Today

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

Example Memory Hierarchy

- CPU registers hold words retrieved from the L1 cache.
- L1 cache holds cache lines retrieved from the L2 cache.
- L2 cache holds cache lines retrieved from main memory.
- Main memory holds disk blocks retrieved from local disks.
- Local disks hold files retrieved from disks on remote servers.
- Main memory holds disk blocks retrieved from local disks.
- Remote secondary storage (e.g., Web servers)
- Local secondary storage (local disks)
- L2 cache (SRAM)
- L1 cache (SRAM)
- CPU registers
- L0: Preg

General Cache Concept

- Smaller, faster, more expensive memory caches a subset of the blocks.

General Cache Organization (S, E, B)

- Cache size: \( C = S \times E \times B \) data bytes
- E = \( 2^b \) lines per set
- S = \( 2^b \) sets
- B = \( 2^b \) bytes per cache block (the data)

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 cache
- Typical system structure:
Cache Read

\[ E = 2 \text{ lines per set} \]
\[ S = 2 \text{ sets} \]
\[ B = 2^t \text{ bytes per cache block (the data)} \]

Address of word:
- 1 bits = 0 bits set tag block offset
- data begins at this offset

Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes

<table>
<thead>
<tr>
<th>Set 0</th>
<th>Set 1</th>
<th>Set 2</th>
<th>Set 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ E = 1 ]</td>
<td>[ E = 1 ]</td>
<td>[ E = 1 ]</td>
<td>[ E = 1 ]</td>
</tr>
</tbody>
</table>

Valid? + match: assume yes = hit

If tag doesn't match: old line is evicted and replaced

Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>set</td>
<td>block offset</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Address of int:
- 1 bits = 0 bits set tag block offset
- int (4 bytes) is here

Direct-Mapped Cache Simulation

E = 16 bytes (4-bit addresses), B = 2 bytes/block, S = 4 sets, E = 1 Blocks/set

Address trace (reads, one byte per read):
0 \[ \{0000\} \], miss
1 \[ \{0001\} \], hit
7 \[ \{0111\} \], miss
0 \[ \{0000\} \], miss

<table>
<thead>
<tr>
<th>v Tag Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0 [ 1 ] 0 M[0-1]</td>
</tr>
<tr>
<td>Set 1 [ 0 ] 1 M[5-7]</td>
</tr>
<tr>
<td>Set 1 [ 1 ] 0 M[0-1]</td>
</tr>
<tr>
<td>Set 2 [ 0 ] 1 M[5-7]</td>
</tr>
</tbody>
</table>

Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes

<table>
<thead>
<tr>
<th>Set 0</th>
<th>Set 1</th>
<th>Set 2</th>
<th>Set 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ E = 1 ]</td>
<td>[ E = 1 ]</td>
<td>[ E = 1 ]</td>
<td>[ E = 1 ]</td>
</tr>
</tbody>
</table>

Valid? + match: assume yes = hit

If tag doesn't match: old line is evicted and replaced

Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>set</td>
<td>block offset</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Address of int:
- 1 bits = 0 bits set tag block offset
- int (4 bytes) is here

E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>set</td>
<td>block offset</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Address of short int:
- 1 bits = 0 bits set tag block offset
- find set
E-way Set Associative Cache (Here: E = 2)
E = 2: Two lines per set
Assume: cache block size 8 bytes

Address of short int:

<table>
<thead>
<tr>
<th>Valid?</th>
<th>match</th>
<th>tag</th>
<th>v</th>
<th>Block Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

No match:
- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), ...

What about writes?
- Multiple copies of data exist:
  - L1, L2, L3, 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)
    + Need a dirty bit (line different from memory or not)
- What to do on a write-miss?
  - Write-allocate (load into cache, update line in cache)
    + Good if more writes to the location follow
  - No-write-allocate (writes straight to memory, does not load into cache)
- Typical
  - Write-through + No-write-allocate
  - Write-back + Write-allocate

L1 i-cache and d-cache:
- 32 KB, 8-way, Access: 4 cycles
L2 unified cache:
- 256 KB, 8-way, Access: 10 cycles
L3 unified cache:
- 8 MB, 16-way, Access: 40-75 cycles

Block size: 64 bytes for all caches.

