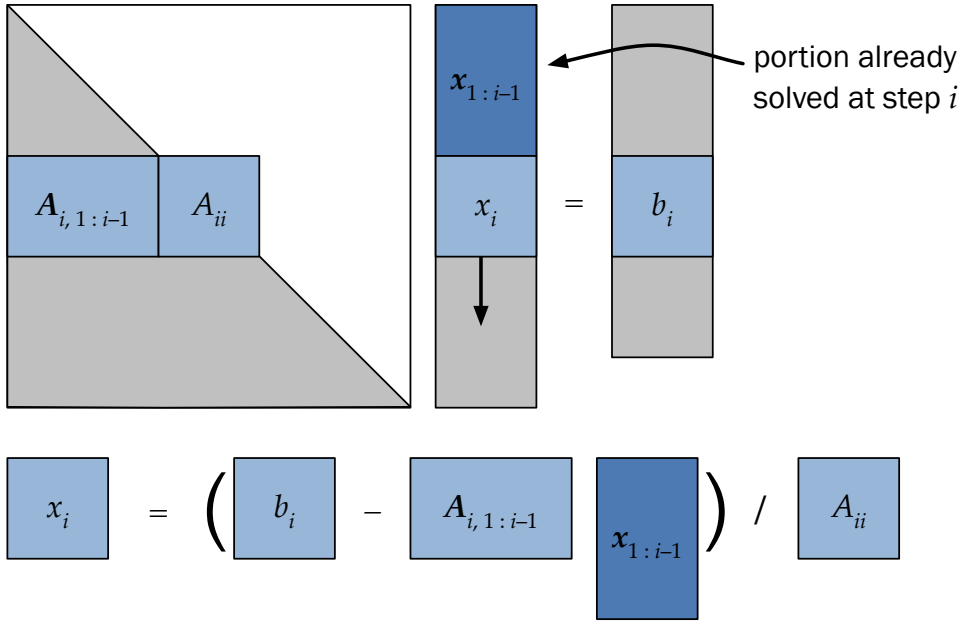
Parallel matrix multiply

- Communication bounds
  - 2D gemm (Cannon, Fox, SUMMA) are asymptotically communication optimal if memory is not replicated
  - 3D gemm communicates less, but replicates the matrices
  - Solomonik and Demmel (2011) proposed 2.5D gemm
    - Replicates memory to achieve lower bounds

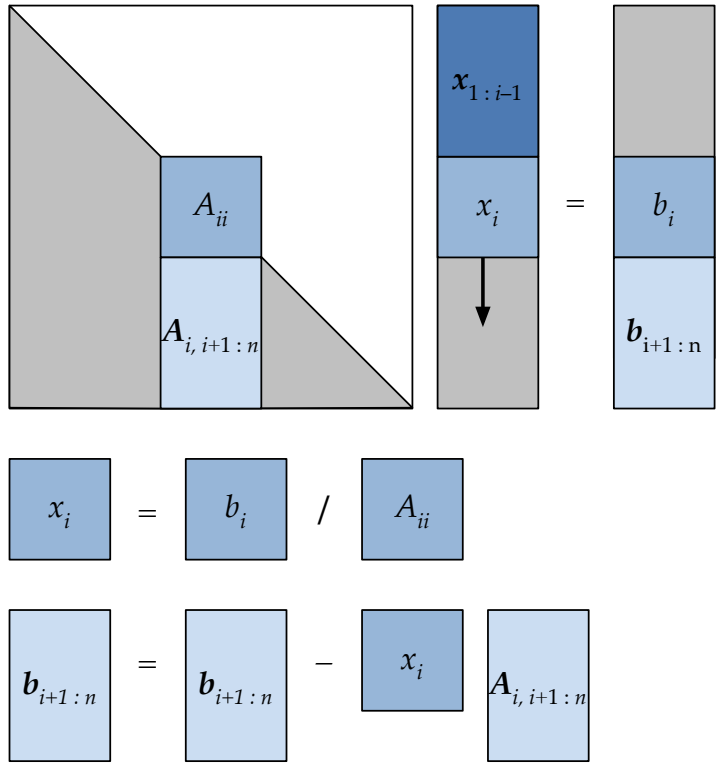
Triangular solve (trsv/trsm)

- $Ax = b$ is easy to solve for triangular matrix A
- Forward / back substitution for lower / upper triangular

dot product ($x^T y$) implementation



axpy ($y = \alpha x + y$) implementation

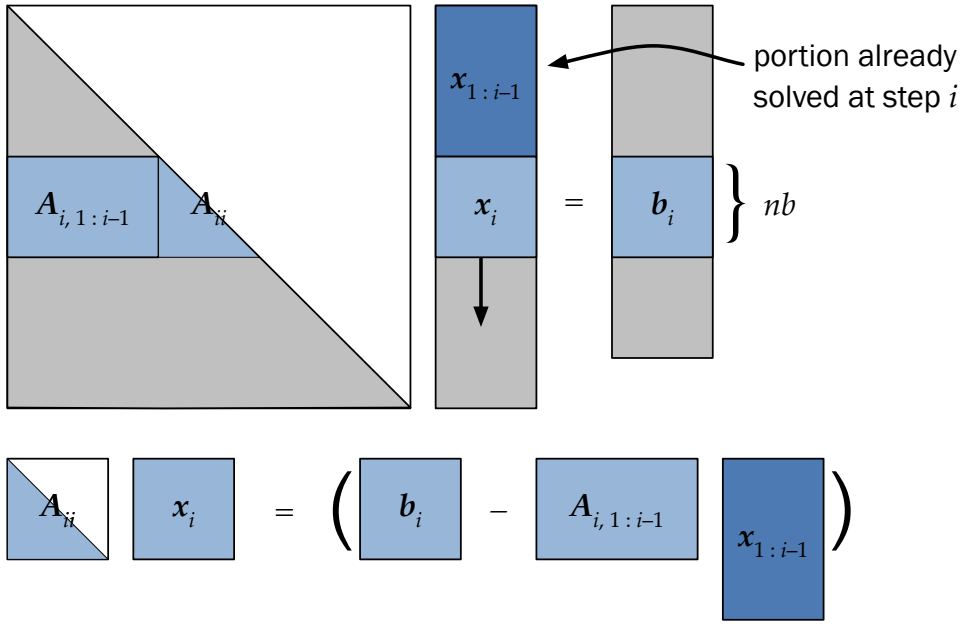


- Cost: n^2 flops

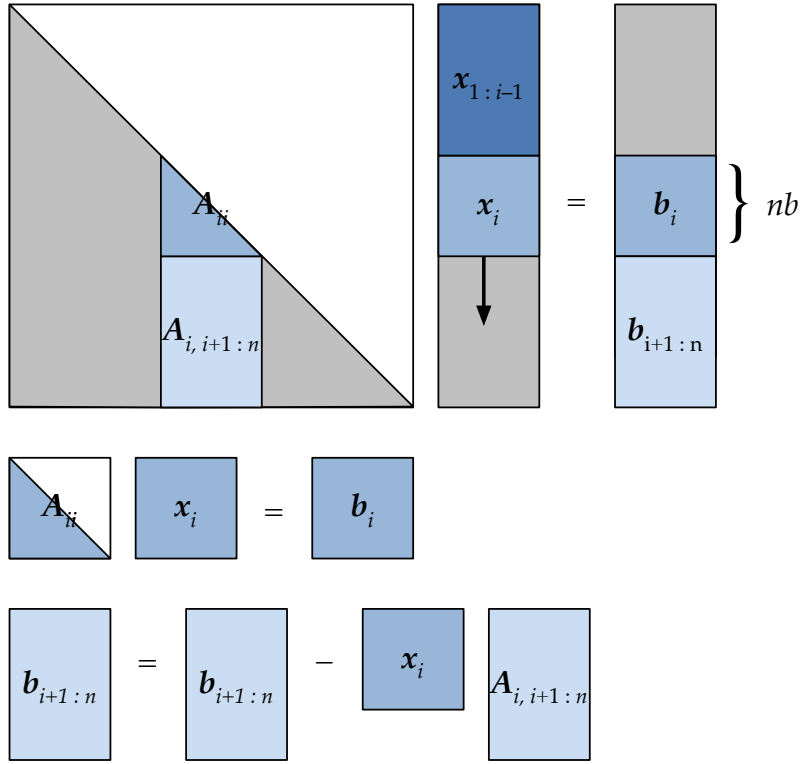
Triangular solve (trsv/trsm)

- Block implementation for higher performance
  - Replace division by A_{ii} with block triangular solve (trsv/trsm)

"block dot product" ($x^T y$) implementation



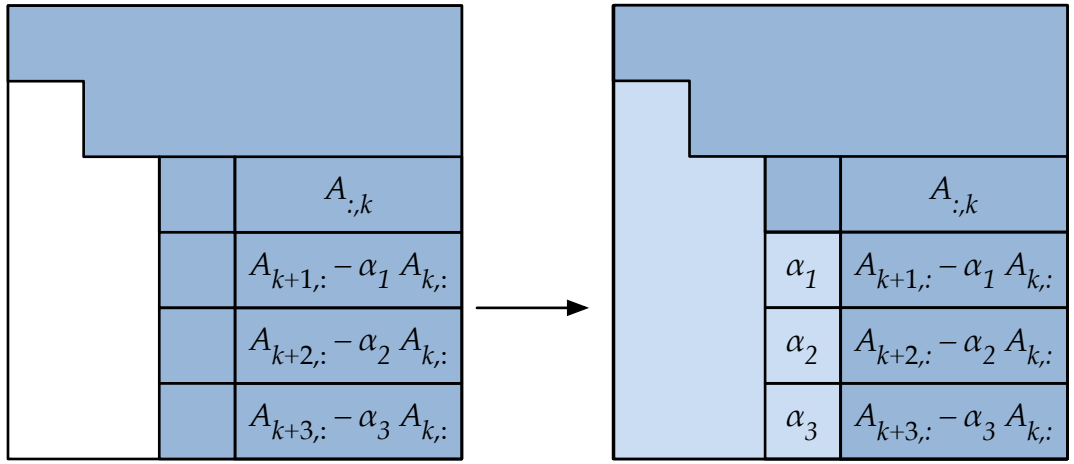
"block axpy" ($y = \alpha x + y$) implementation



LU factorization (Gaussian elimination)

- Goal: transform A into triangular matrices LU, which we know how to solve

```
// Basic no-pivoting version (unstable!)
for k = 1 to n-1
    for i = k+1 to n
        // Subtract multiply of row A(k,:) to zero out entry A(i,k).
        // Store multiplier in place of A(i,k).
        A( i, k ) /= A( k, k )
        A( i, k+1:n ) -= A( i, k ) * A( k, k+1:n )
```



LU factorization (Gaussian elimination)

- Goal: transform A into triangular matrices LU, which we know how to solve

```
// Basic no-pivoting version (unstable!)
for k = 1 to n-1
    for i = k+1 to n
        // Subtract multiply of row A(k,:) to zero out entry A(i,k).
        // Store multiplier in place of A(i,k).
        A( i, k ) /= A( k, k )
        A( i, k+1:n ) -= A( i, k ) * A( k, k+1:n )
```

- What happens with these well-conditioned matrices?

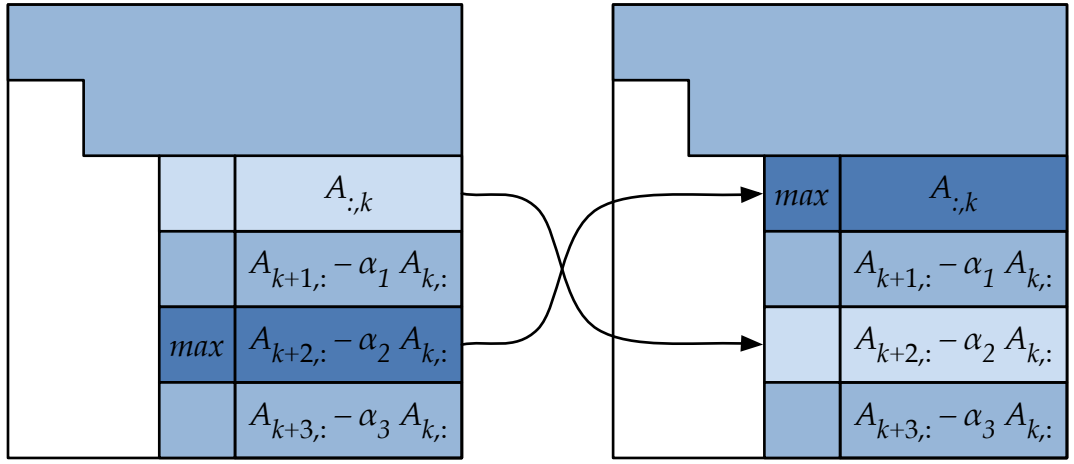
$$A = \begin{bmatrix} 0 & 5 \\ 2 & 4 \end{bmatrix} \qquad A = \begin{bmatrix} \varepsilon & 5 \\ 2 & 4 \end{bmatrix}$$

- LU without pivoting is **unstable**!

LU factorization (Gaussian elimination)

- Solution: swap rows so diagonal has largest value

```
// Basic partial pivoting version (stable!)
for k = 1 to n-1
    p = iamax( A( k:n, k ) ) + k - 1  // index of largest element
    swap( A( k, : ), A( p, : ) )
    for i = k+1 to n
        // Subtract multiply of row A(k,:) to zero out entry A(i,k).
        // Store multiplier in place of A(i,k).
        A( i, k ) /= A( k, k )
        A( i, k+1:n ) -= A( i, k ) * A( k, k+1:n )
```



LU factorization (Gaussian elimination)

- Solution: swap rows so diagonal has largest value

```
// Basic partial pivoting version (stable!)
for k = 1 to n-1
    p = iamax( A( k:n, k ) ) + k - 1  // index of largest element
    swap( A( k, : ), A( p, : ) )
    for i = k+1 to n
        // Subtract multiply of row A(k,:) to zero out entry A(i,k).
        // Store multiplier in place of A(i,k).
        A( i, k ) /= A( k, k )
        A( i, k+1:n ) -= A( i, k ) * A( k, k+1:n )
```

- Have $PA = LU$, where P is permutation matrix
  - P is orthogonal, so $P^{-1} = P^T$
  - Solve $Ax = (P^T L U)\, x = b$ as $x = U^{-1} L^{-1} P b$, where U^{-1} and L^{-1} mean do triangular solves (trsm), don't compute inverses!

