Cache Memories

15-213: Introduction to Computer Systems
12th Lecture, October 6th, 2016

Instructor:
Randy Bryant
Today

- Cache memory organization and operation
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
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
Example Memory Hierarchy

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

Smaller, faster, and costlier (per byte) storage devices
General Cache Concepts

*Everything handled in hardware. Invisible to programmer*

Cache

![Cache diagram with numbers 4, 9, 10, 3 indicating locations and 10 as a transfer unit.]

Memory

![Memory diagram with numbers 0 to 15 indicating locations. Rows 4 and 8 are marked with a reddish color and the numbers 4 and 10 are highlighted in green.]

- Smaller, faster, more expensive memory caches a subset of the blocks.
- Data is copied in block-sized transfer units.
- Larger, slower, cheaper memory viewed as partitioned into “blocks.”

General Cache Concepts: Hit

Request: 14

Data in block b is needed

Block b is in cache:
Hit!
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)
General Caching Concepts:
Types of Cache Misses

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

- **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.

- **Capacity miss**
  - Occurs when the set of active cache blocks (working set) is larger than the cache.
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 a computer system with CPU chip, cache memory, register file, ALU, bus interface, I/O bridge, system bus, memory bus, and main memory.]
What it Really Looks Like

Desktop PC

CPU (Intel Core i7)

Motherboard

Main memory (DRAM)

Source: Dell

Source: PC Magazine

Source: techreport.com

Source: Dell

Source: Dell

Source: Dell
What it Really Looks Like (Cont.)

Intel Sandy Bridge Processor Die

L1: 32KB Instruction + 32KB Data
L2: 256KB
L3: 3–20MB
Recap from Lecture 10:

Modern CPU Design

Instruction Control

Execution

Instruction Cache

Fetch Control

Instruction Decode

Register File

Retirement Unit

Prediction OK?

Register Updates

Data Cache

Fetch Address

Instruction Cache

Operations

Instruction

Address

Branch

Arith

Arith

Arith

Load

Store

Arith Operation Results

Addr.

Data

Addr.

Data

Operation Results
General Cache Organization \((S, E, B)\)

- \(E = 2^e\) lines per set
- \(S = 2^s\) sets
- \(B = 2^b\) bytes per cache block (the data)

Cache size:
\[ C = S \times E \times B \text{ data bytes} \]
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
  - block offset
  - data begins at this offset

E = \(2^e\) lines per set

S = \(2^s\) sets

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 8 bytes

S = $2^5$ sets

Address of int:

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

Direct mapped: One line per set
Assume: cache block size 8 bytes
Example: Direct Mapped Cache (E = 1)

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

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

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

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>t=1</th>
<th>s=2</th>
<th>b=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>xx</td>
<td>x</td>
</tr>
</tbody>
</table>

0       [0000₂], miss
1       [0001₂], hit
7       [0111₂], miss
8       [1000₂], miss
0       [0000₂] miss

<table>
<thead>
<tr>
<th>t=1</th>
<th>s=2</th>
<th>b=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>xx</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Set 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 3</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
E-way Set Associative Cache (Here: E = 2)

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

Address of short int:

\[ \text{t bits} \quad 0...01 \quad 100 \]

find set

S sets

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

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

valid? + match: yes = hit

compare both

Address of short int:

block offset

E = 2:
Two lines per set
Assume:
cache block size 8 bytes
E-way Set Associative Cache (Here: E = 2)

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

Address of short int:

```
 0...01 100
```

compare both

valid? + match: yes = hit

short int (2 Bytes) is here

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

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):

<table>
<thead>
<tr>
<th>Address</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>M[0-1]</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>M[6-7]</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>M[8-9]</td>
</tr>
</tbody>
</table>

Trace:
0 [0000₂], miss
1 [0001₂], hit
7 [0111₂], miss
8 [1000₂], miss
0 [0000₂], hit
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
Why Index Using Middle Bits?

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

S = 2^s sets

Standard Method: Middle bit indexing

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>0 1 2 3 4 5 6 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>tag</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>v</td>
<td>tag</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>v</td>
<td>tag</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
</tbody>
</table>

Address of int:

| t bits | 0...01 | 100 |

find set

Alternative Method: High bit indexing

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>0 1 2 3 4 5 6 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>tag</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>v</td>
<td>tag</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
</tbody>
</table>

find set

Address of int:

| 1...11 | t bits | 100 |
Illustration of Indexing Approaches

- 64-byte memory
  - 6-bit addresses
- 16 byte, direct-mapped cache
- Block size = 4 (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>0100xx</td>
<td>0101xx</td>
<td>0110xx</td>
<td>0111xx</td>
</tr>
<tr>
<td>1000xx</td>
<td>1001xx</td>
<td>1010xx</td>
<td>1011xx</td>
</tr>
<tr>
<td>1100xx</td>
<td>1101xx</td>
<td>1110xx</td>
<td>1111xx</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

<table>
<thead>
<tr>
<th>Set 0</th>
<th>Set 1</th>
<th>Set 2</th>
<th>Set 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

High Bit Indexing

- Addresses of form $SSTTBB$
  - $SS$ Set index bits
  - $TT$ Tag bits
  - $BB$ Offset bits

- Program with high spatial locality would generate lots of conflicts

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition
Intel Core i7 Cache Hierarchy

Processor package

Core 0
- Regs
- L1 d-cache
- L1 i-cache
- L2 unified cache

Core 3
- Regs
- L1 d-cache
- L1 i-cache
- L2 unified cache

... (other cores)

L3 unified cache (shared by all cores)

Main memory

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.

