Linear Algebra

Lecture 2
Jack Dongarra

Tuning for Caches

1. Preserve locality.
2. Reduce cache thrashing.
3. Loop blocking when out of cache.
4. Software pipelining.

Indirect Addressing

\[
\begin{align*}
&d = 0 \\
&\text{do } i = 1,n \\
&\quad j = \text{ind}(i) \\
&\quad d = d + \sqrt{x(i)^2 + y(j)^2 + z(i)^2} \\
&\text{end do}
\end{align*}
\]

- Change loop statement to

\[
\begin{align*}
&d = d + \sqrt{r(1,j)^2 + r(2,j)^2 + r(3,j)^2} \\
&\text{end do}
\end{align*}
\]

- Note that \(r(1,j), r(3,j)\) are in contiguous memory and probably are in the same cache line (\(d\) is probably in a register and is irrelevant). The original form uses 3 cache lines at every instance of the loop and can cause cache thrashing.

Optimizing Matrix Addition for Caches

- Dimension A(n,n), B(n,n), C(n,n)
- A, B, C stored by column (as in Fortran)
- Algorithm 1:
  - for \(i=1:n\), for \(j=1:n\), \(A(i,j) = B(i,j) + C(i,j)\)
- Algorithm 2:
  - for \(j=1:n\), for \(i=1:n\), \(A(i,j) = B(i,j) + C(i,j)\)
- What is “memory access pattern” for Algs 1 and 2?
- Which is faster?
- What if A, B, C stored by row (as in C)?

Loop Fusion Example

/* Before */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
    \[ a[i][j] = 1/b[i][j] * c[i][j]; \]
    \[ d[i][j] = a[i][j] + c[i][j]; \]
/* After */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
    \[ a[i][j] = 1/b[i][j] * c[i][j]; \]
    \[ d[i][j] = a[i][j] + c[i][j]; \]

2 misses per access to \(a\) \& \(c\) vs. one miss per access; improve spatial locality

Improving Ratio of Floating Point Operations to Memory Accesses

subroutine mult(n1,nd1,n2,nd2,y,a,x)
implicit real*8 (a-h,o-z)
dimension a(nd1,nd2),y(nd2),x(nd1)
do 10, i=1,n1
  \[ t=0.d0 \]
do 20, j=1,n2
  \[ t=t+a(j,i)*x(j) \]
10 \[ y(i)=t \]
return
end

**** 2 FLOPS
**** 2 LOADS
Improving Ratio of Floating Point Operations to Memory Accesses

Works correctly when \( n_1, n_2 \) are multiples of 4

dimension \( a(n_1, n_2), y(n_2), x(n_1) \)

\[
\text{do } i = 1, n_1 - 3, 4 \\
t1 = 0.0 \\
t2 = 0.0 \\
t3 = 0.0 \\
t4 = 0.0 \\
\text{do } j = 1, n_2 - 3, 4 \\
t1 = t1 + a(j+0,i+0)*x(j+0) + a(j+1,i+0)*x(j+1) + a(j+2,i+0)*x(j+2) + a(j+3,i+1)*x(j+3) \\
t2 = t2 + a(j+0,i+1)*x(j+0) + a(j+1,i+1)*x(j+1) + a(j+2,i+1)*x(j+2) + a(j+3,i+0)*x(j+3) \\
t3 = t3 + a(j+0,i+2)*x(j+0) + a(j+1,i+2)*x(j+1) + a(j+2,i+2)*x(j+2) + a(j+3,i+2)*x(j+3) \\
t4 = t4 + a(j+0,i+3)*x(j+0) + a(j+1,i+3)*x(j+1) + a(j+2,i+3)*x(j+2) + a(j+3,i+3)*x(j+3) \\
\text{enddo} \\
y(i+0) = t1 \\
y(i+1) = t2 \\
y(i+2) = t3 \\
y(i+3) = t4 \\
\text{enddo}
\]

32 FLOPS
20 LOADS

Optimizing Matrix Multiply for Caches

- Several techniques for making this faster on modern processors
  - heuristically studied
- Some optimizations done automatically by compiler, but can do much better
- In general, you should use optimized libraries (often supplied by vendor) for this and other very common linear algebra operations
  - BLAS = Basic Linear Algebra Subroutines
- Other algorithms you may want are not going to be supplied by vendor, so need to know these techniques

Using a Simple Model of Memory to Optimize

- Assume just 2 levels in the hierarchy, fast and slow
- All data initially in slow memory
  - \( m \) = number of memory elements (words) moved between fast and slow memory
  - \( t_m \) = time per slow memory operation
  - \( f \) = number of arithmetic operations
  - \( t_f \) = time per arithmetic operation << \( t_m \)
  - \( q = f/m \) = average number of flops per slow memory access

Minimum possible time = \( f \cdot t_f \) when all data in fast memory

Actual time

\[
= f \cdot t_f \cdot (1 + t_m / t_f \cdot 1/q)
\]

- Larger \( q \) means time closer to minimum \( f \cdot t_f \)
- \( q \geq 2 \cdot t_m / t_f \) needed to get at least half of peak speed

Warm up: Matrix-vector multiplication

\( y = y + A \cdot x \)