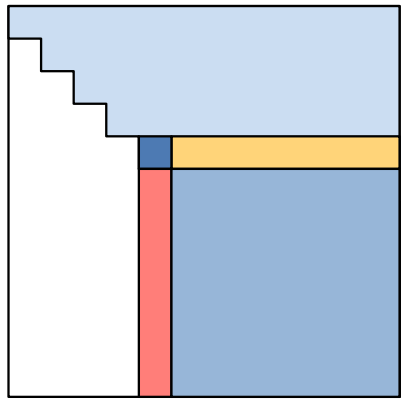
Example swaps first 2 rows

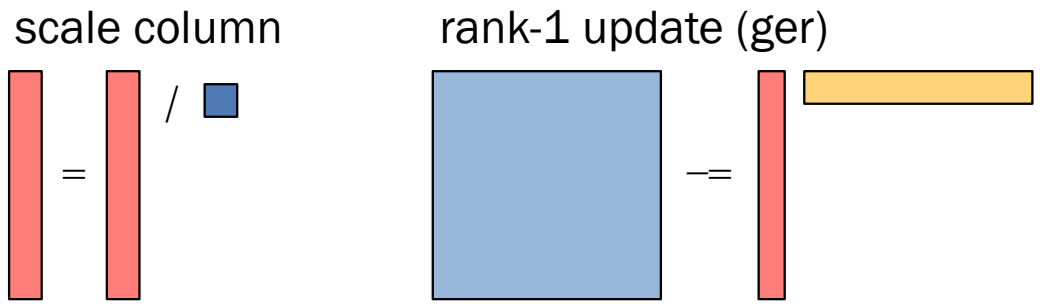
$$P = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

LU factorization (Gaussian elimination)

- Improve performance using BLAS

```
// Partial pivoting version, Level 1 and 2 BLAS
for k = 1 to n-1
    p = iamax( A( k:n, k ) ) + k - 1  // index of largest element
    swap( A( k, : ), A( p, : ) )

    // scale column
    A( k+1:n, k ) /= A( k, k )

    // rank-1 update (ger)
    A( k+1:n, k+1:n ) -= A( k+1:n, k ) * A( k, k+1:n )
```

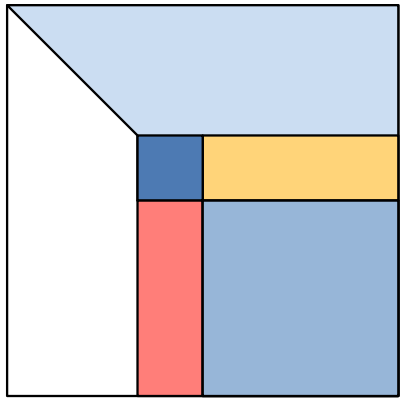




LU factorization (Gaussian elimination)

- Blocked version using Level 3 BLAS

```
// Partial pivoting version, blocked
for k = 1 to n by nb
    kb = min( nb, n - k + 1 )
    getrf( A( k:n, k:k+kb-1 ), pivots )        // panel
    apply_pivots( A( k:n, 1:k-1  ), pivots )   // left of panel
    apply_pivots( A( k:n, k+kb:n ), pivots )   // right of panel

    // update block row of U (trsm)
    Lkk = lower( A( k:k+kb-1, k:k+kb-1 ) )
```

- 

In (Sca)LAPACK:

gesv solves $Ax = b$ by calling:

- **getrf** factors $PA = LU$
- **getrs** solves $P^T LUx = b$

gesvx is an expert version

dsgesv / **zcgesv** are mixed precision versions

getri computes A^{-1} from $PA = LU$

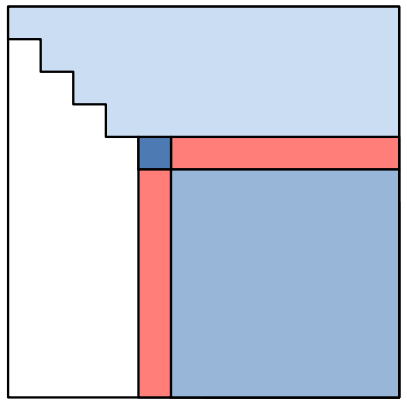
Cholesky (LL^T)

- If A is Hermitian (symmetric) positive definite, reformulate LU symmetrically
  - Square root of diagonal
  - No pivoting required for stability
  - Have $A = LL^T = U^T U$
  - Solve $Ax = (LL^T)\, x = b$ as
    $x = L^{-T}\, L^{-1}\, b$, where L^{-1} means triangular solve,
    and L^{-T} means transposed triangular solve
- Cost: $\frac{1}{3}n^3$ flops

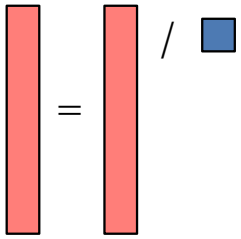
Cholesky (LL^T)

- If A is Hermitian (symmetric) positive definite, reformulate LU symmetrically
  - Square root of diagonal

```
// Level 1 and 2 BLAS
for k = 1 to n-1
    A( k, k ) = sqrt( A( k, k ) )

    // scale column
    A( k+1:n, k ) /= A( k, k )

    // symmetric rank-1 update (syr)
    A( k+1:n, k+1:n ) -= A( k+1:n,
```



scale column



In (Sca)LAPACK:

posv solves $Ax = b$ by calling:

- **potrf** factors $A = LL^T$
- **potrs** solves $LL^Tx = b$

posvx is an expert version

dsposv / **zcposv** are mixed precision versions

potri computes A^{-1} from $A = LL^T$

LDL^T

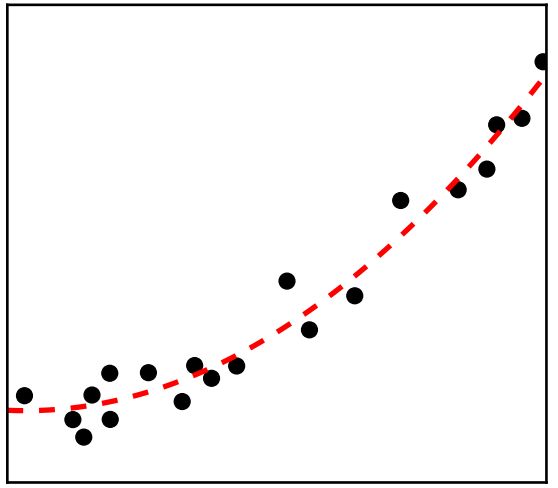
- Symmetric Indefinite
- Pivoting strategies: Bunch-Kaufmann, Rook, Aasen's, Block Aasen's
  - Symmetric pivoting: both row & col swaps
- Cost: $\frac{1}{3}n^3$ flops
  - But more expensive than Cholesky due to row & col swaps (data movement)

In LAPACK:
sysv/hesv* solves $Ax = b$ by calling:
  sytrf/hetrf* factors $A = LDL^T$
  sytrs/hetrs* solves $LDL^Tx = b$
sysvx/hesvx is an expert version
sytri* computes A^{-1} from $A = LDL^T$

* several versions available.
Not in ScaLAPACK. No mixed precision.

Least squares (over-determined)

- Solve $Ax \cong b$, where A is tall $(m \geq n)$
  - Minimize residual $\|r\|_2 = \|b - Ax\|_2$
  - QR
    - $QRx \cong b$
    - $X = R^{-1} Q^T b$, where as usual, R^{-1} means solve triangular system
  - Normal equations
    - $A^T Ax = A^T b$
    - $A^T A$ is symmetric positive definite (assuming A is full rank) and only $n \times n$
    - Generally avoid forming $A^T A$, it squares condition number!
  - SVD
    - $U\Sigma V^T x \cong b$
    - $x = V\Sigma^{-1} U^T b$, inverse of Σ just invert each diagonal element
    - $A^+ = V\Sigma^{-1} U^T$ is the pseudo-inverse
    - Especially good to use if A is rank-deficient ($\text{rank} < \min(m, n)$)



In (Sca)LAPACK:
gels uses QR/LQ
gelsy uses QR with pivoting
gelsd uses SVD (divide and conquer)
gelss uses SVD (QR iteration)

Minimum norm (under-determined)

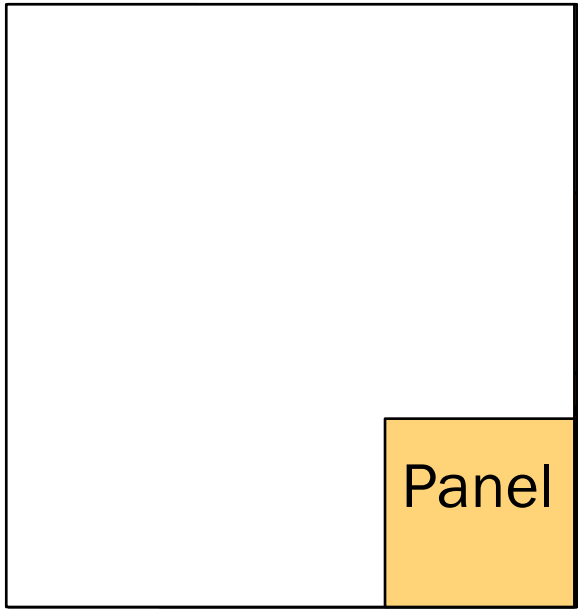
- Solve $Ax = b$, where A is wide ($m < n$)
  - No unique solution, so find x with minimum norm
  - LQ
    - $A = LQ = (\hat{Q}R)^T \quad \text{where} \quad \hat{Q}R = A^T$
    - $LQx = b$
    - $x = Q^T L^{-1} b$
  - SVD
    - Same as before

ICL THE UNIVERSITY OF TENNESSEE KNOXVILLE

Hybrid & Native GPU algorithms

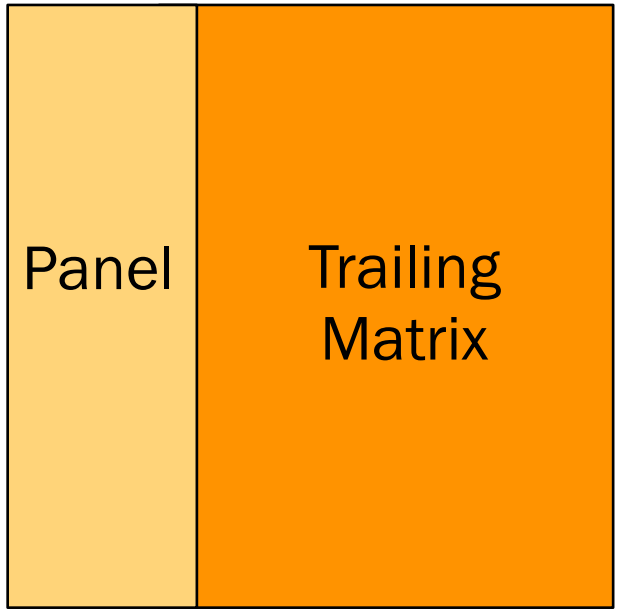
Linear algebra routines

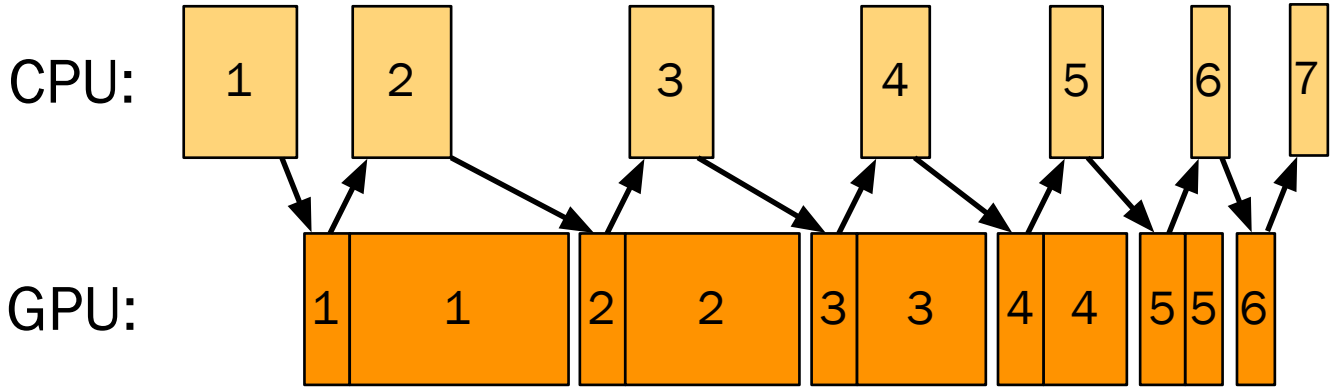
- Iterate two steps:
  - Panel factorization
    - Level 1–2 BLAS
    - Control flow
    - Data dependent (pivoting, etc.)
  - Trailing matrix update
    - Level 3 BLAS



Hybrid CPU–GPU algorithms

- Assign panel to CPU
- Assign trailing matrix to GPU
- Communicate panel between CPU ⇔ GPU
- Overlap next panel during trailing matrix update

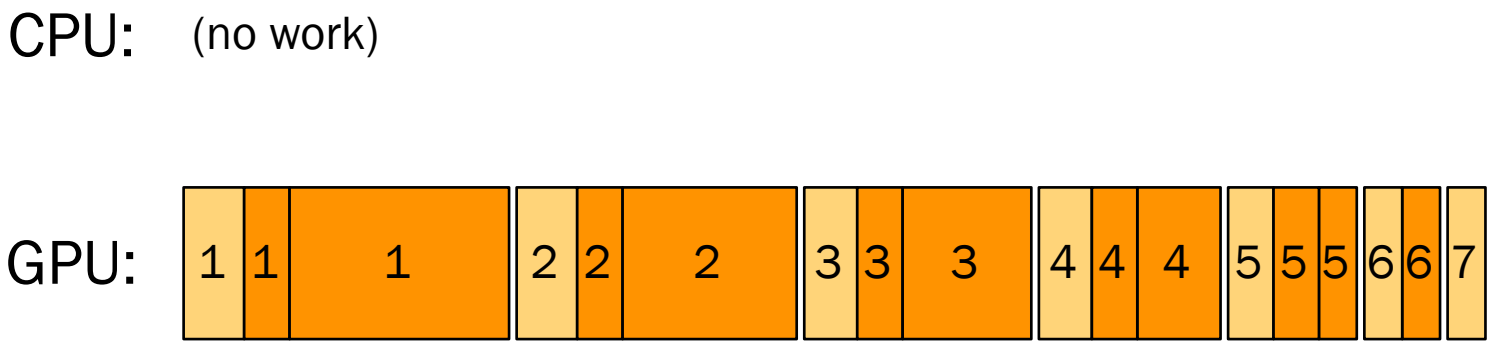




GPU–only algorithms

- Assign both panel and trailing matrix to GPU
- No CPU ⇔ GPU communication

- CPU available for other tasks
- No overlap
  - Some algorithms don't allow overlap anyhow

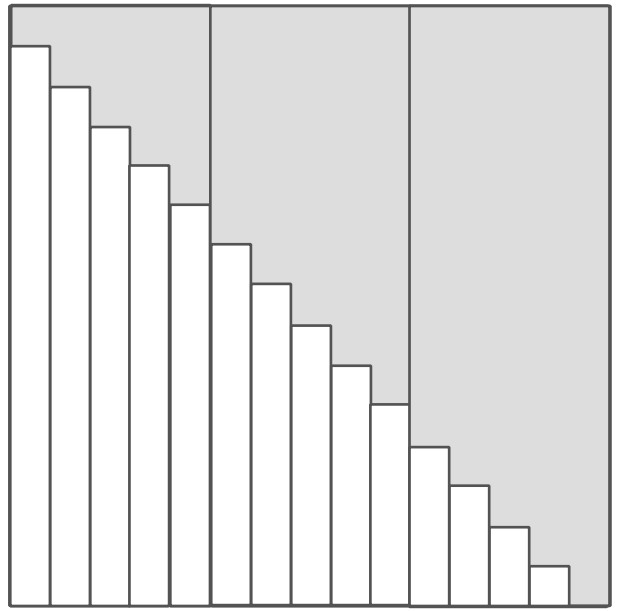


Householder–based algorithms

- QR factorization (geqrf)
  - $A = QR$
  - Least squares, etc.
- QR with column pivoting (geqp3)
  - $AP = QR$
  - More stable, esp. for rank-deficient matrices
- Hessenberg reduction (gehrd)
  - $Q^H A Q = H$
  - Non-symmetric eigenvalues

QR factorization

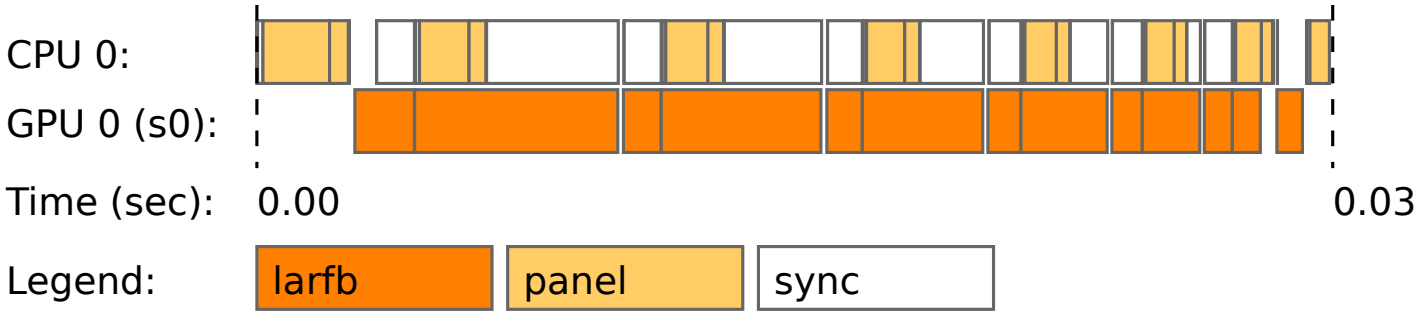
- Panel (nb columns)
  - for each column
    - apply previous reflectors
    - annihilate entries below diagonal
- Trailing matrix
  - update next panel (look-ahead)
  - update rest of A
- Overlap next panel & trailing matrix update



Execution trace

(NOTE: outdated results.

