Outline

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

- Finite elements
  - Partial differential equation (PDE)
  - Discretization yields $Ax = b$
- Nonlinear solve
  - Newton’s Method to solve nonlinear system $f(x) = 0$
  - Linearize: $x_{k+1} = x_k - J(x_k)^{-1} f(x_k)$ with Jacobian $J(x)$ of $f(x)$
  - Solve $J(x_k) s_k = f(x_k)$ then $x_{k+1} = x_k - s_k$
- Optimization
  - BFGS to find minimum of $f(x)$
  - $x_{k+1} = x_k - \alpha H_k^{-1} \nabla f(x_k)$ with approximate Hessian $H_k$ of $f(x)$
  - Near minimum, $H_k$ is SPD
Problems

• Eigenvalues
  • Air flow over bridges
  • Airplane design
  • Chemical engineering reaction-diffusion
  • Molecular resonance
  • Integrated circuit design

• SVD
  • Principal component analysis
  • Latent semantic indexing
  • Image processing
  • Compression
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

• \( Ax \approx b \)
  • \( A \) 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, ...

- \( C = \alpha AB + \beta C \) matrix-matrix multiply (BLAS)
  - General, symmetric, Hermitian, triangular, ...

\[ A \]
\[ x = \lambda x \]

\[ A = U \Sigma V^T \]

\[ U, V \text{ unitary (orthogonal)} \]

\[ C = \alpha A B + \beta C \]
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

```
A
A
```

- Dense but low rank, numerically compressible
  - Hierarchical \(H\) matrices

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

no solution: meet in 3 parallel lines

infinitely many solutions: meet in 1 line

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

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

\[
\begin{bmatrix}
2 & 3 & 5 \\
3 & -2 & 5 \\
-5 & -1 & -10
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z
\end{bmatrix}
= 
\begin{bmatrix}
-4 \\
-4 \\
8
\end{bmatrix}
\]
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:** \( \|x\|_\infty = \max_i(|x_i|) \)

\[
\begin{align*}
\|v\|_2 &= \sqrt{|x|^2 + |y|^2 + |z|^2} = \sqrt{6^2 + 2^2 + 3^2} = 7 \\
\|v\|_1 &= |x| + |y| + |z| = 6 + 2 + 3 = 11 \\
\|v\|_\infty &= \max(|x|, |y|, |z|) = \max(6, 2, 3) = 6
\end{align*}
\]
Matrix norms

- Induced by vector norm:
  - \( \|A\|_p = \max_{\|x\|_p=1} \|Ax\|_p \)
    - Maximum stretching of a vector
  - Instances
    - \( \|A\|_2 = \sigma_{\text{max}} \) 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
    - \( \|A\|_{\text{max}} = \max_{i,j} |A_{i,j}| \)
Norms

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

- Some matrix norms also satisfy
  - $\|AB\| \leq \|A\| \cdot \|B\|$ submultiplicativc
  - Induced norms, Frobenius norm
  - Not max norm
• 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.
Condition number

• For solving $Ax = b$,

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

• 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

<table>
<thead>
<tr>
<th></th>
<th>bits</th>
<th>precision</th>
<th>$\epsilon$ (2 × unit roundoff)</th>
<th>underflow (min normal)</th>
<th>overflow (max)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bfloat16</td>
<td>16</td>
<td>8 bits</td>
<td>$\approx$ 2 digits</td>
<td>$7.81 \times 10^{-3}$</td>
<td>$1.18 \times 10^{-38}$</td>
</tr>
<tr>
<td>half</td>
<td>16</td>
<td>11 bits</td>
<td>$\approx$ 3 digits</td>
<td>$9.77 \times 10^{-4}$</td>
<td>$6.10 \times 10^{-5}$</td>
</tr>
<tr>
<td>single (float)</td>
<td>32</td>
<td>24 bits</td>
<td>$\approx$ 7 digits</td>
<td>$1.19 \times 10^{-7}$</td>
<td>$1.18 \times 10^{-38}$</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>53 bits</td>
<td>$\approx$ 16 digits</td>
<td>$2.22 \times 10^{-16}$</td>
<td>$2.23 \times 10^{-308}$</td>
</tr>
<tr>
<td>double-double</td>
<td>64x2</td>
<td>107 bits</td>
<td>$\approx$ 32 digits</td>
<td>$1.23 \times 10^{-32}$</td>
<td>$2.23 \times 10^{-308}$</td>
</tr>
<tr>
<td>quad</td>
<td>128</td>
<td>113 bits</td>
<td>$\approx$ 34 digits</td>
<td>$1.93 \times 10^{-34}$</td>
<td>$3.36 \times 10^{-4932}$</td>
</tr>
</tbody>
</table>

