Outline

- Cache memory organization and operation
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
- Cache-Friendly Code - Using blocking to improve temporal locality

Review: An Example Memory Hierarchy

Cache Memories

- Cache memories are small, fast SRAM-based memories managed automatically in hardware.
  - Hold frequently accessed blocks of main memory
  - CPU looks first for data in caches (e.g., L1 and L2), then in main memory.
- Typical system structure:

Caching: A Simple Data Layout (Placement)

Direct-Mapped Cache Simulation
**Direct Mapped Cache**

Assume cache block size is 8 bytes: byte 000 through byte 111

<table>
<thead>
<tr>
<th>Valid</th>
<th>Tag</th>
<th>Data Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14g</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>1</td>
<td>14g</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>1</td>
<td>14g</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>1</td>
<td>14g</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>1</td>
<td>14g</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>1</td>
<td>14g</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>1</td>
<td>14g</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
</tbody>
</table>

Address of int:

$1 \text{ bits}$ $0, 01, 100$

5 = $2^2$ blocks

find block (5 bits)

**Direct Mapped Cache**

Assume: cache block size 8 bytes

<table>
<thead>
<tr>
<th>Valid</th>
<th>Tag</th>
<th>Data Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14g</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>1</td>
<td>14g</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>1</td>
<td>14g</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>1</td>
<td>14g</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>1</td>
<td>14g</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>1</td>
<td>14g</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>1</td>
<td>14g</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
</tbody>
</table>

Address of int:

$1 \text{ bits}$ $0, 01, 100$

block offset

**E-way Set Associative Cache**

Arrange cache lines into sets

$E = 2$: Two cache lines per set

Assume: cache block size 8 bytes

<table>
<thead>
<tr>
<th>Valid</th>
<th>Tag</th>
<th>Data Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14g</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>1</td>
<td>14g</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>1</td>
<td>14g</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>1</td>
<td>14g</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>1</td>
<td>14g</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>1</td>
<td>14g</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>1</td>
<td>14g</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
</tbody>
</table>

Address of short int:

$1 \text{ bits}$ $0, 01, 100$

find set

**A Higher Level Example**

Ignore the variables $\text{sum}$, $i$, $j$

```
int sum_array_rows(double a[16][16]) {
  int i, j;
  double sum = 0;
  for (i = 0; i < 16; i++)
    for (j = 0; j < 16; j++)
      sum += a[i][j];
  return sum;
}
```

32 B = 4 doubles

**E-way Set Associative Cache**

$E = 2$: Two cache lines per set

Assume: cache block size 8 bytes

<table>
<thead>
<tr>
<th>Valid</th>
<th>Tag</th>
<th>Data Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14g</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>1</td>
<td>14g</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>1</td>
<td>14g</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>1</td>
<td>14g</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>1</td>
<td>14g</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>1</td>
<td>14g</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>1</td>
<td>14g</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
</tbody>
</table>

Address of short int:

$1 \text{ bits}$ $0, 01, 100$

block offset

compare both simultaneously

match: yes = hit

valid? + match: assume yes = hit

block offset
General Cache Organization (S, E, B)

Assume: cache block size 8 bytes

E = 2^2 lines per set

S = 2^2 sets

B = 2 bytes per cache block (the data)

Cache size: C = S x E x B data bytes

Address Calculation Table

- \(2^2 = 4\)
- \(2^3 = 8\)
- \(2^4 = 16\)
- \(2^5 = 32\)
- \(2^6 = 64\)
- \(2^7 = 128\)
- \(2^8 = 256\)
- \(2^9 = 512\)
- \(2^{10} = 1024\)
- \(2^{11} = 2^{10} \times 2^1 = 2^{11}\)
- \(2^{12} = 2^{10} \times 2^2 = 2^{12}\)
- \(2^{13} = 2^{10} \times 2^3 = 2^{13}\)
- \(2^{14} = 2^{10} \times 2^4 = 2^{14}\)
- \(2^{15} = 2^{10} \times 2^5 = 2^{15}\)
- \(2^{16} = 2^{10} \times 2^6 = 2^{16}\)
- \(2^{17} = 2^{10} \times 2^7 = 2^{17}\)
- \(2^{18} = 2^{10} \times 2^8 = 2^{18}\)
- \(2^{19} = 2^{10} \times 2^9 = 2^{19}\)