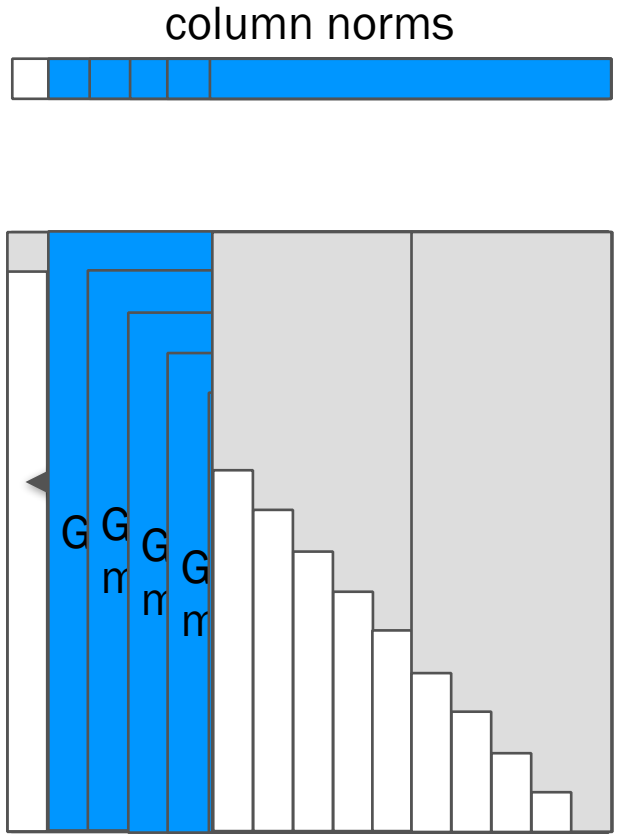
- Hybrid CPU–GPU



- GPU–only

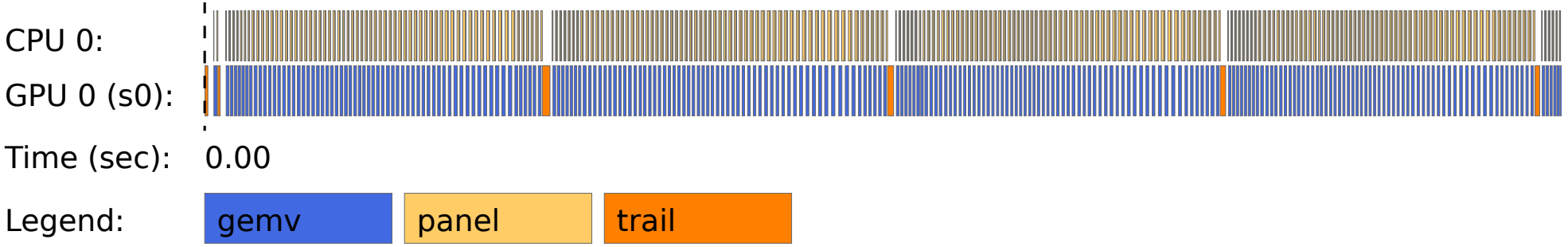
QR with column pivoting

- Compute column norms
- Panel (nb columns)
  - for each column
    - swap with column of max norm
    - apply previous reflectors
    - annihilate entries below diagonal
    - GEMV with trailing matrix on GPU
    - update column norms
- Trailing matrix
  - update rest of A
- Dependencies prevent overlap

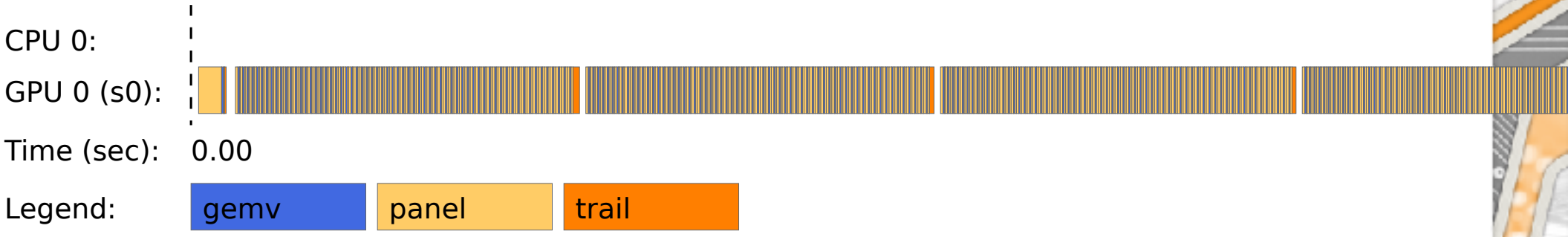


Execution trace

- Hybrid CPU–GPU

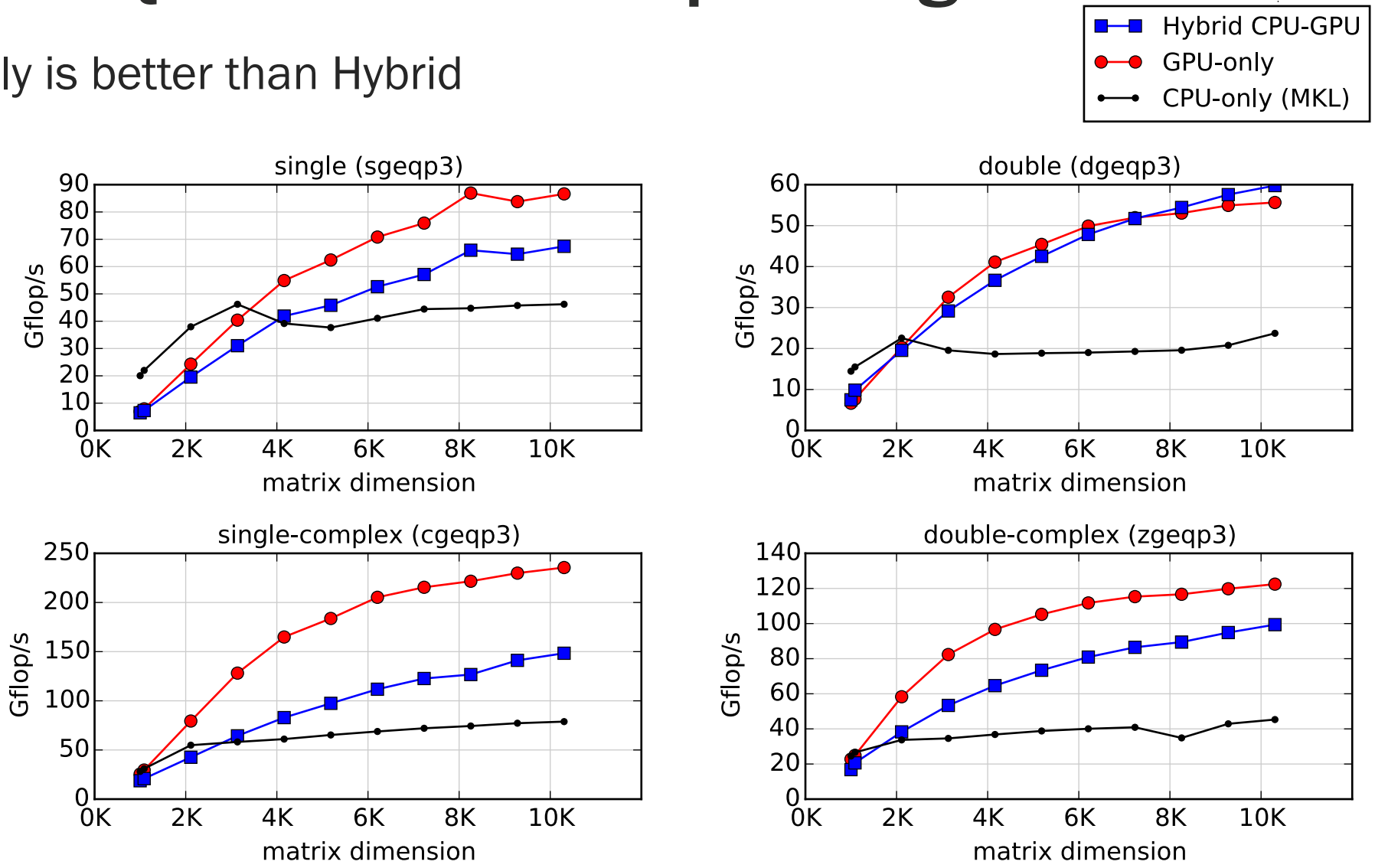


- GPU–only



Results: QR with column pivoting

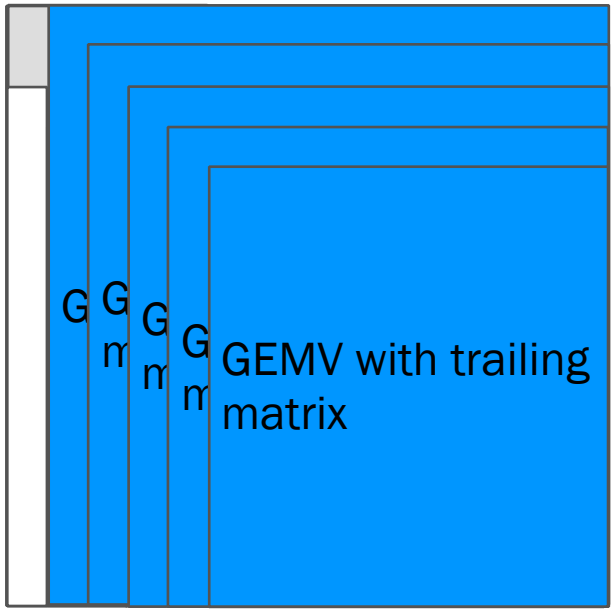
- GPU-only is better than Hybrid



2 x 8 core Intel Sandy Bridge E5-2670, NVIDIA K40c

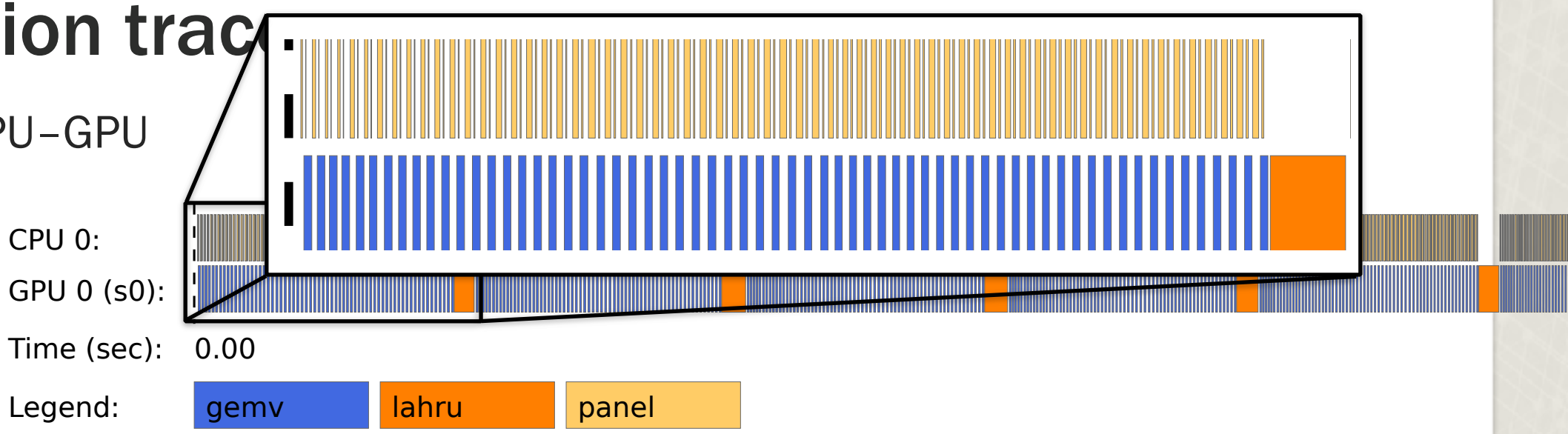
Hessenberg reduction

- Panel (nb columns)
  - for each column
    - apply previous reflectors (from right and left)
    - annihilate entries below sub-diagonal
    - GEMV with trailing matrix on GPU
- Trailing matrix
  - update rest of A from right and left
- Dependencies prevent overlap

