Tuning for Caches

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

\[ d = 0 \]

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

- 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];
    }
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

```fortran
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)
    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 \( j=1, n2-3, 4 \)
  \( t1=t1+a(j+0, i+0) * x(j+0) + a(j+1, i+0) * x(j+1) + \)
  \( a(j+2, i+0) * x(j+2) + a(j+3, i+0) * x(j+3) \)
  \( t2=t2+a(j+0, i+1) * x(j+0) + a(j+1, i+1) * x(j+1) + \)
  \( a(j+2, i+1) * x(j+2) + a(j+3, i+1) * x(j+3) \)
  \( t3=t3+a(j+0, i+2) * x(j+0) + a(j+1, i+2) * x(j+1) + \)
  \( a(j+2, i+2) * x(j+2) + a(j+3, i+2) * x(j+3) \)
  \( t4=t4+a(j+0, i+3) * x(j+0) + a(j+1, i+3) * x(j+1) + \)
  \( a(j+2, i+3) * x(j+2) + a(j+3, i+3) * x(j+3) \)
endo
\( y(i+0)=t1 \)
\( y(i+1)=t2 \)
\( y(i+2)=t3 \)
\( y(i+3)=t4 \)
endo
High Cost of Data Movement

<table>
<thead>
<tr>
<th>Operation</th>
<th>Energy consumed</th>
<th>Time needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-bit multiply-add</td>
<td>200 pJ</td>
<td>1 nsec</td>
</tr>
<tr>
<td>Read 64 bits from cache</td>
<td>800 pJ</td>
<td>3 nsec</td>
</tr>
<tr>
<td>Move 64 bits across chip</td>
<td>2000 pJ</td>
<td>5 nsec</td>
</tr>
<tr>
<td>Execute an instruction</td>
<td>7500 pJ</td>
<td>1 nsec</td>
</tr>
<tr>
<td>Read 64 bits from DRAM</td>
<td>12000 pJ</td>
<td>70 nsec</td>
</tr>
</tbody>
</table>

Notice that 12000 pJ @ 3 GHz = 36 watts!

Algorithms & Software: minimize data movement; perform more work per unit data movement.
Peak Performance - Per Core

Floating point operations per cycle per core

- Most of the recent computers have FMA (Fused multiple add): (i.e. $x \leftarrow x + y*z$ in one cycle)
- Intel Xeon earlier models and AMD Opteron have SSE2
  - 2 flops/cycle DP & 4 flops/cycle SP
- Intel Xeon Nehalem ('09) & Westmere ('10) have AVX
  - 4 flops/cycle DP & 8 flops/cycle SP
- Intel Xeon Sandy Bridge ('11) & Ivy Bridge ('12) have AVX
  - 8 flops/cycle DP & 16 flops/cycle SP
- Intel Xeon Haswell ('13) & (Broadwell ('14)) AVX2
  - 16 flops/cycle DP & 32 flops/cycle SP
- Xeon Phi (per core) is at 16 flops/cycle DP & 32 flops/cycle SP
- Intel Xeon Skylake ('15)
  - 32 flops/cycle DL & 64 flops/cycle SP

FLOPS = cores × clock × \frac{FLOPs}{cycle}
Memory transfer
(Its All About Data Movement)
Example on my laptop: One level of memory

The model IS simplified (see next slide) but it provides an upper bound on
performance as well. I.e., we will never go faster than what the model predicts.
( And, of course, we can go slower ... )
FMA: fused multiply-add

**AXPY:**

\[
\alpha \times x + y
\]

for \( j = 0; j < n; j++ \)

\[
y[i] += a \times x[i];
\]

(without increment)

\[
\text{n MUL}
\]

\[
\text{n ADD}
\]

\[
2n \text{ FLOP}
\]

\[
n \text{ FMA}
\]

**DOT:**

\[
\alpha \leftarrow x^T \times y
\]

alpha = \(0e+00\);

for \( j = 0; j < n; j++ \)

\[
\text{alpha} += x[i] \times y[i];
\]

(without increment)

\[
\text{n MUL}
\]

\[
\text{n ADD}
\]

\[
2n \text{ FLOP}
\]

\[
n \text{ FMA}
\]

Note: It is reasonable to expect the one loop codes shown here to perform as well as their Level 1 BLAS counterpart (on multicore with an OpenMP pragma for example).