2-Way Set Associative Cache Simulation

M = 16-byte addresses;
B = 2 bytes/block;
S = 2 sets; E = 2 blocks/set

Address trace (reads, one byte per read):

- 0 \([0000]\)_i, miss
- 1 \([0001]\)_i, hit
- 7 \([0111]\)_i, miss
- 8 \([1000]\)_i, hit

set 0

1 00 M(0-1)
1 10 M(8-9)

set 1

1 01 M(6-7)
0

1) Locate set
2) Check if any line in set has matching tag
3) Yes = line valid ⇒ hit
4) Locate data starting at offset

Cached Address Calculation

- valid?
  - +
- tag
  - \(v\)
- block

Data begins at this offset

Cache Read

Address of word:

- tag
- set index
- block offset

Locate set
Locate data starting at offset
Check if any line in set has matching tag
Yes = line valid ⇒ hit

int sum_array_rows(double a[16][16])
{
    int i, j, sum = 0;
    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}

Ignore the variables sum, i, j
Assume: cold (empty) cache
An Example: 2-Way Set Associative

```c
int sum_array_rows(double a[16][16]) {
    int i, j;
    double sum = 0;
    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```c
int sum_array_cols(double a[16][16]) {
    int i, j;
    double sum = 0;
    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

What about writes?

- **Multiple copies of data exist:**
  - L1, L2, Main Memory, Disk

- **What to do on a write-hit?**
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until replacement of line)

- **What to do on a write-miss?**
  - Write-allocate (load into cache, update line in cache)

Typical

- Write-through, + No-write-allocate

Cache Misses: 3 C’s

- **Cold (Compulsory) miss**
  - Cold misses occur because the cache is empty.

- **Conflict miss**
  - Most caches limit blocks at level k to a small subset of the block positions at level k+1
  - E.g. Block i at level k must be placed in block (i mod 4) at level k+1
  - **Conflict misses** occur when level k cache is large enough, but multiple data objects all map to the same level k block.
  - E.g. Referencing blocks 0, 8, 16, 24, 32, ... would miss every time = thrashing

- **Capacity miss**
  - Occurs when the set of active cache blocks (working set) is larger than the cache.

Cache Performance Metrics

- **Miss Rate**
  - Fraction of memory references not found in cache, i.e. \( \frac{N_{miss}}{N_{access}} \approx 1 - \text{hit rate} \)

- **Typical numbers (in percentages):**
  - 3-10% for L1

- **Hit Time**
  - Time to deliver a line in the cache
  - Includes time to determine whether the line is in the cache

- **Typical numbers:**
  - 1-2 clock cycle for L1

- **Miss Penalty**
  - Additional time required because of a miss
    - Typically 50-200 cycles for main memory (Trend: increasing)

Intel Core i7 Cache Hierarchy

- **L1-cache and D-cache:** 32 KB, 8-way, Hit Time: 4 cycles
- **L2 unified cache:** 256 KB, 8-way, Hit Time: 11 cycles
- **L3 unified cache:** 8 MB, 16-way, Hit Time: 30-40 cycles

Block size: 64 bytes for all caches.

lets think about those numbers

- **Huge difference between a hit and a miss:**
  - Could be 100x, if just L1 and main memory

- **Would you believe 99% hits is twice as good as 97%?**

  - Consider:
    - cache hit time of 1 cycle
    - miss penalty of 100 cycles

  - **Average access time:**
    - 97% hits: 1 cycle + 0.03 * 100 cycles = 4 cycles
    - 99% hits: 1 cycle + 0.01 * 100 cycles = 2 cycles

  - This is why “miss rate” is used instead of “hit rate”, and why it is so important to keep miss rate low
Writing Cache Friendly Code

- Make the common case go fast
  - Focus on 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.

Outline

- Cache organization and operation
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Cache-Friendly Code - 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.

Memory Mountain Test Function

def double data[MAXELEMS]; /* The global array to be traversed */

/* The test function */
void test(int elems, int stride) {
    int i;
    double result = 0.0;
    volatile double sink; /* declare sink as volatile */
    for (i = 0; i < elems; i += stride)
        result += data[i]; /* result will be kept in register */
    sink = result; /* So compiler doesn't optimize away the loop */
}

/* Run test(elems, stride) and return read throughput (MB/s) */
def double run(int size, int stride, double Mhz) {
    double cycles;
    int elems = size / sizeof(double);
    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
64-byte line size
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
64-byte line size
48 caches on-chip
Outline

• Cache organization and operation
• Performance impact of caches
  • The memory mountain
• Rearranging loops to improve spatial locality
• Cache-Friendly Code - Using blocking to improve temporal locality

Matrix Multiplication Example

• Description:
  • Multiply \( n \times n \) matrices
  • \( O(n^3) \) total operations
  • \( n \) reads per source element
  • \( n \) values summed per destination
  • But may be able to hold in register

