STORAGE OF C ARRAYS

Consider the C array declared by

dooble A[5][4];

Since memory is linear, the array must be unpacked in some systematic way.

C does it in row-major order.

\[
\begin{array}{cccccc}
  a & a + 8 & a + 16 & a + 24 \\
  a + 32 & a + 40 & a + 48 & a + 56 \\
  a + 64 & a + 72 & a + 80 & a + 88 \\
  a + 96 & a + 104 & a + 112 & a + 120 \\
  a + 128 & a + 136 & a + 144 & a + 152 \\
\end{array}
\]
Locality Example

**Claim:** Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.

**Question:** Does this function have good locality?

```c
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];

    return sum
}
```
Locality Example

Question: Does this function have good locality?

```c
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum
}
```
### STORAGE OF FORTRAN ARRAYS

Consider the Fortran array

- **double precision** \( A(4,5) \)

It is stored in **column-major order**.

<table>
<thead>
<tr>
<th>( a )</th>
<th>( a + 40 )</th>
<th>( a + 80 )</th>
<th>( a + 120 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A(1,1) )</td>
<td>( A(1,2) )</td>
<td>( A(1,3) )</td>
<td>( A(1,4) )</td>
</tr>
<tr>
<td>( a + 8 )</td>
<td>( a + 48 )</td>
<td>( a + 88 )</td>
<td>( a + 128 )</td>
</tr>
<tr>
<td>( A(2,1) )</td>
<td>( A(2,2) )</td>
<td>( A(2,3) )</td>
<td>( A(2,4) )</td>
</tr>
<tr>
<td>( a + 16 )</td>
<td>( a + 56 )</td>
<td>( a + 96 )</td>
<td>( a + 136 )</td>
</tr>
<tr>
<td>( A(3,1) )</td>
<td>( A(3,2) )</td>
<td>( A(3,3) )</td>
<td>( A(3,4) )</td>
</tr>
<tr>
<td>( a + 24 )</td>
<td>( a + 64 )</td>
<td>( a + 104 )</td>
<td>( a + 144 )</td>
</tr>
<tr>
<td>( A(4,1) )</td>
<td>( A(4,2) )</td>
<td>( A(4,3) )</td>
<td>( A(4,4) )</td>
</tr>
<tr>
<td>( a + 32 )</td>
<td>( a + 72 )</td>
<td>( a + 112 )</td>
<td>( a + 152 )</td>
</tr>
<tr>
<td>( A(5,1) )</td>
<td>( A(5,2) )</td>
<td>( A(5,3) )</td>
<td>( A(5,4) )</td>
</tr>
</tbody>
</table>
ACCESSING ARRAYS FOR CACHE EFFICIENCY

If we have to access all the elements of an array, we should access them with unit stride. Thus in C we should access the array by rows:

\[
A[0][0] \ A[0][1] \ A[0][2] \ A[0][3] \ A[1][0] \ A[1][1] \ldots
\]

In Fortran we should access the array by rows.

\[
A(1,1) \ A(2,1) \ A(3,1) \ A(4,1) \ A(5,1) \ A(1,2) \ A(2,2) \ldots
\]

This means that we must code algorithms differently in C and Fortran.
Writing Cache Friendly Code

Repeated references to variables are good (temporal locality)

Stride-1 reference patterns are good (spatial locality)

Examples:

- cold cache, 4-byte words, 4-word cache blocks

```c
int sumarrayrows(int a[M][N]) {
    int i, j, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```c
int sumarraycols(int a[M][N]) {
    int i, j, sum = 0;
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Miss rate = 1/4 = 25%

Miss rate = 100%
The Memory Mountain

Read throughput (read bandwidth)
- Number of bytes read from memory per second (MB/s)

Memory mountain
- Measured read throughput as a function of spatial and temporal locality.
- Compact way to characterize memory system performance.
/* The test function */
void test(int elems, int stride) {
    int i, result = 0;
    volatile int sink;

    for (i = 0; i < elems; i += stride)
        result += data[i];
    sink = result; /* So compiler doesn't optimize away the loop */
}

/* Run test(elems, stride) and return read throughput (MB/s) */
double run(int size, int stride, double Mhz)
{
    double cycles;
    int elems = size / sizeof(int);

    test(elems, stride); /* warm up the cache */
    cycles = fcyc2(test, elems, stride, 0); /* call test(elems,stride) */
    return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}
Memory Mountain Main Routine