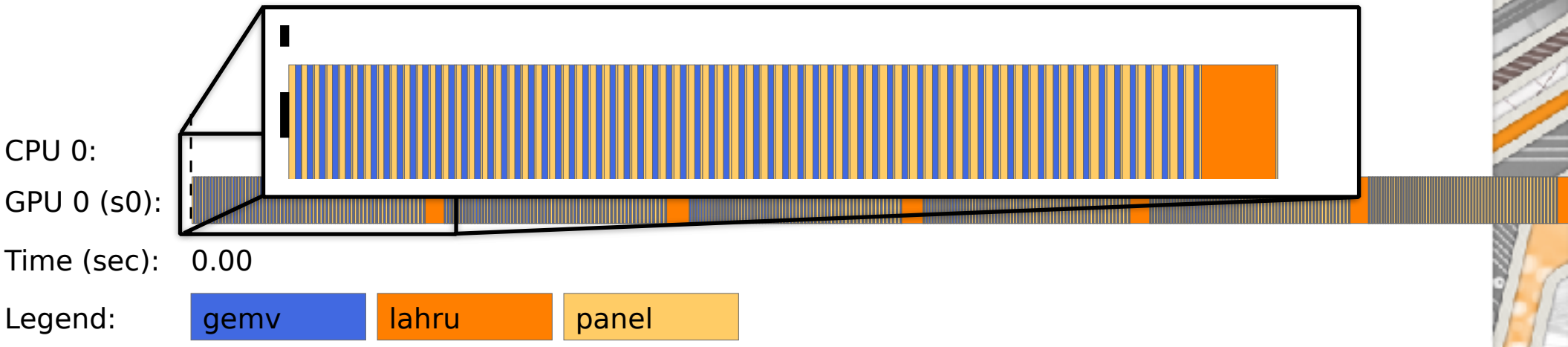


Execution trac

- Hybrid CPU–GPU

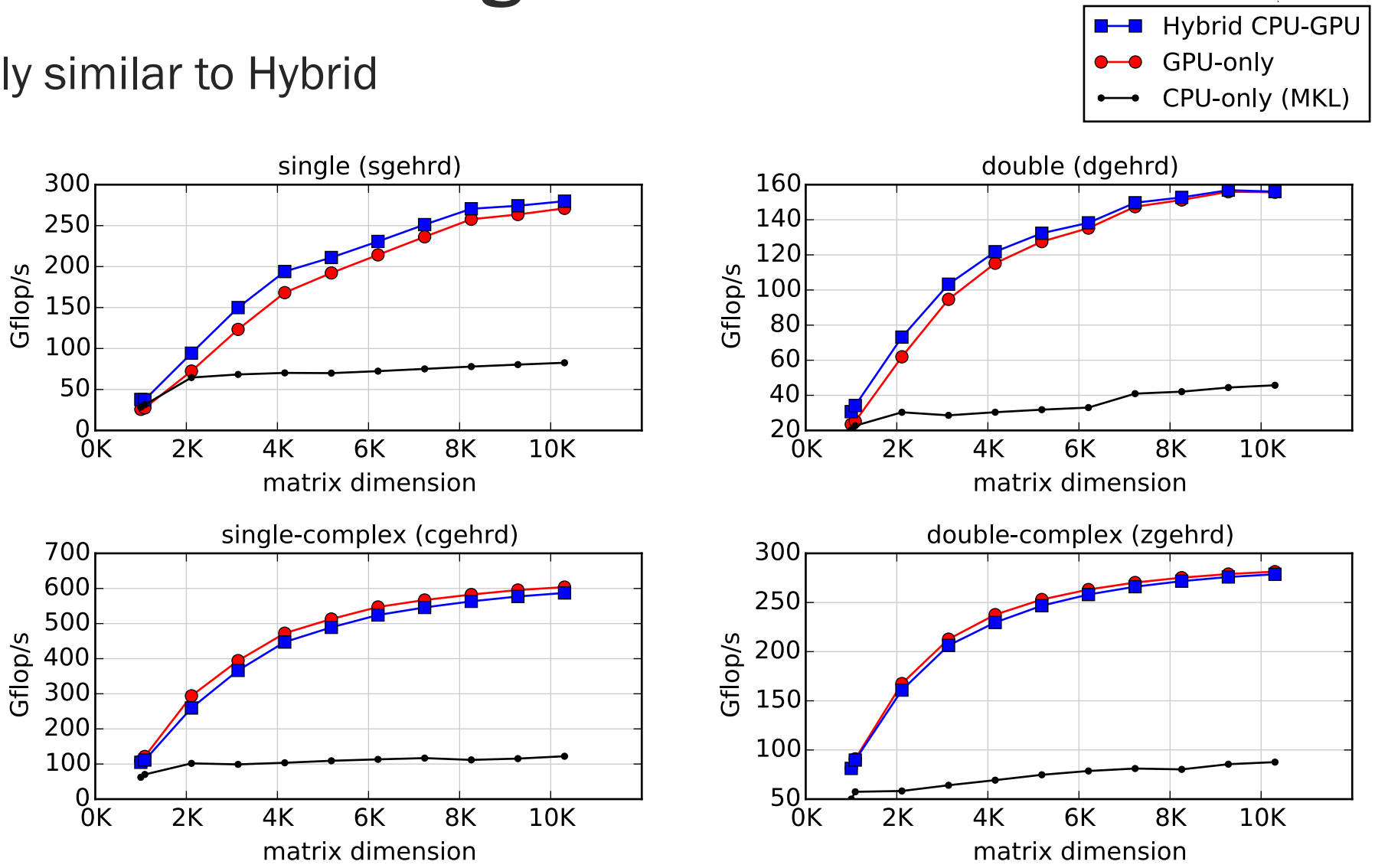


- GPU-only



Results: Hessenberg

- GPU-only similar to Hybrid



2 x 8 core Intel Sandy Bridge E5-2670, NVIDIA K40c

GPU–only kernels & optimizations

- Householder reflectors
  - Generate — vector norm and scaling (larfg)
    - save extra copies of tau in T, etc.
  - Apply — dot product and axpy (larf)
- Custom norm update for QR with pivoting
- Optimized gemv
  - Tall matrix transposed * vector: $V^T a_j$
- Use gemv, faster than trmv
  - Store V and T with explicit 0's and 1's
  - Merge trmv+gemv into one gemv

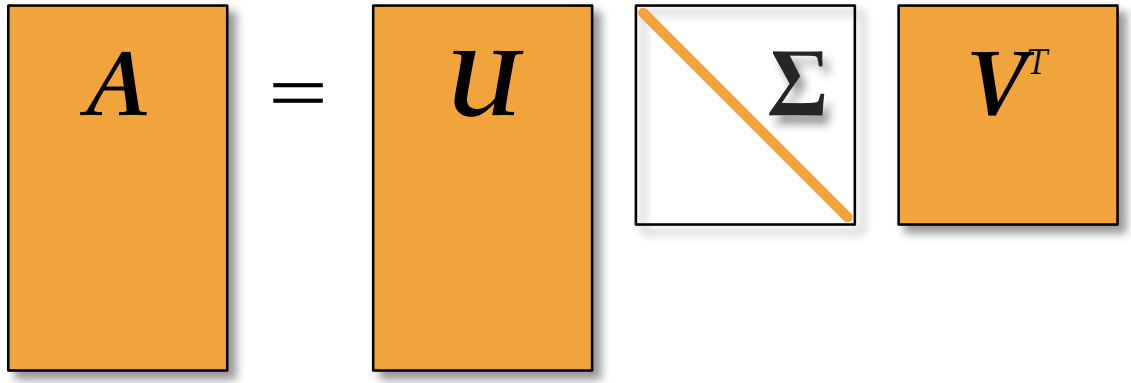
Lessons Learned

- Panels
  - Lack parallelism
  - Significant control flow
  - Many separate function calls
    - Perform poorly on GPUs
  - Requires programming custom GPU kernels
  - Merge kernels together to reduce overheads

- GPU-only reduces communication
  - Modest win for QR with pivoting
  - No improvement for Hessenberg

SVD

What is the SVD?

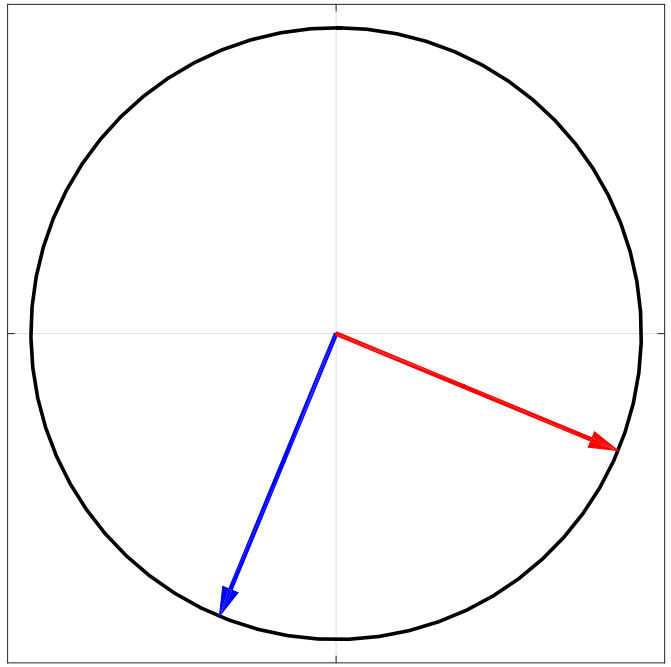


- U and V are orthogonal matrices of left and right **singular vectors**
- Σ is diagonal matrix of **singular values**, $\sigma_i \geq 0$

What is the SVD?

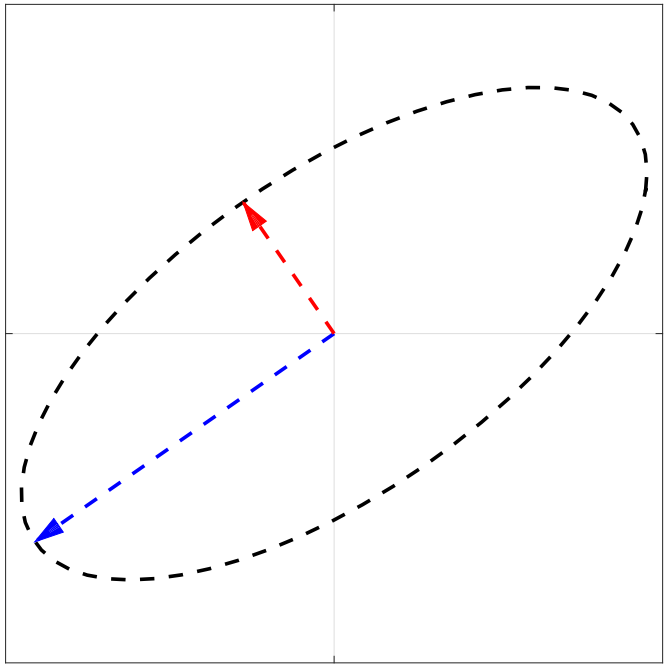
- 2 x 2 matrix A maps unit circle to ellipse
  - Axes are singular vectors

V, left singular vectors



A

$U\Sigma$, right singular vectors

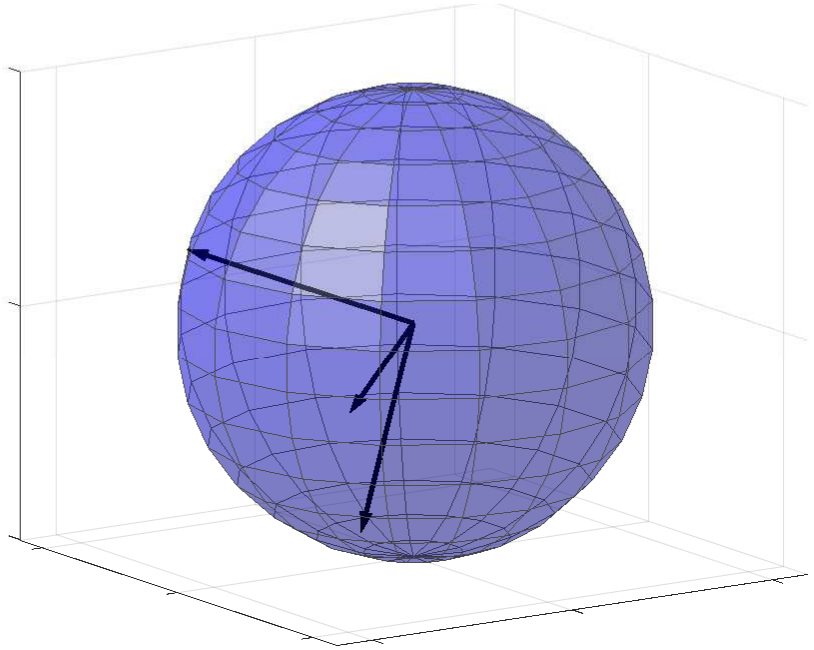


ICL THE UNIVERSITY OF TENNESSEE KNOXVILLE

What is the SVD?

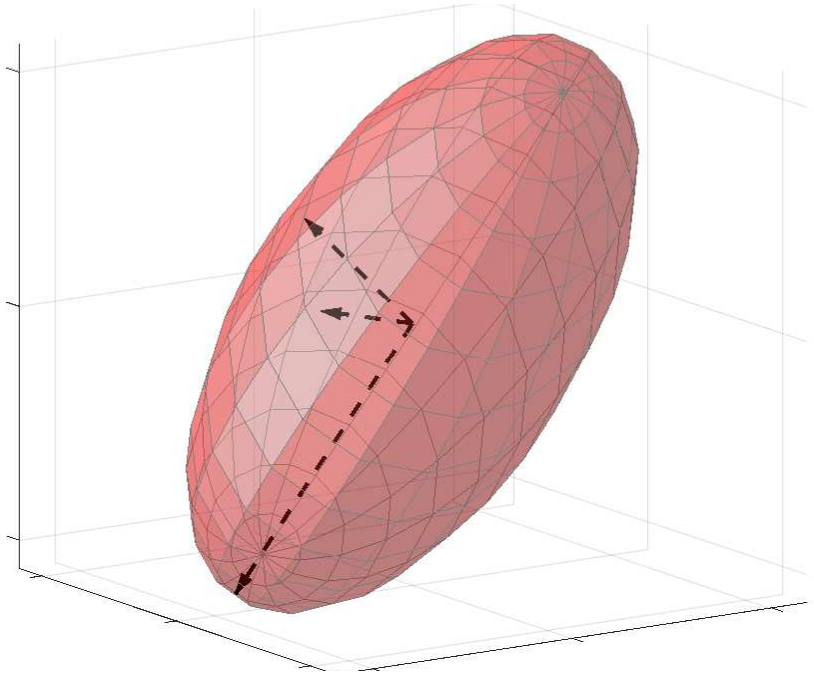
- 3 x 3 matrix A maps unit sphere to ellipsoid
  - Axes are singular vectors

V, left singular vectors



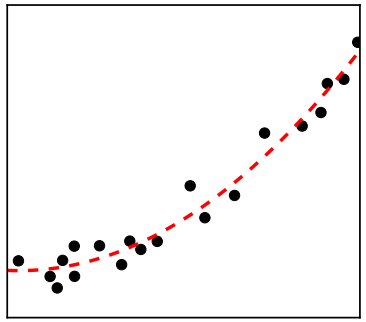
A

$U\Sigma$, right singular vectors



ICL THE UNIVERSITY OF TENNESSEE KNOXVILLE

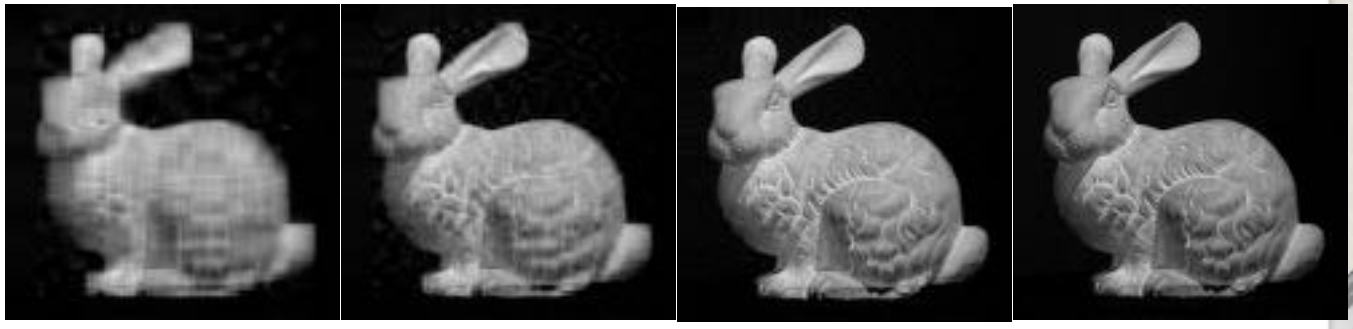
Applications



- Often most robust, but most expensive, method
  - "Elephant gun" of linear algebra
- Rank, 2-norm, condition number
- Least squares
- Best low-rank approximation

