Tuning for Caches

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

\[ d = 0 \]
\[
\text{do } i = 1, n \\
\quad j = \text{ind}(i) \\
\quad d = d + \sqrt{ x(j) \cdot x(j) + y(j) \cdot y(j) + z(j) \cdot z(j) } \\
\text{end do}
\]

- Change loop statement to
  \[ d = d + \sqrt{ r(1,j) \cdot r(1,j) + r(2,j) \cdot r(2,j) + r(3,j) \cdot r(3,j) } \]

- 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 \(i=1:n, j=1:n\), \(A(i,j) = B(i,j) + C(i,j)\)
- Algorithm 2:
  - for \(j=1:n, i=1:n\), \(A(i,j) = B(i,j) + C(i,j)\)
- 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]; }
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)+
      a(j+2,i+0)*x(j+2)+a(j+3,i+0)*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+1)*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)
  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

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

- Larger \( q \) means time closer to minimum \( f^* t_f \)
  - \( q \geq \frac{t_m}{t_f} \) needed to get at least half of peak speed

\( q: \) flops/memory reference

**Computational Intensity:** Key to algorithm efficiency

**Machine Balance:** Key to machine efficiency
Warm up: Matrix-vector multiplication

\[ y = y + A \times x \]

for \( i = 1:n \)

\[
\begin{align*}
  \text{for j} &= 1:n \\
  y(i) &= y(i) + A(i,j) \times x(j)
\end{align*}
\]
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 = \( 2*n^2 \)
° \( q = f/m \approx 2 \)
° Matrix-vector multiplication limited by slow memory speed
Matrix Multiply $C = C + A \times B$

for $i = 1$ to $n$

for $j = 1$ to $n$

for $k = 1$ to $n$

$C(i,j) = C(i,j) + A(i,k) \times B(k,j)$
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}
Matrix Multiply $C=C+A*B$
(unblocked, or untiled)

Number of slow memory references on unblocked matrix multiply

$$m = n^3 \text{ read each column of } B \text{ } n \text{ times}$$
$$+ n^2 \text{ read each row of } A \text{ once for each } i$$
$$+ 2*n^2 \text{ read and write each element of } C \text{ once}$$
$$= n^3 + 3*n^2$$

So $q = f/m = (2*n^3)/(n^3 + 3*n^2)$
$$\approx 2 \text{ for large } n, \text{ no improvement over matrix-vector mult}$$
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\}
      \(C(i,j) = C(i,j) + A(i,k) \times B(k,j)\) \{do a matrix multiply on blocks\}
    \{write block \(C(i,j)\) back to slow memory\}

\[\begin{array}{ccc}
  C(i,j) & = & C(i,j) \\
  & + & A(i,k) \times B(k,j)
\end{array}\]
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
      end do
    end do
  end do
end do
Matrix Multiply (blocked or tiled)

Why is this algorithm correct?

Number of slow memory references on blocked matrix multiply

\[ m = N * n^2 \text{ read each block of } B \text{ } N^3 \text{ times } (N^3 * n/N * n/N) \]
\[ + N * n^2 \text{ read each block of } A \text{ } N^3 \text{ times } (N^3 * n/N * n/N) \]
\[ + 2 * n^2 \text{ read and write each block of } C \text{ once} \]
\[ = (2*N + 2)*n^2 \]

So \( q = f/m = 2*n^3 / ((2*N + 2)*n^2) \)
\[ \approx n/N = b \text{ for large } n \]

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: \( 3*b^2 \leq M \), so \( q \approx b \leq \sqrt{M/3} \)

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
  $$\begin{align*}
  p_1 &= (a_{11} + a_{22}) \times (b_{11} + b_{22}) \\
  p_2 &= (a_{21} + a_{22}) \times b_{11} \\
  p_3 &= a_{11} \times (b_{12} - b_{22}) \\
  p_4 &= a_{22} \times (b_{21} - b_{11}) \\
  p_5 &= (a_{11} + a_{12}) \times b_{22} \\
  p_6 &= (a_{21} - a_{11}) \times (b_{11} + b_{12}) \\
  p_7 &= (a_{12} - a_{22}) \times (b_{21} + b_{22})
  \end{align*}$$

Then
  $$\begin{align*}
  m_{11} &= p_1 + p_4 - p_5 + p_7 \\
  m_{12} &= p_3 + p_5 \\
  m_{21} &= p_2 + p_4 \\
  m_{22} &= p_1 + p_3 - p_2 + p_6
  \end{align*}$$

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)

\[
\begin{pmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\
A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\
B_{21} & B_{22} \end{pmatrix}
\]

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)

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

° 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 \( O(n^{2.376..}) \)
Potential Project
Implement Strassen’s Method

- Write your matrix multiply using Strassen's method as discussed in class. Use the manufactured version of DGEMM to perform the matrix multiply parts you will need. Also compare the performance of your version of Strassen's matrix multiply with the ATLAS version. Be sure that you include a verification that you have the correct result.

- Do this sequentially then in parallel.