- 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 $A_{ij} = A_{ji}$
  • $\lambda$ are real, eigenvectors are orthogonal (if real-symmetric)

• Hermitian
  • $A = A^H$ (conjugate-transpose)
  • Entries $A_{ij} = \text{conj}(A_{ji})$
  • Implies diagonal is real
  • $\lambda$ are real (even if $A$ is complex), eigenvectors are unitary
Matrix types: symmetric / Hermitian

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

$$H = \begin{bmatrix}
\frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\
\frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y^2}
\end{bmatrix}$$
Matrix types: orthogonal / unitary

- Orthogonal (unitary if complex)
  - Rotation, reflection, or combination
  - Columns are orthogonal to each other, and unit length
    
    \[ Q^T_{i:j} Q_{i:j} = \begin{cases} 
    0 & \text{if } i \neq j \\
    1 & \text{if } i = j 
    \end{cases} \]

    or

    \[ Q^T Q = I \quad (Q^H Q = I \text{ if complex}) \]

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

• Givens rotation
  • Rotation with $\theta$ chosen to eliminate $x_2$
    \[ Q = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix}, \quad Q \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \alpha \\ 0 \end{bmatrix} \]

• Householder reflection
  • Reflect vector $x$ onto axis
    \[ H = I - \tau vv^T, \quad v = x \pm \|x\|_2 e_1, \quad \tau = \frac{2}{vv^T} \]
    \[ H \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} \alpha \\ 0 \\ \vdots \\ 0 \end{bmatrix} \]

• Block reflector
  \[ H = H_k \cdots H_1 = I - VTV^T \]
Column-major vs. row-major

- **Col-major:** \( A( i, j ) = A[ i + j \times \text{lda} ] \), where \( \text{lda} \geq m \)
- **Row-major:** \( A( i, j ) = A[ i \times \text{lda} + j ] \), where \( \text{lda} \geq n \)
  - Leading dimension \( \text{lda} \)
  - \( \text{lda} \) 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 col-major
// access in Fortran. lda >= m
do j = 1, n
    do i = 1, m
        A(i,j) = ...
    continue
continue

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

// 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 ] = ...

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

```c
// 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

```plaintext
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)

```c
// 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, $A_{ij}B_{lj}$
- $C$ is $4 \times 6$, $A$ is $4 \times 4$, $B$ is $4 \times 6$

2D grouping:
- $2 \times 2 \times k$ blocks

3D grouping:
- $2 \times 2 \times 2$ blocks
Parallelization strategy

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

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

$C$ is $4 \times 6$, $A$ is $4 \times 4$, $B$ is $4 \times 6$

1D column grouping
$m \times 1 \times k$ blocks

1D row grouping
$1 \times n \times k$ blocks
Parallel matrix multiply: Cannon (1969)

- Square matrices $A$, $B$, $C$
- Simple block distribution, $N$ blocks x $N$ blocks (here, 3x3)
- Cycles $A$ and $B$ through nodes
- Post-processing step to return $A$ and $B$ to original nodes

- Diagrams for Cannon keep colors associated with original position; MPI ranks are fixed position
Parallel matrix multiply: Cannon (1969)

• Setup step
• Multiply \((k = 0)\)

// A, B, C are \(N\) blocks \(\times N\) 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 \(N-1\)
    // ... see next slide
    parallel for each \(i, j\)
    // by current position of blocks!
    \(C(i, j) += A(i, j) \times B(i, j)\)
Parallel matrix multiply: Cannon (1969)

- Shift
- Multiply \( (k = 1) \)

// Multiplication step
for \( k = 0 \) to \( N-1 \)
if \( (k > 0) \)
    shift blocks of \( A \) right 1 col, mod \( N \)
    shift blocks of \( B \) down 1 row, mod \( N \)
parallel for each \( i, j \)
// by current position of blocks!
\[ C(i, j) += A(i, j) \times B(i, j) \]
Parallel matrix multiply: Cannon (1969)

- Shift
- Multiply \((k = 2)\)

