Linear Algebra

Lecture 2

Jack Dongarra

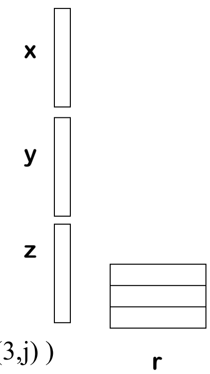
Tuning for Caches

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

2

Indirect Addressing

```
d = 0
  do i = 1,n
    j = ind(i)
    d = d + sqrt( x(j)*x(j) + y(j)*y(j) + z(j)*z(j) )
  end do
```



- Change loop statement to

  d = d + sqrt(r(1,j)*r(1,j) + r(2,j)*r(2,j) + r(3,j)*r(3,j))

- 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];
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
      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
20        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

```
c works correctly when n1,n2 are multiples of 4
   dimension a(nd1,nd2), y(nd2), x(nd1)
   do i=1,n1-3,4
     t1=0.d0
     t2=0.d0
     t3=0.d0
     t4=0.d0
     do j=1,n2-3,4
      t1=t1+a(j+0,i+0)*x(j+0)+a(j+1,i+0)*x(j+1)+
   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)+
   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)+
   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)+
   1     a(j+2,i+3)*x(j+2)+a(j+3,i+3)*x(j+3)
     enddo
     y(i+0)=t1
     y(i+1)=t2
     y(i+2)=t3
     y(i+3)=t4
   enddo
```

32 FLOPS
20 LOADS

Optimizing Matrix Multiply for Caches

- Several techniques for making this faster on modern processors
  - heavily 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

***Computational Intensity:* Key to algorithm efficiency**

- Minimum possible time = $f * t_f$ when all data in fast memory
- Actual time
  - $f * t_f + m * t_m = f * t_f * (1 + \boxed{t_m/t_f} * 1/q)$

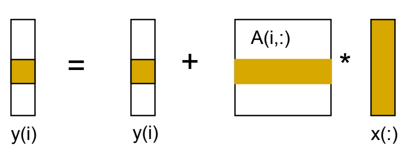
***Machine Balance:*Key to machine efficiency**

- Larger q means time closer to minimum $f * t_f$
  - q [illegible] t_m/t_f needed to get at least half of peak speed

q: flops/memory reference

Warm up: Matrix-vector multiplication y = y + A*x

```
for i = 1:n
        for j = 1:n
                y(i) = y(i) + A(i,j)*x(j)
```



Warm up: Matrix-vector multiplication
y = y + A*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}

    for j = 1:n

        y(i) = y(i) + A(i,j)*x(j)

{write y(1:n) back to slow memory}

- m = number of slow memory refs = $3*n + n^2$
- f = number of arithmetic operations = $2*n^2$
- q = f/m ~= 2
- Matrix-vector multiplication limited by slow memory speed

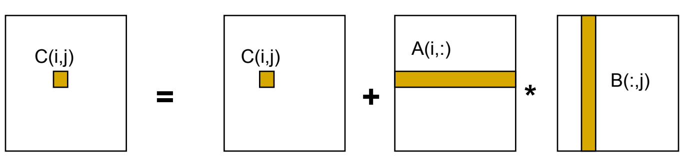
Matrix Multiply C=C+A*B

for i = 1 to n

    for j = 1 to n

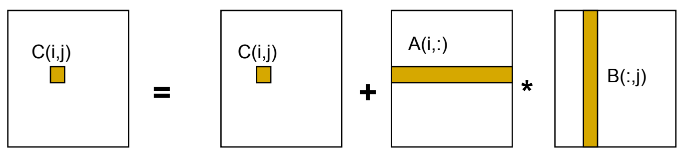
        for k = 1 to n

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



Matrix Multiply C=C+A*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) * B(k,j)
     {write C(i,j) back to slow memory}
```



Matrix Multiply C=C+A*B (unblocked, or untiled)

q=ops/slow mem ref

Number of slow memory references on unblocked matrix multiply

$m = n^3$ read each column of B n times

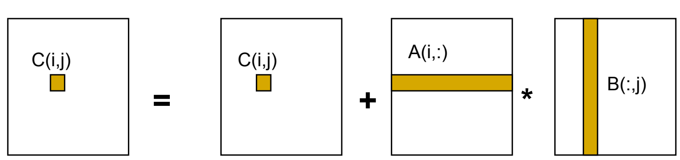
$+ n^2$ read each row of A once for each i

$+ 2*n^2$ read and write each element of C once

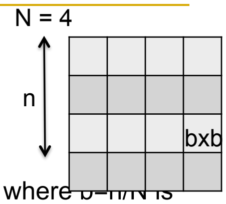
$= n^3 + 3*n^2$

So $q = f/m = (2*n^3)/(n^3 + 3*n^2)$

~= 2 for large n, no improvement over matrix-vector mult



Matrix Multiply (blocked, or tiled)



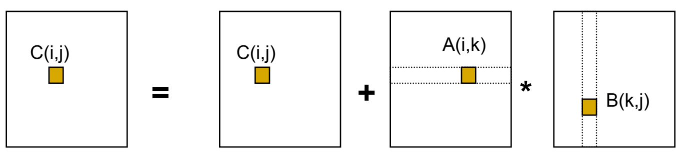
Consider A,B,C to be N by N matrices of b by b subblocks where b=n/N is called the blocksize

```
for i = 1 to N
    for j = 1 to N
        {read block C(i,j) into fast memory}
        for k = 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) * B(k,j) {do a matrix multiply on blocks}
        {write block C(i,j) back to slow memory}
```

N^2 times

N^3 times

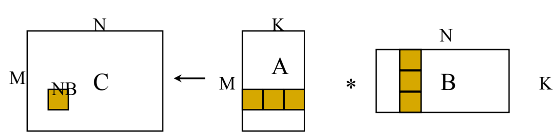
N^3 times



Cache Blocking

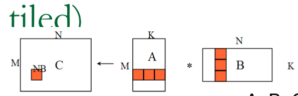
Looping over the blocks