- \{read \( x(1:n) \) into fast memory\}
- \{read \( y(1:n) \) into fast memory\}

for \( i = 1:n \)

- \{read row \( i \) of \( A \) into fast memory\}
- \( y(i) = y(i) + A(i,:) \cdot x(j) \)
- \{write \( y(1:n) \) back to slow memory\}

\( m = \) number of slow memory refs = \( 3 \cdot n + n^2 \)
\( f = \) number of arithmetic operations = \( 2 \cdot n^2 \)
\( q = m / t_f \approx 2 \)

Matrix-vector multiplication limited by slow memory speed

\[
\text{Machine Balance: Key to algorithm efficiency}
\]

\[
\text{Computational Intensity: Key to algorithm efficiency}
\]

Matrix Multiply \( C = C + A \cdot B \)

for \( i = 1 \) to \( n \)

- \{read \( A(i,:) \) into fast memory\}
- \{read \( B(:,j) \) into fast memory\}
- \( C(i,j) = C(i,j) + A(i,k) \cdot B(k,j) \)

\( C(i,j) \) = \( C(i,j) \) + \( A(i,k) \) * \( B(k,j) \)

Warm up: Matrix-vector multiplication

\( y = y + A \cdot x \)

\{read \( x(1:n) \) into fast memory\}
\{read \( y(1:n) \) into fast memory\}

for \( i = 1:n \)

- \{read row \( i \) of \( A \) into fast memory\}
- \( y(i) = y(i) + A(i,:) \cdot x(j) \)
- \{write \( y(1:n) \) back to slow memory\}

- \m = number of slow memory refs = \( 3 \cdot n + n^2 \)
- \( f = \) number of arithmetic operations = \( 2 \cdot n^2 \)
- \( q = m / t_f \approx 2 \)

Matrix-vector multiplication limited by slow memory speed
Matrix Multiply \( C = C + A \times B \)

(unblocked, or untiled)

for \( i = 1 \) to \( n \)

\{ read row \( i \) of \( A \) into fast memory \}

for \( j = 1 \) to \( n \)

\{ read \( C(i,j) \) into fast memory \}

\{ read column \( j \) of \( B \) into fast memory \}

for \( k = 1 \) to \( n \)

\[ C(i,j) = C(i,j) + A(i,k) \times B(k,j) \]

\{ write \( C(i,j) \) back to slow memory \}

\[ \begin{array}{c}
\text{number of slow memory references on unblocked matrix multiply} \\
m = n^3 \\
\end{array} \]

\[ \begin{array}{c}
\text{read each column of } B \text{ } n \text{ times} \\
\text{read each row of } A \text{ once for each } i \\
\text{read and write each element of } C \text{ once} \\
\end{array} \]

\[ q = \frac{\text{ops}}{\text{slow mem ref}} \]

Matrix Multiply (blocked, or tiled)

Consider \( A,B,C \) to be \( N \) by \( N \) matrices of \( b \) by \( b \) subblocks where \( b = \frac{n}{N} \) is called the blocksize.

for \( i = 1 \) to \( N \)

\{ read block \( C(i,j) \) into fast memory \}

for \( j = 1 \) to \( N \)

\{ read block \( A(i,k) \) into fast memory \}

\{ read block \( B(k,j) \) into fast memory \}

\[ C(i,j) = C(i,j) + A(i,k) \times B(k,j) \] (do a matrix multiply on blocks)

\{ write block \( C(i,j) \) back to slow memory \}

\[ \begin{array}{c}
\text{number of slow memory references on blocked matrix multiply} \\
m = N^2 \times n \\
\text{read each block of } B \text{ } N^3 \times (\frac{n}{N} \times \frac{n}{N}) \times N \\
\text{read each block of } A \text{ } N^3 \times (\frac{n}{N} \times \frac{n}{N}) \times N \\
\text{read and write each block of } C \text{ once} \\
\end{array} \]

\[ q = \frac{\text{ops}}{\text{slow mem ref}} \]

Cache Blocking

Looping over the blocks

\[ \begin{array}{c}
\text{do } kk = 1 \text{, } n\text{, blk} \\
\text{do } jj = 1 \text{, } n\text{, blk} \\
\text{do } ii = 1 \text{, } n\text{, blk} \\
\text{do } k = kk \ldots kk+nblk-1 \\
\text{do } j = jj \ldots jj+nblk-1 \\
\text{do } i = ii \ldots ii+nblk-1 \\
\end{array} \]

\[ c(i,j) = c(i,j) + a(i,k) \times b(k,j) \]

end do

end do

end do

Strassen's Matrix Multiply

- The traditional algorithm (with or without tiling) has \( O(n^3) \) flops
- Strassen discovered an algorithm with asymptotically lower flops
  - \( O(n^{2.81}) \)
- Consider a 2x2 matrix multiply, normally 8 multiplies and 4 additions

\[ \begin{array}{c}
\text{Strassen formulation does 7 multiplies and 18 additions} \\
\text{Extend to run by divide and conquer} \\
\end{array} \]

\[ \begin{array}{c}
\text{Theorem (Hong, Kung, 1981): Any reorganization of this algorithm} \\
\text{that uses only associativity is limited to } q = \Theta(\sqrt{M}) \\
\end{array} \]
Strassen algorithm (1)