Cache Performance Metrics
- Miss Rate
  - Fraction of memory references not found in cache (misses / accesses)
  - 1 – hit rate
  - Typical numbers (in percentages):
    - 3-10% for L1
    - can be quite small (e.g., < 1%) for L2, depending on size, etc.
- Hit Time
  - Time to deliver a line in the cache to the processor
  + includes time to determine whether the line is in the cache
  - Typical numbers:
    - 4 clock cycle for L1
    - 10 clock cycles for L2
- Miss Penalty
  - Additional time required because of a miss
    + typically 50-200 cycles for main memory (Trend: increasing)
Let’s 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”

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

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

Rows/Columns Example

```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 (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

Ignore the variables sum, i, j

Assume: cold (empty) cache, a[0][0] goes here, 2-way set associative

32 B = 4 doubles

Rows/Columns Example

```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 (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

Ignore the variables sum, i, j

Assume: cold (empty) cache, a[0][0] goes here, 2-way set associative

32 B = 4 doubles
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
long data[MAXELEMS]; /* Global array to traverse */
/* test - Iterate over first "elems" elements of *array "data" with stride of "stride", using *        using 4x4 loop unrolling. */
long test(int elems, int stride) {
    long i, sx2 = stride * 2, sx3 = stride * 3, sx4 = stride * 4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;
    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i + stride];
        acc2 = acc2 + data[i + sx2];
        acc3 = acc3 + data[i + sx3];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}
```

Call `test()` with many combinations of `elems` and `stride`.

1. Call `test()` once to warm up the caches.
2. Call `test()` again and measure the read throughput (MB/s).

---

**Matrix Multiplication Example**

- **Description**:
  - Multiply N x N matrices
  - Matrix elements are doubles (8 bytes)
  - O(N^3) total operations
  - N reads per source element
  - N values summed per destination
  - but may be able to hold in register

```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;
    }
}
```

---

**Today**

- **Cache organization and operation**
- **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**:
  - Block size = 32B (big enough for four doubles)
  - 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

```c
<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>k</td>
</tr>
</tbody>
</table>
```

```
A = i
C
B
X
```

---

**The Memory Mountain**

- **Core i7 Haswell**
  - 2.1 GHz
  - 32 KB L1 d-cache
  - 256 KB L2 cache
  - 8 MB L3 cache
  - 64 B block size

---

**Matrix Multiplication Example**

- **Description**:
  - Multiply N x N matrices
  - Matrix elements are doubles (8 bytes)
  - O(N^3) total operations
  - N reads per source element
  - N values summed per destination
  - but may be able to hold in register

```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;
    }
}
```
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++)
    - sum += a[0][i];
  - accesses successive elements
  - if block size (B) > sizeof(a) bytes, exploit spatial locality
    - miss rate = sizeof(a) / B
- Stepping through rows in one column:
  - for (i = 0; i < n; i++)
    - sum += a[i][0];
  - accesses distant elements
    - 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>A</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>A</td>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>

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];
    }
}
```

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>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Matrix Multiplication (ikj)

```c
/* 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>A</td>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>
Matrix Multiplication (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;
    }
}
```

Matrix Multiplication (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;
    }
}
```

Summary of Matrix Multiplication

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

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

ji (jki):
- 2 loads, 1 store
- misses/iter = 2.0

Example: Matrix Multiplication

```c
double **a = malloc(sizeof(double *) * n);
for (i = 0; i < n; i++)
    a[i] = malloc(sizeof(double) * n);

void matrix_multiply(double *a, double *b, double *c, int n) {
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            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 (much smaller than n)
- First iteration:
  - n/8 + n = 9n/8 misses
- Afterwards in cache:
  - (schematic)
Cache Miss Analysis

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

- **Second iteration:**
  - Again:
    - \( n/8 + n = 9n/8 \) misses

- **Total misses:**
  - \( 9n/8 * n^2 = (9/8) * n^3 \)

Blocked Matrix Multiplication

```c
// (double *) calloc(sizeof(double), n*n);
/* Multiply n x n matrices a and b */
void mm(double *a, double *b, double *c, int n) {
    for (i = 0; i < n; i+=B) {
        for (j = 0; j < n; j+=B) {
            for (k = 0; k < n; k+=B) {
                c[i*n+j] += a[i*n+k] * b[k*n + j];
            }
        }
    }
}
```

Cache Summary

- **Cache memories can have significant performance impact**
  - You can write your programs to exploit this!
    - Focus on the inner loops, where bulk of computations and memory accesses occur.
    - Try to maximize spatial locality by reading data objects with sequentially with stride 1.
    - Try to maximize temporal locality by using a data object as often as possible once it’s read from memory.