```
do kk = 1,n,nblk
    do jj = 1,n,nblk
        do ii = 1,n,nblk
            do k = kk,kk+nblk-1
                do j = jj,jj+nblk-1
                    do i = ii,ii+nblk-1
                        c(i,j) = c(i,j) + a(i,k) * b(k,j)
                    end do
            . . .
end do
```



Matrix multiply of block

Matrix Multiply (blocked or tiled)



A, B, C made up of
NxN blocks of size b (n/N)

Why is this algorithm correct?

Number of slow memory references on blocked matrix multiply

$m = N*n^2$ read each block of B N^3 times ($N^3 * n/N * n/N$)
$+ N*n^2$ read each block of A N^3 times ($N^3 * n/N * n/N$)
$+ 2*n^2$ read and write each block of C once
$= (2*N + 2)*n^2$

So $q = f/m = 2*n^3 / ((2*N + 2)*n^2)$
$\sim= n/N = b$ for large n

q=ops/slow mem ref

n size of matrix
b blocksize (n/N)
N number of blocks

So we can improve performance by increasing the blocksize b
Can be much faster than matrix-vector multiply (q=2)

Limit: All three blocks from A,B,C must fit in fast memory (cache), so we cannot make these blocks arbitrarily large: $3*b^2 <= M$, so $q \sim= b <= sqrt(M/3)$

Theorem (Hong, Kung, 1981): Any reorganization of this algorithm (that uses only associativity) is limited to q =O(sqrt(M))

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

$$M = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

- Strassen formulation does 7 multiples and 18 additions.

Let p1 = (a11 + a22) * (b11 + b22)
p2 = (a21 + a22) * b11
p3 = a11 * (b12 - b22)
p4 = a22 * (b21 – b11)

p5 = (a11 + a12) * b22
p6 = (a21 – a11) * (b11 + b12)
p7 = (a12 - a22) * (b21 + b22)

Then m11 = p1 + p4 – p5 + p7
m12 = p3 + p5
m21 = p2 + p4
m22 = p1 + p3 – p2 + p6

Extends to nxn by divide&conquer

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
- $O(2^{log(7)}) = O(2^{2.807})$ complexity (recursively)
- Applicable to 2 x 2 block matrices

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

Phase 1 (temporary sums)

T1 = A11+A22 T2 = A21+A22 T3 = A11+A12
T4 = A21-A11 T5 = A12-A22 T6 = B11+B22
T7 = B12-B22 T8 = B21-B11 T9 = B11+B12 (corrected 3/2/09)
T10 = B21+B22

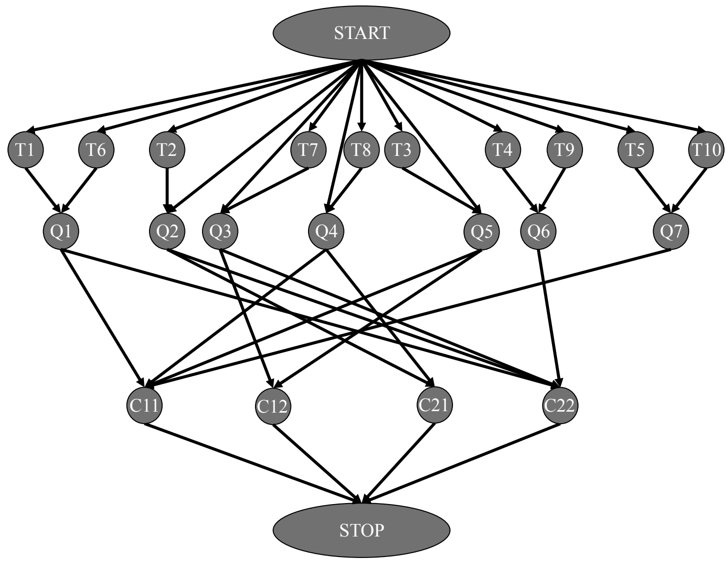
Phase 2 (temporary products)

Q1 = T1*T6 Q2 = T2*B11 Q3 = A11*T7
Q4 = A22*T8 Q5 = T3*B22 Q6 = T4*T9
Q7 = T5*T10

Phase 3

C11 = Q1+Q4-Q5+Q7
C12 = Q3+Q5 C21 = Q2+Q4
C22 = Q1-Q2+Q3+Q6

Strassen's task graph



Strassen (continued)

T(n) = Cost of multiplying nxn matrices

= $7*T(n/2) + 18*(n/2)^2$

= $O(n^{\log_2 7})$

= $O(n^{2.81})$

- 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 $O(n^{2.376..})$

CS 594 - Applications of Parallel Computing
Homework Week 2
January 18th, 2012
Due February 1th, 2012

Part1:
Implement, in Fortran or C, the six different ways to perform matrix multiplication by interchanging the loops.

C = C + A*B

(Use 64-bit arithmetic.) Make each implementation a subroutine, like:

```
subroutine ijk ( c, m, n, ldc, a, k, lda, b, ldb )
subroutine ikj ( c, m, n, ldc, a, k, lda, b, ldb )
...
```

Construct a driver program to generate random matrices and calls each matrix multiply routine with square matrices of orders 10, 20, 30, …, 200, 250, 300, …, 500, timing the calls and computing the Mflop/s rate. Make sure you verify the correctness of your results. You can use the ATLAS routine's (see below) results to verify your routines' results. You should compute something like:

$$\frac{\| C_{ijk} - C_{atlas} \|}{\| C_{atlas} \| * machep}$$

Run your program on a processor of the clusters.

Use the highest level of optimization the compiler allows and experiment with this.
For measuring the time, number of operations, and rate of execution use PAPI.
See http://www.cs.utk.edu/~terpstra/using_papi/
Include in your timing routine a call to the BLAS matrix multiply routine DGEMM from ATLAS.

Download and build ATLAS for this part.