$$A_r = U_{1:r}\,\Sigma_{1:r}\,V_{1:r}{}^T \approx A$$

- Compression
- Facial recognition
- Latent semantic analysis



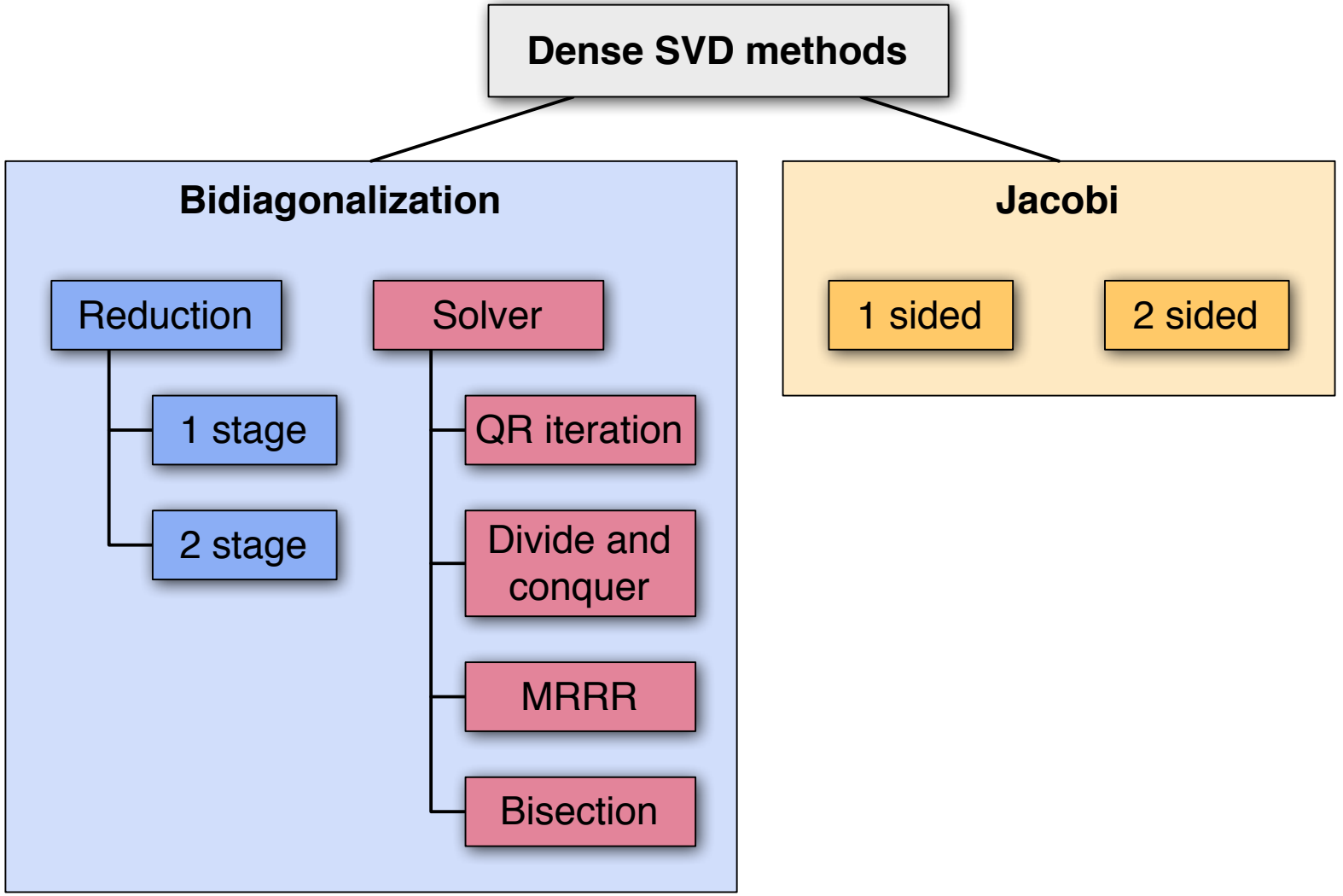
rank 10 | rank 20 | rank 50 | original rank 217

average | component 1 | component 2

eigenfaces

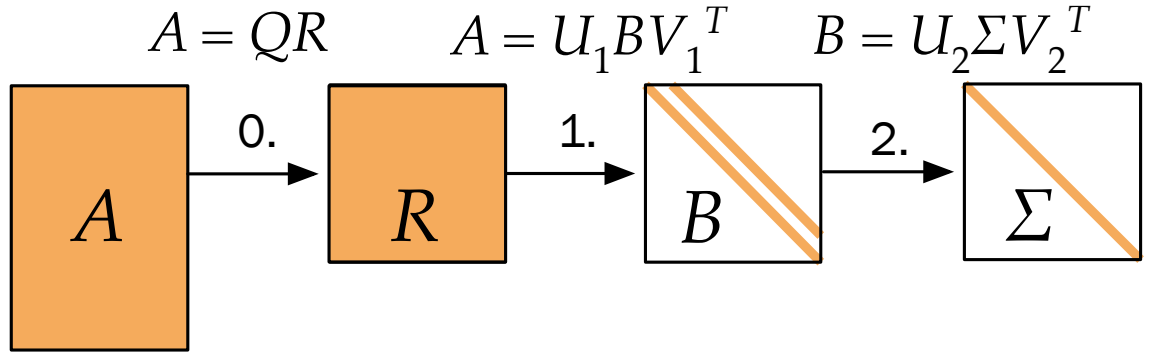
Bunny from Stanford 3D Scanning Repository

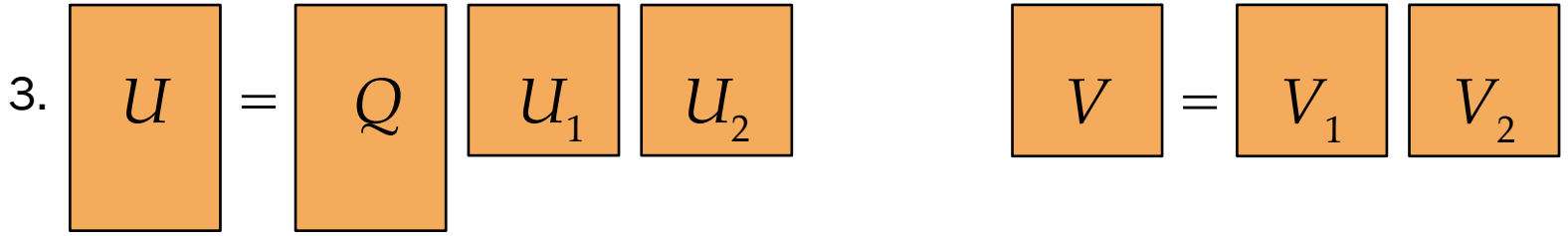
Computing the SVD



Bidiagonalization phases

0. Optional initial QR to reduce tall matrix to square
1. Reduction to bidiagonal form (1- or 2-stage)
2. SVD of bidiagonal (QR iteration, D&C, MRRR, Bisection)
3. Computation of singular vectors





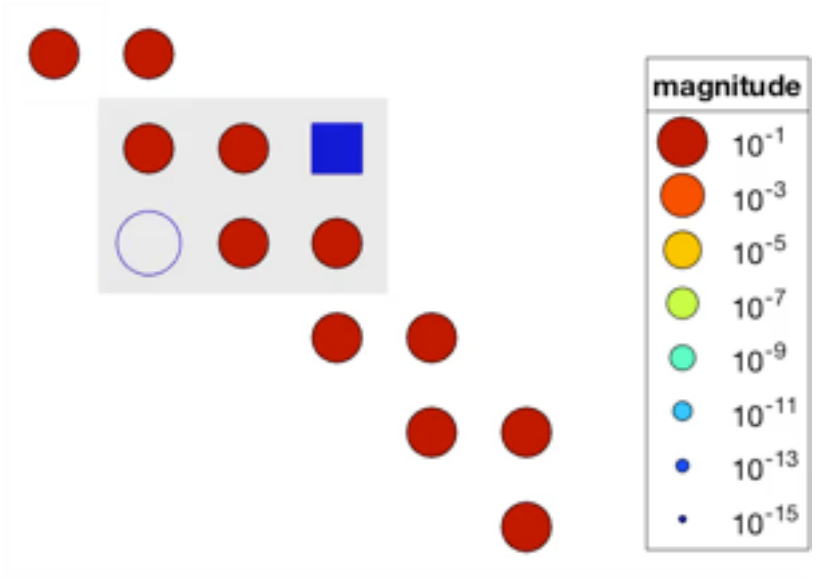
Bidiagonal reduction

- for $i = 1$ to n
  - Apply Householder on left to zero column i
  - Apply Householder on right to zero row i
- $\frac{8}{3}n^3$ flops

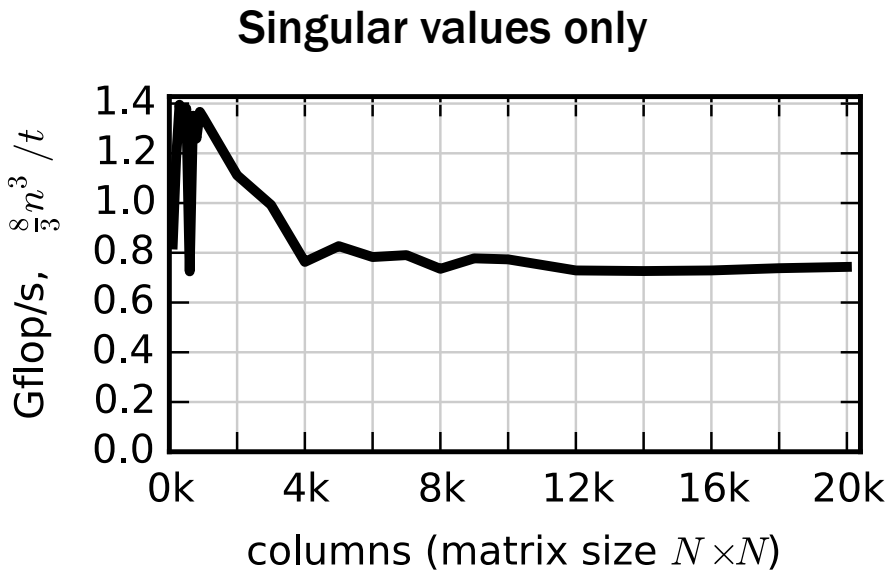
ICL THE UNIVERSITY OF TENNESSEE KNOXVILLE

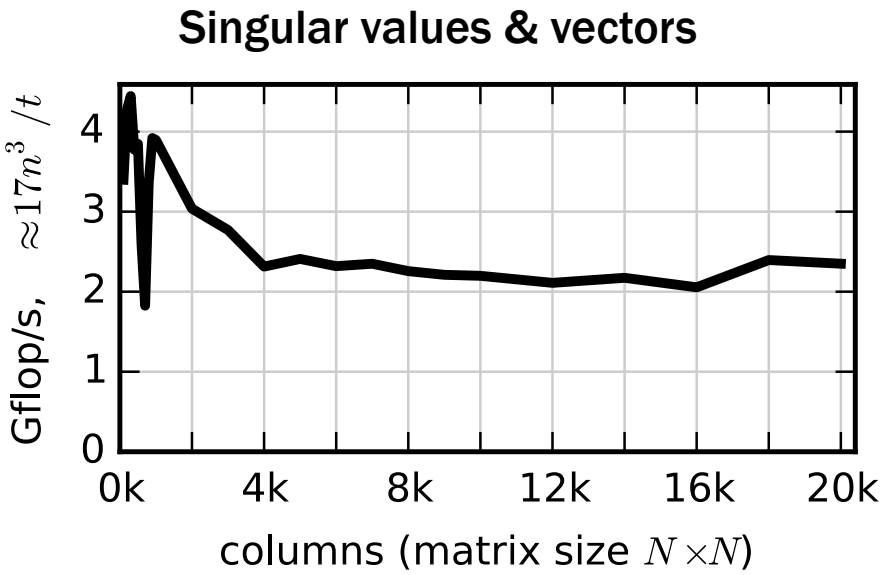
QR iteration

- Iteratively apply Givens rotations on left & right to reduce off-diagonal terms
- $O(n^2)$ singular values only, ~$12n^3$ with vectors



EISPACK

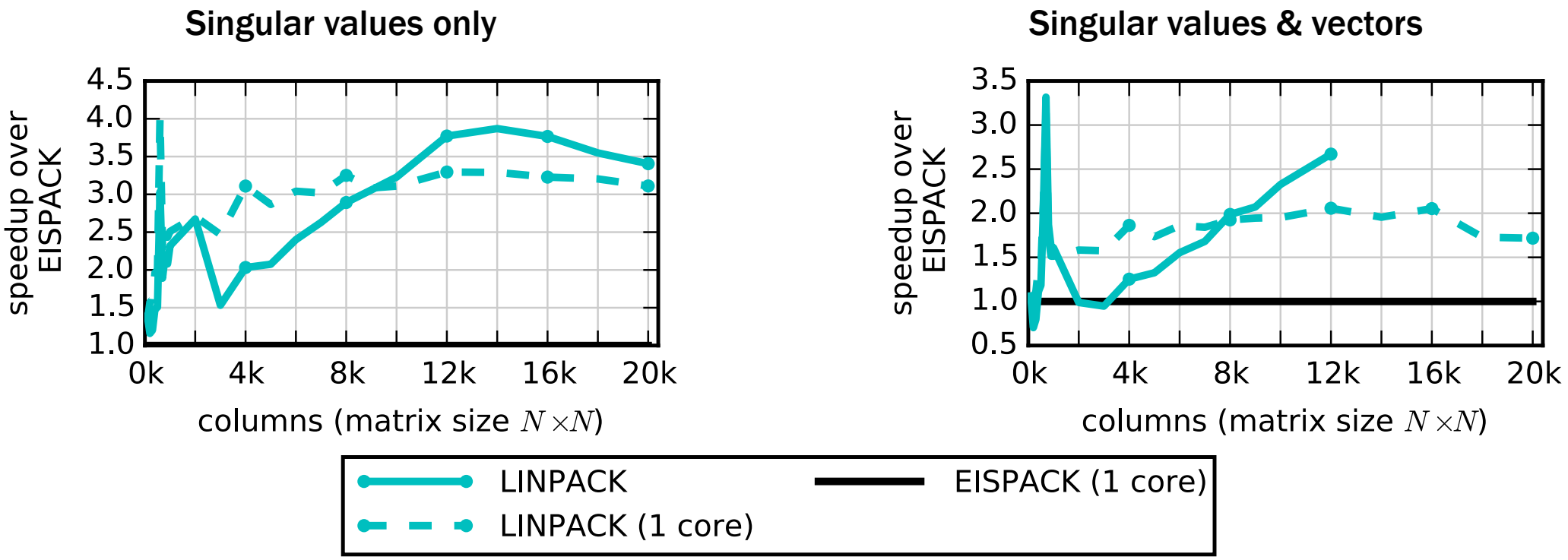




- Fortran transliteration of Algol (Golub & Reinsch)
- Row major access
- Single core

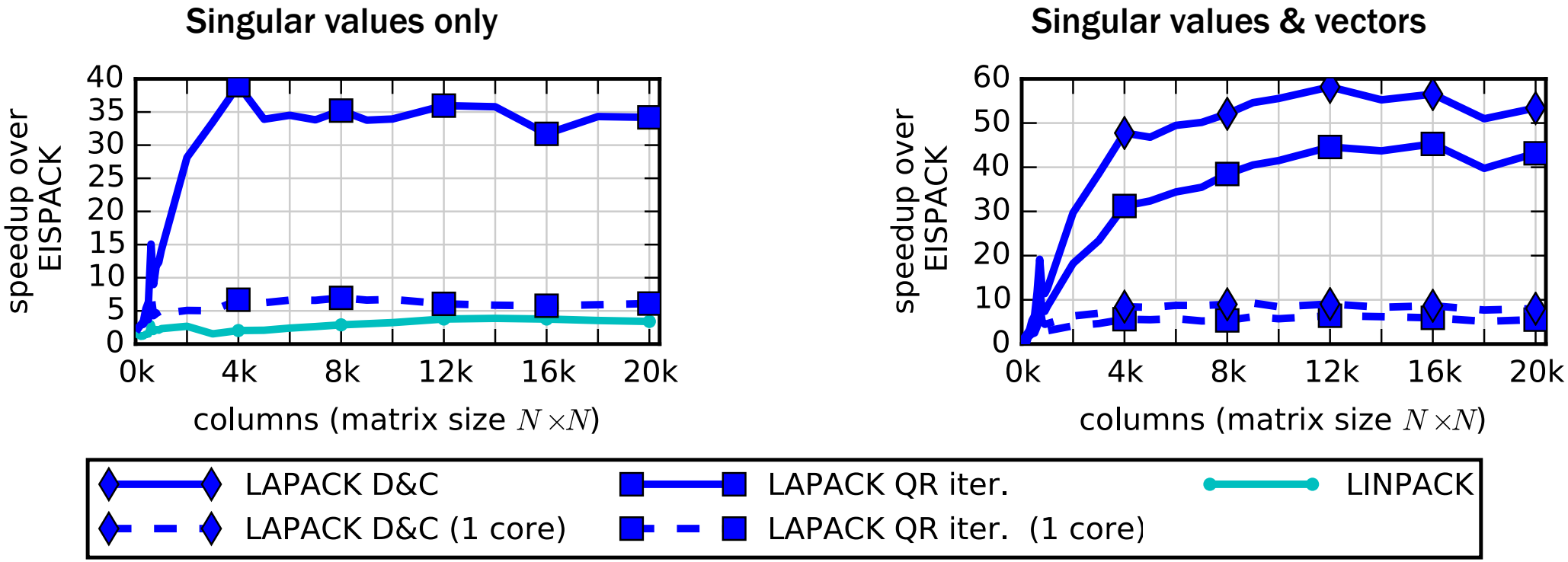
Results on 2x8 core 2.6 GHz Intel Sandy Bridge