The true gain these days with using the BLAS is (1) Level 3 BLAS, and (2) portability.
• Take two double precision vectors \( x \) and \( y \) of size \( n = 375,000 \).

• Data size:
  – \( (375,000 \text{ double}) \times (8 \text{ Bytes/\text{double}}) = 3 \text{ MBytes per vector} \)

  (Two vectors fit in cache (6 MBytes). OK.)

• Time to move the vectors from memory to cache:
  – \( (6 \text{ MBytes}) / (25.6 \text{ GBytes/sec}) = 0.23 \text{ ms} \)

• Time to perform computation of DOT:
  – \( (2n \text{ flop}) / (56 \text{ Gflop/sec}) = 0.01 \text{ ms} \)
Vector Operations

\[
\text{total\_time} \geq \max ( \text{time\_comm}, \text{time\_comp} ) \\
= \max (0.23\text{ms}, 0.01\text{ms}) = 0.23\text{ms}
\]

Performance = \(\frac{(2 \times 375,000 \text{ flops})}{0.23\text{ms}} = 3.2 \text{ Gflop/s}\)

**Performance for DOT \leq 3.2 \text{ Gflop/s}**

Peak is 56 Gflop/s

We say that the operation is communication bounded. No reuse of data.
Level 1, 2 and 3 BLAS

Level 1 BLAS Matrix-Vector operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Formula</th>
<th>FLOP</th>
<th>Memory Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXPY</td>
<td>$y \leftarrow \alpha x + y$</td>
<td>$2n$</td>
<td>$2n$ READ, $n$ WRITE</td>
</tr>
<tr>
<td>DOT</td>
<td>$\alpha \leftarrow x^T y$</td>
<td>$2n$</td>
<td>$2n$ READ</td>
</tr>
</tbody>
</table>

Level 2 BLAS Matrix-Vector operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Formula</th>
<th>FLOP</th>
<th>Memory References</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEMV</td>
<td>$y \leftarrow \alpha A x + y$</td>
<td>$2n^2$</td>
<td>$n^2$</td>
</tr>
</tbody>
</table>

Level 3 BLAS Matrix-Matrix operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Formula</th>
<th>FLOP</th>
<th>Memory References</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEMM</td>
<td>$C \leftarrow \alpha A B + \beta C$</td>
<td>$2n^3$</td>
<td>$3n^2$ READ, $n^2$ WRITE</td>
</tr>
</tbody>
</table>

RATIO:
- Level 1: 1
- Level 2: 2
- Level 3: $\frac{2}{3} n$
• Double precision matrix $A$ and vectors $x$ and $y$ of size $n=860$.

• Data size:
  
  $-(860^2 + 2\times860 \text{ double}) \times (8 \text{ Bytes} / \text{ double}) \sim 6 \text{ MBytes}$

  Matrix and two vectors fit in cache (6 MBytes).

• Time to move the data from memory to cache:
  
  $-(6 \text{ MBytes}) / (25.6 \text{ GBytes/sec}) = 0.23 \text{ ms}$

• Time to perform computation of DOT:
  
  $-(2n^2 \text{ flop}) / (56 \text{ Gflop/sec}) = 0.26 \text{ ms}$
Matrix - Vector Operations

total_time \geq \max ( \text{time}_{\text{comm}}, \text{time}_{\text{comp}} )

= \max ( 0.23\text{ms}, 0.26\text{ms} ) = 0.26\text{ms}

Performance = (2 \times 860^2 \text{flops})/0.26\text{ms} = 5.7 \text{Gflop/s}

**Performance for GEMV \leq 5.7 \text{Gflop/s}**

**Performance for DOT \leq 3.2 \text{Gflop/s}**

Peak is 56 Gflop/s

We say that the operation is communication bounded. Very little reuse of data.