```
call dgemm('No', 'No', n, n, n, 1.0d0, a, lda, b,
           ldb,1.0d0, c, ldc )
```

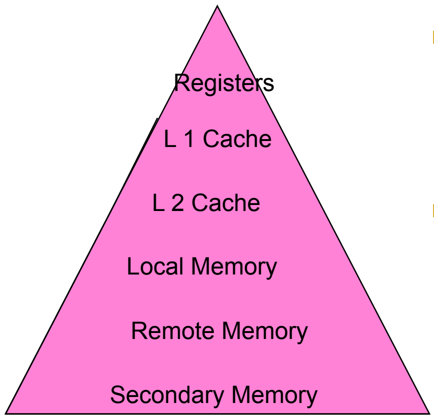
For ATLAS see http://www.netlib.org/atlas/.
For PAPI see: http://icl.cs.utk.edu/papi/

Write-up a description of the timing and describe why the routines perform as they do.

Part 2:
The goal is to optimize matrix multiplication on these machines. Use whatever optimization techniques you can to improve the performance.

BLAS

Memory Hierarchy



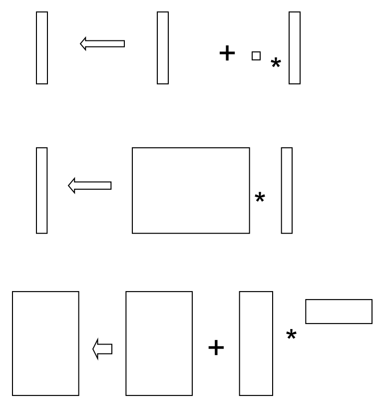
- Key to high performance in effective use of memory hierarchy
- True on all architectures

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. $O(n^2)$ data and $O(n^3)$ work.

BLAS (Basic Linear Algebra Subroutines)

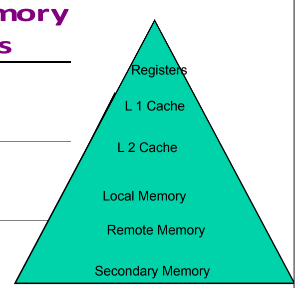
- Industry standard interface(evolving)
- Vendors, others supply optimized implementations
- History
  - BLAS1 (1970s):
    - vector operations: dot product, saxpy (y=∝*x+y), etc
    - m=2*n, f=2*n, q ~1 or less

      q: flops/memory reference
  - BLAS2 (mid 1980s)
    - matrix-vector operations: matrix vector multiply, etc
    - m=n^2, f=2*n^2, q~2, less overhead
    - somewhat faster than BLAS1
  - BLAS3 (late 1980s)
    - matrix-matrix operations: matrix matrix multiply, etc
    - m >= 4n^2, f=O(n^3), so q can possibly be as large as n, so BLAS3 is potentially much faster than BLAS2
- Good algorithms used BLAS3 when possible (LAPACK)
- www.netlib.org/blas, www.netlib.org/lapack

Why Higher Level BLAS?

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

BLAS	Memory Refs	Flops	Flops / Memory Refs
Level 1 $y=y+\alpha x$	3 n	2 n	2 / 3
Level 2 $y=y+Ax$	n^2	2 n^2	2
Level 3 $C=C+AB$	4 n^2	2 n^3	n/2



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 * 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 * x, for scalar a and vectors x,y
- sdot computes $s = s + \sum^{n}_{i=1} x(i)*y(i)$

Level 1 BLAS

- Other routines doing reduction operations
  - Compute different vector norms of vector *x*
  - Compute the sum of the entries of vector *x*
  - Find the smallest or biggest component of vector *x*
  - Compute the sum of squares of the entries of vector *x*
- Routines doing rotation operations
  - Generate Givens plane rotation
  - Generate Jacobi rotation
  - Generate Householder transformation

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*x**
  - **where A is m-by-n, x is n-by-1 and y is m-by-1.**
- ***sger:* rank-one update**
  - **$A = A + y*x^T$, i.e., A(i,j) = A(i,j)+y(i)*x(j)**
  - **where A is m-by-n, y is m-by-1, x is n-by-1,**
  - **strsv: triangular solve**
  - **solves y=T*x for x, where T is triangular**

Level 2 BLAS

- Routines of Level 2
  - Compute different matrix vector products
  - Do addition of scaled matrix vector products
  - Compute multiple matrix vector products
  - Solve triangular equations
  - Perform rank one and rank two updates
  - Some operations use symmetric or triangular matrices

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 *nxn* matrix *A* is stored in a one-dimensional array `a`
