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
The CPU-Memory Gap

The gap widens between DRAM, disk, and CPU speeds.

The graph shows the time (in nanoseconds) for different components over a period from 1985 to 2015. The y-axis represents time in nanoseconds, ranging from 0.0 to 100,000,000.0. The x-axis represents years from 1985 to 2015.

- **Disk seek time**
- **SSD access time**
- **DRAM access time**
- **SRAM access time**
- **CPU cycle time**
- **Effective CPU cycle time**

The graph illustrates how the access times for each component decrease over time, with the disk seek time being the longest and the CPU cycle time being the shortest. The effective CPU cycle time shows a trend that is influenced by the other components, indicating a widening gap between the different types of memory and the CPU.
Locality to the Rescue!

The key to bridging this CPU-Memory gap is a fundamental property of computer programs known as 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.
Real Memory Reference Patterns

Typical Memory Reference Patterns

- Instruction fetches
- Stack accesses
- Data accesses

- Address vs. Time
  - n loop iterations
  - subroutine call
  - subroutine return
  - argument access
  - vector access
  - scalar accesses
Memory Reference Patterns

Locality Example

```c
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- **Data references**
  - Reference array elements in succession (stride-1 reference pattern).
  - Reference variable `sum` each iteration.

- **Instruction references**
  - Reference instructions in sequence.
  - Cycle through loop repeatedly.

**Temporal locality**
- Reference variable `sum` each iteration.
- Cycle through loop repeatedly.

**Spatial locality**
- Reference array elements in succession (stride-1 reference pattern).
- Reference instructions in sequence.
**Locality Example**

- **Question:** Does this function have good locality with respect to array $a$?

```c
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];

    return sum;
}
```
Locality Example

**Question:** Can you permute the loops so that the function scans the 3-d array $a$ with a stride-1 reference pattern (and thus has good spatial locality)?

```c
int sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                sum += a[k][i][j];

    return sum;
}
```
Qualitative Estimates of Locality

Claim: Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.

Question: Does this function have good locality with respect to array $a$?

```c
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];