```c
/* mountain.c - Generate the memory mountain. */
#define MINBYTES (1 << 10)  /* Working set size ranges from 1 KB */
#define MAXBYTES (1 << 23)  /* ... up to 8 MB */
#define MAXSTRIDE 16        /* Strides range from 1 to 16 */
#define MAXELEMS MAXBYTES/sizeof(int)

int data[MAXELEMS];         /* The array we'll be traversing */

int main()
{
    int size;        /* Working set size (in bytes) */
    int stride;      /* Stride (in array elements) */
    double Mhz;      /* Clock frequency */

    init_data(data, MAXELEMS); /* Initialize each element in data to 1 */
    Mhz = mhz(0);              /* Estimate the clock frequency */
    for (size = MAXBYTES; size >= MINBYTES; size >>= 1) {
        for (stride = 1; stride <= MAXSTRIDE; stride++)
            printf("%.1f\t", run(size, stride, Mhz));
        printf("\n");
    }
    exit(0);
}
```
Ridges of Temporal Locality

Slice through the memory mountain with stride=1
- illuminates read throughputs of different caches and memory

![Bar chart showing read throughputs for different working set sizes in various memory regions: main memory region, L2 cache region, and L1 cache region. The x-axis represents working set size in bytes (8m, 4m, 2m, 1024k, 512k, 256k, 128k, 64k, 32k, 16k, 8k, 4k, 2k, 1k) and the y-axis represents read throughput in MB/s (0 to 1200 MB/s).]
A Slope of Spatial Locality

Slice through memory mountain with size=256KB

- shows cache block size.

![Bar graph showing read throughput (MB/s) vs stride (words) with one access per cache line.](image-url)
MATRICES-VECTOR MULTIPLICATION (COLUMN ORIENTED)

Consider the problem of computing $Ax$, where $A$ is $n \times n$.

Partition $A$ by columns:

$$
\begin{pmatrix}
    x_1 \\
    x_2 \\
    \vdots \\
    x_n
\end{pmatrix}
= x_1 a_1 + x_2 a_2 + \cdots + a_n x_n.
$$

This gives the algorithm (in Matlab)

```matlab
y = 0;
for j = 1:n
    y = y + x(j)*A(:,j);  \% This is an AXPY.
end
```

In scalar form

```matlab
for i=1:n
    y(i) = 0;
end
for j=1:n
    for i=1:n
        y(i) = y(i) + x(j)*A(i,j);
    end
end
```

MATRICES – VECTOR MULTIPLICATION (ROW ORIENTED)

Partition $A$ in the form

$$
\begin{pmatrix}
  y_1 \\
  y_2 \\
  \vdots \\
  y_n \\
\end{pmatrix}
= 
\begin{pmatrix}
  a_1^T \\
  a_2^T \\
  \vdots \\
  a_n^T \\
\end{pmatrix}
\begin{pmatrix}
  x \\
\end{pmatrix}
= 
\begin{pmatrix}
  a_1^T x \\
  a_2^T x \\
  \vdots \\
  a_n^T x \\
\end{pmatrix}
$$

This gives the program

```
for i=1:n
    y(i) = A(i,:)*x;  % This is a DOT.
end
```

In scalar form

```
for i=1:n
    y(i) = 0;
    for j=1:n
        y(i) = y(i) + A(i,j)*x(j);
    end
end
```
MATRIX-MATRIX MULTIPLICATION (COLUMN ORIENTED)

Consider the product $C = AB$ and partition $C$ and $B$ by columns.

$$(c_1 \ c_2 \ \cdots \ c_n) = A(b_1 \ b_2 \ \cdots \ b_n) = (Ab_1 \ Ab_2 \ \cdots \ Ab_n).$$

We can then use AXPY’s to compute the products $Ab_k$.