LINPACK



- Level 1 BLAS for vector supercomputers
- Fortran column major access

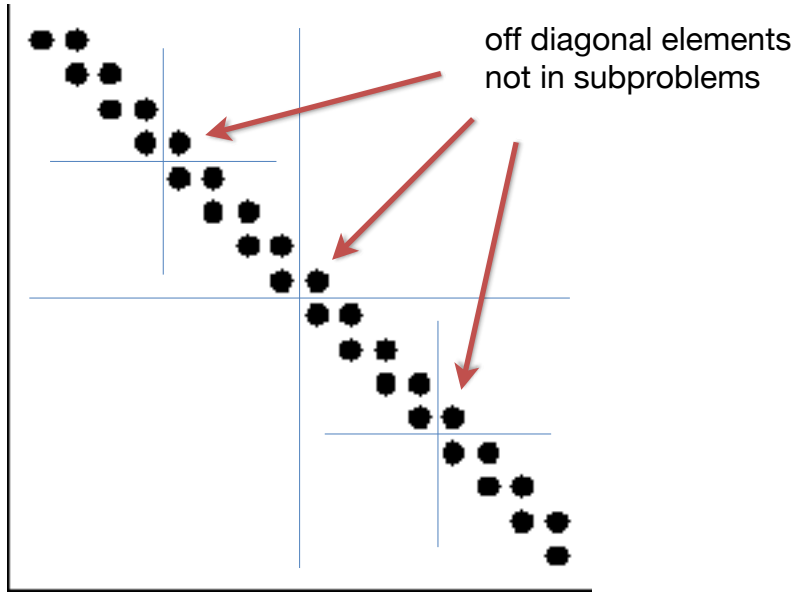
LAPACK

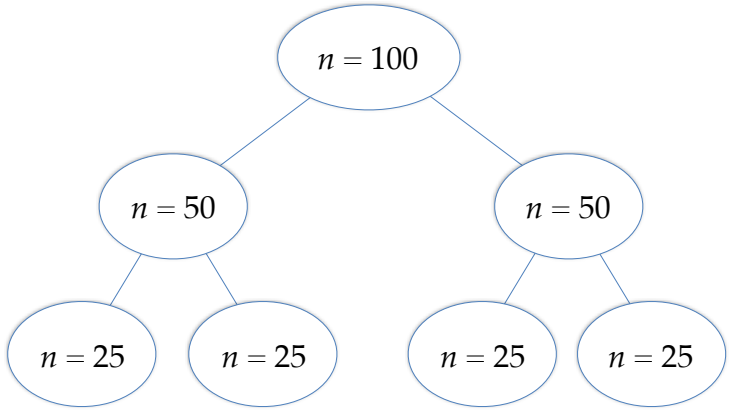


- Blocked reduction to band: half Level 2 BLAS, half Level 3 BLAS
  - Limited to 2x gemv performance
- QR iteration (gesvd) and Divide & conquer (gesdd)

Divide and Conquer (Gu & Eisenstat)

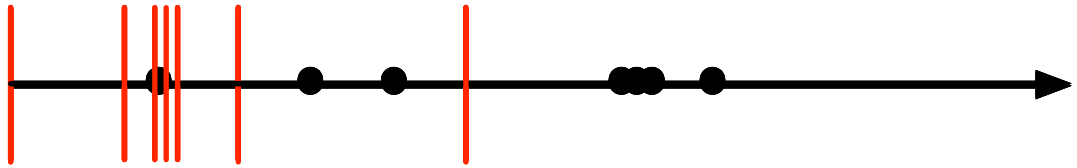
- Divide bidiagonal in half recursively
- Solve leaf nodes (n ≈ 25) using QR iteration
- Combine subproblems by solving secular equation
- Reduces flops from $\sim 15n^3$ to $7n^3$ and uses Level 3 BLAS





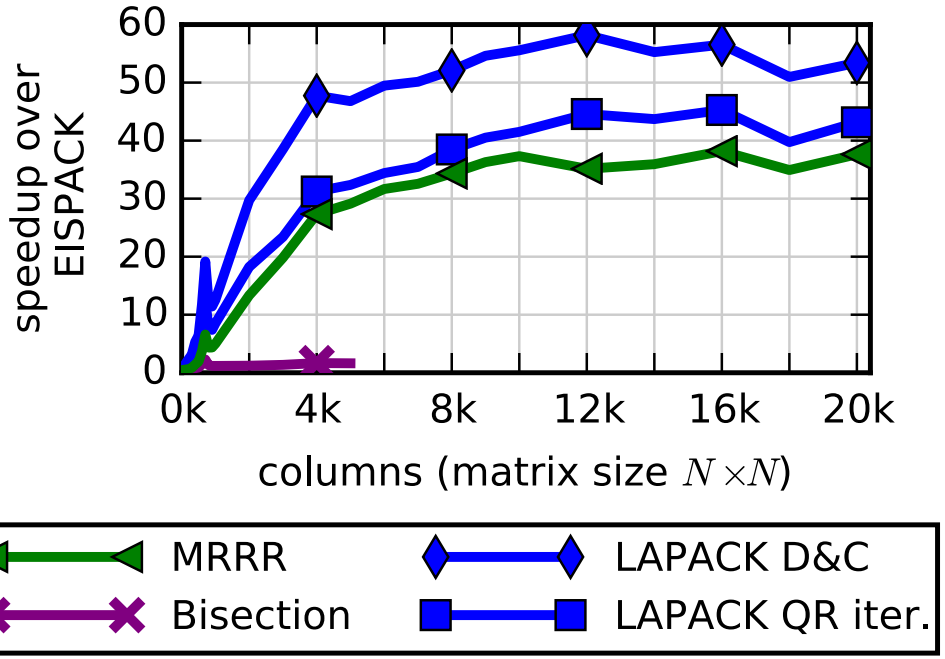
Bisection & MRRR

- Sylvester's inertia theorem gives # singular values in range
- Bisect range until singular value isolated to desired accuracy
- Difficulties with tight clusters
- MRRR (multiple relatively robust representations) uses shifts and special representation to deal with clusters

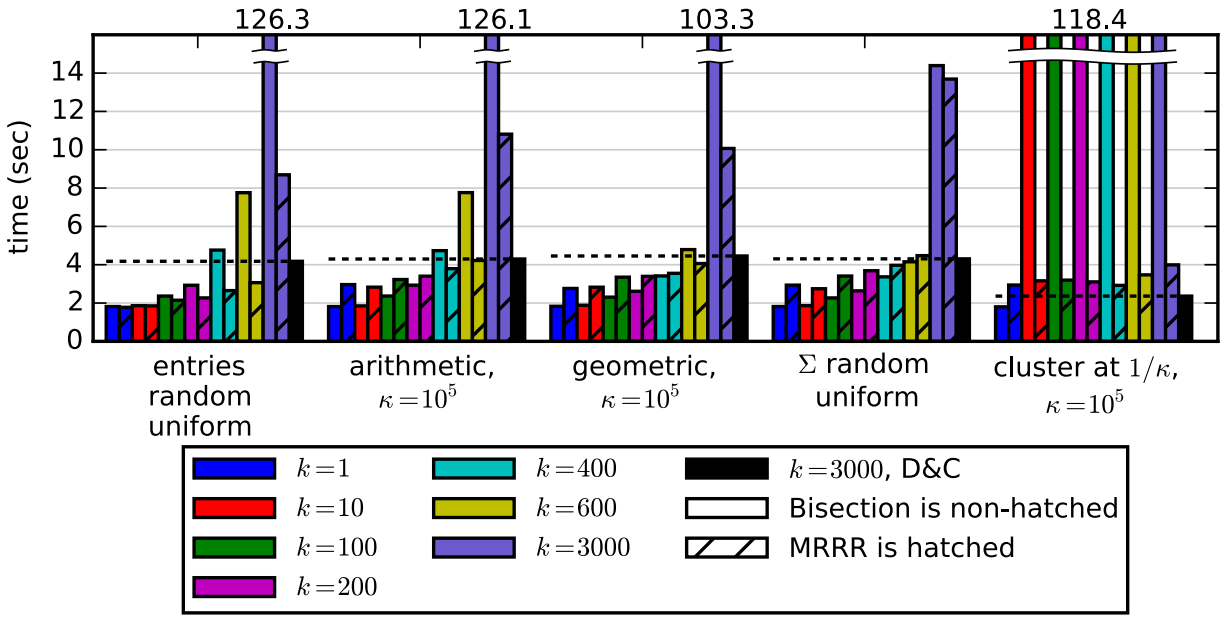


LAPACK

Singular values & vectors

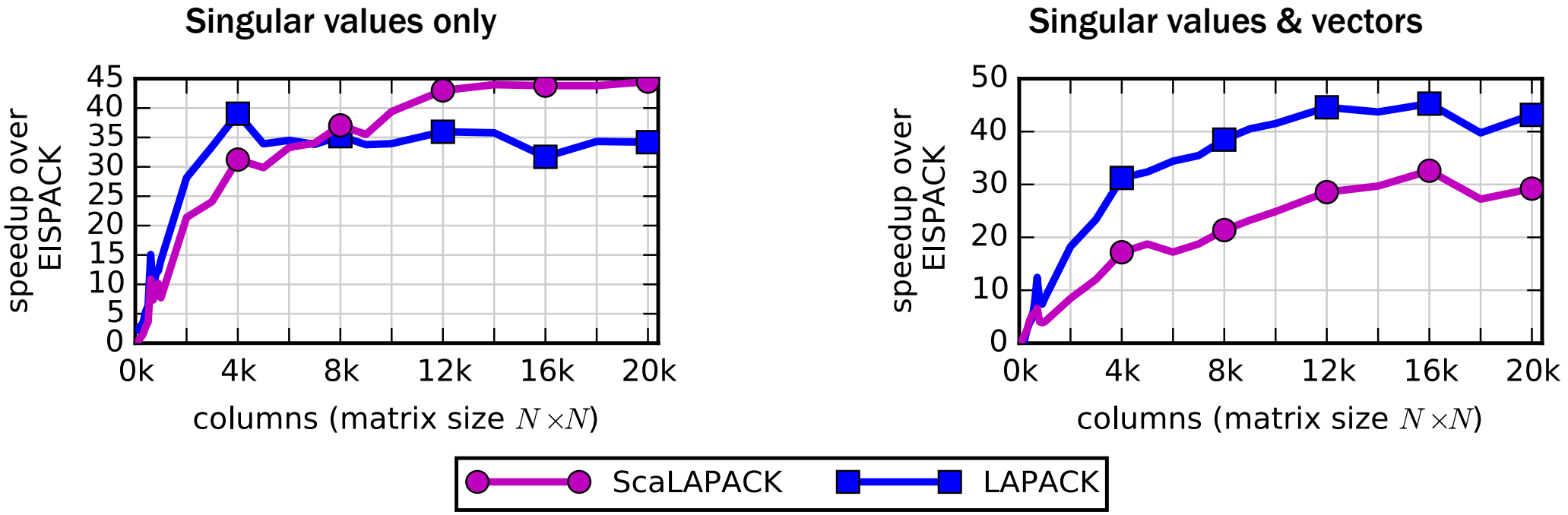


Subset of singular values & vectors



- Bisection (gesvdx; Marques & Vasconcelos)
- MRRR (Willems, Lang, & Vömel)
- Compute subset of singular vectors

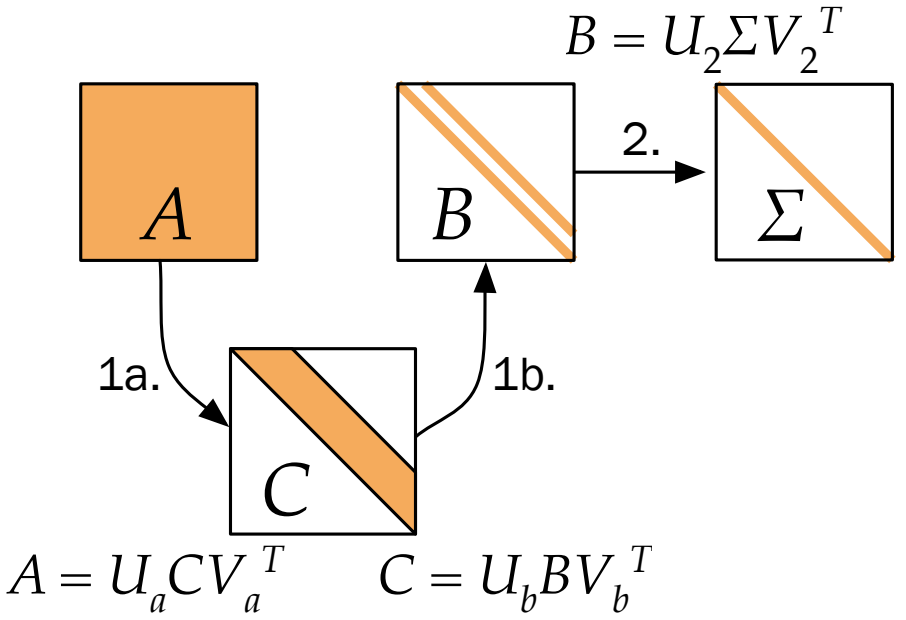
ScaLAPACK



- Distributed memory, $p \times q$ process grid
- Uses LAPACK's QR iteration solver
  - Parallelism limited to p for $\boldsymbol{V}$ and q for $\boldsymbol{U}$

2-stage Bidiagonalization (Großer & Lang)