    - a_{ij} `=> a[i+j*s]` (C, column-wise storage)
  - If $s=n$, rows (columns) will be contiguous in memory
  - If $s>n$, there will be a gap of $(s\text{-}n)$ 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 op

  $y \leftarrow \alpha Ax + \beta y$ where $A = A^T$

  $x \leftarrow \alpha Ax$ or $x \leftarrow \alpha A^T x$ where A is triangular

  $y \leftarrow \alpha Ax + \beta Bx$

  $x \leftarrow \beta A^T y$, $w \leftarrow \alpha Ax$

  $x \leftarrow A^T y$, $w \leftarrow Az$ where A is triangular

  as well as many others
- 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

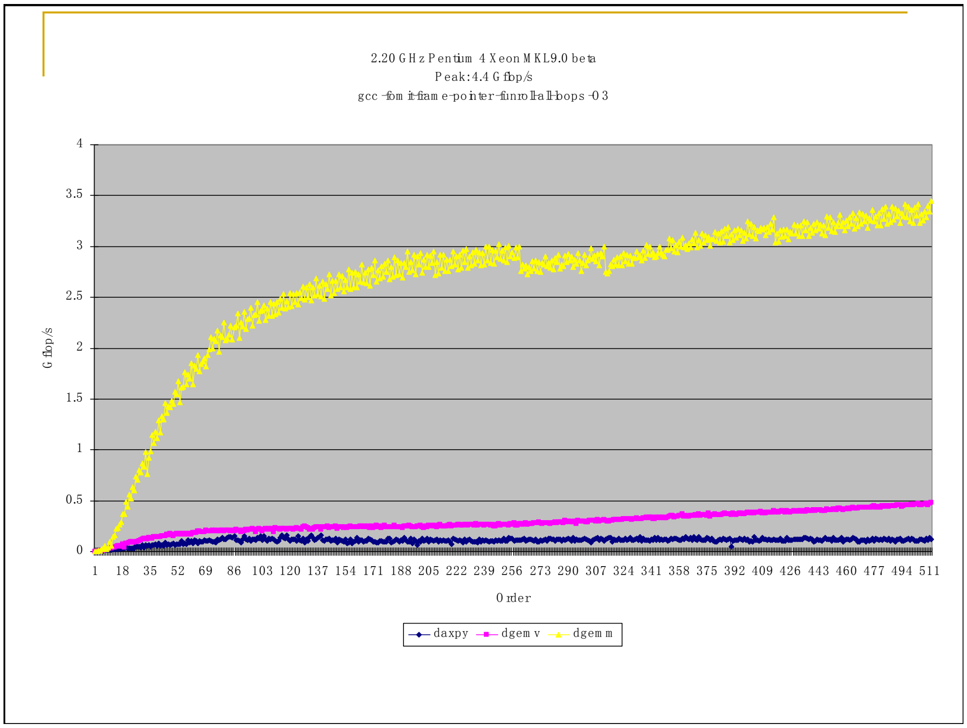
- Routine names
  - 5/6 character name: **xYYZZZ**
  - **x** – data type and precision
    - S – real single precision
    - D – real double precision
    - C – complex single precision
    - Z – complex double precision
  - **YY** – matrix type
    - GE – general rectangular
    - SY – symmetric
    - 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
- Routine names – continued
  - **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

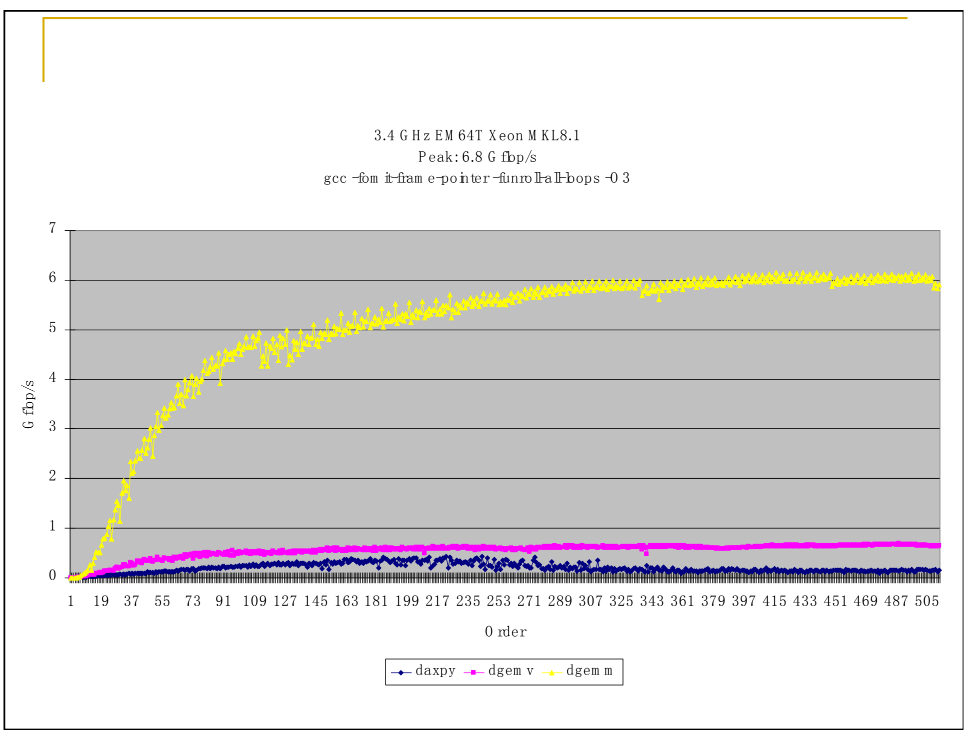
Parameter Naming Conventions

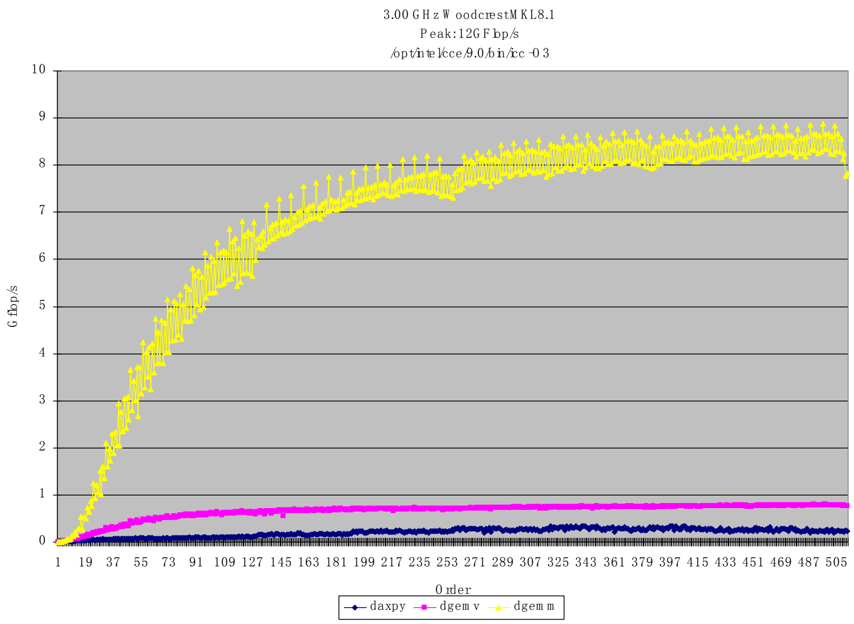
- Vector names
  - X, Y
- Vector strides
  - INCX, INCY
- Matrix names
  - A, B, C
- Matrix leading dimensions
  - LDA, LDB, LDC
- Matrix and/or vector dimensions
  - M, N, K
- Scalars
  - ALPHA, BETA
- Which side matrix operates on
  - SIDE
  - LEFT, RIGHT
- Transformation of the matrix before operation
  - TRANS
  - NOTRANS, TRANSPOSE, CONJUGATE TRANSPOSE
- Which part of the matrix to operate on
  - UPLO
  - UPPER, LOWER
- Whether to treat diagonal as ones
  - DIAG
  - UNIT, NONUNIT

BLAS Names

- The first letter of the subprogram 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
  - IxAMAX - 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
  - xSYRK - symmetric rank-k update
  - xSYR2K - symmetric rank-2k update

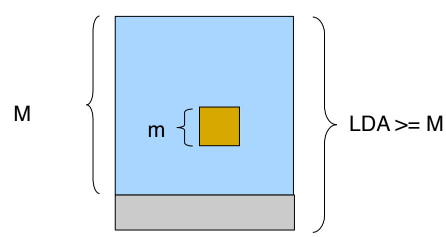

2.20 GHz Pentium 4 Xeon MKL9.0 beta
Peak: 4.4 Gflop/s
gcc -fomit-frame-pointer -funroll-all-loops -O3
Gflop/s
Order
daxpy
dgemv
dgemm


3.4 GHz EM64T Xeon MKL8.1
Peak: 6.8 Gflop/s
gcc -fomit-frame-pointer -funroll-all-loops -O3
Gflop/s
Order
daxpy
dgemv
dgemm



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



A, LDA, M

A(i,j), LDA, m

Packed Storage

- 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 *nxn* matrix *A* may be stored in a one-dimensional array `a`
    - `a`$_{ij}$`(i£j) => a[j+i*(2*n-i-1)/2]` (C, row-wise storage)
- Example.

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ 0 & a_{11} & a_{12} \\ 0 & 0 & a_{22} \end{pmatrix} \Rightarrow \begin{matrix} a_{00} & a_{01} & a_{02} & a_{11} & a_{12} & a_{22} \end{matrix}$$

Band Storage

- Band storage
  - A compact storage scheme for band matrices
- Consider Fortran and a column-wise storage scheme