```
/* 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 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 (i.e. stride) of inner loop

```
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;
  }
}
```

Matrix Multiplication Example

```
/* 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 (review)

• C arrays allocated in row-major order
  • Each row in contiguous memory locations
• Stepping through columns in one row:
  • for \( (i = 0; i < n; i++) \)
    • \( \text{sum} += a[i][j] \)
  • 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 \( (j = 0; j < n; j++) \)
    • \( \text{sum} += a[i][j] \)
  • Accesses distant elements
  • No spatial locality!
  • Compulsory miss rate = 1 (i.e. 100%)

```
```
Matrix Multiplication (\(kij\))

```c
/* kij */
for (i=0; icn; i++) { 
    for (k=0; kn; k++) { 
        x = a[i][k];
        c[i][j] += x * 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>0.0</td>
<td>0.25</td>
<td>0.25</td>
<td></td>
</tr>
</tbody>
</table>

Summary of Matrix Multiplication

- \(ijk\) & \(ikj\):
  - 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

- misses/iter = 2.0
- misses/iter = 1.25
- misses/iter = 0.5
Outline

- Cache organization and operation
- Performance impact of caches
  - The memory mountain
- Rearranging loops to improve spatial locality
- Cache-Friendly Code - Using blocking to improve temporal locality

Cache Miss Analysis

- Assume:
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size $C = n^2$ (much smaller than $n$)

- First iteration:
  - $n/8 + n = 9n/8$ misses

- Afterwards in cache:
  - (schematic)

Example: Matrix Multiplication

```c
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 = 8 doubles
  - Cache size $C = n^2$ (much smaller than $n$)

- Second iteration:
  - Again: $n/8 + n = 9n/8$ misses

- Total misses:
  - $9n/8 \times n = (9/8) \times n^3$

Blocked Matrix Multiplication

```c
n = (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++)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i; i1++)
                    for (j1 = j; j1 < j; j1++)
                        for (k1 = k; k1 < n; k1++)
                            c[(i1*n+j1)] += a[i1*n + k1]*b[k1*n + j1];
}
```

Cache Miss Analysis

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

- First (block) iteration:
  - $B/8$ misses for each block
  - $(2n/B) \times B/8 = nb/4$
  - (omitting matrix c)

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

• Assume:
  • Cache block = 8 doubles
  • Cache size C << n (C 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
  (omitting matrix c)

• Total misses:
  • nB/4 * (n/B)^2 = n^3/(4B)

Summary

• Total Misses
  • No blocking: (9/8) * n^3
  • Blocking: 1/(4B) * n^3
  (if B = 8 \rightarrow 1/4B = 1/32)
  • (9/8)/(1/32) = \approx 36X improvement

• Suggest largest possible block size B, but limit 3B^2 < C!
  • But matrix b and matrix c might have conflicts

• Reason for dramatic difference:
  • Matrix multiplication has inherent temporal locality:
    • Input data: 3n^3, 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 – Use it in Cache Lab!

• 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)
    • Re-order memory accesses (e.g. interchange loops) to keep stride small (spatial locality)