Writing Cache Friendly Code
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.
Today

- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using tiling to improve temporal locality
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.
Memory Mountain Test Function

```c
/* 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 */
}
```
The Memory Mountain

Intel Core i7
32 KB L1 i-cache
32 KB L1 d-cache
256 KB unified L2 cache
8M unified L3 cache
All caches on-chip
The Memory Mountain

Stride (x8 bytes)  Working set size (bytes)

Read throughput (MB/s)

Slopes of spatial locality

Intel Core i7
32 KB L1 i-cache
32 KB L1 d-cache
256 KB unified L2 cache
8M unified L3 cache
All caches on-chip
The Memory Mountain

Intel Core i7
32 KB L1 i-cache
32 KB L1 d-cache
256 KB unified L2 cache
8M unified L3 cache
All caches on-chip

Ridges of Temporal locality
Slopes of spatial locality
Warm-up example

- **Description:**
  - 1 array of length n
  - Walk array m times

- **Assumptions**
  - 32B cache blocks (fit four, 64-bit (8B) doubles)
  - n is large (array much bigger than cache)
  - m is large (approximate 1/m as 0)

- **Average miss rate?**
  - A) 0  B) 1/8  C) 1/4  D) 1/2  E) 1

```c
double sum = 0.0;
double a[n];
for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        sum += a[j];
    }
}
```
Loop inversion swaps the indexing variable of the inner loop (j i indexes the inner loop)

- **Same Assumptions**
  - 32B cache blocks (fit four, 64-bit (8B) doubles)
  - n is large (array much bigger than cache)
  - m is large (approximate 1/m as 0)

- **Average miss rate?**
  - A) 0
  - B) 1/8
  - C) 1/4
  - D) 1/2
  - E) 1

```c
double sum = 0.0;
double a[n];
for (j = 0; j < n; j ++) {
    for (i = 0; i < n; i ++) {
        sum += a[j];
    }
}
```
Loop Fusion joins two loops that traverse the same cache blocks, increasing temporal locality

```
for(int j = 0; j < LARGE; j++) {
    sum += A[j];
}
```
```
for(int j = 0; j < LARGE; j++) {
    product *= A[j];
}
```
```
for(int j = 0; j < LARGE; j++) {
    sum += A[j];
    product *= A[j];
}
```
Loop Fission separates loops that disrupt each other’s temporal locality

```c
for(int j = 0; j < LARGE; j++) {
    sum += A[j];  // Large misses
    for(int k = 0; k < LARGE; k++) {
        other_sum += B[j][k]; // Large/4 misses
    }
}
```

```c
for(int j = 0; j < LARGE; j++) {
    sum += A[j]; // Large/4 misses
}
for(j = 0; j < LARGE; j++)
    for(int k = 0; k < LARGE; k++) {
        other_sum += B[j][k] // Large/4 misses
    }
```
Accessing two arrays in the same inner loop

- Assumptions
  - 32B cache blocks (fit four, 64-bit (8B) doubles)
  - n & m are large (arrays much bigger than cache)

- Average misses per inner loop iteration?
  - A) 0  B) 1/8  C) 1/4  D) 1/2  E) $\geq 1$

  \[
  \frac{1}{4} + \frac{1}{1024}
  \]
Tiling creates a “sliding window” of data, increasing temporal locality.

Average misses per inner loop iteration?

A) 0  B) 1/8  C) 1/4  D) 1/2  E) 1
Example: Multiply two NxN matrices

- Assume:
  - 32B cache blocks (fit four, 64-bit (8B) doubles)
  - Matrix dimension \( N \) is very large (Approximate \( 1/N \) as 0.0)
  - Cache is not even big enough to hold multiple rows

```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
Matrices are allocated in row-major order in memory (rows are contiguous in memory)

for (i = 0; i < N; i++) //row traversal
    sum += a[0][i];
    ▪ accesses successive elements
    ▪ if block size (B) > 4 bytes, exploit spatial locality
      ▪ compulsory miss rate = 8 bytes / B

for (i = 0; i < n; i++) //col traversal
    sum += a[i][0];
    ▪ accesses distant elements
    ▪ no spatial locality!
      ▪ compulsory miss rate = 1 (i.e. 100%)

<table>
<thead>
<tr>
<th></th>
<th>a[0][0]</th>
<th>a[0][1]</th>
<th>a[0][2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[1][0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[1][1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[1][2]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[2][0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[2][1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[2][2]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0xC1…00  a[0][0]  
0xC1…08  a[0][1]  
0xC1…10  a[0][2]  
0xC1…18  a[1][0]  
0xC1…20  a[1][1]  
•••        •••      •••
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;
    }
}
```