  - An *mxn* band matrix *A* with *l* subdiagonals and *u* superdiagonals may be stored in a 2-dimensional array `A` with *l+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} => `A(u+i-j,j)` for *max(0,j-u) £i £min(m-1,j+l)*
- Example.

$$\begin{pmatrix} a_{00} & a_{01} & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & 0 & 0 \\ a_{20} & a_{21} & a_{22} & a_{23} & 0 \\ 0 & a_{31} & a_{32} & a_{33} & a_{34} \\ 0 & 0 & a_{42} & a_{43} & a_{44} \end{pmatrix} \Rightarrow \begin{matrix} * & a_{01} & a_{12} & a_{23} & a_{34} \\ a_{00} & a_{11} & a_{22} & a_{33} & a_{44} \\ a_{10} & a_{21} & a_{32} & a_{43} & * \\ a_{20} & a_{31} & a_{42} & * & * \end{matrix}$$

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

- C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, *Basic Linear Algebra Subprograms for Fortran Usage,* ACM Transactions on Mathematical Software, 5:308--325, 1979.
- J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson, *An Extended Set of Fortran Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, 14(1):1--32, 1988.
- J. Dongarra, J. Du Croz, I. Duff, S. Hammarling, *A Set of Level 3 Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, 16(1):1--17, 1990.

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?

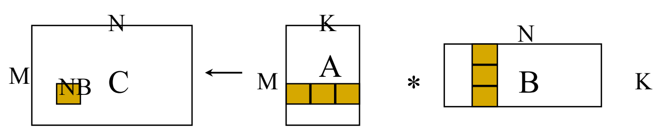
- Today's processors can achieve high-performance, but this requires extensive machine-specific hand tuning.
- Routines have a large design space w/many parameters
  - blocking sizes, loop nesting permutations, loop unrolling depths, software pipelining strategies, register allocations, and instruction 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 tranpose cases coerced through data copy to 1 case of on-chip multiply
  - Only 1 case generated per platform



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

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)

Hand Code (assembly)

+ Timing → Algorithm selection

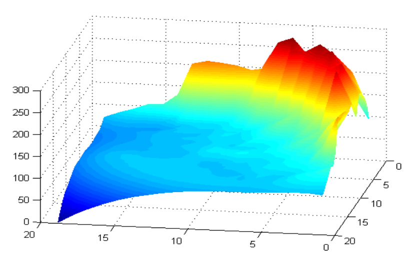
Runtime: decision based on data

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

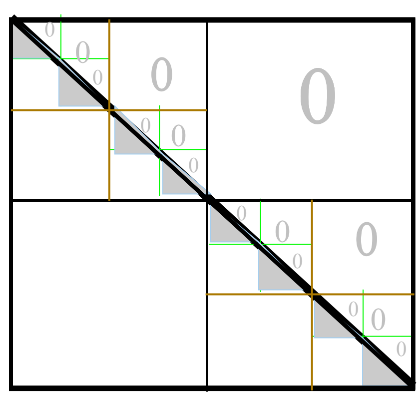
- RISC (Reduced Instr. Set C.)
  - POWERX
  - MIPS
  - xSPARC X
  - HP/DEC/Digital Alpha
- CISC
  - AMD 32/64-bit
  - Intel 32/64-bit
  - HyperThreading™
  - Multi-core (Duo, Trio, Q...)
- VLIW (Very Long Instruction Word)
  - Itanium
  - Itanium 2

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

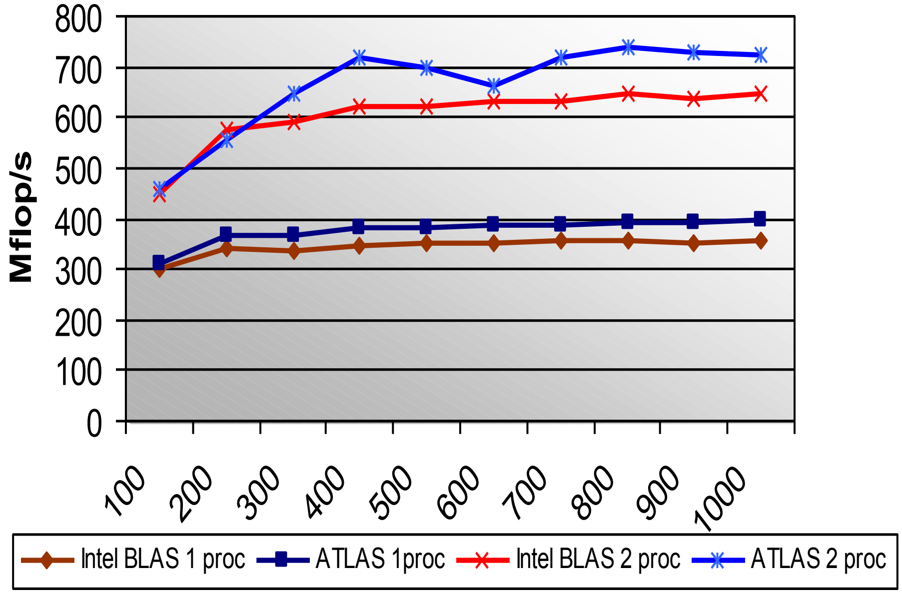
Recursive Approach for Other Level 3 BLAS

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

Recursive TRMM



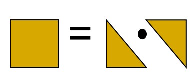
Multi-Threaded DGEMM
Intel PIII 550 MHz



Gaussian Elimination Basics

Solve $A\,x = b$

<u>Step 1</u> A = L U



<u>Step 2</u> Forward Elimination

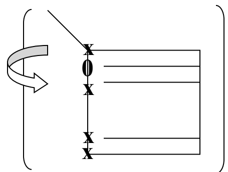
Solve $L\,y = b$

<u>Step 3</u> Backward Substitution

Solve $U\,x = y$

Note: Changing RHS does not imply to recompute LU factorization

Gaussian Elimination



Standard Way

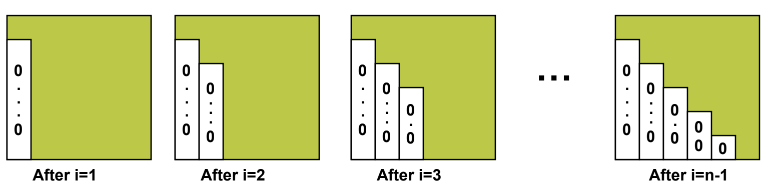
subtract a multiple of a row

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