- Matrix multiplication algorithms
- Reduction of multiplication number in 2 x 2 matrices product
  - Strassen: 7 products and 18 additions
  - Classic algorithm: 8 products and 4 additions
- $O(2^{log_7^2}) = O(2^{2.807})$ complexity (recursively)
- Applicable to 2 x 2 block matrices

Strassen algorithm (cont’d) (Corrected slide 3/2/09)

Phase 1 (temporary sums)

\[
\begin{align*}
T1 &= A11 + A22 \\
T2 &= A21 + A22 \\
T3 &= A11 + A12 \\
T4 &= A21 - A11 \\
T5 &= A12 - A22 \\
T6 &= B11 + B22 \\
T7 &= B12 - B22 \\
T9 &= B11 + B12 \\
T10 &= B21 + B22
\end{align*}
\]

Phase 2 (temporary products)

\[
\begin{align*}
Q1 &= T1 * T6 \\
Q2 &= T2 * B11 \\
Q3 &= A11 * T7 \\
Q4 &= A22 * T8 \\
Q5 &= T3 * B22 \\
Q6 &= T4 * T9 \\
Q7 &= T5 * T10
\end{align*}
\]

Phase 3

\[
\begin{align*}
C11 &= Q1 + Q4 - Q5 + Q7 \\
C12 &= Q3 + Q5 \\
C21 &= Q2 + Q4 \\
C22 &= Q1 - Q2 + Q3 + Q6
\end{align*}
\]

Strassen’s task graph

Strassen (continued)

\[
T(n) = \text{Cost of multiplying } n \times n \\text{ matrices}
= 7 T(n/2) + 18 n^2
= O(n^{\log_2^7})
= O(n^{2.81})
\]

- Available in several libraries
- Up to several time faster if n large enough (100s)
- Needs more memory than standard algorithm
- Can be less accurate because of roundoff error
- Current world’s record is $O(n^{2.376})$

BLAS
Memory Hierarchy

- Key to high performance in effective use of memory hierarchy
- True on all architectures

Array Libraries

- Vector and matrix operations appear over and over again in many applications.
  - Simple linear algebra kernels such as vector, matrix-vector, matrix-matrix multiply
- More complicated algorithms can be built from these basic kernels.
- Standards for optimization and portibility
  - The libraries are supposed to be optimised for each particular computer
- One of the most well-known and well-designed array libraries is the Basic Linear Algebra Subprograms (BLAS)
  - Provides basic array operations for numerical linear algebra
  - Available for most modern systems
- Led to portable libraries for vector and shared memory parallel machines.

Level 1, 2 and 3 BLAS

- Level 1 BLAS
  - Vector-Vector operations
- Level 2 BLAS
  - Matrix-Vector operations
- Level 3 BLAS
  - Matrix-Matrix operations

BLAS

- BLAS is an acronym for Basic Linear Algebra Subroutines.
- The source code for BLAS is available through Netlib.
  - Many computer vendors will have a special version of BLAS tuned for maximal speed and efficiency on their computer.
  - This is one of the main advantages of BLAS: the calling sequences are standardized so that programs that call BLAS will work on any computer that has BLAS installed.
  - If you have a fast version of BLAS, you will also get high performance on all programs that call BLAS.
  - Hence BLAS provides a simple and portable way to achieve high performance for calculations involving linear algebra. LAPACK is a higher-level package built on the same ideas.
- The BLAS subroutines can be divided into three levels:
  - Level 1: Vector-vector operations. $O(n)$ data and $O(n)$ work.
  - Level 2: Matrix-vector operations. $O(n^2)$ data and $O(n^2)$ work.
  - Level 3: Matrix-matrix operations. $O(n^3)$ data and $O(n^3)$ work.

Why Higher Level BLAS?

- Can only do arithmetic on data at the top of the hierarchy
- Higher level BLAS lets us do this

<table>
<thead>
<tr>
<th>BLAS</th>
<th>Memory Refs</th>
<th>Flops</th>
<th>Flops/ Memory Refs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1</td>
<td>$3n$</td>
<td>$2n$</td>
<td>$2/3$</td>
</tr>
<tr>
<td>Level 2</td>
<td>$n^2$</td>
<td>$2n^2$</td>
<td>$2$</td>
</tr>
<tr>
<td>Level 3</td>
<td>$4n^2$</td>
<td>$2n^2$</td>
<td>$n/2$</td>
</tr>
<tr>
<td>$C=C+AB$</td>
<td>$4n^2$</td>
<td>$2n^2$</td>
<td>$n/2$</td>
</tr>
</tbody>
</table>
**Level 1 BLAS**

- Operate on vectors or pairs of vectors
  - perform \( O(n) \) operations;
  - return either a vector or a scalar.
- **saxpy**
  - \( y(i) = a \cdot x(i) + y(i) \), for \( i = 1 \) to \( n \).
  - \( s \) stands for single precision, \( daxpy \) is for double precision, \( caxpy \) for complex, and \( zaxpy \) for double complex,
- **sscal** \( y = a \cdot x \), for scalar \( a \) and vectors \( x,y \)
- **sdot** computes \( s = \sum_{i=1}^{n} x(i) \cdot y(i) \)