**Inner loop:**

- **Row-wise Misses:**
  - A
  - B
  - C

- **Col-wise Misses:**
  - A
  - B

- **Fixed:**
  - A

Misses per inner loop iteration:

\[
\begin{align*}
A & = 0.25 \\
B & = 1.0 \\
C & = 0.0
\end{align*}
\]

\[
\frac{1.25 \text{ misses}}{n^3}
\]

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:**

- a) 0
- b) 0.25
- c) 0.5
- d) 1
- e) 1.25
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:

**Misses per inner loop iteration:**

\[
\begin{array}{ccc}
A & B & C \\
\_ & \_ & .25 \\
\_ & .5 & \_ \\
\_ & \_ & .75 \\
\_ & \_ & 1.25 \\
\_ & \_ & 2 \\
\end{array}
\]

**Total misses per inner loop iteration**

a) .25  b) .5  c) .75  d) 1.25  e) 2
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];
  }
}

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>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Total misses per inner loop iteration:

a) .25  b) .5  c) .75  d) 1.25  e) 2
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;
    }
}
```

**Inner loop:**

<table>
<thead>
<tr>
<th>Misses per <strong>inner loop iteration:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>1.0</td>
</tr>
</tbody>
</table>
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;
    }
}
Core i7 Matrix Multiply Performance

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

- jki / kji
- ijk / jik
- kij / ikj
From where comes the performance?

a) Spatial locality
b) Temporal locality
We can using **tiling** on matrices, just like we did with one-dimensional arrays

```c
// Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n+j] += a[i*n + k]*b[k*n + j];
}
```
Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache block = 4 doubles
  - Cache size C << n (much smaller than n)

- **First iteration:**
  - $\frac{n}{4} + n = \frac{5n}{4}$ misses

- **Afterwards in cache:**
  - (schematic)
Cache Miss Analysis

Assume:
- Matrix elements are doubles
- Cache block = 4 doubles
- Cache size C \ll n (much smaller than n)

Second iteration:
- Again:
  \( n/4 + n = 5n/4 \) misses

Total misses:
- \( 5n/4 \times n^2 = (5/4) \times n^3 \)
Create tiles in two dimensions (row & col)

c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                for (i1 = i; i1 < i+B; i1++)
                    for (j1 = j; j1 < j+B; j1++)
                        for (k1 = k; k1 < k+B; k1++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
Cache Miss Analysis

- **Assume:**
  - Cache block = 4 doubles
  - Cache size $C \ll n$ (much smaller than $n$)
  - Three tiles fit into cache: $3B^2 < C$

- **First (tile) iteration:**
  - $B^2/4$ misses for each tile
  - $2n/B * B^2/4 = nB/2$ (omitting matrix $c$)
  - Afterwards in cache (schematic)
Cache Miss Analysis

- Assume:
  - Cache block = 4 doubles
  - Cache size $C \ll n$ (much smaller than $n$)
  - Three tiles fit into cache: $3B^2 < C$

- Second (tile) iteration:
  - Same as first iteration
  - $2n/B \times B^2/4 = nB/2$

- Total misses:
  - $nB/2 \times (n/B)^2 = n^3/(2B)$
Summary

- No tiling: \((5/4) \times n^3\)
- Tiling: \((1/(2B)) \times n^3\)

- Suggest largest possible tile size \(B\), but limit \(3B^2 < C\)!

- Reason for dramatic difference:
  - Matrix multiplication has inherent temporal locality:
    - Input data: \(3n^2\), computation \(2n^3\)
    - Every array elements used \(O(n)\) times!
  - But program has to be written properly
Concluding Observations

- Programmer can optimize for cache performance
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
    - Tiling 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)