```
… for each column i
… zero it out below the diagonal by adding multiples of row i to later rows
for i = 1 to n-1
   … for each row j below row i
   for j = i+1 to n
      … 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)
```



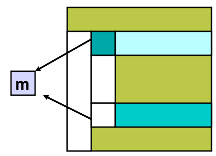
Refine GE Algorithm (1)

- Initial Version

```
… for each column i
… zero it out below the diagonal by adding multiples of row i to later rows
for i = 1 to n-1
   … for each row j below row i
   for j = i+1 to n
      … 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)
```

- Remove computation of constant tmp/A(i,i) from inner loop.

```
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)
```



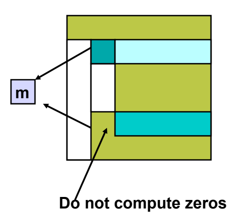
Refine GE Algorithm (2)

- 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)
```

- Don't compute what we already know: zeros below diagonal in column i

```
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)
```



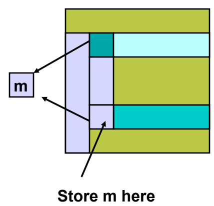
Refine GE Algorithm (3)

- 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)
```

- Store multipliers m below diagonal in zeroed entries for later use

```
for i = 1 to n-1
    for j = i+1 to n
        A(j,i) = A(j,i)/A(i,i)
        for k = i+1 to n
            A(j,k) = A(j,k) - A(j,i) * A(i,k)
```



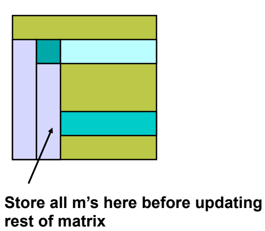
Refine GE Algorithm (4)

- Last version

```
for i = 1 to n-1
    for j = i+1 to n
        A(j,i) = 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**

```
for i = 1 to n-1
    for j = i+1 to n
        A(j,i) = A(j,i)/A(i,i)
    for j = i+1 to n
        for k = i+1 to n
            A(j,k) = A(j,k) - A(j,i) * A(i,k)
```



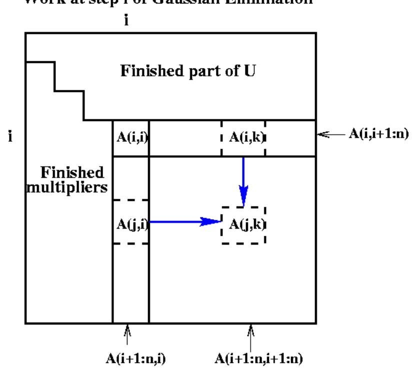
Refine GE Algorithm (5)

- Last version

```
for i = 1 to n-1
   for j = i+1 to n
      A(j,i) = A(j,i)/A(i,i)
   for j = i+1 to n
      for k = i+1 to n
         A(j,k) = A(j,k) - A(j,i) * A(i,k)
```

- Express using matrix operations (BLAS)

Work at step i of Gaussian Elimination




```
for i = 1 to n-1
   A(i+1:n,i) = A(i+1:n,i) * ( 1 / A(i,i) )
   A(i+1:n,i+1:n) = A(i+1:n , i+1:n )
        - A(i+1:n , i) * A(i , i+1:n)
```

What GE really computes

```
for i = 1 to n-1
   A(i+1:n,i) = A(i+1:n,i) / A(i,i)
   A(i+1:n,i+1:n) = A(i+1:n , i+1:n ) - A(i+1:n , i) * A(i , i+1:n)
```

- 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*U
- Solving A*x=b using GE
  - Factorize A = L*U using GE (cost = 2/3 n^3 flops)
  - Solve L*y = b for y, using substitution (cost = n^2 flops)
  - Solve U*x = y for x, using substitution (cost = n^2 flops)
- Thus A*x = (L*U)*x = L*(U*x) = L*y = b as desired

Pivoting in Gaussian Elimination

- A = [0 1] fails completely because can't divide by A(1,1)=0
  [1 0]
- But solving Ax=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

Gaussian Elimination with Partial Pivoting (GEPP)

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

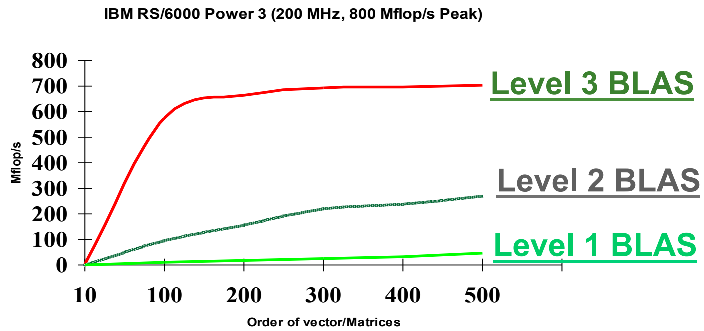
```
for i = 1 to n-1
    find and record k where |A(k,i)| = max{i <= j <= n} |A(j,i)|
        … i.e. largest entry in rest of column i
    if |A(k,i)| = 0
        exit with a warning that A is singular, or nearly so
    elseif  k != i
        swap rows i and k of A
    end if
    A(i+1:n,i) = A(i+1:n,i) / 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) * A(i , i+1:n)
```

- *Lemma*: This algorithm computes A = P*L*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

Problems with basic GE algorithm

- What if some A(i,i) 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…)

```
for i = 1 to n-1
    A(i+1:n,i) = A(i+1:n,i) / A(i,i)          … BLAS 1 (scale a vector)
    A(i+1:n,i+1:n) = A(i+1:n , i+1:n )   … BLAS 2 (rank-1 update)
            - A(i+1:n , i) * A(i , i+1:n)
```



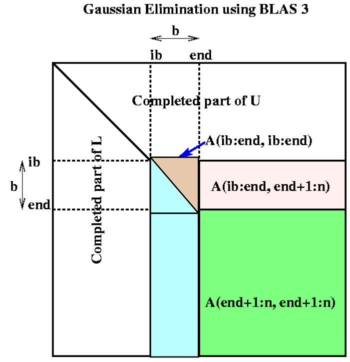
Converting BLAS2 to BLAS3 in GEPP

- 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

Blocked GEPP (www.netlib.org/lapack/single/sgetrf.f)

```
for   ib = 1 to n-1 step b        … Process matrix b columns at a time
    end = ib + b-1                … Point to end of block of b columns
    apply BLAS2 version of GEPP to  get A(ib:n , ib:end) = P' * L' * U'
    … let LL denote the strict lower triangular part of A(ib:end , ib:end) + I
    A(ib:end , end+1:n) = LL^-1 * A(ib:end , end+1:n)       … update next b rows of U
    A(end+1:n , end+1:n ) = A(end+1:n , end+1:n )
              - A(end+1:n , ib:end) * A(ib:end , end+1:n)
                           … apply delayed updates with single matrix-multiply
                           … with inner dimension b
```

