Outline

• What is Dense Linear Algebra?
• Where does the time go in an algorithm?
  • Moving Data in Memory hierarchies
• Case studies
  • Matrix Multiplication
  • Gaussian Elimination
What is dense linear algebra?

- Not just Basic Linear Algebra Subroutines (eg matmul)
- Linear Systems: $Ax=b$
- Least Squares: choose $x$ to minimize $||Ax-b||_2$
  - Overdetermined or underdetermined
  - Unconstrained, constrained, weighted
- Eigenvalues and vectors of Symmetric Matrices
  - Standard ($Ax = \lambda x$), Generalized ($Ax=\lambda Bx$)
- Eigenvalues and vectors of Unsymmetric matrices
  - Eigenvalues, Schur form, eigenvectors, invariant subspaces
  - Standard, Generalized
- Singular Values and vectors (SVD)
  - Standard, Generalized
- Different matrix structures
  - Real, complex; Symmetric, Hermitian, positive definite; dense, triangular, banded …
- Level of detail
  - Simple Driver
  - Expert Drivers with error bounds, extra-precision, other options
  - Lower level routines (“apply certain kind of orthogonal transformation”, …)
Matrix-multiply, optimized several ways

Speed of n-by-n matrix multiply on Sun Ultra-1/170, peak = 330 MFlops
Where does the time go?

- Hardware organized as Memory Hierarchy
  - Try to keep frequently accessed data in fast, but small memory
  - Keep less frequently accessed data in slower, but larger memory
- \( \text{Time(flops)} \approx \text{Time(on-chip cache access)} \ll \text{Time(slower mem access)} \)
  - Need algorithms that minimize accesses to slow memory, i.e. “minimize communication”

### Memory Hierarchy

- **On-chip cache (SRAM)**: 
  - Speed: 1ns
  - Size: KB

- **Main memory (DRAM)**: 
  - Speed: 10ns
  - Size: MB

- **Secondary storage (Disk)**: 
  - Speed: 100ns
  - Size: GB

- **Tertiary storage (Disk/Tape)**: 
  - Speed: 10ms
  - Size: TB

- **Speed**
  - 1ns
  - 10ns
  - 100ns
  - 10ms
  - 10sec

- **Size**
  - KB
  - MB
  - GB
  - TB
  - PB
Minimizing communication gets more important every year

- Memory hierarchies are getting deeper
  - Processors get faster more quickly than memory

“Moore’s Law”

Processor-Memory Performance Gap: (grows 50% / year)

µProc 60%/yr.

DRAM 7%/yr.
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 $\ll t_m$
  • $q = \frac{f}{m}$ average number of flops per slow memory access

• Minimum possible time $= f \cdot t_f$ when all data in fast memory

• Actual time
  • $f \cdot t_f + m \cdot t_m = f \cdot t_f \cdot (1 + \frac{t_m}{t_f} \cdot \frac{1}{q})$

• Larger $q$ means time closer to minimum $f \cdot t_f$
  • $q \geq \frac{t_m}{t_f}$ needed to get at least half of peak speed
    $\geq 1000$
Warm up: Matrix-vector multiplication

\{\text{implement } y = y + A^x\}

for \(i = 1:n\)

\begin{align*}
&\quad \text{for } j = 1:n \\
&\quad \quad y(i) = y(i) + A(i,j) \times x(j)
\end{align*}
Warm up: Matrix-vector multiplication

{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 $j = 1:n$
        $y(i) = y(i) + A(i,j) \times x(j)$
    {write $y(1:n)$ back to slow memory}

- $m =$ number of slow memoryrefs $= 3n + n^2$
- $f =$ number of arithmetic operations $= 2n^2$
- $q =$ $f / m \approx 2$

- Matrix-vector multiplication limited by slow memory speed
Naïve Matrix Multiply

\{\text{implements } C = C + A*B\}

\text{for } i = 1 \text{ to } n
\hspace{1cm} \text{for } j = 1 \text{ to } n
\hspace{2cm} \text{for } k = 1 \text{ to } n
\hspace{3cm} C(i,j) = C(i,j) + A(i,k) \times B(k,j)

Algorithm has $2*\text{n}^3 = O(\text{n}^3)$ Flops and
operates on $3*\text{n}^2$ words of memory

$q$ potentially as large as \(\frac{2*\text{n}^3}{3*\text{n}^2} = O(\text{n})\)
Naïve Matrix Multiply