• Take two double precision vectors $x$ and $y$ of size $n=500$.

• Data size:
  
  $- (500^2 \text{ double}) \times (8 \text{ Bytes} / \text{ double}) = 2 \text{ MBytes per matrix}$

  (Three matrices fit in cache (6 MBytes). OK.)

• Time to move the matrices in cache:
  
  $- (6 \text{ MBytes}) / (25.6 \text{ GBytes/sec}) = 0.23 \text{ ms}$

• Time to perform computation in GEMM:
  
  $- (2n^3 \text{ flop}) / (56 \text{ Gflop/sec}) = 4.46 \text{ ms}$
Matrix Matrix Operations

total_time ≥ \text{max} (\text{time\_comm, time\_comp})

= \text{max}(0.23\text{ms}, 4.46\text{ms}) = 4.46\text{ms}

For this example, communication time is less than 6% of the computation time.

Performance = (2 \times 500^3 \text{flops})/4.69\text{ms} = 53.3 \text{Gflop/s}

There is a lot of data reuse in a GEMM; 2/3n per data element. Has good temporal locality.

If we assume total_time \approx time\_comm + time\_comp, we get

\text{Performance for GEMM} \approx 53.3 \text{Gflop/sec}

\text{Performance for DOT} \leq 3.2 \text{Gflop/s}
\text{Performance for GEMV} \leq 5.7 \text{Gflop/s}

(Out of 56 \text{Gflop/sec possible, so that would be 95\% peak performance efficiency.})
Level 1, 2 and 3 BLAS

1 core Intel Haswell i7-4850HQ, 2.3 GHz (Turbo Boost at 3.5 GHz);
Peak = 56 Gflop/s

Matrix (Vector) Size N

1 core Intel Haswell i7-4850HQ, 2.3 GHz, Memory: DDR3L-1600MHz
6 MB shared L3 cache, and each core has a private 256 KB L2 and 64 KB L1.
The theoretical peak per core double precision is 56 Gflop/s per core.
Compiled with gcc and using Veclib
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 $C = C + A \times B$
(unblocked, or untiled)

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 $i$
+ $2n^2$  read and write each element of $C$ once
= $n^3 + 3n^2$

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

$\sim 2$ 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}
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

\[ m = N \cdot n^2 \quad \text{read each block of } B \quad N^3 \times (N^3 \cdot n/N \cdot n/N) \]

\[ + N \cdot n^2 \quad \text{read each block of } A \quad N^3 \times (N^3 \cdot n/N \cdot n/N) \]

\[ + 2 \cdot n^2 \quad \text{read and write each block of } C \text{ once} \]

\[ = (2 \cdot N + 2) \cdot n^2 \]

So \( q = \frac{f}{m} = \frac{2 \cdot n^3}{(2 \cdot N + 2) \cdot n^2} \)

\[ \sim n/N = b \quad \text{for large } n \]

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

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 multiplies and 18 additions.

Let $p1 = (a_{11} + a_{22}) \times (b_{11} + b_{22})$

$p2 = (a_{21} + a_{22}) \times b_{11}$

$p3 = a_{11} \times (b_{12} - b_{22})$

$p4 = a_{22} \times (b_{21} - b_{11})$

Then $m_{11} = p1 + p4 - p5 + p7$

$m_{12} = p3 + p5$

$m_{21} = p2 + p4$

$m_{22} = p1 + p3 - p2 + p6$

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)

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

Phase 1 (temporary sums)

\begin{align*}
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 \text{ (corrected 3/2/09)} \\
T10 &= B21 + B22
\end{align*}

Phase 2 (temporary products)

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

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

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

= \(O(n^{2.81})\)

° 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 \(O(n^{2.376..})\)
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.