**Level 2 BLAS**

- Operate on a matrix and a vector;
  - return a matrix or a vector;
  - \( O(n^2) \) operations
- **sgemv**: matrix-vector multiply
  - \( y = y + A \cdot x \)
  - where \( A \) is \( m \times n \), \( x \) is \( n \times 1 \) and \( y \) is \( m \times 1 \).
- **sger**: rank-one update
  - \( A = A + y \cdot x^T \), i.e., \( A(i,j) = A(i,j) + y(i) \cdot x(j) \)
  - where \( A \) is \( m \times n \), \( y \) is \( m \times 1 \), \( x \) is \( n \times 1 \),
- **strsv**: triangular solve
  - solves \( y = T \cdot x \) for \( x \), where \( T \) is triangular

**Level 2 BLAS**

- To store matrices, the following schemes are used
  - Column-based and row-based storage
  - Packed storage for symmetric or triangular matrices
  - Band storage for band matrices
- Conventional storage
  - An \( m \times n \) matrix \( A \) is stored in a one-dimensional array \( a \)
    - \( a_{ij} \rightarrow a[1+j \cdot s] \) (C, column-wise storage)
  - If \( s = n \), rows (columns) will be contiguous in memory
  - If \( s > n \), there will be a gap of \( (s-n) \) memory elements
    - between two successive rows (columns)
  - Only significant elements of symmetric/triangular matrices need be set

**Level 2 BLAS**

- Other routines of Level 2 perform the following
  - \( y \leftarrow A \cdot x + b \) where \( A = A^T \)
  - \( x \leftarrow A \cdot x \) or \( x \leftarrow A \cdot x \) where \( A \) is triangular
  - \( y \leftarrow b \cdot y \) or \( w \leftarrow A \cdot w \) where \( A \) is triangular

- For any matrix-vector operation with a specific matrix operand (triangular, symmetric, banded, etc.), there is a routine for each storage scheme that can be used to store the operand
Level 3 BLAS

- Operate on pairs or triples of matrices
  - returning a matrix;
  - complexity is $O(n^3)$.
- sgemm: Matrix-matrix multiplication
  - $C = C + A*B$,
  - where $C$ is m-by-n, $A$ is m-by-k, and $B$ is k-by-n
- strsm: multiple triangular solve
  - solves $Y = T*X$ for $X$,
  - where $T$ is a triangular matrix, and $X$ is a rectangular matrix.

**BLAS/LAPACK Naming Conventions**

- Routine names
  - 5/6 character name: $xYZZZ$
    - $x$ – data type and precision
      - S – real single precision
      - D – real double precision
      - C – complex single precision
      - Z – complex double precision
    - YY – matrix type
      - GE – general rectangular
      - SY – symmetric
      - HE – hermitian
      - TR – triangular
      - GB – general banded
      - SB – symmetric banded
      - HB – hermitian banded
      - TB – triangular banded
      - SP – symmetric Packed
      - HP – hermitian Packed
      - TP – triangular packed
  - ZZZ – operation type
    - MV – matrix-vector multiply
    - MM – matrix-matrix multiply
    - SV – solve on a vector
    - SM – solve on a matrix
    - R – rank update
    - R2 – symmetric rank update
    - R2K – parametrized symmetric rank update
- Parameters
  - X, Y – vectors
  - A, B, C – matrices
  - N, M, K – dimensions
  - LDA, LDB, LDC – leading dimensions
  - ALPHA, BETA – scalars
  - SIDE, TRANS, UPLO, DIAG – operation details

**Parameter Naming Conventions**

- Vector names
  - X, Y
- Vector strides
  - INCX, INCY
- Matrix names
  - A, B, C
- Matrix leading dimensions
  - LDA, LDB, LDC
- Matrix and/or vector dimensions
  - M, N, K
- Scalars
  - ALPHA, BETA

**BLAS Names**

- The first letter of the subroutine name indicates the precision used:
  - S Real single precision
  - D Real double precision
  - C Complex single precision
  - Z Complex double precision
- BLAS 1
  - xCOPY – copy one vector to another
  - xSWAP – swap two vectors
  - xSCAL – scale a vector by a constant
  - xAXPY – add a multiple of one vector to another
  - xDOT – inner product
  - xASUM – 1-norm of a vector
  - xNRM2 – 2-norm of a vector
  - xAMAX – find maximal entry in a vector
- BLAS 2
  - xGEMV – general matrix-vector multiplication
  - xGER – general rank-1 update
  - xSYR2 – symmetric rank-2 update
  - xTRSV – solve a triangular system of equations
- BLAS 3
  - xGEMM – general matrix-matrix multiplication
  - xSYMM – symmetric matrix-matrix multiplication
  - xSYR – symmetric rank-1 update
  - xSYRK – symmetric rank-2k update
Leading Dimension: What and Why

- Leading dimension allows specification of submatrices
- Right choice of leading dimension can increase performance
- Wrong choice of leading dimension can degrade performance
  - Beware of powers of 2
    - Most common reason for memory bank conflicts
  - Vector increment is an extreme case of leading dimension
- Interaction with Fortran 90/95/03
  - Copying of assume-shaped arrays

M

\[ LDA \geq M \]