```
// Multiplication step
for k = 0 to N-1
  if (k > 0)
    shift blocks of A right 1 col, mod N
    shift blocks of B down 1 row, mod N
  parallel for each i, j
    // by current position of blocks!
    C(i, j) += A(i, j) * B(i, j)
```
Parallel matrix multiply: Fox (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

- Diagrams for Fox keep colors associated with original position; MPI ranks are fixed position

\[
\begin{array}{ccc}
A_{00} & A_{01} & A_{02} \\
A_{10} & A_{11} & A_{12} \\
A_{20} & A_{21} & A_{22} \\
\end{array}
\quad
\begin{array}{ccc}
B_{00} & B_{01} & B_{02} \\
B_{10} & B_{11} & B_{12} \\
B_{20} & B_{21} & B_{22} \\
\end{array}
\quad
\begin{array}{ccc}
C_{00} & C_{01} & C_{02} \\
C_{10} & C_{11} & C_{12} \\
C_{20} & C_{21} & C_{22} \\
\end{array}
\]

MPI ranks

\[
\begin{array}{ccc}
0 & 3 & 6 \\
1 & 4 & 7 \\
2 & 5 & 8 \\
\end{array}
\]
Parallel matrix multiply: Fox (1987)

- Broadcast A
- Multiply \( k = 0 \)
- Shift B (next slide)

```plaintext
// A, B, C are N blocks x N blocks
for k = 0 to N-1
    for i = 0 to N-1
        Bcast A(i, (i+k) mod N) across 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: Fox (1987)

• Broadcast A
• Multiply (k = 1)
• Shift B (next slide)

// A, B, C are N blocks x N blocks
for k = 0 to N-1
    for i = 0 to N-1
        Bcast A(i, (i+k) mod N) across 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: Fox (1987)

- Broadcast A
- Multiply (k = 2)
- Shift B (not shown) returns it to original state

// A, B, C are N blocks x N blocks
for k = 0 to N-1
  for i = 0 to N-1
    Bcast A(i, (i+k) mod N) across 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 (1994)

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

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

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

$C = \begin{bmatrix}
  C_{00} & C_{01} & C_{02} \\
  C_{10} & C_{11} & C_{12} \\
  C_{20} & C_{21} & C_{22}
\end{bmatrix}$
Parallel matrix multiply: SUMMA (1994)

- Broadcast col 0 of A
- Broadcast row 0 of B
- Multiply

\[
\begin{array}{ccc}
C_{00} &=& A_{00} \cdot B_{00} \\
C_{01} &=& A_{10} \cdot B_{00} \\
C_{02} &=& A_{20} \cdot B_{00} \\
C_{10} &=& A_{00} \cdot B_{01} \\
C_{11} &=& A_{10} \cdot B_{01} \\
C_{12} &=& A_{20} \cdot B_{01} \\
C_{20} &=& A_{00} \cdot B_{02} \\
C_{21} &=& A_{10} \cdot B_{02} \\
C_{22} &=& A_{20} \cdot B_{02} \\
\end{array}
\]