    return sum;
}
```
Memory Hierarchies

- Some fundamental and enduring properties of hardware and software:
  - Fast storage technologies cost more per byte, have less capacity, and require more power (heat!).
  - The gap between CPU and main memory speed is widening.
  - Well-written programs tend to exhibit good locality.

- These fundamental properties complement each other beautifully.

- They suggest an approach for organizing memory and storage systems known as a memory hierarchy.
Example Memory Hierarchy

L6: Remote secondary storage (e.g., Web servers)
L5: Local secondary storage (local disks)
L4: Main memory (DRAM)
L3: L3 cache (SRAM)
L2: L2 cache (SRAM)
L1: L1 cache (SRAM)
L0: CPU registers

Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices

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 L3 cache.
L3 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.
Management of Memory Hierarchy

- **Small/fast storage, e.g., registers**
  - Address usually specified in instruction
  - Generally implemented directly as a register file
    - but hardware might do things behind software’s back, e.g., stack management, register renaming

- **Larger/slower storage, e.g., main memory**
  - Address usually computed from values in register
  - Generally implemented as a hardware-managed cache hierarchy
    - hardware decides what is kept in fast memory
    - but software may provide “hints”, e.g., don’t cache or prefetch
Caches

- **Cache**: A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.

- **Fundamental idea of a memory hierarchy**: For each $k$, the faster, smaller device at level $k$ serves as a cache for the larger, slower device at level $k+1$.

- **Why do memory hierarchies work?**
  - Because of locality, programs tend to access the data at level $k$ more often than they access the data at level $k+1$.
  - Thus, the storage at level $k+1$ can be slower, and thus larger and cheaper per bit.

- **Big Idea**: The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.
General Cache Concepts

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”.

Cache

Memory

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

Diagram:

- Cache contains blocks 4, 9, 10, and 3.
- Memory contains blocks 0 to 15, with blocks 4, 8, 9, and 12 highlighted.

Legend:

- Red blocks: Blocks in the cache.
- Green blocks: Blocks in memory that are not in the cache.
- Yellow blocks: Blocks in memory that are in the cache.

The diagram illustrates how data is accessed and copied between cache and memory.
General Cache Concepts: Hit

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.
## Examples of Caching in the Mem. Hierarchy

<table>
<thead>
<tr>
<th>Cache Type</th>
<th>What is Cached?</th>
<th>Where is it Cached?</th>
<th>Latency (cycles)</th>
<th>Managed By</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>4-8 bytes words</td>
<td>CPU core</td>
<td>0</td>
<td>Compiler</td>
</tr>
<tr>
<td>TLB</td>
<td>Address translations</td>
<td>On-Chip TLB</td>
<td>0</td>
<td>Hardware MMU</td>
</tr>
<tr>
<td>L1 cache</td>
<td>64-byte blocks</td>
<td>On-Chip L1</td>
<td>4</td>
<td>Hardware</td>
</tr>
<tr>
<td>L2 cache</td>
<td>64-byte blocks</td>
<td>On-Chip L2</td>
<td>10</td>
<td>Hardware</td>
</tr>
<tr>
<td>Virtual Memory</td>
<td>4-KB pages</td>
<td>Main memory</td>
<td>100</td>
<td>Hardware + OS</td>
</tr>
<tr>
<td>Buffer cache</td>
<td>Parts of files</td>
<td>Main memory</td>
<td>100</td>
<td>OS</td>
</tr>
<tr>
<td>Disk cache</td>
<td>Disk sectors</td>
<td>Disk controller</td>
<td>100,000</td>
<td>Disk firmware</td>
</tr>
<tr>
<td>Network buffer cache</td>
<td>Parts of files</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>NFS client</td>
</tr>
<tr>
<td>Browser cache</td>
<td>Web pages</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>Web browser</td>
</tr>
<tr>
<td>Web cache</td>
<td>Web pages</td>
<td>Remote server disks</td>
<td>1,000,000,000</td>
<td>Web proxy server</td>
</tr>
</tbody>
</table>
General Cache Concept

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”
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, then in main memory**
- **Typical system structure:**

![diagram](image-url)
Cache Algorithm (Read)

Look at Processor Address, search cache tags to find match. Then either

- Found in cache
  - a.k.a. HIT
  - Return copy of data from cache

- Not in cache
  - a.k.a. MISS
  - Read block of data from Main Memory
  - Wait ...
  - Return data to processor and update cache

Q: Which line do we replace?
Placement Policy

Memory

Set Number

Cache

block 12 can be placed

Fully Associative anywhere

(2-way) Set Associative anywhere in set 0 (12 mod 4)

Direct Mapped only into block 4 (12 mod 8)
Direct-Mapped Cache

- Tag
- Index
- Block Offset

V Tag Data Block

2^k lines

HIT

Data Word or Byte
Direct Map Address Selection

higher-order vs. lower-order address bits

Index | Tag | Block Offset

$V$ Tag Data Block

$2^k$ lines

HIT

Data Word or Byte
2-Way Set-Associative Cache

- Tag
- Index
- Block Offset
- V
- Block Offset
- Data Block
- V
- Tag
- Data Block
- V
- Tag
- Data Block
- =
- Data Word or Byte
- HIT
Fully Associative Cache

- **V**: Tag
- **Tag**: Data Block
- **Data Block**: 
- **Hit**: Data Word or Byte

Diagram shows the interaction between these components.
Replacement Policy

In an associative cache, which block from a set should be evicted when the set becomes full?

- Random

- Least-Recently Used (LRU)
  - LRU cache state must be updated on every access
  - true implementation only feasible for small sets (2-way)
  - pseudo-LRU binary tree often used for 4-8 way

- First-In, First-Out (FIFO) a.k.a. Round-Robin
  - used in highly associative caches

- Not-Most-Recently Used (NMRU)
  - FIFO with exception for most-recently used block or blocks

- Most Recently Used (?)

Replacement only happens on misses

This is a second-order effect. Why?
Block Size and Spatial Locality

Block is unit of transfer between the cache and memory

<table>
<thead>
<tr>
<th>Tag</th>
<th>Word0</th>
<th>Word1</th>
<th>Word2</th>
<th>Word3</th>
</tr>
</thead>
</table>

4 word block, \( b = 2 \)

Split CPU address

block address

offset\(_b\)

32-b bits

2\(^b\) = block size a.k.a line size (in bytes)

b bits

Larger block size has distinct hardware advantages

- less tag overhead
- exploit fast burst transfers from DRAM
- exploit fast burst transfers over wide busses

What are the disadvantages of increasing block size?

*Fewer blocks \(\Rightarrow\) more conflicts. Can waste bandwidth.*
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**
Itanium-2 On-Chip Caches
(Intel/HP, 2002)

Level 1: 16KB, 4-way s.a., 64B line, quad-port (2 load+2 store), single cycle latency

Level 2: 256KB, 4-way s.a., 128B line, quad-port (4 load or 4 store), five cycle latency

Level 3: 3MB, 12-way s.a., 128B line, single 32B port, twelve cycle latency
Power 7 On-Chip Caches [IBM 2009]

32KB L1 I$/core
32KB L1 D$/core
3-cycle latency

256KB Unified L2$/core
8-cycle latency

32MB Unified Shared L3$
Embedded DRAM (eDRAM)
25-cycle latency to local slice
IBM z196 Mainframe Caches 2010

- 96 cores (4 cores/chip, 24 chips/system)
  - Out-of-order, 3-way superscalar @ 5.2GHz
- L1: 64KB I-$$/core + 128KB D-$$/core
- L2: 1.5MB private/core (144MB total)
- L3: 24MB shared/chip (eDRAM) (576MB total)
- L4: 768MB shared/system (eDRAM)
Intel Core i7 Cache Hierarchy

Processor package

Core 0

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

Core 3

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

... (remaining 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.
**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 * 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
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

Aggressive prefetching

Ridges of temporal locality

Slopes of spatial locality

Core i7 Haswell
2.1 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
  - Using blocking to improve temporal locality
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
/* 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 = 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

\[
C_{i,j} = A_{i,k} \times B_{k,j}
\]
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;
    }
}
```