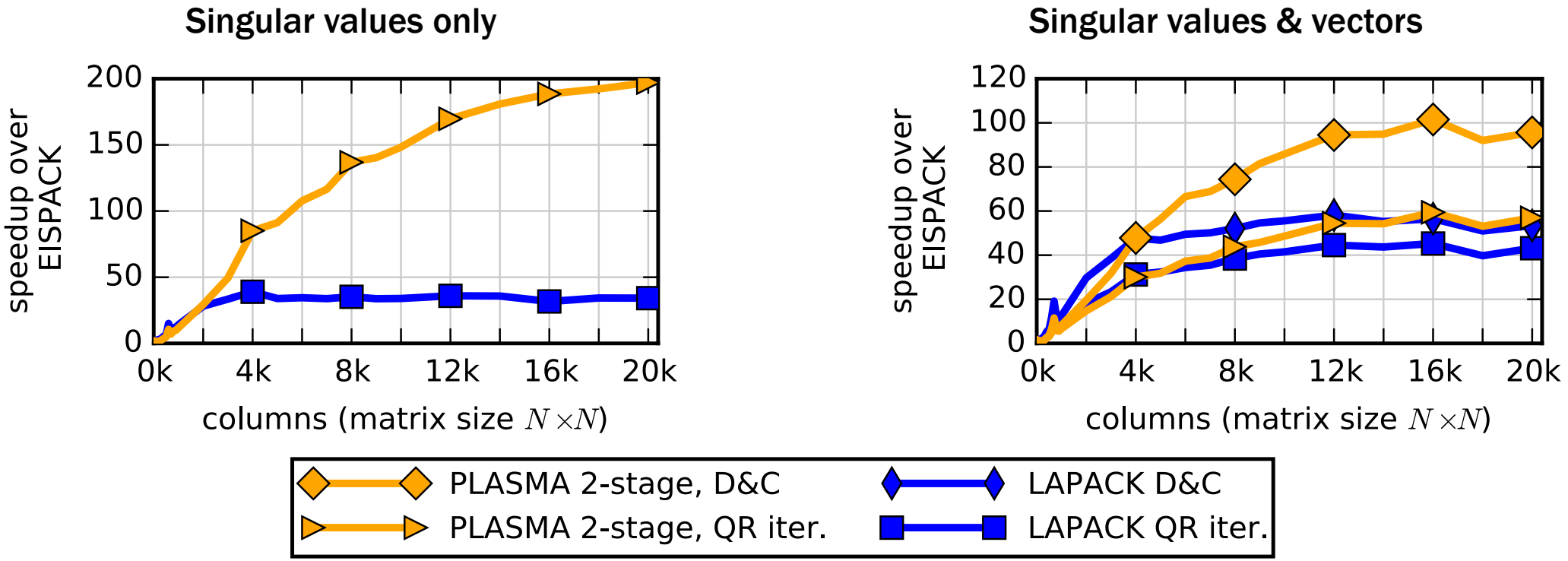
1. Reduction to bidiagonal form (2-stage)
   a. Reduce to band
   b. Band to bidiagonal (bulge chasing)
3. But adds U_b, V_b, doubling work in computing singular vectors



3. $U = U_a \; U_b \; U_2$

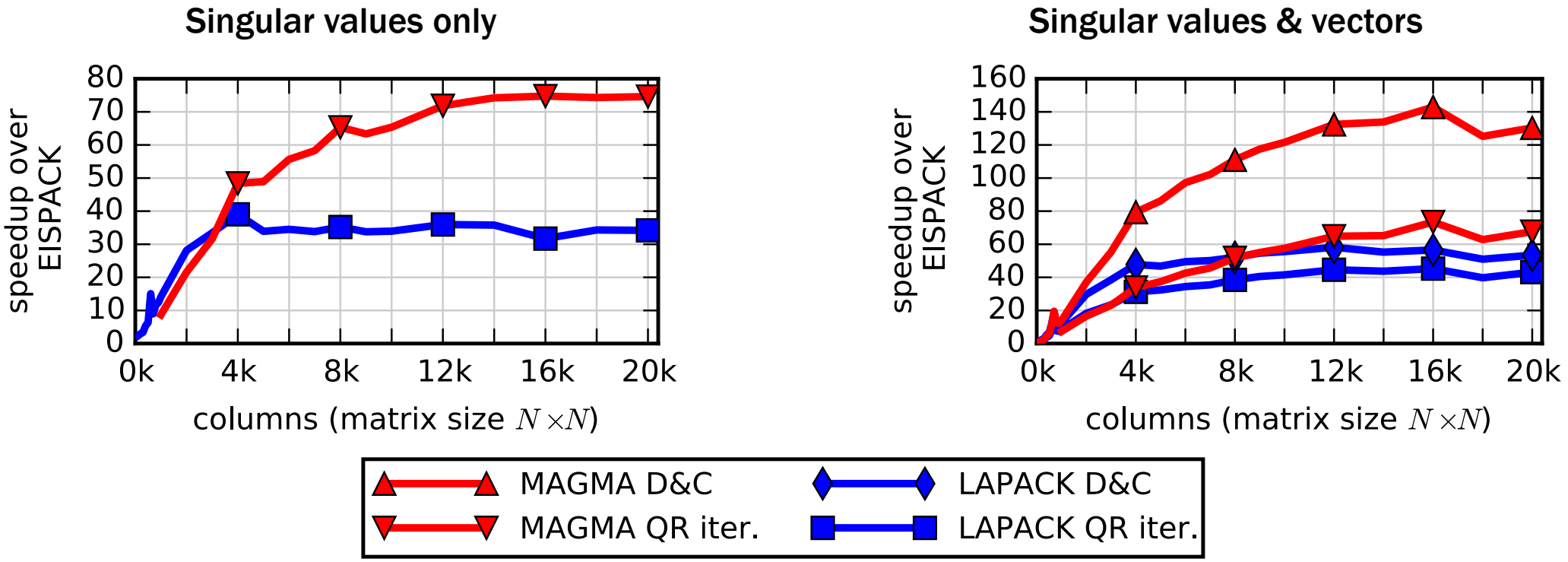
$V = V_a \; V_b \; V_2$

PLASMA



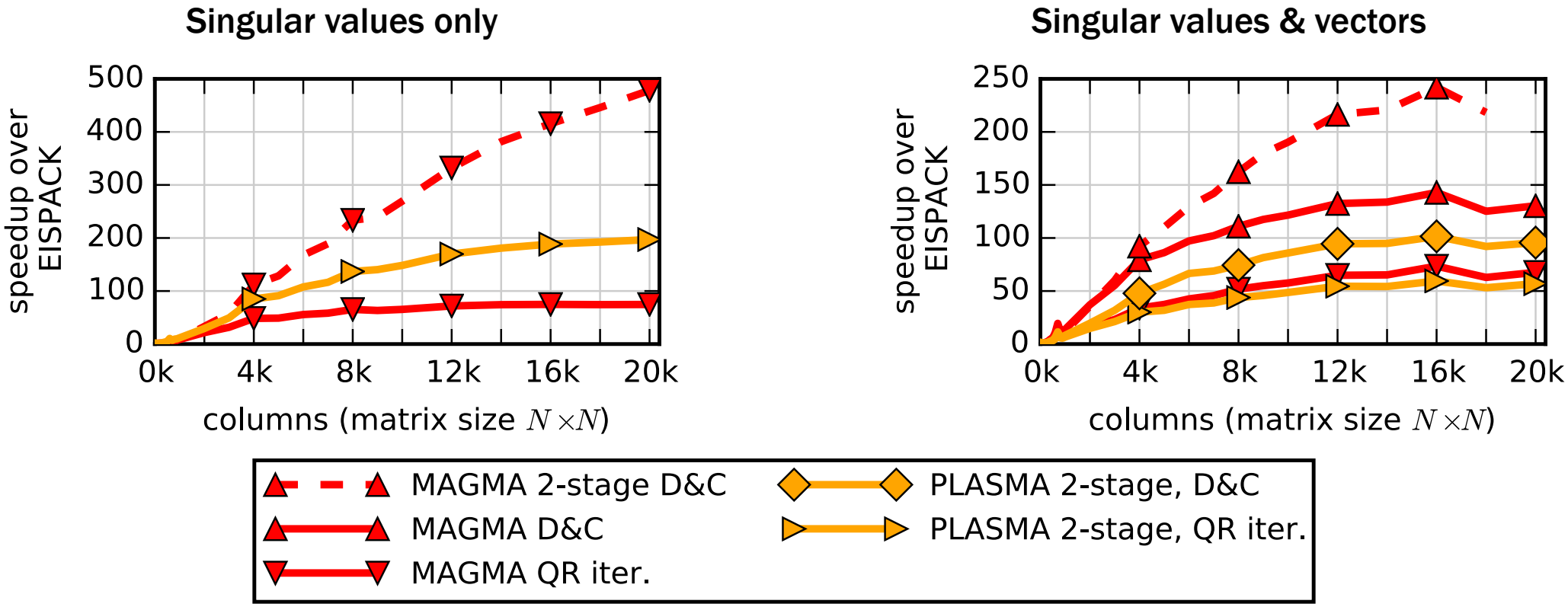
- 2-stage bidiagonal reduction
- Tile algorithm with task scheduler (DAG)

MAGMA



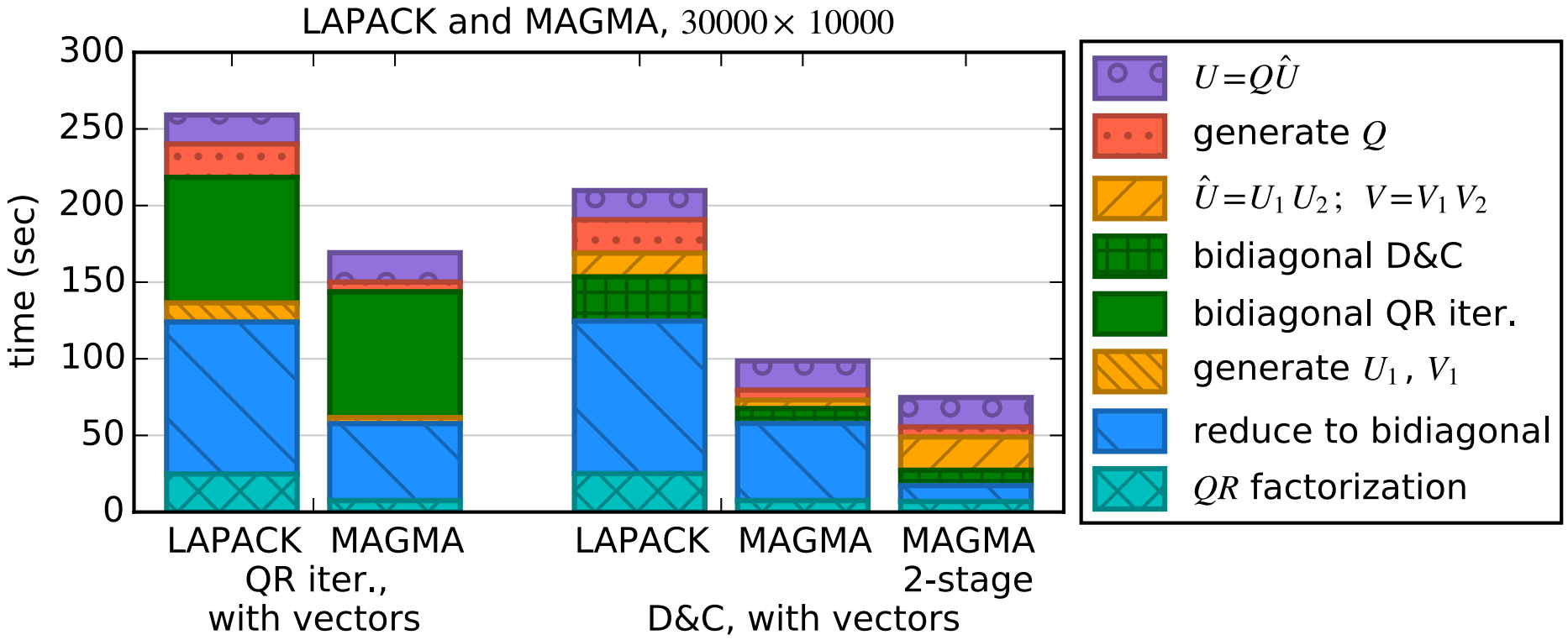
- GPU accelerated 1-stage bidiagonal reduction
- GPU accelerated divide & conquer

MAGMA 2-stage



- GPU accelerate 1st stage reduction to band
- Uses PLASMA 2nd stage (band to bidiagonal)

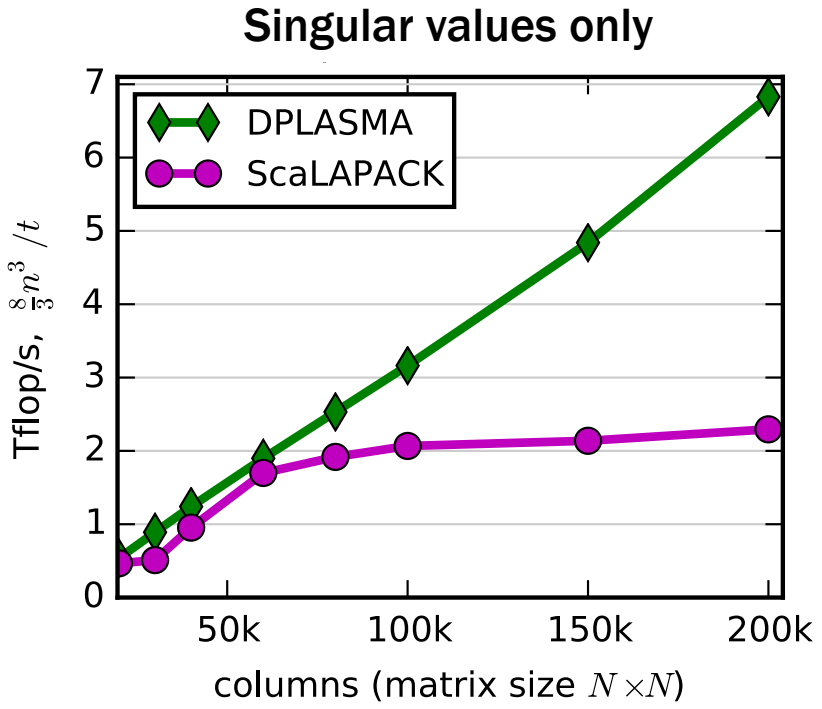
LAPACK and MAGMA Profile



- Need to accelerate as many phases as possible

Distributed: ScaLAPACK & DPLASMA

- DPLASMA uses 2-stage PLASMA algorithm with ParSEC runtime
- Uses LAPACK QR iteration for solve



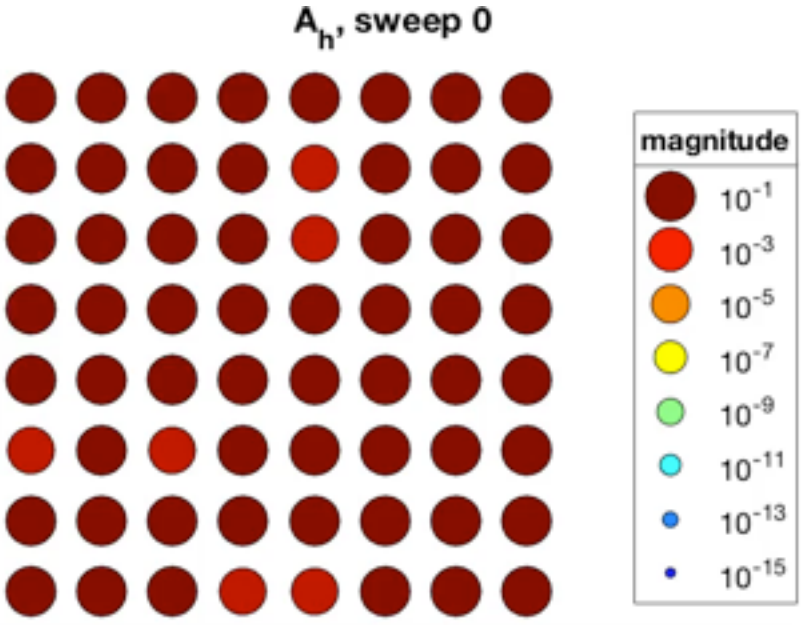
Results on 49 nodes, each 2x18 core 2.6 GHz Intel Broadwell, total 1764 cores.
Thanks to Intel for machine access.