\{\text{implements } C = C + A*B\}

\text{for } i = 1 \text{ to } n

\{\text{read row } i \text{ of } A \text{ into fast memory}\}

\text{for } j = 1 \text{ to } n

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

\{\text{read column } j \text{ of } B \text{ into fast memory}\}

\text{for } k = 1 \text{ to } n

\hspace{1em} C(i,j) = C(i,j) + A(i,k) \times B(k,j)

\{\text{write } C(i,j) \text{ back to slow memory}\}
Naïve Matrix Multiply

\{\text{implements } C = C + A \times B\}

\text{for } i = 1 \text{ to } n

\quad \{\text{read row } i \text{ of } A \text{ into fast memory } \ldots \text{ } n^2 \text{ total reads}\}

\text{for } j = 1 \text{ to } n

\quad \{\text{read } C(i,j) \text{ into fast memory } \ldots \text{ } n^2 \text{ total reads}\}

\quad \{\text{read column } j \text{ of } B \text{ into fast memory } \ldots \text{ } n^3 \text{ total reads}\}

\text{for } k = 1 \text{ to } n

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

\quad \{\text{write } C(i,j) \text{ back to slow memory } \ldots \text{ } n^2 \text{ total writes}\}
Naïve Matrix Multiply

Number of slow memory references on unblocked matrix multiply

\[ m = n^3 \] to read each column of B \( n \) times
\[ + n^2 \] to read each row of A once
\[ + 2n^2 \] to read and write each element of C once
\[ = n^3 + 3n^2 \]

So \( q = f / m = 2n^3 / (n^3 + 3n^2) \)
\[ \approx 2 \] for large \( n \), no improvement over matrix-vector multiply

Inner two loops are just matrix-vector multiply, of row i of A times B
Similar for any other order of 3 loops

\[ C(i,j) = C(i,j) + A(i,:) \ast B(:,j) \]
Consider $A, B, C$ to be $(n/b)$-by-$(n/b)$ matrices of $b$-by-$b$ blocks where $b$ is called the block size; assume fast memory holds 3 $b$-by-$b$ blocks.

for $i = 1$ to $n/b$
  for $j = 1$ to $n/b$
    {read block $C(i,j)$ into fast memory}
    for $k = 1$ to $n/b$
      {read block $A(i,k)$ into fast memory}
      {read block $B(k,j)$ into fast memory}
      $C(i,j) = C(i,j) + A(i,k) \times B(k,j)$  {matrix multiply on $b$-by-$b$ blocks}
    {write block $C(i,j)$ back to slow memory}
Blocked (Tiled) Matrix Multiply

Recall:
- $m$ is amount memory traffic between slow and fast memory
- matrix is $n$-by-$n$, arranged as $(n/b)$-by-$(n/b)$ matrix of $b$-by-$b$ blocks
- $f = 2n^3$ = number of floating point operations
- $q = f / m$ = “computational intensity”

So:
- $m = \frac{n^3}{b}$ read each block of $B$ $(n/b)^3$ times, so $(n/b)^3 \cdot b^2 = \frac{n^3}{b}$
- $+ \frac{n^3}{b}$ read each block of $A$ $(n/b)^3$ times
- $+ 2n^2$ read and write each block of $C$ once
- $= \frac{2n^3}{b} + 2n^2$

So computational intensity $q = f / m = \frac{2n^3}{\frac{2n^3}{b} + 2n^2}$
- $\approx \frac{2n^3}{(2n^3/b)} = b$ for large $n$

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

• \#\text{slow\_memory\_references} = m = \frac{2n^3}{b} + 2n^2
• Increasing b reduces m. How big can we make b?
• Recall assumption that 3 b-by-b blocks \( C(i,j), A(i,k) \) and \( B(k,j) \) fit in fast memory, say of size M
  • Constraint: \( 3b^2 \leq M \)
• Tiled matrix multiply cannot reduce m below \( \approx 2 \cdot 3^{1/2} \cdot \frac{n^3}{M^{1/2}} \)

• Theorem (Hong & Kung, 1981): You can’t do any better than this: Any reorganization of this algorithm (that uses only commutativity and associativity) is limited to \#\text{slow\_memory\_references} = \Omega(n^3 / M^{1/2})
  \( (f(x) = \Omega(g(x)) \) means that \( |f(x)| \geq C \cdot |g(x)| \).
**Limits to Optimizing All Dense Linear Algebra**