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

Stack Address: 0x00007f7262a1e010
Set index: 0x0
Tag: 0x7f7262a1e

Block offset: 0x10

Hex  | Decimal | Binary
--- | --- | ---
0 | 0 | 0000
1 | 1 | 0001
2 | 2 | 0010
3 | 3 | 0011
4 | 4 | 0100
5 | 5 | 0101
6 | 6 | 0110
7 | 7 | 0111
8 | 8 | 1000
9 | 9 | 1001
A | 10 | 1010
B | 11 | 1011
C | 12 | 1100
D | 13 | 1101
E | 14 | 1110
F | 15 | 1111

Address of word:

<table>
<thead>
<tr>
<th>t bits</th>
<th>s bits</th>
<th>b bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>set</td>
<td>block</td>
</tr>
</tbody>
</table>

Block offset: 6 bits
Set index: 6 bits
Tag: 35 bits
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:
    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
Today

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

Slopes of spatial locality

Aggressive prefetching

Ridges of temporal locality

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

Read throughput (MB/s)

Size (bytes)

Stride (x8 bytes)
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

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

Miss rate = s/8

Miss rate = 1.0

Measured

Stride s
Modeling Block Size Effects from Memory Mountain

Throughput for size = 128K

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

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

Today

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

**Description:**
- Multiply $N \times N$ matrices
- Matrix elements are doubles (8 bytes)
- $2N^3$ total FP 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;
    }
}
```

Variable `sum` held in register

`matmult/mm.c`
Miss Rate Analysis for Matrix Multiply

- **Assume:**
  - Block size = 64B (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
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:
  - \texttt{for} (\(i = 0; \ i < N; \ i++\))
    \begin{itemize}
    \item \texttt{sum += a[0][i];}
    \item accesses successive elements
    \item if block size (\(B\)) > \texttt{sizeof(a\_ij)} bytes, exploit spatial locality
      \begin{itemize}
      \item miss rate = \(\frac{\text{sizeof(a\_ij)}}{B}\)
      \end{itemize}
    \end{itemize}

- Stepping through rows in one column:
  - \texttt{for} (\(i = 0; \ i < n; \ i++\))
    \begin{itemize}
    \item \texttt{sum += a[i][0];}
    \item accesses distant elements
    \item no spatial locality!
      \begin{itemize}
      \item miss rate = 1 (i.e. 100%)
      \end{itemize}
    \end{itemize}
Matrix Multiplication (ijk)

/* 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>0.125</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
    }
}
```

Inner loop:

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

<table>
<thead>
<tr>
<th>Misses per inner loop iteration:</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>0.125</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></td>
<td>0.0</td>
<td>0.125</td>
<td>0.125</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];
    }
}
```

Inner loop:

<table>
<thead>
<tr>
<th>(i,k)</th>
<th>(k,*</th>
<th>(i,*</th>
<th>Fixed</th>
<th>Row-wise</th>
<th>Row-wise</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>0.0</td>
<td>0.125</td>
<td>0.125</td>
</tr>
</tbody>
</table>

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>0.125</td>
<td>0.125</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;
    }
}

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>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
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>
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;
    }
}

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;
    }
}

**ijk (& jik):**
- 2 loads, 0 stores
- misses/iter = 1.125

**kij (& ikj):**
- 2 loads, 1 store
- misses/iter = 0.25

**jki (& kji):**
- 2 loads, 1 store
- misses/iter = 2.0
2008-era Matrix Multiply Performance

Nanoseconds per floating-point operation. Measured on 2.4GHz Core 2 Duo

- $jki / kji (2.0)$
- $ijk / jik (1.125)$
- $kij / ikj (0.25)$
2014-era Matrix Multiply Performance

Nanoseconds per floating-point operation. Measured on 3.1 Ghz Haswell

- $jki / kji (2.0)$
- $ijk / jik (1.125)$
- $kij / ikj (0.25)$
2008 Memory Mountain

No prefetching

Core 2 Duo
2.4 GHz
32 KB L1 d-cache
6MB L2 cache
64 B block size

Read throughput (MB/s)

Stride (x8 bytes)

Size (bytes)
2014 Memory Mountain

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

Read throughput (MB/s)

Stride (x8 bytes)

Size (bytes)
EXTRA SLIDES
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:
    \[ \frac{n}{8} + n = 9\frac{n}{8} \text{ misses} \]

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

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

```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+=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];
}
```

Bryant and O’Hallaron, Computer Systems: A Programmer’s Perspective

Bryant and O’Hallaron, Computer Systems: A Programmer’s Perspective
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
- $2n/B \times B^2/8 = nB/4$

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

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

- 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
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.