Cache Memories

15-213/18-213/15-513: Introduction to Computer Systems
12th Lecture, February 26, 2019
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
Recall: Locality

- **Principle of Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently

- **Temporal locality:**
  - Recently referenced items are likely to be referenced again in the near future

- **Spatial locality:**
  - Items with nearby addresses tend to be referenced close together in time
Recall: Memory Hierarchy

Smaller, faster, and costlier (per byte) storage devices

L0: Regs
CPU registers hold words retrieved from the L1 cache.

L1:
L1 cache (SRAM)
L1 cache holds cache lines retrieved from the L2 cache.

L2:
L2 cache (SRAM)
L2 cache holds cache lines retrieved from L3 cache.

L3:
L3 cache (SRAM)
L3 cache holds cache lines retrieved from main memory.

L4:
Main memory (DRAM)
Main memory holds disk blocks retrieved from local disks.

L5:
Local secondary storage (local disks)
Local disks hold files retrieved from disks on remote servers.

L6:
Remote secondary storage (e.g., Web servers)

Larger, slower, and cheaper (per byte) storage devices
Recall: General Cache Concepts

Cache

Data is copied in block-sized transfer units

Memory

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

Larger, slower, cheaper memory viewed as partitioned into “blocks”
General Cache Concepts: Hit

Cache

Data in block b is needed

Block b is in cache:
Hit!

Memory
General Cache Concepts: Miss

Data in block b is needed

Block b is not in cache: Miss!

Block b is fetched from memory

Block b is stored in cache

- Placement policy: determines where b goes
- Replacement policy: determines which block gets evicted (victim)
Recall: General Caching Concepts: 3 Types of Cache Misses

- **Cold (compulsory) miss**
  - Cold misses occur because the cache starts empty and this is the first reference to the block.

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

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

![Diagram of computer system structure](image-url)
Recall: Modern CPU Design

**Instruction Control**

- **Fetch Control**
- **Instruction Decode**
- **Instruction Cache**

- **Retirement Unit**
  - **Register File**

- **Address**
- **Instructions**
- **Operations**

- **Prediction OK?**
  - **Register Updates**

**Execution**

- **Branch**
- **Arith**
  - **Arith**
  - **Arith**
- **Load**
- **Store**

- **Data Cache**
- **Operation Results**
- **Addr.**
- **Data**
What it Really Looks Like

Nehalem

AMD FX 8150

Core i7-3960X
What it Really Looks Like (Cont.)

Intel Sandy Bridge Processor Die

L1: 32KB Instruction + 32KB Data
L2: 256KB
L3: 3–20MB
General Cache Organization (S, E, B)

E = 2^e lines per set

S = 2^s sets

Cache size:
= S x E x B data bytes

valid bit

B = 2^b bytes per cache block (the data)
Cache Read

- Locate set
- Check if any line in set has matching tag
- Yes + line valid: hit
- Locate data starting at offset

Address of word:

- `t` bits
- `s` bits
- `b` bits

- Tag
- Set
- Index
- Offset

- Data begins at this offset

- Valid bit
- `B = 2^b` bytes per cache block (the data)
Example: Direct Mapped Cache (E = 1)

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

\[ S = 2^s \text{ sets} \]

Address of int:
\[ \text{t bits} \quad 0...01 \quad 100 \]

find set
Example: Direct Mapped Cache (E = 1)

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

Address of int:

<table>
<thead>
<tr>
<th>t bits</th>
<th>0...01</th>
<th>100</th>
</tr>
</thead>
</table>

Block offset

valid? + match: assume yes (= hit)

v | tag | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Example: Direct Mapped Cache (E = 1)

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

If tag doesn’t match (= miss): old line is evicted and replaced
Direct-Mapped Cache Simulation

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

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>Address</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>M[0-1]</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>hit</td>
</tr>
<tr>
<td>7</td>
<td>011</td>
<td>miss</td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>miss</td>
</tr>
<tr>
<td>0</td>
<td>000</td>
<td>miss</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set 0</th>
<th>1</th>
<th>0</th>
<th>M[0-1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 2</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 3</td>
<td>1</td>
<td>0</td>
<td>M[6-7]</td>
</tr>
</tbody>
</table>
E-way Set Associative Cache (Here: E = 2)

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

Address of short int:

<table>
<thead>
<tr>
<th>t bits</th>
<th>0...01</th>
<th>100</th>
</tr>
</thead>
</table>

2 lines per set

S sets

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

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

Address of short int:

Block offset:

Compare both:

Valid? + match: yes (= hit)
E-way Set Associative Cache (Here: E = 2)

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

- Address of short int:
  - t bits: 0...01 100

Compare both valid? + match: yes (= hit)

- No match or not valid (= miss):
  - One line in set is selected for eviction and replacement
  - Replacement policies: random, least recently used (LRU), ...

