Memory Hierarchy

Computer Systems Organization (Spring 2015)
CSCI-UA 201, Section 3

Instructor: Joanna Klukowska
Teaching Assistants: Paige Connelly & Carlos Guzman

Slides adapted from
Andrew Case, Jinyang Li, Mohamed Zahran, Stewart Weiss, Randy Bryant and Dave O'Hallaron
Random-Access Memory (RAM)

• Key features
  • RAM is traditionally packaged as a chip.
  • Basic storage unit is normally a cell (one bit per cell).
  • Multiple RAM chips form a memory.

• Static RAM (SRAM)
  • Each cell stores a bit with a four or six-transistor circuit.
  • Retains value indefinitely, as long as it is kept powered.
  • Relatively insensitive to electrical noise (EMI), radiation, etc.
  • Faster and more expensive than DRAM.

• Dynamic RAM (DRAM)
  • Each cell stores bit with a capacitor. One transistor is used for access.
  • Value must be refreshed every 10-100 ms.
  • More sensitive to disturbances (EMI, radiation,...) than SRAM.
  • Slower and cheaper than SRAM.
## SRAM vs DRAM Summary

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SRAM</strong></td>
<td>4 or 6</td>
<td>1X</td>
<td>No</td>
<td>Maybe</td>
<td>100X</td>
<td>L1-L3 cache memories</td>
</tr>
<tr>
<td><strong>DRAM</strong></td>
<td>1</td>
<td>10X</td>
<td>Yes</td>
<td>Yes</td>
<td>1X</td>
<td>Main memories, frame buffers</td>
</tr>
</tbody>
</table>
Nonvolatile Memories

• DRAM and SRAM are volatile memories
  • Lose information if powered off.
• Nonvolatile memories retain value even if powered off
  • Read-only memory (ROM): programmed during production
  • Programmable ROM (PROM): can be programmed once
  • Eraseable PROM (EPROM): can be bulk erased (UV, X-Ray)
  • Electrically eraseable PROM (EEPROM): electronic erase capability
  • Flash memory: EEPROMs with partial (sector) erase capability
    • Wears out after about 100,000 erasings.

• Uses for Nonvolatile Memories
  • Firmware programs stored in a ROM (BIOS, controllers for disks, network cards, graphics accelerators, security subsystems,...)
  • Solid state disks (replace rotating disks in thumb drives, smart phones, mp3 players, tablets, laptops,...)
  • Disk caches
Traditional Bus Structure Connecting CPU and Memory

• A **bus** is a collection of parallel wires that carry address, data, and control signals.
• Buses are typically shared by multiple devices.
Memory Read Transaction (1)

- CPU places address A on the memory bus.
Memory Read Transaction (2)

• Main memory reads A from the memory bus, retrieves word x, and places it on the bus.

Load operation: \texttt{movl A, %eax}
Memory Read Transaction (3)

• CPU read word $x$ from the bus and copies it into register %eax.

Load operation: `movl A, %eax`
Memory Write Transaction (1)

- CPU places address A on bus. Main memory reads it and waits for the corresponding data word to arrive.

Store operation: `movl %eax, A`
Memory Write Transaction (2)

- CPU places data word \( y \) on the bus.
Memory Write Transaction (3)

• Main memory reads data word \( y \) from the bus and stores it at address \( A \).

Store operation: `movl %eax, A`
The CPU-Memory Gap

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

- Disk seek time
- Flash SSD access time
- DRAM access time
- SRAM access time
- CPU cycle time
- Effective CPU cycle time
The CPU-Memory Gap

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

```
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 $\text{sum}$ each iteration.

- **Instruction references**
  - Reference instructions in sequence.
  - Cycle through loop repeatedly.
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;
}
```
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;
}
```
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.
An Example Memory Hierarchy

Smaller, faster, costlier per byte:
- CPU registers: hold words retrieved from L1 cache
- L1 cache: (SRAM) holds cache lines retrieved from L2 cache
- L2 cache: (SRAM) holds cache lines retrieved from main memory
- Main memory: (DRAM) holds disk blocks retrieved from local disks
- Local secondary storage: (local disks) hold files retrieved from disks on remote network servers

Larger, slower, cheaper per byte:
- Remote secondary storage: (tapes, distributed file systems, Web servers)
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**.

* The word cache has two meaning. In this context it refers to any type of memory from the memory hierarchy. The other meaning refers to some particular levels of the memory hierarchy.

** Note that in our book the top/highest level has the smallest number; the bottom/lowest level has the largest number.
General Cache Concepts

Level k
(ex. L3 cache)

Data is copied in block-sized transfer units

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

Larger, slower, cheaper memory viewed as partitioned into “blocks”

Level k+1
(ex. 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.
Cache Analogy: Hungry! must eat!

• Option 1: go to a refrigerator
  • Found -> eat!
  • Latency = 1 minute

• Option 2: go to a store
  • Found -> purchase, take home, eat!
  • Latency = 20-30 minutes

• Option 3: grow food!
  • Plant, wait ... wait ... wait ..., harvest, eat!
  • Latency = ~250,000 minutes (~ 6 months)
## Examples of Caching in the 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</td>
</tr>
<tr>
<td>L1 cache</td>
<td>64-bytes block</td>
<td>On-Chip L1</td>
<td>1</td>
<td>Hardware</td>
</tr>
<tr>
<td>L2 cache</td>
<td>64-bytes block</td>
<td>On/Off-Chip L2</td>
<td>10</td>
<td>Hardware</td>
</tr>
<tr>
<td>Virtual Memory</td>
<td>4-KB page</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>AFS/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 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:**

\[ C = S \times E \times B \text{ data bytes} \]

* Notice that this does not include the space required by valid bit and the tag.
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)

valid bit
Direct Mapped Cache ($E = 1$)

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

$S = 2^s$ sets

Address of int:

$\begin{array}{l}
\text{t bits} \\
\hline
0...01 \\
100
\end{array}$

find set
Direct Mapped Cache (E = 1)

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

valid?  +  match: assume yes = hit

Address of int:

v  tag  0 1 2 3 4 5 6 7

t bits 0...01 100

block offset
Direct Mapped Cache (E = 1)

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

No match: old line is evicted and replaced
DNHI: A Higher Level 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; i < 16; i++)
        for (i = 0; j < 16; j++)
            sum += a[i][j];

    return sum;
}
```