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 (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>

Inner loop:
- Row-wise
- Column-wise
- Fixed
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>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>
Matrix Multiplication (ijk)

/* 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:
(i,k)  (k,*)  (i,*)
A      B      C

Fixed  Row-wise  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>
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;
    }
}
```

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

**ijk (& jik):**
- 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

Cycles per inner loop iteration vs. Array size (n)

- jki / kji
- ijk / jik
- kij / ikj
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 \( C \ll n \) (much smaller than \( n \))

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

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

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

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

**Total misses:**
- $9n/8 \times n^2 = (9/8) \times n^3$
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 */
                    /* 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];
}
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
- $2n/B * B^2/8 = nB/4$ (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 $3B^2 < C$

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

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

- Speculate on future instruction and data accesses and fetch them into cache(s)
  - Instruction accesses easier to predict than data accesses

- Varieties of prefetching
  - Hardware prefetching
  - Software prefetching
  - Mixed schemes

- What types of misses does prefetching affect?
Issues in Prefetching

- Usefulness – should produce hits
- Timeliness – not late and not too early
- Cache and bandwidth pollution
Hardware Instruction Prefetching

Instruction prefetch in Alpha AXP 21064

- Fetch two blocks on a miss; the requested block (i) and the next consecutive block (i+1)
- Requested block placed in cache, and next block in instruction stream buffer
- If miss in cache but hit in stream buffer, move stream buffer block into cache and prefetch next block (i+2)
Hardware Data Prefetching

- **Prefetch-on-miss:**
  - Prefetch \( b + 1 \) upon miss on \( b \)

- **One Block Lookahead (OBL) scheme**
  - Initiate prefetch for block \( b + 1 \) when block \( b \) is accessed
  - *Why is this different from doubling block size?*
  - Can extend to N-block lookahead

- **Strided prefetch**
  - If observe sequence of accesses to block \( b, b+N, b+2N \), then prefetch \( b+3N \) etc.

Example: IBM Power 5 [2003] supports eight independent streams of strided prefetch per processor, prefetching 12 lines ahead of current access
Software Prefetching

\[
\text{for}(i=0; \ i < \ N; \ i++) \ \{ \\
\quad \text{prefetch}( \ &a[i + 1] \ ); \\
\quad \text{prefetch}( \ &b[i + 1] \ ); \\
\quad \text{SUM} = \text{SUM} + \ a[i] \ * \ b[i]; \\
\}
\]
Software Prefetching Issues

Timing is the biggest issue, not predictability

- If you prefetch very close to when the data is required, you might be too late
- Prefetch too early, cause pollution
- Estimate how long it will take for the data to come into L1, so we can set P appropriately
- *Why is this hard to do?*

```c
for(i=0; i < N; i++) {
    prefetch( &a[i + P] );
    prefetch( &b[i + P] );
    SUM = SUM + a[i] * b[i];
}
```

Must consider cost of prefetch instructions
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.
Acknowledgements

These slides contain material developed and copyright by:
- Arvind (MIT)
- Krste Asanovic (MIT/UCB)
- Joel Emer (Intel/MIT)
- James Hoe (CMU)
- John Kubiatowicz (UCB)
- David Patterson (UCB)

MIT material derived from course 6.823
UCB material derived from course CS252