Short int (2 Bytes) is here
2-Way Set Associative Cache Simulation

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

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>Address</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000_2</td>
<td>miss</td>
</tr>
<tr>
<td>1</td>
<td>0001_2</td>
<td>hit</td>
</tr>
<tr>
<td>7</td>
<td>0111_2</td>
<td>miss</td>
</tr>
<tr>
<td>8</td>
<td>1000_2</td>
<td>miss</td>
</tr>
<tr>
<td>0</td>
<td>0000_2</td>
<td>hit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set 0</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00</td>
<td>M[0-1]</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>M[8-9]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set 1</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01</td>
<td>M[6-7]</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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)
    - Each cache line needs a dirty bit (set if data differs from memory)

- **What to do on a write-miss?**
  - **Write-allocate** (load into cache, update line in cache)
    - Good if more writes to the location will follow
  - **No-write-allocate** (writes straight to memory, does not load into cache)

- **Typical**
  - Write-through + No-write-allocate
  - Write-back + Write-allocate
Why Index Using Middle Bits?

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

Standard Method: Middle bit indexing

Address of int:
\[ t \text{ bits} \quad 0...01 \quad 100 \]

find set

Alternative Method: High bit indexing

Address of int:
\[ 1...11 \quad t \text{ bits} \quad 100 \]

find set

S = 2^s sets
Illustration of Indexing Approaches

- 64-byte memory
  - 6-bit addresses
- 16 byte, direct-mapped cache
- Block size = 4. Thus 4 sets.
- 2 bits tag, 2 bits index, 2 bits offset

<table>
<thead>
<tr>
<th>Set 0</th>
<th>Set 1</th>
<th>Set 2</th>
<th>Set 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000xx</td>
<td>0001xx</td>
<td>0010xx</td>
<td>0011xx</td>
</tr>
<tr>
<td>0011xx</td>
<td>0100xx</td>
<td>0101xx</td>
<td>0110xx</td>
</tr>
<tr>
<td>0111xx</td>
<td>1000xx</td>
<td>1001xx</td>
<td>1010xx</td>
</tr>
<tr>
<td>1011xx</td>
<td>1100xx</td>
<td>1101xx</td>
<td>1110xx</td>
</tr>
<tr>
<td>1111xx</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Middle Bit Indexing

- Addresses of form **TTSSBB**
  - **TT** Tag bits
  - **SS** Set index bits
  - **BB** Offset bits

- Makes good use of spatial locality
  - Adjacent memory blocks map to different sets

```
Set 0
Set 1
Set 2
Set 3
```

```
0000xx
0001xx
0010xx
0011xx
0100xx
0101xx
0110xx
0111xx
1000xx
1001xx
1010xx
1011xx
1100xx
1101xx
1110xx
1111xx
```
High Bit Indexing

- Addresses of form \( SS\ TT\ BB \)
  - \( SS \) Set index bits
  - \( TT \) Tag bits
  - \( BB \) Offset bits

- Program with high spatial locality would generate lots of conflicts
  - Adjacent blocks map to same set
Intel Core i7 Cache Hierarchy

Processor package

Core 0

- Regs
- L1 i-cache
- L1 d-cache
- L2 unified cache

Core 3

- Regs
- L1 i-cache
- L1 d-cache
- L2 unified cache

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.

Main memory
Example: Core i7 L1 Data Cache

32 kB 8-way set associative
64 bytes/block
47 bit address range

B =
S = , s =
E = , e =
C =

Stack Address: 0x00007f7262a1e010
Block offset: 0x??
Set index: 0x??
Tag: 0x??
Example: Core i7 L1 Data Cache

32 kB 8-way set associative
64 bytes/block
47 bit address range

B = 64
S = 64, s = 6
E = 8, e = 3
C = 64 x 64 x 8 = 32,768

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

Address of word:

<table>
<thead>
<tr>
<th>t bits</th>
<th>s bits</th>
<th>b bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Stack Address:

| 0x00007f7262a1e010 |

Block offset:

| 0x10 |

Set index:

| 0x0 |

Tag:

| 0x7f7262a1e |

0000 0001 0000
Cache Performance Metrics

- **Miss Rate**
  - Fraction of memory references not found in cache (misses / accesses)
  - $= 1 – \text{hit rate}$
  - 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 this simplified example:
    - cache hit time of 1 cycle
    - miss penalty of 100 cycles

  - Average access time:
    - 97% hits: 1 cycle + 0.03 x 100 cycles = 4 cycles
    - 99% hits: 1 cycle + 0.01 x 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
Quiz Time!