- Theorem [Ballard, Demmel, Holtz, Schwartz, 2011]: Consider any algorithm that is “like 3 nested loops in matrix-multiply”, running with fast memory size $M$. Then no matter how you optimize it, using only associativity and commutativity,

$$\#\text{slow}_\text{memory}_\text{references} = \Omega \left( \frac{\#\text{flops}}{M^{1/2}} \right)$$

- This applies to
  - Basic Linear Algebra Subroutines like matmul, triangular solve, …
  - Gaussian elimination, Cholesky, other variants …
  - Least squares
  - Eigenvalue problems and the SVD
  - Some operations on tensors, graph algorithms …
  - Some whole programs that call a sequence of these operations
  - Multiple levels of memory hierarchy, not just “fast and slow”
  - Dense and sparse matrices
    - $\#\text{flops} = O(n^3)$ for dense matrices, usually much less for sparse
  - Sequential and parallel algorithms
    - Parallel case covered in CS267
Can we attain these lower bounds?

• Do algorithms in standard dense linear algebra libraries attain these bounds?
  • LAPACK (sequential), ScaLAPACK (parallel), versions offered by vendors
  • For some problems (eg matmul) they do, but mostly not
  • Note: these libraries are still fast, and should be your first choice on most problems!

• Are there other algorithms that do attain them?
  • Yes, some known for a while, some under development

• Two examples
  • Matmul (again, but for any memory hierarchy)
  • Gaussian Elimination
What if there are more than 2 levels of memory?

• Goal is to minimize communication between all levels
• The tiled algorithm requires finding a good block size
  • Machine dependent: \( b \) depends on fast memory size \( M \)
  • Need to “block” \( b \times b \) matrix multiply in inner most loop
    • 1 level of memory \( \Rightarrow \) 3 nested loops (naïve algorithm)
    • 2 levels of memory \( \Rightarrow \) 6 nested loops
    • 3 levels of memory \( \Rightarrow \) 9 nested loops …

• Cache Oblivious Algorithms offer an alternative
  • Treat \( n \times n \) matrix multiply as a set of smaller problems
  • Eventually, these will fit in cache
  • Will minimize # words moved between every level of memory hierarchy (between L1 and L2 cache, L2 and L3, L3 and main memory etc.) – at least asymptotically
Recursive Matrix Multiplication (RMM) (1/2)

• For simplicity: square matrices with \( n = 2^m \)

\[
\begin{align*}
C &= \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = A \cdot B = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \\
&= \begin{pmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{pmatrix}
\]

• True when each \( A_{ij} \) etc \( 1 \times 1 \) or \( n/2 \times n/2 \)

```
func C = RMM (A, B, n)
    if n = 1, C = A * B, else
    {  C_{11} = RMM (A_{11}, B_{11}, n/2) + RMM (A_{12}, B_{21}, n/2) \\
        C_{12} = RMM (A_{11}, B_{12}, n/2) + RMM (A_{12}, B_{22}, n/2) \\
        C_{21} = RMM (A_{21}, B_{11}, n/2) + RMM (A_{22}, B_{21}, n/2) \\
        C_{22} = RMM (A_{21}, B_{12}, n/2) + RMM (A_{22}, B_{22}, n/2)  }
    return
```
Recursive Matrix Multiplication (2/2)

```
func C = RMM (A, B, n)
    if n=1, C = A * B, else
        {  C_{11} = RMM (A_{11}, B_{11}, n/2) + RMM (A_{12}, B_{21}, n/2)
            C_{12} = RMM (A_{11}, B_{12}, n/2) + RMM (A_{12}, B_{22}, n/2)
            C_{21} = RMM (A_{21}, B_{11}, n/2) + RMM (A_{22}, B_{21}, n/2)
            C_{22} = RMM (A_{21}, B_{12}, n/2) + RMM (A_{22}, B_{22}, n/2)  }
    return
```

\[ A(n) = \# \text{arithmetic operations in } RMM(\ .\ ,\ .\ ,\ n) \]
\[ = 8 \cdot A(n/2) + 4(n/2)^2 \text{ if } n > 1, \quad \text{else } 1 \]
\[ = 2n^3 \quad \ldots \text{same operations as usual, in different order} \]

\[ W(n) = \# \text{words moved between fast, slow memory by } RMM(\ .\ ,\ .\ ,\ n) \]
\[ = 8 \cdot W(n/2) + 4(n/2)^2 \text{ if } 3n^2 > M, \quad \text{else } 3n^2 \]
\[ = O( n^3 / (M )^{1/2} + n^2 ) \quad \ldots \text{same as blocked matmul} \]