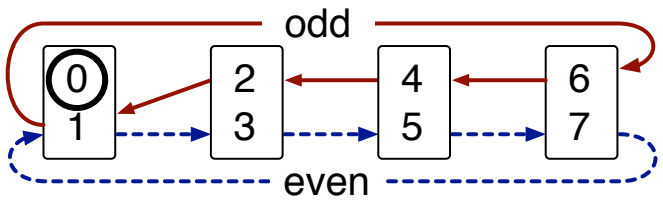
Two-sided Jacobi (Kogbetliantz)

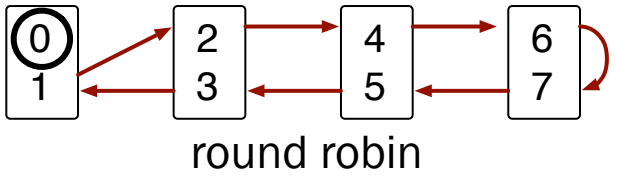
- Apply Jacobi rotations on both sides diagonalize A

$$A_{h+1} = J_h^T A_h K_h \rightarrow \Sigma \text{ as } h \rightarrow \infty$$

- until norm(offdiag(A)) < tol
  - for each pair (i, j)
    - J_h and K_h solve 2x2 SVD of $\begin{bmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{bmatrix}$
    - update rows i, j and cols i, j of A
- $n/2$ rotations can be done in parallel







One-sided Jacobi (Hestenes)

- Apply Jacobi rotations on right to implicitly solve eigenvalues of $A^{\mathrm{T}}A$

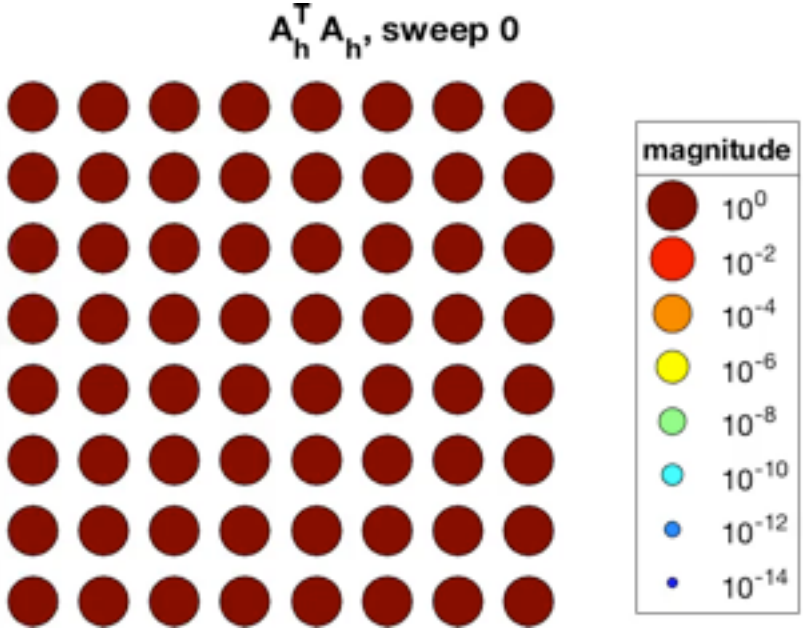
$$A_{h+1} = A_h J_h \to U\Sigma \text{ as } h \to \infty$$

$$A_h^T A_h \to \Sigma^2 \text{ as } h \to \infty$$

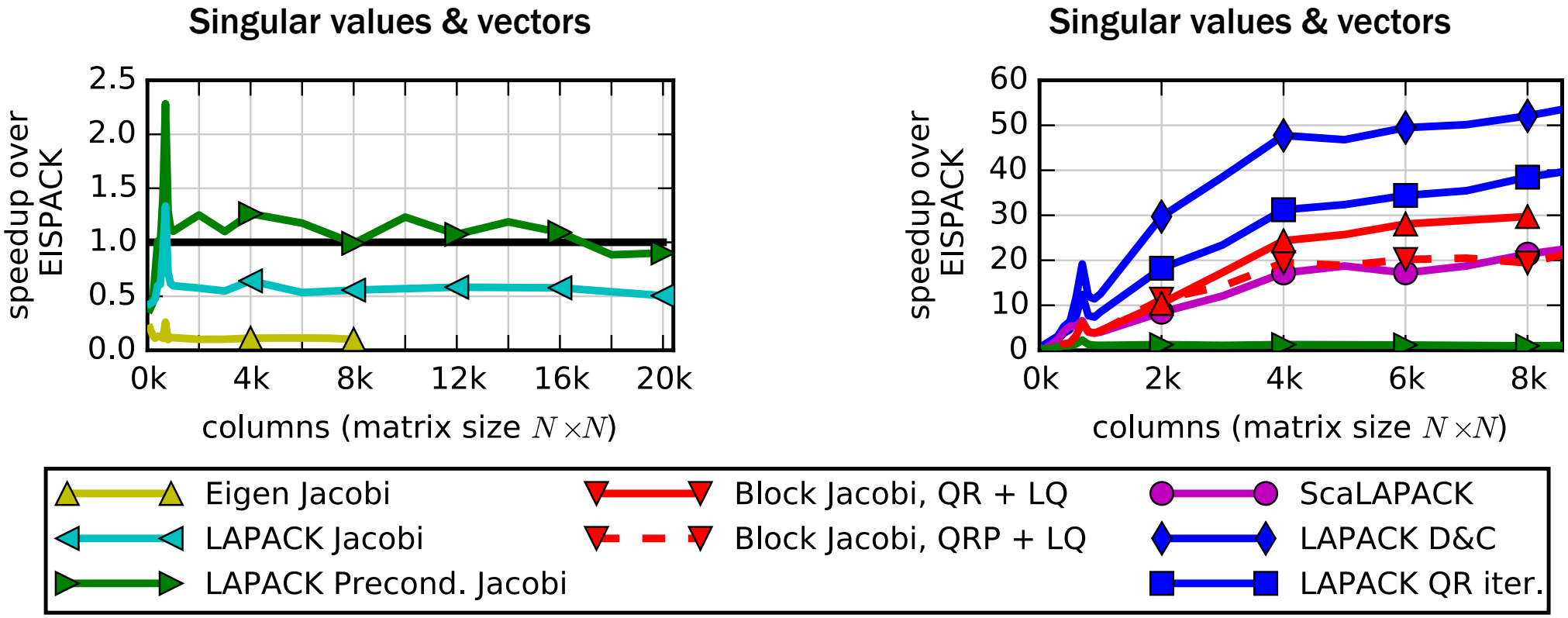
- until no changes
  - for each pair (i, j)
    - J_{h} solves 2x2 eigenvalue problem

$$\begin{bmatrix} b_{ii} & b_{ij} \\ b_{ij} & b_{jj} \end{bmatrix} \text{ where } b_{ij} = a_i^T a_j$$

    - update cols i, j of A
- $n/2$ rotations can be done in parallel



Jacobi & Block Jacobi

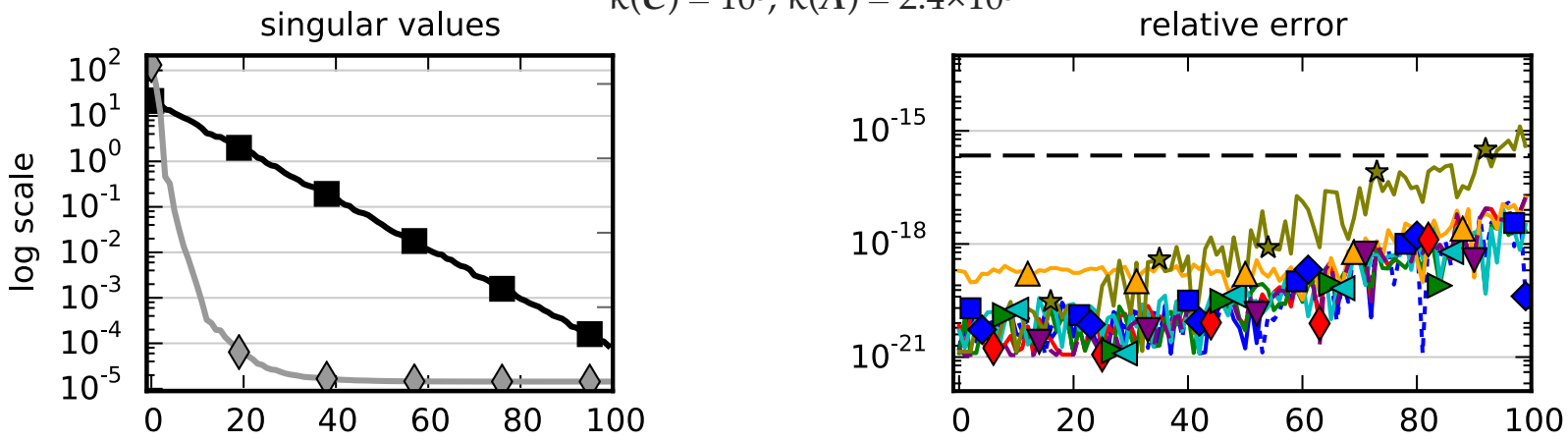


- Serial Jacobi in LAPACK (gejsv and gesvj; Drmač & Veselić)
- Parallel block Jacobi (Bečka, Okša, & Vajteršic)

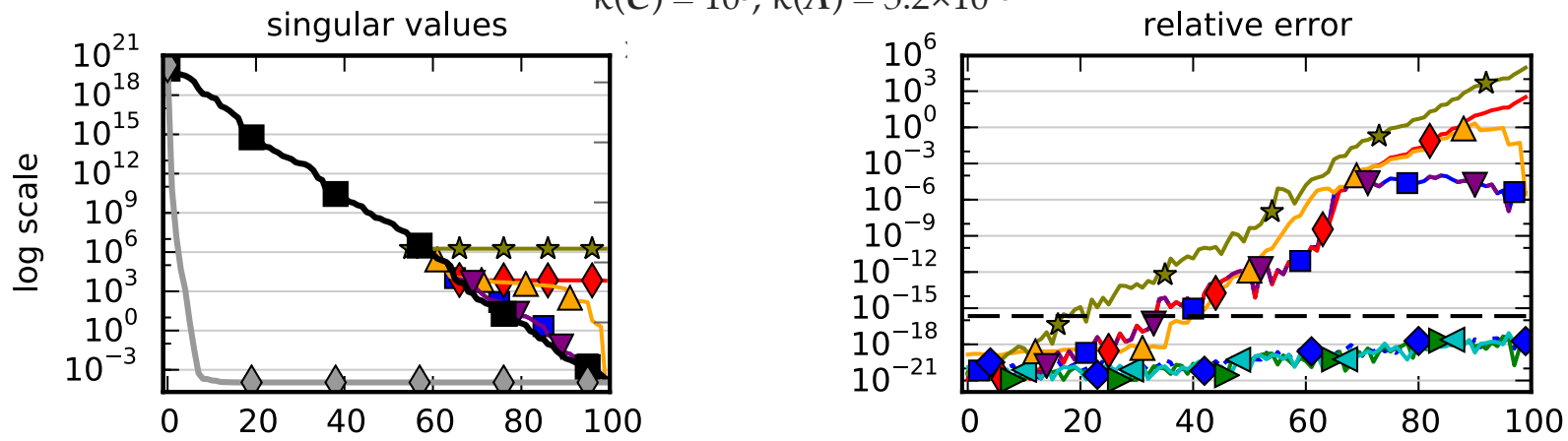
Accuracy

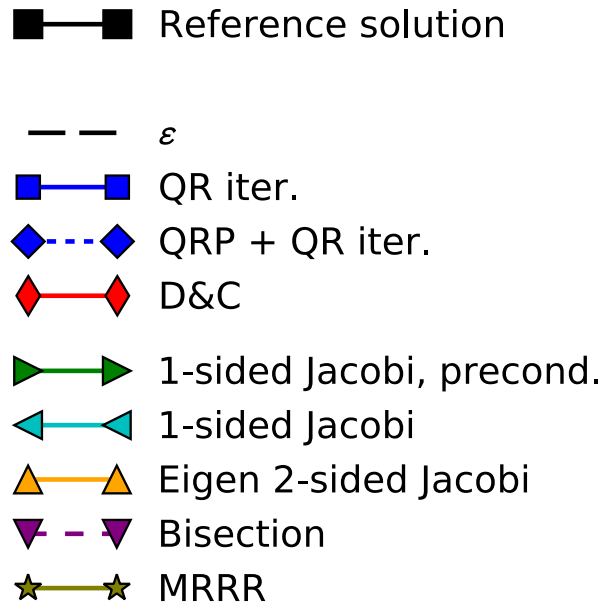
- $A = CD$, where D diagonal scaling, $\kappa(A) \gg \kappa(C)$

$\kappa(C) = 10^5$, $\kappa(A) = 2.4\times10^5$



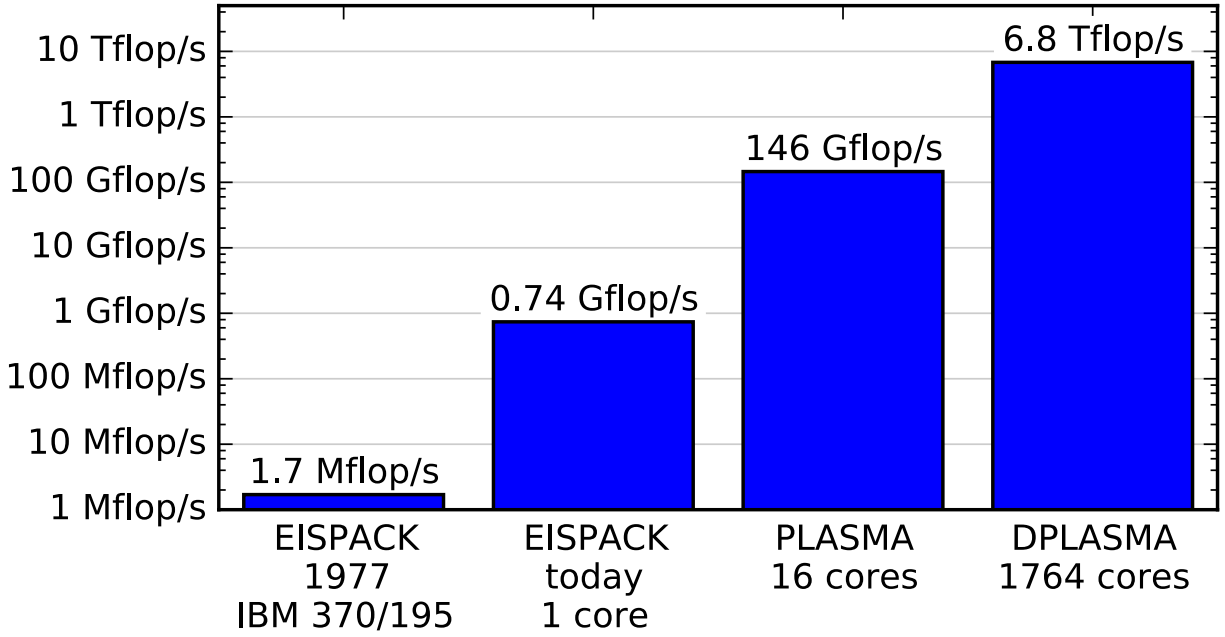
$\kappa(C) = 10^5$, $\kappa(A) = 5.2\times10^{23}$





Historical perspective, 1977 to today

- EISPACK is 435× faster
- PLASMA is 85 000× faster, or 197× EISPACK today
- DPLASMA is 4 000 000× faster, or 9189× EISPACK today
- Increase problem size from n = 80 to n = 200 000



Summary

- Bidiagonalization
  - Use of Level 3 BLAS is critical for performance
    - LAPACK blocked algorithm
    - PLASMA 2-stage reduction
    - Divide & conquer
  - Operation count ≠ time
    - More operations can be faster!
- Jacobi
  - Basic version is easy, parallel, accurate, but slow (Level 1 BLAS)
  - Block Jacobi can be competitive