Many of the following slides are taken with permission from

**Complete Powerpoint Lecture Notes for Computer Systems: A Programmer's Perspective (CS:APP)**

*Randal E. Bryant* and *David R. O'Hallaron*

http://csapp.cs.cmu.edu/public/lectures.html

The book is used explicitly in CS 2505 and CS 3214 and as a reference in CS 2506.
Locality Example (1)

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

**Question:** Which of these functions has 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;
}

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;
}
```
Layout of C Arrays in Memory

C arrays allocated in contiguous memory locations with addresses ascending with the array index:

```c
int32_t A[10] = {0, 1, 2, 3, 4, ..., 8, 9};
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>80430000</td>
<td>0</td>
</tr>
<tr>
<td>80430004</td>
<td>1</td>
</tr>
<tr>
<td>80430008</td>
<td>2</td>
</tr>
<tr>
<td>8043000C</td>
<td>3</td>
</tr>
<tr>
<td>80430010</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>80430048</td>
<td>8</td>
</tr>
<tr>
<td>8043004C</td>
<td>9</td>
</tr>
</tbody>
</table>
Layout of C Arrays in Memory

Two-dimensional C arrays allocated in *row-major order* - each row in contiguous memory locations:

```c
int32_t A[3][5] =
{ { 0,  1,  2,  3,  4},
{10, 11, 12, 13, 14},
{20, 21, 22, 23, 24},
};
```
int32_t A[3][5] = 
{  { 0,  1,  2,  3,  4},
   {10, 11, 12, 13, 14},
   {20, 21, 22, 23, 24},
};

Stepping through columns in one row:

for (i = 0; i < 3; i++)
   for (j = 0; j < 5; j++)
      sum += A[i][j];

- accesses successive elements in memory

- if cache block size B > 4 bytes, exploit spatial locality
  compulsory miss rate = 4 bytes / B
Layout of C Arrays in Memory

```c
int32_t A[3][5] =
{ { 0,  1,  2,  3,  4},
  {10, 11, 12, 13, 14},
  {20, 21, 22, 23, 24},
};
```

Stepping through rows in one column:

```c
for (j = 0; i < 5; i++)
  for (i = 0; i < 3; i++)
    sum += a[i][j];
```

accesses distant elements

no spatial locality!

compulsory miss rate = 1 (i.e. 100%)
Writing Cache Friendly Code

Repeated references to variables are good (temporal locality)

Stride-1 reference patterns are good (spatial locality)

Assume an initially-empty cache with 16-byte 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;
}
```

Miss rate = $1/4 = 25\%$
Consider the previous slide, but assume that the cache uses a block size of 64 bytes instead of 16 bytes.

```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;
}
```

Miss rate = 1/16 = 6.25%
"Skipping" accesses down the rows of a column do not provide 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;
}
```

**Miss rate = 100%**

(That's actually somewhat pessimistic... depending on cache geometry.)
Locality Example (2)

**Question:** Can you permute the loops so that the function scans the 3D array `a[]` with a stride-1 reference pattern (and thus has good spatial locality)?

```c
int sumarray3d(int a[N][N][N])
{
    int i, j, k, sum = 0;

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

    return sum;
}
```
It's easy to write an array traversal and see the addresses at which the array elements are stored:

```c
int A[5] = {0, 1, 2, 3, 4};
for (i = 0; i < 5; i++)
    printf("%d:  %X\n",  
i, (unsigned)&A[i]);
```

We see there that for a 1D array, the index varies in a stride-1 pattern.

<table>
<thead>
<tr>
<th>i</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>28ABE0</td>
</tr>
<tr>
<td>1</td>
<td>28ABE4</td>
</tr>
<tr>
<td>2</td>
<td>28ABE8</td>
</tr>
<tr>
<td>3</td>
<td>28ABEC</td>
</tr>
<tr>
<td>4</td>
<td>28ABF0</td>
</tr>
</tbody>
</table>

} stride-1 : addresses differ by the size of an array cell (4 bytes, here)
We see that for a 2D array, the second index varies in a stride-1 pattern.

But the first index does not vary in a stride-1 pattern.

```
int B[3][5] = { ... };
for (i = 0; i < 3; i++)
  for (j = 0; j < 5; j++)
    printf("%d %3d: %X\n",
           i, j, (unsigned)&B[i][j]);
```
### Layout of C Arrays in Memory

```c
int C[2][3][5] = { ... };

for (i = 0; i < 2; i++)
    for (j = 0; j < 3; j++)
        for (k = 0; k < 5; k++)
            printf("%3d %3d %3d: %d
", i, j, k, (unsigned)&C[i][j][k]);
```

We see that for a 3D array, the third index varies in a stride-1 pattern:

<table>
<thead>
<tr>
<th>i-j-k order:</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0:</td>
<td>28CC1C</td>
</tr>
<tr>
<td>0 0 1:</td>
<td>28CC20</td>
</tr>
<tr>
<td>0 0 2:</td>
<td>28CC24</td>
</tr>
<tr>
<td>0 0 3:</td>
<td>28CC28</td>
</tr>
<tr>
<td>0 0 4:</td>
<td>28CC2C</td>
</tr>
<tr>
<td>0 1 0:</td>
<td>28CC30</td>
</tr>
<tr>
<td>0 1 1:</td>
<td>28CC34</td>
</tr>
<tr>
<td>0 1 2:</td>
<td>28CC38</td>
</tr>
</tbody>
</table>

But... if we change the order of access, we no longer have a stride-1 pattern:

<table>
<thead>
<tr>
<th>k-j-i order:</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0:</td>
<td>28CC24</td>
</tr>
<tr>
<td>1 0 0:</td>
<td>28CC60</td>
</tr>
<tr>
<td>0 1 0:</td>
<td>28CC38</td>
</tr>
<tr>
<td>1 1 0:</td>
<td>28CC74</td>
</tr>
<tr>
<td>0 2 0:</td>
<td>28CC4C</td>
</tr>
<tr>
<td>1 2 0:</td>
<td>28CC88</td>
</tr>
<tr>
<td>0 0 1:</td>
<td>28CC28</td>
</tr>
<tr>
<td>0 1 1:</td>
<td>28CC3C</td>
</tr>
<tr>
<td>0 2 1:</td>
<td>28CC50</td>
</tr>
<tr>
<td>0 0 2:</td>
<td>28CC2C</td>
</tr>
<tr>
<td>0 1 2:</td>
<td>28CC40</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i-k-j order:</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0:</td>
<td>28CC24</td>
</tr>
<tr>
<td>0 1 0:</td>
<td>28CC38</td>
</tr>
<tr>
<td>0 2 0:</td>
<td>28CC4C</td>
</tr>
<tr>
<td>0 0 1:</td>
<td>28CC28</td>
</tr>
<tr>
<td>0 1 1:</td>
<td>28CC3C</td>
</tr>
<tr>
<td>0 2 1:</td>
<td>28CC50</td>
</tr>
<tr>
<td>0 0 2:</td>
<td>28CC2C</td>
</tr>
<tr>
<td>0 1 2:</td>
<td>28CC40</td>
</tr>
</tbody>
</table>
Locality Example (2)

Question: Can you permute the loops so that the function scans the 3D array \(a[]\) with a stride-1 reference pattern (and thus has good spatial locality)?

```c
int sumarray3d(int a[N][N][N])
{
    int i, j, k, sum = 0;
    
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                sum += a[k][i][j];
    return sum;
}
```

This code does not yield good locality at all.

The inner loop is varying the first index, worst case!
Locality Example (3)

**Question:** Which of these two exhibits better spatial locality?

// struct of arrays
struct soa {
    float *x;
    float *y;
    float *z;
    float *r;
};

compute_r(struct soa s) {
    for (i = 0; ...) {
        s.r[i] = s.x[i] * s.x[i]
            + s.y[i] * s.y[i]
            + s.z[i] * s.z[i];
    }
}

// array of structs
struct aos {
    float x;
    float y;
    float z;
    float r;
};

compute_r(struct aos *s) {
    for (i = 0; ...) {
        s[i].r = s[i].x * s[i].x
            + s[i].y * s[i].y
            + s[i].z * s[i].z;
    }
}
Locality Example (3)

// struct of arrays
struct soa {
    float *x;
    float *y;
    float *z;
    float *r;
};
struct soa s;
s.x = malloc(8*sizeof(float));
...

// array of structs
struct aos {
    float x;
    float y;
    float r;
};
struct aos s[8];
Locality Example (3)

Question: Which of these two exhibits better spatial locality?

// struct of arrays
compute_r(struct soa s) {
    for (i = 0; ...) {
        s.r[i] = s.x[i] * s.x[i]
        + s.y[i] * s.y[i]
        + s.z[i] * s.z[i];
    }
}

// array of structs
compute_r(struct aos *s) {
    for (i = 0; ...) {
        s[i].r = s[i].x * s[i].x
        + s[i].y * s[i].y
        + s[i].z * s[i].z;
    }
}
Locality Example (4)

Question: Which of these two exhibits better spatial locality?

// struct of arrays
sum_r(struct soa s) {
    sum = 0;
    for (i = 0; ...) {
        sum += s.r[i];
    }
}

// array of structs
sum_r(struct aos *s) {
    sum = 0;
    for (i = 0; ...) {
        sum += s[i].r;
    }
}
QTP: How would this compare to the previous two?

```
// array of pointers to structs
struct aos {
    float x;
    float y;
    float z;
    float r;
};

struct *aops[8];

for (i = 0; i < 8; i++)
    apos[i] = malloc(sizeof(struct aos));
```
Writing Cache Friendly Code

Make the common case go fast
  - Focus on the inner loops of the core functions

Minimize the misses in the inner loops
  - Repeated references to variables are good (temporal locality)
  - Stride-1 reference patterns are good (spatial locality)

Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories.
Miss Rate Analysis for Matrix Multiply

Assume:

- Line size = 32B (big enough for four 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
Matrix Multiplication Example

Description:
- Multiply N x N matrices
- \(O(N^3)\) total operations
- \(N\) reads per source element
- \(N\) values summed per destination

```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;
    }
}
```

Variable `sum` held in register
Matrix Multiplication (ijk)

/* 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;
    }
}

Inner loop:

<table>
<thead>
<tr>
<th></th>
<th>(i,*)</th>
<th>(*)</th>
<th>(i,j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Row-wise  Column-wise  Fixed

Misses per inner loop iteration:

A: 0.25    B: 1.0    C: 0.0
Matrix Multiplication (kij)

```c
/* 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];
    }
}
```

Inner loop:

- **A** (i,k)
- **B** (k,*)
- **C** (i,*)

Fixed: Row-wise

Row-wise

Row-wise

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>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>
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>
Summary of Matrix Multiplication

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;
    }
}

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];
    }
}

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;
}

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

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

jki (& kji):
- 2 loads, 1 store
- misses/iter = 2.0
Core i7 Matrix Multiply Performance

- jki / kji
- ijk / jik
- kij / ikj

Cycles per inner loop iteration vs. Array size (n)
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)