```matlab
for k=1:n
    C(:,k) = 0;
    for j=1:n
        for i=1:n
            C(i,k) = C(i,k) + A(i,j)*B(j,k);
        end
    end
end
```
SPATIAL AND TEMPORAL LOCALITY OF REFERENCE

for k=1:n
    C(:,k) = 0;
    for j=1:n
        for i=1:n
            C(i,k) = C(i,k) + A(i,j)*B(j,k);
        end
    end
end

Locality of reference comes in two types.

Spatial locality means that when a memory reference to an address is made, it is surrounded by references to nearby addresses.

The above algorithm has good spatial locality (in a column oriented language) because it references its arrays by columns.

Temporal locality means that when a memory reference to an address is made, the address is reused frequently before it must be swapped out.

The above algorithm does not have good temporal locality because it makes n passes over the array A.
Matrix Multiplication Example

Major Cache Effects to Consider

- Total cache size
  - Exploit temporal locality and keep the working set small (e.g., by using blocking)
- Block size
  - Exploit spatial locality

Description:

- Multiply N x N matrices
- O(N^3) total operations
- Accesses
  - N reads per source element
  - N values summed per destination
  - but may be able to hold in register

```
/* ijk */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```
Miss Rate Analysis for Matrix Multiply

Assume:

- Line size = 32B (big enough for 4 64-bit words)
- Matrix dimension (N) is very large
  - Approximate 1/N as 0.0
- Cache is not even big enough to hold multiple rows

Analysis Method:

- Look at access pattern of inner loop
Layout of C Arrays in Memory (review)

C arrays allocated in row-major order
- each row in contiguous memory locations

Stepping through columns in one row:
- for (i = 0; i < N; i++)
  \[ \text{sum} += a[0][i]; \]
- accesses successive elements
- if block size (B) > 4 bytes, exploit spatial locality
  - compulsory miss rate = \(4 \text{ bytes} / B\)

Stepping through rows in one column:
- for (i = 0; i < n; i++)
  \[ \text{sum} += a[i][0]; \]
- accesses distant elements
- no spatial locality!
  - compulsory miss rate = 1 (i.e. 100%)
Matrix Multiplication (ijk)

```c
/* ijk */
for (i=0; i<n; i++)  {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

**Misses per Inner Loop Iteration:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.25</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Matrix Multiplication (jik)

```c
/* jik */
for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum
    }
}
```

**Misses per Inner Loop Iteration:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.25</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Matrix Multiplication (kij)

/* kij */
for (k=0; k<n; k++) {
    for (i=0; i<n; i++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}

Misses per Inner Loop Iteration:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misses</td>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>
Matrix Multiplication (ikj)

/* ikj */
for (i=0; i<n; i++) {
    for (k=0; k<n; k++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}

Inner loop:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>

**Misses per Inner Loop Iteration:**
Matrix Multiplication (jki)

```c
/* jki */
for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}
```

**Misses per Inner Loop Iteration:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Inner loop:

- Column-wise
- Fixed
- Column-wise
Matrix Multiplication (kji)

```c
/* kji */
for (k=0; k<n; k++) {
    for (j=0; j<n; j++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}
```

**Misses per Inner Loop Iteration:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misses</td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
### Summary of Matrix Multiplication

**ijk (& jik):**
- 2 loads, 0 stores
- misses/iter = 1.25

```c
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

**kij (& ikj):**
- 2 loads, 1 store
- misses/iter = 0.5

```c
for (k=0; k<n; k++) {
    for (i=0; i<n; i++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}
```

**jki (& kji):**
- 2 loads, 1 store
- misses/iter = 2.0

```c
for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}
```
Miss rates are helpful but not perfect predictors.