A(i,j), LDA, m

Packed Storage

- The relevant triangle of a symmetric/triangular matrix is packed by columns or rows in a one-dimensional array
- The upper triangle of an \( nn \) matrix \( A \) may be stored in a one-dimensional array \( A \)

\[ a_{ij} \Rightarrow a[j+i*2*(n-i-1)/2] \] (C, row-wise storage)

Example.

\[
\begin{pmatrix}
  a_{00} & a_{01} & a_{02} \\
  0 & a_{11} & a_{12} \\
  0 & 0 & a_{22}
\end{pmatrix}
\]

Band Storage

- Band storage
  - A compact storage scheme for band matrices
  - Consider Fortran and a column-wise storage scheme

An \( nn \) band matrix \( A \) with \( l \) subdiagonals and \( u \) superdiagonals may be stored in a 2-dimensional array \( A \) with \( 2l+u+1 \) rows and \( n \) columns

- Columns of matrix \( A \) are stored in corresponding columns of array \( A \)
- Diagonals of matrix \( A \) are stored in rows of array \( A \)

\[ a_{ij} \Rightarrow A(u+i-j,j) \] for \( \max(0,j-u) \leq i \leq \min(m-1,j+l) \)

Example.

\[
\begin{pmatrix}
  a_0 & a_1 & a_2 & 0 & 0 \\
  a_1 & a_2 & a_3 & 0 & 0 \\
  a_2 & a_3 & a_4 & 0 & 0 \\
  0 & a_2 & a_3 & a_4 & 0 \\
  0 & 0 & a_3 & a_4 & a_5 \\
  0 & 0 & 0 & a_4 & a_5 \\
  0 & 0 & 0 & 0 & a_5 
\end{pmatrix} \Rightarrow
\]

BLAS -- References

- BLAS software and documentation can be obtained via:
  - WWW: http://www.netlib.org/blas,
  - (anonymous) ftp ftp.netlib.org: cd blas; get index
  - email netlib@www.netlib.org with the message: send index from blas

- Comments and questions can be addressed to: lapack@cs.utk.edu

BLAS Papers

Performance of BLAS

- BLAS are specially optimized by the vendor
- Big payoff for algorithms that can be expressed in terms of the BLAS3 instead of BLAS2 or BLAS1.
- The top speed of the BLAS3
- Algorithms like Gaussian elimination organized so that they use BLAS3

How To Get Performance From Commodity Processors?

- Today’s processors can achieve high-performance, but this requires extensive machine-specific hand tuning.
- Routines have a large design space w/many parameters
  - blocking sizes, loop nesting permutations, loop unrolling depths, software pipelining strategies, register allocations, and instruction schedules.
  - Complicated interactions with the increasingly sophisticated microarchitectures of new microprocessors.
- A few months ago no tuned BLAS for Pentium for Linux.
- Need for quick/dynamic deployment of optimized routines.
- ATLAS - Automatic Tuned Linear Algebra Software
  - PhiPac from Berkeley

Optimizing in practice

- Tiling for registers
  - loop unrolling, use of named “register” variables
- Tiling for multiple levels of cache
- Exploiting fine-grained parallelism within the processor
  - super scalar
  - pipelining
- Complicated compiler interactions
- Hard to do by hand (but you’ll try)
- Automatic optimization an active research area
  - PHIPAC: www.icsi.berkeley.edu/~bilmes/phipac
  - www.cs.berkeley.edu/~iyer/asci_slides.ps
  - ATLAS: www.netlib.org/atlas/index.html

Adaptive Approach for Level 3

- Do a parameter study of the operation on the target machine, done once.
- Only generated code is on-chip multiply
- BLAS operation written in terms of generated on-chip multiply
- All transpose cases coerced through data copy to 1 case of on-chip multiply
  - Only 1 case generated per platform

\[
\begin{bmatrix}
  M & \bullet & \bullet \\
  \bullet & \bullet & \bullet \\
  \bullet & \bullet & N
\end{bmatrix} \quad \rightarrow \quad
\begin{bmatrix}
  M & \bullet & \bullet \\
  \bullet & \bullet & \bullet \\
  \bullet & \bullet & N
\end{bmatrix} \quad \times \quad
\begin{bmatrix}
  \bullet & \bullet & \bullet \\
  \bullet & \bullet & \bullet \\
  \bullet & \bullet & \bullet
\end{bmatrix}
\]

ATLAS

- Keep a repository of kernels for specific machines.
- Develop a means of dynamically downloading code
- Extend work to allow sparse matrix operations
- Extend work to include arbitrary code segments
- See: http://www.netlib.org/atlas/
Implementation

Installation: self-tuning

Code generation (C)

Timing

Algorithm selection

Runtime: decision based on data

Small, contiguous data

No-copy code

Otherwise

Code with data copy

Performance does not depend on data, unless:
- Special numerical properties exist
- Diagonal dominance for LU factorization (5%-10% speed improvement)
- NaNs, INFs in the vector/matrix

Code Generation Strategy

- On-chip multiply optimizes for:
  - TLB access
  - L1 cache reuse
  - FP unit usage
  - Memory fetch
  - Register reuse
  - Loop overhead minimization
  - Takes a couple of hours to run.