// C is M x N, A is M x K, 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: SUMMA (1994)

- Broadcast col 1 of A
- Broadcast row 1 of B
- Multiply

// C is M x N, A is M x K,
// 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: SUMMA (1994)

- Broadcast col 2 of A
- Broadcast row 2 of B
- Multiply

// C is M x N, A is M x K, 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

\[ x_i = \left( b_j - A_{i,1:i-1} x_{1:i-1} \right) / A_{ii} \]

\[ x_{1:i-1} = b_i \]

\[ b_{i+1:n} = b_{i+1:n} - x_j A_{i+1:n} \]

- Cost: $n^2$ flops
Triangular solve (trsv/trsm)

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

```
\begin{align*}
\begin{array}{c}
\text{“block dot product” ($\mathbf{x}^T\mathbf{y}$) implementation} \\
A_{i,1:i-1} x_i + x_i = b_{i} \quad \left\{ \begin{array}{c} \\
\text{portion already solved at step } i \\
\end{array} \right. \\
A_i x_i = \left( b_i - A_{i,1:i-1} x_{1:i-1} \right) \\
\end{array}
\end{align*}
```

```
\begin{align*}
\begin{array}{c}
\text{“block axpy” ($\mathbf{y} = \alpha\mathbf{x} + \mathbf{y}$) implementation} \\
A_{i,i+1:n} x_i + x_i = b_{i} \quad \left\{ \begin{array}{c} \\
\text{portion already solved at step } i \\
\end{array} \right. \\
A_i x_i = \left( b_i - \mathbf{x}_i A_{i,i+1:n} \right) \\
\end{array}
\end{align*}
```

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

“block dot product” ($\mathbf{x}^T\mathbf{y}$) implementation

“block axpy” ($\mathbf{y} = \alpha\mathbf{x} + \mathbf{y}$) 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 )
```

Store multipliers in lower triangle; they form $L$ matrix
LU factorization (Gaussian elimination)

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

```c
// 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?

\[
A = \begin{bmatrix} 0 & 5 \\ 2 & 4 \end{bmatrix} \quad A = \begin{bmatrix} \epsilon & 5 \\ 2 & 4 \end{bmatrix}
\]

- 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

```c
// 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 $Ax = (P^T LU)x = b$ 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

```c
// 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

```c
// 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 \( P^T L U x = b \)
- `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 (LLᵀ)**

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

```plaintext
// 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, k ) * A( k+1:n, k )\^T
```

In (Sca)LAPACK:
- `posv` solves $Ax = b$ by calling:
  - `potrf` factors $A = LLᵀ$
  - `potrs` solves $LLᵀx = b$
- `posvx` is an expert version
- `dsposv / zcposv` are mixed precision versions
- `potri` computes $A^{-1}$ from $A = LLᵀ$
\( \text{LDL}^T \)

- 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 \( A = \text{LDL}^T \)
  - `sytrs/hetrs*` solves \( \text{LDL}^T x = b \)
- `sysvx/hesvx` is an expert version
- `sytri*` computes \( A^{-1} \) from \( A = \text{LDL}^T \)

* several versions available.
Not in ScaLAPACK. No mixed precision.
Least squares (over-determined)

- Solve $Ax \approx b$, where $A$ is tall ($m \geq n$)
  - Minimize residual $\|r\|_2 = \|b - Ax\|_2$
- QR
  - $QRx \approx b$
  - $X = R^{-1} Q^T b$, 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
  - $U \Sigma V^T x \approx b$
  - $x = V \Sigma^{-1} U^T b$, inverse of $\Sigma$ 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
  - $A = LQ = (\hat{QR})^T$ where $\hat{QR} = A^T$
  - $LQx = b$
  - $x = Q^T L^{-1} b$
- SVD
  - Same as before
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

**CPU:** (no work)

**GPU:**