- Code scheduling matters, too.
Improving Temporal Locality by Blocking

Example: Blocked matrix multiplication

- “block” (in this context) does not mean “cache block”.
- Instead, it mean a sub-block within the matrix.
- Example: \( N = 8 \); sub-block size = 4

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

Key idea: Sub-blocks (i.e., \( A_{xy} \)) can be treated just like scalars.

\[
C_{11} = A_{11}B_{11} + A_{12}B_{21} \\
C_{12} = A_{11}B_{12} + A_{12}B_{22} \\
C_{21} = A_{21}B_{11} + A_{22}B_{21} \\
C_{22} = A_{21}B_{12} + A_{22}B_{22}
\]
We can increase temporal locality by blocking. Partition $C = AB$ in the form

$$
\begin{pmatrix}
C_{11} & C_{12} & \cdots & C_{1\ell} \\
C_{21} & C_{22} & \cdots & C_{2\ell} \\
\vdots & \vdots & \ddots & \vdots \\
C_{\ell 1} & C_{\ell 2} & \cdots & C_{\ell\ell}
\end{pmatrix}
= \begin{pmatrix}
A_{11} & A_{12} & \cdots & A_{1\ell} \\
A_{21} & A_{22} & \cdots & A_{2\ell} \\
\vdots & \vdots & \ddots & \vdots \\
A_{\ell 1} & A_{\ell 2} & \cdots & A_{\ell\ell}
\end{pmatrix}
\begin{pmatrix}
B_{11} & B_{12} & \cdots & B_{1\ell} \\
B_{21} & B_{22} & \cdots & B_{2\ell} \\
\vdots & \vdots & \ddots & \vdots \\
B_{\ell 1} & B_{\ell 2} & \cdots & B_{\ell\ell}
\end{pmatrix},
$$

where all the blocks are $m \times m$.

The our blocked algorithm is

1. for $k=1:m:n$
2. $kk = k+m-1$;
3. $C(:,k:kk) = 0$;
4. for $j=1:m:n$
5. $jj = j+m-1$;
6. for $i=1:m:n$
7. $ii = i+m-1$;
9. end
10. end
11. end
1. for k=1:m:n
2.   kk = k+m-1;
3.   C(:,k:kk) = 0;
4.   for j=1:m:n
5.     jj = j+m-1;
6.     for i=1:m:n
7.       ii = i+m-1;
8.       C(i:ii,k:kk) = C(i:ii,k:kk) + A(i:ii,j:jj)*B(j:jj,k:kk);
9.     end
10.   end
11. end

If the blocks A(i:ii,j:jj), B(j:jj,k:kk), an C(i:ii,k:kk) all fit into cache, then it does not matter how they are multiplied.

We now make only n/m passes over A.
Blocked Matrix Multiply (bijk)

```c
for (jj=0; jj<n; jj+=bsize) {
    for (i=0; i<n; i++)
        for (j=jj; j < min(jj+bsize,n); j++)
            c[i][j] = 0.0;
    for (kk=0; kk<n; kk+=bsize) {
        for (i=0; i<n; i++) {
            for (j=jj; j < min(jj+bsize,n); j++) {
                sum = 0.0
                for (k=kk; k < min(kk+bsize,n); k++) {
                    sum += a[i][k] * b[k][j];
                }
                c[i][j] += sum;
            }
        }
    }
}
```
Block Matrix Multiply Analysis

- Innermost loop pair multiplies a $1 \times bsize$ sliver of $A$ by a $bsize \times bsize$ block of $B$ and accumulates into $1 \times bsize$ sliver of $C$

- Loop over $i$ steps through $n$ row slivers of $A$ & $C$, using same $B$

```c
for (i=0; i<n; i++) {
    for (j=jj; j < min(jj+bsize,n); j++) {
        sum = 0.0
        for (k=kk; k < min(kk+bsize,n); k++) {
            sum += a[i][k] * b[k][j];
        }
        c[i][j] += sum;
    }
}
```

- $i$ and $j$ are row slivers accessed $bsize$ times
- $k$ is the block reused $n$ times in succession
- Update successive elements of sliver
Blocking (bijk and bikj) improves performance by a factor of two over unblocked versions (ijk and jik)

- relatively insensitive to array size.
Concluding Observations

**Programmer can optimize for cache performance**

- How data structures are organized
- How data are accessed
  - Nested loop structure
  - Blocking is a general technique

**All systems favor “cache friendly code”**

- Getting absolute optimum performance is very platform specific
  - Cache sizes, line sizes, associativities, etc.
- Can get most of the advantage with generic code
  - Keep working set reasonably small (temporal locality)
  - Use small strides (spatial locality)