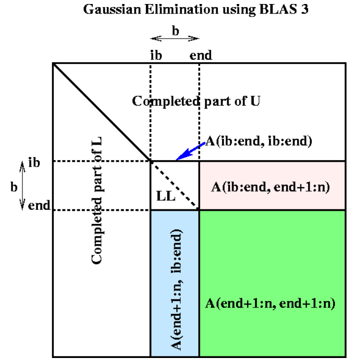
Gaussian Elimination using BLAS 3



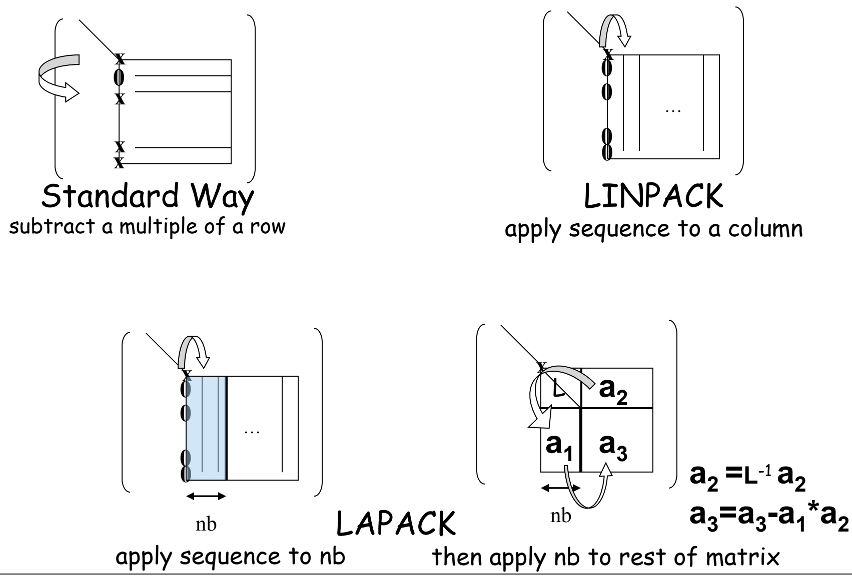
Review: BLAS 3 (Blocked) GEPP

```
for   ib = 1 to n-1 step b        … Process matrix b columns at a time
    end = ib + b-1                … Point to end of block of b columns
    apply BLAS2 version of GEPP to  get A(ib:n , ib:end) = P' * L' * U'
    … let LL denote the strict lower triangular part of A(ib:end , ib:end) + I
BLAS 3 {
    A(ib:end , end+1:n) = LL^-1 * A(ib:end , end+1:n)       … update next b rows of U
    A(end+1:n , end+1:n ) = A(end+1:n , end+1:n )
              - A(end+1:n , ib:end) * A(ib:end , end+1:n)
}
                           … apply delayed updates with single matrix-multiply
                           … with inner dimension b
```

Gaussian Elimination using BLAS 3



Gaussian Elimination



History of Block Partitioned Algorithms

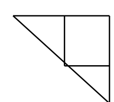
- Early algorithms involved use of small main memory using tapes as secondary storage.
- Recent work centers on use of vector registers, level 1 and 2 cache, main memory, and “out of core” memory.

Blocked Partitioned Algorithms

- LU Factorization
- Cholesky factorization
- Symmetric indefinite factorization
- Matrix inversion
- QR, QL, RQ, LQ factorizations
- Form Q or $Q^T C$
- Orthogonal reduction to:
  - (upper) Hessenberg form
  - symmetric tridiagonal form
  - bidiagonal form
- Block QR iteration for nonsymmetric eigenvalue problems

Derivation of Blocked Algorithms
Cholesky Factorization $A = U^T U$

$$\begin{pmatrix} A_{11} & a_j & A_{13} \\ a_j^T & a_{jj} & \alpha_j^T \\ A_{13}^T & \alpha_j & A_{33} \end{pmatrix} = \begin{pmatrix} U_{11}^T & 0 & 0 \\ u_j^T & u_{jj} & 0 \\ U_{13}^T & \mu_j & U_{33}^T \end{pmatrix} \begin{pmatrix} U_{11} & u_j & U_{13} \\ 0 & u_{jj} & \mu_j^T \\ 0 & 0 & U_{33} \end{pmatrix}$$



Equating coefficient of the j^{th} column, we obtain

$$a_j = U_{11}^T u_j$$
$$a_{jj} = u_j^T u_j + u_{jj}^2$$

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

$$U_{11}^T u_j = a_j$$
$$u_{jj}^2 = a_{jj} - u_j^T u_j$$

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
   20    CONTINUE
         S = A( J, J ) - S
C     ...EXIT
         IF( S.LE.0.0E0 ) GO TO 40
         A( J, J ) = SQRT( S )
   30 CONTINUE
```

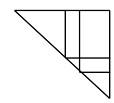
LAPACK Implementation

```
      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

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{12}^T & A_{22} & A_{12} \\ A_{13}^T & A_{12}^T & A_{33} \end{pmatrix} = \begin{pmatrix} U_{11}^T & 0 & 0 \\ U_{12}^T & U_{22}^T & 0 \\ U_{13}^T & U_{23}^T & U_{33}^T \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23}^T \\ 0 & 0 & U_{33} \end{pmatrix}$$



Equating coefficient of second block of columns, we obtain

$$A_{12} = U_{11}^T U_{12}$$

$$A_{22} = U_{12}^T U_{12} + U_{22}^T U_{22}$$

Hence, if U_{11} has already been computed, we can compute U_{12} as the solution of the following equations by a call to the Level 3 BLAS routine STRSM:

$$U_{11}^T U_{12} = A_{12}$$

$$U_{22}^T U_{22} = A_{22} - U_{12}^T U_{12}$$

LAPACK Blocked Algorithms

```
 DO 10 J = 1, N, NB
    CALL STRSM( 'Left', 'Upper', 'Transpose','Non-Unit', J-1, JB, ONE, A, LDA,
$            A( 1, J ), LDA )
    CALL SSYRK( 'Upper', 'Transpose', JB, J-1,-ONE, A( 1, J ), LDA, ONE,
$            A( J, J ), LDA )
    CALL SPOTF2( 'Upper', JB, A( J, J ), LDA, INFO )
    IF( INFO.NE.0 ) GO TO 20
10 CONTINUE
```

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

Intel Pentium 4 1.7 GHz N = 500	Rate of Execution
Linpack variant (L1B)	238 Mflop/s
Level 2 BLAS Variant	312 Mflop/s
Level 3 BLAS Variant	1262 Mflop/s