Ignore the variables sum, i, j
assume: cold (empty) cache, a[0][0] goes here

Figure out how many cache misses and cache hits occur when each of the functions is run.
Assume direct mapped cache.
E-way Set Associative Cache (Here: $E = 2$)

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

Address of short int:

- **t bits**: 0...01 100

**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:

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

compare both

v tag 0 1 2 3 4 5 6 7 v tag 0 1 2 3 4 5 6 7

block offset

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

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

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

Ignore the variables `sum`, `i`, `j`

assume: cold (empty) cache, `a[0][0]` goes here

```c
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

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

    return sum;
}
```

Figure out how many cache misses and cache hits occur when each of the functions is run.
Assume 2-way mapped cache.
What about writes?

• Multiple copies of data exist:
  • L1 cache, L2 cache, 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 immediately to memory)

• Typical
  • Write-through + No-write-allocate
  • Write-back + Write-allocate
Intel Core i7 Cache Hierarchy

Processor package

Core 0
- Regs
- L1 d-cache
- L1 i-cache
- L2 unified cache
- L3 unified cache (shared by all cores)

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

... 

L1 i-cache and d-cache: 32 KB, 8-way, Access: 4 cycles
- i-cache stores instructions
- d-cache stores data

L2 unified cache: 256 KB, 8-way, Access: 11 cycles

L3 unified cache: 8 MB, 16-way, Access: 30-40 cycles

Block size: 64 bytes for all caches.
Cache Performance Metrics

• **Miss Rate**
  • Fraction of memory references not found in cache (misses / accesses)
  • Typical numbers (in percentages):
    • 3-10% for L1
    • can be quite small (e.g., < 1%) for L2, depending on size, etc.

• **Hit Rate** = 1 – Miss Rate

• **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:
    • 1-2 clock cycle for L1
    • 5-20 clock cycles for L2

• **Miss Penalty**
  • Additional time required because of a miss
    • typically 50-200 cycles for main memory (Trend: increasing!)
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”
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)
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 of inner loop

\[
\begin{align*}
\text{A} & \quad \downarrow \quad \downarrow \quad \downarrow \\
\quad k & \quad \quad i \\
\text{B} & \quad \downarrow \\
\quad k & \quad \downarrow \quad j \\
\text{C} & \quad \downarrow \\
\quad i & \quad \downarrow \quad j
\end{align*}
\]
Matrix Multiplication Example

• Description:
  • Multiply N x 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

```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
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) > 4 bytes, exploit spatial locality
    • compulsory miss rate = 4 bytes / B

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

/* 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>
Matrix Multiplication (kij)

/* 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:

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)

```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:
- 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;
    }
}
```

Inner loop:

- Column-wise
- Fixed
- Column-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>1.0</td>
<td>0.0</td>
<td>1.0</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

```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;
    }
}
```

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

```c
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
- misses/iter = 0.5

```c
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
- 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 *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++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n+j] += a[i*n + k]*b[k*n + j];
}
```

\[
\begin{array}{c}
c \\
\end{array}
= \begin{array}{c}
a \\
i \\
\end{array}
* \begin{array}{c}
b \\
j \\
\end{array}
\]
Cache Miss Analysis

• Assume:
  • Matrix elements are doubles
  • Cache block = 8 doubles
  • Cache size $C \ll n$ (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 \( C \ll n \) (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 */
                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
  • $2n/B \times B^2/8 = nB/4$ (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:
  • $nB/4 \times (n/B)^2 = n^3/(4B)$
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
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

• 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)
    • Use small strides (spatial locality)