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 blocking 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.
/* 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

Read throughput (MB/s)

Working set size (bytes)

Stride (x8 bytes)

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

Read throughput (MB/s)

Stride (x8 bytes)

Working set size (bytes)

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

Slopes of spatial locality

Ridges of Temporal locality
Today

- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality
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

---

\[
\begin{align*}
\text{A} & \quad \text{B} & \quad \text{C} \\
\begin{array}{c}
\text{} \quad \text{} \quad \text{} \quad \text{} \\
\downarrow \quad \downarrow \quad \downarrow \\
i \quad k \quad j \quad i
\end{array}
\end{align*}
\]
Matrix Multiplication Example

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

```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;
    }
}
```
Layout of C Arrays in Memory

- **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++)
    - sum += a[0][i];
  - accesses successive elements
  - if block size (B) > 4 bytes, exploit spatial locality
    - compulsory miss rate = 4 bytes / B

- **Stepping through rows in one column:**
  - for (i = 0; i < n; i++)
    - 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>

Inner loop:

- **Row-wise**
- **Column-wise**
- **Fixed**
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>a)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b)</td>
<td>.25</td>
<td>.25</td>
<td>.25</td>
</tr>
<tr>
<td>c)</td>
<td>.75</td>
<td>.75</td>
<td>.75</td>
</tr>
<tr>
<td>d)</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>e)</td>
<td>2.0</td>
<td>2.0</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Inner loop:

- **Row-wise**
- **Column-wise**
- **Fixed**
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:

A   B   C

Inner loop:

(i,k) (k,*) (i,*)

Fixed Row-wise Row-wise

A   B   C

Misses:

a) 0  
b) .25  
c) .75  
d) 1.0  
e) 2.0
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>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>
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:

Misses per inner loop iteration:

<table>
<thead>
<tr>
<th></th>
<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>
<td>0.25</td>
</tr>
</tbody>
</table>
Matrix Multiplication (jki)

/* 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:
A   B   C

Inner loop:

(A,*) (k,j) (*,j)

Column-wise

Fixed

Column-wise

a) 0  b) .25  c) .75  d) 1.0  e) 2.0
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;
    }
}
```

Inner loop:

- Column-wise
- Fixed
- Column-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>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
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></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;
    }
}
```

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

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

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

- Spatial locality
- Temporal locality

```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;
    }
}
```

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

```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];
    }
}
```

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

```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;
    }
}
```

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

- **Performance impact of caches**
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality
Example: Matrix Multiplication

```c

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

![Matrix Multiplication Diagram]
Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size $C << n$ (much smaller than $n$)

- **First iteration:**
  - $n/8 + n = 9n/8$ misses
  - Afterwards *in cache:* (schematic)

\[ n \quad = \quad * \quad \text{8 wide} \]
Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)

- **Second iteration:**
  - Again:
    - \( \frac{n}{8} + n = \frac{9n}{8} \) misses

- **Total misses:**
  - \( \frac{9n}{8} \times n^2 = \frac{9}{8} \times n^3 \)
Blocked Matrix Multiplication

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)
                /* B x B mini matrix multiplications */
                    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];
}

Block size B x B
Cache Miss Analysis

- Assume:
  - Cache block = 8 doubles
  - Cache size \( C \ll n \) (much smaller than \( n \))
  - Three blocks fit into cache: \( 3B^2 < C \)

- First (block) iteration:
  - \( B^2/8 \) misses for each block
  - \( 2n/B \times B^2/8 = nB/4 \) (omitting matrix \( c \))
  - Afterwards in cache (schematic)
Cache Miss Analysis

- Assume:
  - Cache block = 8 doubles
  - Cache size $C \ll n$ (much smaller than $n$)
  - Three blocks fit into cache: $3B^2 < C$

- Second (block) iteration:
  - Same as first iteration
  - $2n/B * B^2 / 8 = nB/4$

- Total misses:
  - $nB/4 * (n/B)^2 = n^3/(4B)$
Summary

- No blocking: \((9/8) \times n^3\)
- Blocking: \((1/(4B)) \times n^3\)

- Suggest largest possible block 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
    - 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)