Check out:

https://canvas.cmu.edu/courses/8555
Today

- Cache organization and operation
- 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.
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 4x4 loop unrolling. */
int 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`.

For each `elems` and `stride`:

1. Call `test()` once to warm up the caches.

2. Call `test()` again and measure the read throughput (MB/s)
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

Aggressive prefetching

Ridges of temporal locality

Slopes of spatial locality

Read throughput (MB/s)

Stride (x8 bytes)

Size (bytes)
Today

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

Description:
- Multiply $N \times N$ 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
/* 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;
    }
}
```

matmult/mm.c
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

\[
\begin{align*}
\text{C} & \quad = \quad \text{A} \times \text{B}
\end{align*}
\]
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_{ij})` bytes, exploit spatial locality
    - miss rate = `sizeof(a_{ij}) / B`

- **Stepping through rows in one column:**
  - `for (i = 0; i < n; i++)`
    - `sum += a[i][0];`
  - accesses distant elements
  - no spatial locality!
    - 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;
    }
}
```

Miss rate for inner loop iterations:

<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>

Block size = 32B (four doubles)
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**
- **B**
- **C**

Miss rate for inner loop iterations:

<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>

Block size = 32B (four doubles)
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;
    }
}
```

Miss rate for inner loop iterations:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miss rate</td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Block size = 32B (four doubles)
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;
    }
}
```

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

```
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
- avg misses/iter = 0.5

```
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
- avg misses/iter = 2.0
Core i7 Matrix Multiply Performance

Cycles per inner loop iteration

Array size (n)

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

ijk / jik (1.25)

jki / kji (2.0)

kij / ikj (0.5)
Today

- Cache organization and operation
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality
Example: 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++)
        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 (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:**
  - $9n/8 \cdot n^2 = (9/8) \cdot n^3$
Blocked Matrix Multiplication

```c
double *c = 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; i++)
                        for (j1 = j; j1 < j+B; j++)
                            for (k1 = k; k1 < k+B; k++)
                                c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```

matmult/bmm.c
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 << n (much smaller than n)
  - Three blocks fit into cache: $3B^2 < C$

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

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

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

- Use largest block size \(B\), such that \(B\) satisfies \(3B^2 < C\)

- 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
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 sequentially with stride 1.
  - Try to maximize temporal locality by using a data object as often as possible once it’s read from memory.
Supplemental slides
The Memory Mountain

Slopes of spatial locality

Ridges of temporal locality

Aggressive prefetching

Core i5 Haswell
3.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size

Stride (x8 bytes)
Size (bytes)
Read throughput (MB/s)

s1
s3
s5
s7
s9
s11

L1
L2
L3
Mem

32m
128m
512k
128k
2m
32k
2m
512k
128k
2m
32k

0
4000
8000
12000
16000
20000
24000
28000
32000

128m
32m
Cache Capacity Effects from Memory Mountain

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

Slice through memory mountain with stride=8
Cache Block Size Effects from Memory Mountain

Throughput for size = 128K

- Miss rate = s/8
- Miss rate = 1.0

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

MB/sec

s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12

Stride s
Modeling Block Size Effects from Memory Mountain

Throughput for size = 128K

Throughput = \frac{10^6}{8.0s + 24.3}

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

Throughput vs Stride s graph with measured and modeled data points.
2008 Memory Mountain

No prefetching

Core 2 Duo
2.4 GHz
32 KB L1 d-cache
6MB L2 cache
64 B block size
Matrix Multiplication ($j ik$)

```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>

Block size = 32B (four doubles)
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];
    }
}
```

```
matmult/mm.c
```

Inner loop:

- **A**: Fixed
- **B**: Row-wise
- **C**: 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>

Block size = 32B (four doubles)
Matrix Multiplication (kji)

/* 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>

Block size = 32B (four doubles)