Algorithm called “cache oblivious”, because doesn’t depend on M
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

\[
\begin{align*}
&\text{... for each column } i \\
&\text{... zero it out below the diagonal by adding multiples of row } i \text{ to later rows} \\
&\quad \text{for } i = 1 \text{ to } n-1 \\
&\quad \text{... for each row } j \text{ below row } i \\
&\quad \quad \text{for } j = i+1 \text{ to } n \\
&\quad \quad \quad \text{... add a multiple of row } i \text{ to row } j \\
&\quad \quad \quad \quad \text{tmp} = A(j,i); \\
&\quad \quad \quad \quad \text{for } k = i \text{ to } n \\
&\quad \quad \quad \quad \quad A(j,k) = A(j,k) - (\text{tmp}/A(i,i)) \times A(i,k)
\end{align*}
\]
Refine GE Algorithm (1/5)

• Initial Version

```java
... 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.

```java
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 (2/5)

• Last version

```plaintext
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

```plaintext
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)
```

Do not compute zeros
Refine GE Algorithm (3/5)

• Last version

\[
\text{for } i = 1 \text{ to } n-1 \\
\quad \text{for } j = i+1 \text{ to } n \\
\quad \quad m = A(j,i)/A(i,i) \\
\quad \text{for } k = i+1 \text{ to } n \\
\quad \quad A(j,k) = A(j,k) - m \times A(i,k)
\]

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

\[
\text{for } i = 1 \text{ to } n-1 \\
\quad \text{for } j = i+1 \text{ to } n \\
\quad A(j,i) = A(j,i)/A(i,i) \\
\quad \text{for } k = i+1 \text{ to } n \\
\quad \quad A(j,k) = A(j,k) - A(j,i) \times A(i,k)
\]
Refine GE Algorithm (4/5)

• Last version

\[
\begin{align*}
\text{for } & i = 1 \text{ to } n-1 \\
& \text{for } j = i+1 \text{ to } n \\
& \quad A(j,i) = A(j,i)/A(i,i) \\
& \text{for } k = i+1 \text{ to } n \\
& \quad A(j,k) = A(j,k) - A(j,i) \times A(i,k)
\end{align*}
\]

• Split Loop

\[
\begin{align*}
\text{for } & i = 1 \text{ to } n-1 \\
& \text{for } j = i+1 \text{ to } n \\
& \quad A(j,i) = A(j,i)/A(i,i) \\
& \text{for } j = i+1 \text{ to } n \\
& \quad \text{for } k = i+1 \text{ to } n \\
& \quad A(j,k) = A(j,k) - A(j,i) \times A(i,k)
\end{align*}
\]

Store all m’s here before updating rest of matrix

9/26/2019 CS294-73 – Lecture 9 26
Refine GE Algorithm (5/5)

• Last version

\[
\text{for } i = 1 \text{ to } n-1 \\
\quad \text{for } j = i+1 \text{ to } n \\
\quad \quad A(j,i) = A(j,i)/A(i,i) \\
\text{for } j = i+1 \text{ to } n \\
\quad \quad \text{for } k = i+1 \text{ to } n \\
\quad \quad \quad \quad A(j,k) = A(j,k) - A(j,i) * A(i,k)
\]

• Express using matrix operations (BLAS)

Work at step $i$ of Gaussian Elimination

\[
\text{for } i = 1 \text{ to } n-1 \\
\quad \text{for } j = i+1 \text{ to } n \\
\quad \quad A(j,i) = A(j,i)/A(i,i) \\
\text{for } j = i+1 \text{ to } n \\
\quad \quad \text{for } k = i+1 \text{ to } n \\
\quad \quad \quad \quad A(j,k) = A(j,k) - A(j,i) * A(i,k)
\]

for $i = 1 \text{ to } n-1$

\[
A(i+1:n,i) = A(i+1:n,i) * (1/ A(i,i)) \\
\text{... BLAS 1 (scale a vector)} \\
A(i+1:n,i+1:n) = A(i+1:n , i+1:n) \\
\quad - A(i+1:n , i) * A(i , i+1:n) \\
\text{... BLAS 2 (rank-1 update)}
\]
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 $A = L * U$
- Solving $A * x = b$ using GE
  - Factorize $A = L * U$ using GE \( \text{(cost = } \frac{2}{3} n^3 \text{ flops)} \)
  - Solve $L * y = b$ for $y$, using substitution \( \text{(cost = } n^2 \text{ flops)} \)
  - Solve $U * x = y$ for $x$, using substitution \( \text{(cost = } n^2 \text{ flops)} \)
- Thus $A * x = (L * U) * x = L * (U * x) = L * y = b$ as desired
Problems with basic GE algorithm