```
1  1  1  2  2  2  3  3  3  4  4  4  5  5  5  6  6  7
```
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

65
Execution trace

• Hybrid CPU–GPU

CPU 0:  
GPU 0 (s0):  
Time (sec): 0.00 0.12
Legend: larfb panel sync

• GPU–only

CPU 0:  
GPU 0 (s0):  
Time (sec): 0.00
Legend: larfb panel

(NOTE: outdated results.)
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

CPU 0:
GPU 0 (s0):
Time (sec): 0.00
Legend: gemv | panel | trail

• GPU–only

CPU 0:
GPU 0 (s0):
Time (sec): 0.00
Legend: gemv | panel | trail
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 trace

- Hybrid CPU–GPU
  - CPU 0
  - GPU 0 (s0)
  - Time (sec): 0.00
  - Legend: gemv, lahru, panel

- GPU-only
  - CPU 0
  - GPU 0 (s0)
  - Time (sec): 0.00
  - Legend: gemv, lahru, panel
Results: Hessenberg

- GPU-only similar to Hybrid

![Graphs showing performance comparison between Hybrid CPU-GPU, GPU-only, and CPU-only (MKL) for different matrix dimensions and operations (single, double, single-complex, double-complex).]
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
What is the SVD?

- $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices of left and right singular vectors.
- $\mathbf{\Sigma}$ 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

$$V, \text{ left singular vectors} \quad U\Sigma, \text{ right singular vectors}$$
What is the SVD?

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

$V$, left singular vectors

$U\Sigma$, right singular vectors
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

Bunny from Stanford 3D Scanning Repository
Computing the SVD

Dense SVD methods

Bidiagonalization
- Reduction
  - 1 stage
  - 2 stage
- Solver
  - QR iteration
  - Divide and conquer
  - MRRR
  - Bisection

Jacobi
- 1 sided
- 2 sided
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

\[ A = QR \]
\[ A = U_1BV_1^T \]
\[ B = U_2\Sigma V_2^T \]

3. \[ U = Q \quad U_1 \quad U_2 \]
\[ V = V_1 \quad V_2 \]
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
QR iteration

• Iteratively apply Givens rotations on left & right to reduce off-diagonal terms
• $O(n^2)$ singular values only, $\sim 12n^3$ with vectors
• 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
• 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 ($n \approx 25$) 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**

- 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 $V$ and $q$ for $U$
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

$$A = U_aCV_a^T$$

$$C = U_bBV_b^T$$

$$B = U_2\Sigma V_2^T$$

3. $U = U_a U_b U_2$

3. $V = V_a V_b V_2$
PLASMA

- 2-stage bidiagonal reduction
- Tile algorithm with task scheduler (DAG)
• 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 \( \text{norm( offdiag( A ))} < \text{tol} \)
  - 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

\[ A_n, \text{ sweep 0} \]

\[ \text{magnitude} \]
<table>
<thead>
<tr>
<th>( 10^{-1} )</th>
<th>( 10^{-3} )</th>
<th>( 10^{-5} )</th>
<th>( 10^{-7} )</th>
<th>( 10^{-9} )</th>
<th>( 10^{-11} )</th>
<th>( 10^{-13} )</th>
<th>( 10^{-15} )</th>
</tr>
</thead>
</table>

\[ \begin{array}{cccc}
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
\end{array} \]

round robin
One-sided Jacobi (Hestenes)

- Apply Jacobi rotations on right to implicitly solve eigenvalues of $A^T A$
  \[ A_{h+1} = A_h J_h \rightarrow U \Sigma \text{ as } h \rightarrow \infty \]
  \[ A_h^T A_h \rightarrow \Sigma^2 \text{ as } h \rightarrow \infty \]
- 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, \( \kappa(A) \gg \kappa(C) \)

\[ \kappa(C) = 10^5, \kappa(A) = 2.4 \times 10^5 \]

\[ \kappa(C) = 10^5, \kappa(A) = 5.2 \times 10^{23} \]
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