Gaussian Elimination via a Recursive Algorithm

F. Gustavson and S. Toledo

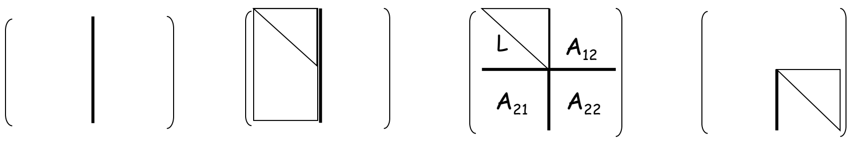
LU Algorithm:

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 $A_{12} = L^{-1}A_{12}$ and
 matrix multiplication $A_{22}=A_{22} - A_{21} * A_{12}$)

4: Apply LU Algorithm to right part

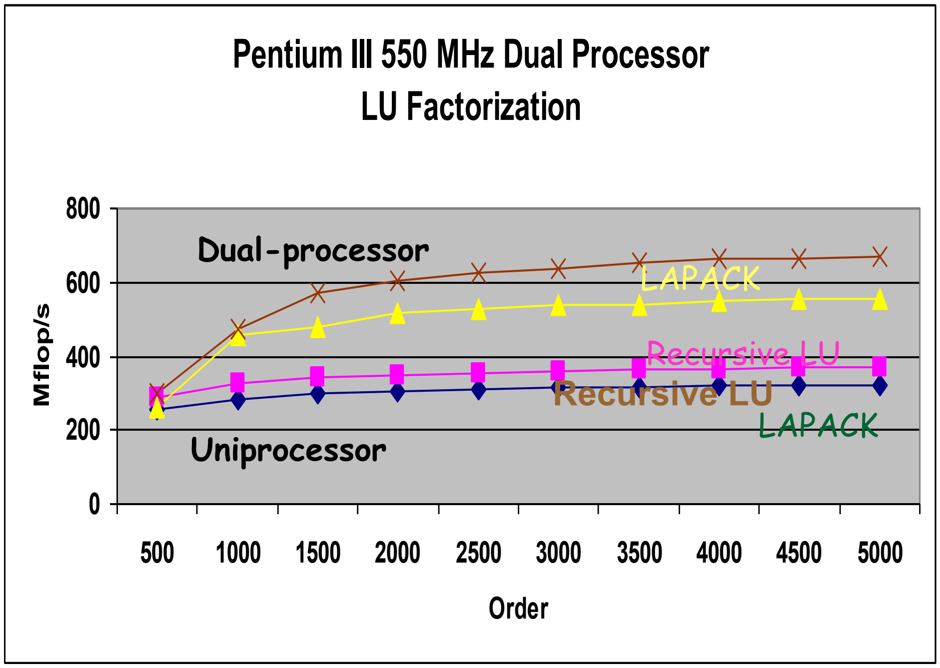


Most of the work in the matrix multiply
Matrices of size n/2, n/4, n/8, …

Source: Jack Dongarra

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).



Source: Jack Dongarra

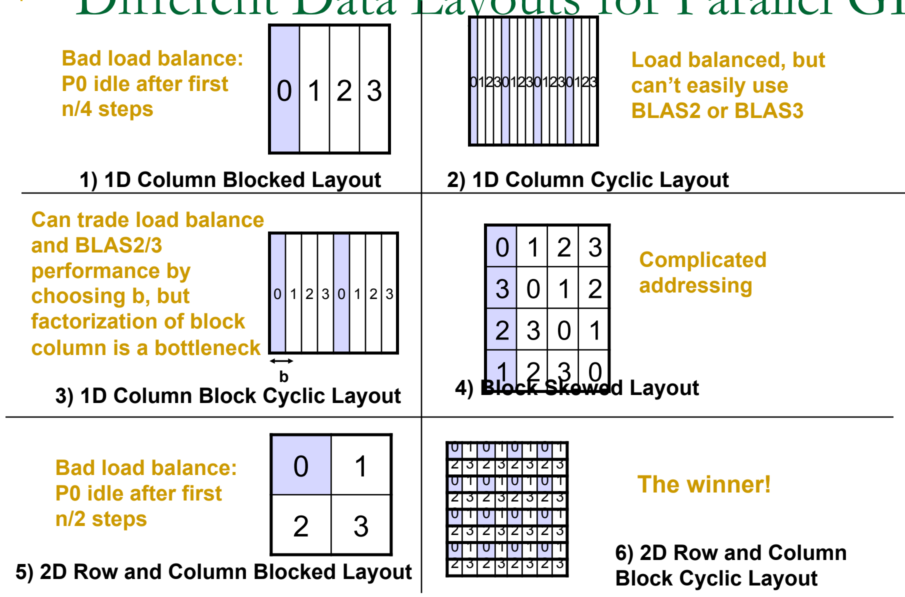
Recursive Algorithms – Limits

- 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

Different Data Layouts for Parallel GE



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

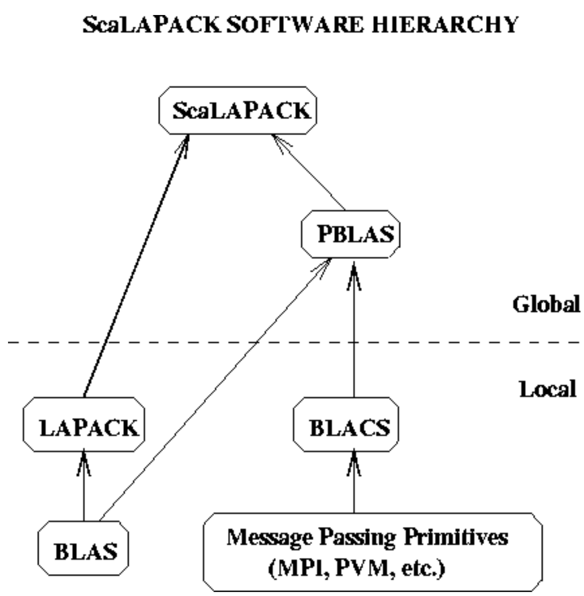
- Similar to the BLAS in functionality and naming.
- Built on the BLAS and BLACS
- Provide global view of matrix

CALL DGEXXX (M, N, A(IA, JA), LDA,...)

⇩

CALL PDGEXXX(M, N, A, IA, JA, DESCA,...)

ScaLAPACK Overview



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
    - Nonsymmetric eigenproblem hardest
- Narrow band problems hardest (to do BLAS3 or parallelize)
  - Solving and eigenproblems
- www.netlib.org/{lapack,scalapack}