for $i = 1$ to $n-1$

\[
A(i+1:n,i) = A(i+1:n,i) / A(i,i) \quad \ldots \text{BLAS 1 (scale a vector)}
\]

\[
A(i+1:n,i+1:n) = A(i+1:n, i+1:n) - A(i+1:n, i) \times A(i, i+1:n) \quad \ldots \text{BLAS 2 (rank-1 update)}
\]

- 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, with low computational intensities ($q \leq 2$), need to use BLAS 3 operations like matmul instead
Recursive, cache-oblivious GE formulation (1/3)

- Toledo (1997)
  - Describe without pivoting for simplicity
  - “Do left half of matrix, then right half”

\[
A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & I \end{pmatrix} \times \begin{pmatrix} U_{11} & U_{12} \\ 0 & S \end{pmatrix} = \begin{pmatrix} L_{11}*U_{11} & L_{11}*U_{12} \\ L_{21}*U_{11} & S+L_{21}*U_{12} \end{pmatrix}
\]

- Four step recursive algorithm
  1. Factor \( A_{11} = L_{11}*U_{11} = \begin{pmatrix} L_{11} \\ L_{21} \end{pmatrix} \times U_{11} \) same problem with half the columns: solve recursively
  2. Solve \( A_{12} = L_{11}*U_{12} \) for \( U_{12} \); call triangular solve (BLAS3)
  3. Solve \( A_{22} = S+L_{21}*U_{12} \) for \( S = A_{22} - L_{21}*U_{12} \); matmul (BLAS3)
  4. Factor \( S = L_{22}*U_{22} \) same problem with half the columns, fewer rows: solve recursively
Recursive, cache-oblivious GE formulation (2/3)

- Toledo (1997)
  - Describe without pivoting for simplicity
  - “Do left half of matrix, then right half”

\[
\begin{align*}
A &= L \times U
\end{align*}
\]

function \([L,U] = \text{RLU}(A)\) ... assume \(A\) is \(m\) by \(n\)

if \((n=1)\)
  \(L = A/A(1,1),\ U = A(1,1)\)
else

  \([L1,U1] = \text{RLU}(A(1:m, 1:n/2))\) ... do left half of \(A\)

  ... let \(L11\) denote top \(n/2\) rows of \(L1\)
  \(A(1:n/2, n/2+1:n) = L11^{-1} \times A(1:n/2, n/2+1:n)\)

  ... update \(A12\) (top \(n/2\) rows of right half of \(A\))
  \(A(n/2+1:m, n/2+1:n) = A(n/2+1:m, n/2+1:n)\)
  - \(A(n/2+1:m, 1:n/2) \times A(1:n/2, n/2+1:n)\)

  ... update rest of right half of \(A\), get \(S\)

  \([L2,U2] = \text{RLU}(A(n/2+1:m, n/2+1:n))\) ... do right half of \(A\)

return \([L1,[0;L2]]\) and \([U1, [A(:,,:) ; U2]]\)
Alternative cache-oblivious GE formulation (3/3)

function \([L,U] = RLU(A)\) … assume \(A\) is \(m\) by \(n\)

if \((n=1)\) \(L = A / A(1,1),\ U = A(1,1)\)

else

\([L1,U1] = RLU(A(1:m, 1:n/2))\) … do left half of \(A\)

… let \(L11\) denote top \(n/2\) rows of \(L1\)

\(A(1:n/2, n/2+1:n) = L11^{-1} * A(1:n/2, n/2+1:n)\)

… update \(A12\) (top \(n/2\) rows of right half of \(A\))

\(A(n/2+1:m, n/2+1:n) = A(n/2+1:m, n/2+1:n) - A(n/2+1:m, 1:n/2) * A(1:n/2, n/2+1:n)\)

… update rest of right half of \(A\), get \(S\)

\([L2,U2] = RLU(A(n/2+1:m, n/2+1:n))\) … do right half of \(A\)

return \([L1, [0; L2]]\) and \([U1, [A(:, :) ; U2]]\)

- Performs same flops as original algorithm, just in a different order
- \(W(m,n) = \#\) words moved to factor \(m\)-by-\(n\) matrix
  \[
  W(m,n) = W(m,n/2) + O(\max(m \cdot n, m \cdot n^2 / M^{1/2})) + W(m-n/2,n/2)
  \leq 2 \cdot W(m,n/2) + O(\max(m \cdot n, m \cdot n^2 / M^{1/2}))
  = O(m \cdot n^2 / M^{1/2} + m \cdot n \cdot \log M)
  = O(m \cdot n^2 / M^{1/2}) \quad \text{if} \ M^{1/2} \cdot \log M = O(n), \ i.e. \ usually \ attains \ lower \ bound
  \]
Other Topics

• Cache oblivious algorithms not a panacea
  • Recursing down to problems of size 1 add too much overhead
  • Need to switch to blocked algorithm for small enough subproblems

• Algorithms for other linear algebra problems (eg least squares, eigenvalue problems) more complicated
  • Minimizing communication requires different mathematical algorithms, not just different order of execution