- Code is iteratively generated & timed until optimal case is found. We try:
  - Differing NBs
  - Breaking false dependencies
  - M, N and K loop unrolling

Supported Hardware

- RISC (Reduced Instr. Set C.)
  - POWERX
  - MIPS
  - xSPARC X
  - HP/DEC/Digital Alpha

- CISC
  - AMD 32/64-bit
  - Intel 32/64-bit
  - HyperThreading™
  - Multi-core (Duo, Trio, Q...)

- VLIW (Very Long Instruction Word)
  - Itanium
  - Itanium 2

- Not supported (= bad performance)
  - Vector CPUs
    - It is slow but it works
    - Vector compilers cannot understand Atlas-generated C code

Recursive Approach for Other Level 3 BLAS

- Recur down to L1 cache block size
- Need kernel at bottom of recursion
- Use gemm-based kernel for portability

Multi-Threaded DGEMM

Intel PIII 550 MHz

Gaussian Elimination Basics

Solve \( A x = b \)

Step 1

\[ A = LU \]

Step 2

Forward Elimination

Solve \( L y = b \)

Step 3

Backward Substitution

Solve \( U x = y \)

Note: Changing RHS does not imply to recompute LU factorization
Gaussian Elimination

Standard Way
subtract a multiple of a row

Overwrite A with L and U
The lower part of A has a representation of "L"

Gaussian Elimination (GE) for Solving $Ax=b$

- Add multiples of each row to later rows to make A upper triangular
- Solve resulting triangular system $Ux = c$ by substitution

Refine GE Algorithm (1)

Initial Version
- For each column $i$
  - Zero it out below the diagonal by adding multiples of row $i$ to later rows
• For each row $j$ below row $i$
  - Add a multiple of row $i$ to row $j$
    - $tmp = A(j,i)$
    - For $k = i$ to $n$
      - $A(j,k) = A(j,k) - (tmp/A(i,i)) * A(i,k)$

Refine GE Algorithm (2)

Last version
- For $i = 1$ to $n-1$
  - For $j = i+1$ to $n$
    - $m = A(j,i)/A(i,i)$
    - For $k = i$ to $n$
      - $A(j,k) = A(j,k) - m * A(i,k)$

Refine GE Algorithm (3)

Last version
- For $i = 1$ to $n-1$
  - For $j = i+1$ to $n$
    - $m = A(j,i)/A(i,i)$
    - For $k = i+1$ to $n$
      - $A(j,k) = A(j,k) - m * A(i,k)$

Refine GE Algorithm (4)

Last version
- For $i = 1$ to $n-1$
  - For $j = i+1$ to $n$
    - $m = A(j,i)/A(i,i)$
    - For $k = i+1$ to $n$
      - $A(j,k) = A(j,k) - A(j,i) * A(i,k)$

Split Loop
Refine GE Algorithm (5)

- Last version

Express using matrix operations (BLAS)

for \( i = 1 \) to \( n-1 \)
\[
A(i+1:n,i) = A(i+1:n,i) \cdot (1 / A(i,i))
\]
\[
A(i+1:n,i+1:n) = A(i+1:n,i+1:n) - A(i+1:n,i) \cdot A(i,i+1:n)
\]

What GE really computes

- Call the strictly lower triangular matrix of multipliers \( M \), and let \( L = I + M \)
- Call the upper triangle of the final matrix \( U \)
- **Lemma (LU Factorization):** If the above algorithm terminates (does not divide by zero) then \( A = L^\top U \)

Solving \( A\mathbf{x} = \mathbf{b} \) using GE

- Factorize \( A = L^\top U \) using GE (cost = \( \frac{2}{3} n^3 \) flops)
- Solve \( L\mathbf{y} = \mathbf{b} \) for \( \mathbf{y} \), using substitution (cost = \( n^2 \) flops)
- Solve \( U\mathbf{x} = \mathbf{y} \) for \( \mathbf{x} \), using substitution (cost = \( n^2 \) flops)

Thus \( A\mathbf{x} = (L^\top U)\mathbf{x} = L^\top(U\mathbf{x}) = L^\top\mathbf{y} = \mathbf{b} \) as desired

Pivoting in Gaussian Elimination

- \( A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \) fails completely because can’t divide by \( A(1,1) = 0 \)
- But solving \( \mathbf{Ax} = \mathbf{b} \) should be easy!
- When diagonal \( A(i,i) \) is tiny (not just zero), algorithm may terminate but get completely wrong answer
- Numerical instability
- Roundoff error is cause
- **Cure:** Pivot (swap rows of \( A \)) so \( A(i,i) \) large

Problems with basic GE algorithm

- What if some \( A(i,j) \) is zero? Or very small?
  - Result may not exist, or be “unstable”, so need to pivot
- Current computation all BLAS 1 or BLAS 2, but we know that BLAS 3 (matrix multiply) is fastest (earlier lectures...)