• Dense linear algebra possible in $O(n^w)$ flops, with $w < 3$
  • Eg: Strassen’s algorithm has $w = \log_2 7 \sim 2.81$
  • Less communication too

• Only a few algorithms known in sparse case that minimize communication

• See Ma221, CS267 for details
How hard is hand-tuning matmul, anyway?

- Results of 22 student teams trying to tune matrix-multiply, in CS267 Spr09
- Students given “blocked” code to start with
  - Still hard to get close to vendor tuned performance (ACML)
- For more discussion, see www.cs.berkeley.edu/~volkov/cs267.sp09/hw1/results/
How hard is hand-tuning matmul, anyway?
Industrial-strength dense linear algebra

- Uses the correct abstractions to formulate the algorithms (block matrix-matrix multiply)
- Software building blocks so that those algorithms can be implemented efficiently with high performance (BLAS)
- Machines are really complicated, so some choices of algorithm parameters must be discovered empirically (autotuning).
Basic Linear Algebra Subroutines (BLAS)

- Industry standard interface (evolving)
  - [www.netlib.org/blas], [www.netlib.org/blas/blast--forum]
- Vendors, others supply optimized implementations
- History
  - BLAS1 (1970s):
    - vector operations: dot product, saxpy ($y = \alpha x + y$), etc
    - $m = 2n$, $f = 2n$, $q \sim 1$ or less
  - BLAS2 (mid 1980s)
    - matrix-vector operations: matrix vector multiply, etc
    - $m = n^2$, $f = 2n^2$, $q \sim 2$, less overhead
    - somewhat faster than BLAS1
  - BLAS3 (late 1980s)
    - matrix-matrix operations: matrix matrix multiply, etc
    - $m \leq 3n^2$, $f = O(n^3)$, so $q = f/m$ can possibly be as large as $n$, so BLAS3 is potentially much faster than BLAS2
- Good algorithms use BLAS3 when possible (LAPACK & ScaLAPACK)
Peak speed = 266 Mflops

BLAS 3 (n-by-n matrix matrix multiply) vs BLAS 2 (n-by-n matrix vector multiply) vs BLAS 1 (saxpy of n vectors)
Dense Linear Algebra: BLAS2 vs. BLAS3

- BLAS2 and BLAS3 have very different computational intensity, and therefore different performance

BLAS3 (MatrixMatrix) vs. BLAS2 (MatrixVector)

Data source: Jack Dongarra
Tuning Code in Practice

• Tuning code can be tedious
  • Lots of code variations to try besides blocking
  • Machine hardware performance hard to predict
  • Compiler behavior hard to predict

• Response: “Autotuning”
  • Let computer generate large set of possible code variations, and search them for the fastest ones
  • Field started with CS267 homework assignment in mid 1990s
    • PHiPAC, leading to ATLAS, incorporated in Matlab
    • We still use the same assignment
  • We (and others) are extending autotuning to other motifs

• Still need to understand how to do it by hand
  • Not every code will have an autotuner
  • Need to know if you want to build autotuners
Search Over Block Sizes

• Performance models are useful for high level algorithms
  • Helps in developing a blocked algorithm
  • Models have not proven very useful for block size selection
    • too complicated to be useful
    • too simple to be accurate
      – Multiple multidimensional arrays, virtual memory, etc.
  • Speed depends on matrix dimensions, details of code, compiler, processor
A 2-D slice of a 3-D register-tile search space. The dark blue region was pruned.
(Platform: Sun Ultra-Ili, 333 MHz, 667 Mflop/s peak, Sun cc v5.0 compiler)
ATLAS (DGEMM n = 500)

- ATLAS is faster than all other portable BLAS implementations and it is comparable with machine-specific libraries provided by the vendor.

Source: Jack Dongarra

Architectures

MLOPS

Vendor BLAS
ATLAS BLAS
F77 BLAS