Converting BLAS2 to BLAS3 in GEPP

- Blocking
  - Used to optimize matrix-multiplication
  - Harder here because of data dependencies in GEPP
- **BIG IDEA:** Delayed Updates
  - Save updates to “trailing matrix” from several consecutive BLAS2 updates
  - Apply many updates simultaneously in one BLAS3 operation
- Same idea works for much of dense linear algebra
  - Open questions remain?
- First Approach: Need to choose a block size \( b \)
  - Algorithm will save and apply \( b \) updates
  - \( b \) must be small enough so that active submatrix consisting of \( b \) columns of \( A \) fits in cache
  - \( b \) must be large enough to make BLAS3 fast

Gaussian Elimination with Partial Pivoting (GEPP)

- Partial Pivoting: swap rows so that \( A(i,i) \) is largest in column

for \( i = 1 \) to \( n-1 \)
find and record \( k \) where \( \max_{i \leq j \leq n} |A(j,i)| \) ... i.e. largest entry in rest of column \( i \)
if \( A(k,k) = 0 \)
exit with a warning that \( A \) is singular, or nearly so else if \( k \neq i \)
swap rows \( i \) and \( k \) of \( A \)
end if
\[
A(i+1:n,i) = A(i+1:n,i) / A(i,i)
\]
... each quotient lies in \([-1,1]\)
\[
A(i+1:n,i+1:n) = A(i+1:n,i+1:n) - A(i+1:n,i) \cdot A(i,i+1:n)
\]

• **Lemma:** This algorithm computes \( A = P^\top L^\top U \), where \( P \) is a permutation matrix.
• This algorithm is numerically stable in practice
• For details see LAPACK code at http://www.netlib.org/lapack/single/sgetf2.f

Level 1 BLAS

Level 2 BLAS

Level 3 BLAS
Blocked GEPP \((\text{www.netlib.org/lapack/single/sgetrf.f})\)

\[
\begin{align*}
\text{for } \text{ib} = 1 \text{ to } n-1 \text{ step } b & \quad \text{Process matrix } b \text{ columns at a time} \\
\text{end} = \text{ib} + b-1 & \quad \text{Point to end of block of } b \text{ columns} \\
\text{apply BLAS2 version of GEPP to get } A(\text{ib:end}, \text{ib:end}) = P^\prime \cdot L^\prime \cdot U^\prime & \\
\text{let } L \text{ denote the strict lower triangular part of } A(\text{ib:end}, \text{ib:end}) + I & \\
A(\text{ib:end}, \text{end+1:n}) = L^{-1} & \\
\text{apply delayed updates with single matrix-multiply} & \\
\text{with inner dimension } b &
\end{align*}
\]

Gaussian Elimination

- **Standard Way**: subtract a multiple of a row
- **LINPACK**: apply sequence to a column
- **LAPACK**: apply sequence to nb, then apply nb to rest of matrix

History of Block Partitioned Algorithms

- Early algorithms involved use of small main memory using tapes as secondary storage.
- Recent work centers on use of vector registers, level 1 and 2 cache, main memory, and “out of core” memory.

Blocked Partitioned Algorithms

- LU Factorization
- Cholesky factorization
- Symmetric indefinite factorization
- Matrix inversion
- QR, OL, RO, LQ factorizations
- Form Q or Q\(^T\)C

Derivation of Blocked Algorithms

**Cholesky Factorization** \(A = U^TU\)

\[
\begin{bmatrix}
A_{11} & a_{12} & a_{13} & \cdots \\
a_{12}^T & a_{22} & a_{23} & \cdots \\
a_{13}^T & a_{23}^T & a_{33} & \cdots \\
\vdots & \vdots & \vdots & \ddots
\end{bmatrix} =
\begin{bmatrix}
U_{11}^T & 0 & 0 & \cdots \\
0 & U_{22} & 0 & \cdots \\
0 & 0 & U_{33} & \cdots \\
\vdots & \vdots & \vdots & \ddots
\end{bmatrix}
\]

Equating coefficient of the \(j\)th column, we obtain

\[
\begin{align*}
a_j &= U_{j1}^T u_j \\
a_{jj} &= u_j^T u_j + u_j^T u_j
\end{align*}
\]

Hence, if \(U_{11}\) has already been computed, we can compute \(u_j\) and \(u_j\) from the equations:

\[
\begin{align*}
U_{j1}^T u_j &= a_j \\
\hat{u}_j^T = a_{jj} - u_j^T u_j
\end{align*}
\]
LINPACK Implementation

Here is the body of the LINPACK routine SPOFA which implements the method:

DO 30 J = 1, N
   INFO = J
   S = 0.0E0
   JM1 = J - 1
   IF( JM1.LT.1 ) GO TO 20
   DO 10 K = 1, JM1
      T = A( K, J ) - SDOT( K-1, A( 1, K ), 1, A( 1, J ), 1 )
      T = T / A( K, K )
      A( K, J ) = T
      S = S + T*T
   10 CONTINUE
   S = A( J, J ) - S
   C        ...EXIT
   IF( S.LE.0.0E0 ) GO TO 40
   A( J, J ) = SQRT( S )
30 CONTINUE

LAPACK Implementation

DO 10 J = 1, N
   CALL STRSV( 'Upper', 'Transpose', 'Non-Unit', J-1, A, LDA, A( 1, J ), 1 )
   S = A( J, J ) - SDOT( J-1, A( 1, J ), 1, A( 1, J ), 1 )
   IF( S.LE.ZERO ) GO TO 20
   A( J, J ) = SQRT( S )
10 CONTINUE

This change by itself is sufficient to significantly improve the performance on a number of machines.

From 238 to 312 Mflop/s for a matrix of order 500 on a Pentium 4-1.7 GHz.

However on peak is 1,700 Mflop/s.

Suggest further work needed.

Derivation of Blocked Algorithms

\[
\begin{bmatrix}
A_1 & A_2 & A_3 \\
A_2' & A_2'' & A_3'' \\
A_3' & A_3'' & A_3''
\end{bmatrix}
= \begin{bmatrix}
U_1' & 0 & 0 \\
U_1'' & U_2 & 0 \\
U_3 & U_3' & U_3''
\end{bmatrix}
\]

Equating coefficient of second block of columns, we obtain

\[
A_1 = U_1' U_1''
\]

\[
A_2 = U_1'' U_1''' + U_2''' U_2'''
\]

Hence, if \( U_2 \) has already been computed, we can compute \( U_3 \) as the solution of the following equations by a call to the Level 3 BLAS routine STRSM:

\[
U_1'' U_1''' = A_1
\]

\[
U_2''' U_2''' = A_2 - U_2'' U_2'''
\]

On Pentium 4, L3 BLAS squeezes a lot more out of 1 proc

<table>
<thead>
<tr>
<th>Proc</th>
<th>Linpack variant (L1B)</th>
<th>Rate of Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 proc</td>
<td>235 Mflop/s</td>
<td>239 Mflop/s</td>
</tr>
<tr>
<td>2 proc</td>
<td>316 Mflop/s</td>
<td>1262 Mflop/s</td>
</tr>
</tbody>
</table>

Gaussian Elimination via a Recursive Algorithm

F. Gustavson and S. Toledo

LU Algorithm:

1. Split matrix into two rectangles (m x n/2)
   if only 1 column, scale by reciprocal of pivot & return
2. Apply LU Algorithm to the left part
3. Apply transformations to right part
   (triangular solve \( \text{U}_n \) & \( \text{I}_n \)
   matrix multiplication \( \text{U}_{n-1} \text{U}_n - \text{U}_{n-1} \text{A} \)
4. Apply LU Algorithm to right part

Most of the work in the matrix multiply
Matrices of size \( n/2 \), \( n/4 \), \( n/8 \)

Recursive Factorizations

Just as accurate as conventional method

Same number of operations

Automatic variable-size blocking

Level 1 and 3 BLAS only

Extreme clarity and simplicity of expression

Highly efficient

The recursive formulation is just a rearrangement of the point-wise LINPACK algorithm

The standard error analysis applies (assuming the matrix operations are computed the “conventional” way).
Recursive Algorithms – Limits
- Two kinds of dense matrix compositions
  - One Sided
    - Sequence of simple operations applied on left of matrix
      - Gaussian Elimination: $A = L^*U$ or $A = P^*L^*U$
      - Symmetric Gaussian Elimination: $A = L^*D*L^T$
      - Cholesky: $A = L*L^T$
      - QR Decomposition for Least Squares: $A = Q*R$
      - Can be nearly 100% BLAS 3
      - Susceptible to recursive algorithms
  - Two Sided
    - Sequence of simple operations applied on both sides, alternating
      - Eigenvalue algorithms, SVD
    - At least ~25% BLAS 2
    - Seem impervious to recursive approach?
    - Some recent progress on SVD (25% vs 50% BLAS2)

ScaLAPACK
- Library of software dealing with dense & banded routines
- Distributed Memory - Message Passing
- MIMD Computers and Networks of Workstations
- Clusters of SMPs

Programming Style
- SPMD Fortran 77 with object based design
- Built on various modules
  - PBLAS Interprocessor communication
  - BLACS
    - PVM, MPI, IBM SP, CRI T3, Intel, TMC
    - Provides right level of notation.
    - BLAS
  - LAPACK software expertise/quality
    - Software approach
    - Numerical methods

Overall Structure of Software
- Object based - Array descriptor
  - Contains information required to establish mapping between a global array entry and its corresponding process and memory location.
  - Provides a flexible framework to easily specify additional data distributions or matrix types.
  - Currently dense, banded, & out-of-core
- Using the concept of context
PBLAS

- Similar to the BLAS in functionality and naming.
- Built on the BLAS and BLACS
- Provide global view of matrix
  
  ```
  CALL DGEXXX ( M, N, A( IA, JA ), LDA,... )
  CALL PDGEXXX( M, N, A, IA, JA, DESCA,... )
  ```

LAPACK and ScaLAPACK Status

- "One-sided Problems" are scalable
  - In Gaussian elimination, A factored into product of 2 matrices $A = LU$ by premultiplying $A$ by sequence of simpler matrices
  - Asymptotically 100% BLAS3
  - LU ("Linpack Benchmark")
  - Cholesky, QR
- "Two-sided Problems" are harder
  - A factored into product of 3 matrices by pre and post multiplication
  - Half BLAS2, not all BLAS3
  - Eigenproblems, SVD
  - Non-symmetric eigenproblem hardest
- Narrow band problems hardest (to do BLAS3 or parallelize)
  - Solving and eigenproblems
- www.netlib.org/lapack,scalapack