Memory Hierarchy

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
The CPU-Memory Gap

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

![Graph showing the comparison between Disk seek time, SSD access time, DRAM access time, SRAM access time, CPU cycle time, and Effective CPU cycle time over the years 1985 to 2015.]
Principle of Locality

• Temporal locality
  – Recently referenced items are likely to be referenced in the near future

• Spatial locality
  – Items with nearby addresses tend to be referenced close together in time
Principle of Locality: Example

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

• Data
  – Reference array elements in succession  Spatial locality
  – Reference `sum` each iteration  Temporal locality

• Instructions
  – Reference instructions in sequence  Spatial locality
  – Cycle through loop repeatedly  Temporal locality
Memory Hierarchy

• Some fundamental and enduring properties of hardware and software
  – Fast storage technologies cost more per byte, have less capacity, and require more power
  – 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
Memory Hierarchy: Example

- **L0:** Registers
- **L1:** L1 cache (SRAM)
- **L2:** L2 cache (SRAM)
- **L3:** L3 cache (SRAM)
- **L4:** Main memory (DRAM)
- **L5:** Local secondary storage (local disks)
- **L6:** Remote secondary storage (e.g. Web cache)

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

Larger, slower, and cheaper (per byte) storage devices
Exploiting Locality

• How to exploit temporal locality?
  – Speed up data accesses by caching data in faster storage
  – Caching in multiple levels: form a memory hierarchy
  – The lower levels of the memory hierarchy tend to be slower, but larger and cheaper

• How to exploit spatial locality?
  – Larger cache line size
  – Cache nearby data together
Cache

• A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device
• Improves the average access time
• Exploits both temporal and spatial locality
General Cache Concepts

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”

Memory
General Cache Concepts: Hit

Data in block b is needed

Block b is in cache:
Hit!

Cache

8 9 14 3

Request: 14

Memory

0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15

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)
CPU Cache Design Issues

• Cache size
  – 8KB ~ 64KB for L1

• Cache line size
  – Typically, 32 ~ 64 bytes for L1

• Lookup
  – Fully associative
  – Set associative: 2-way, 4-way, 8-way, 16-way, etc.
  – Direct mapped

• Replacement
  – LRU (Least Recently Used)
  – FIFO (First-In First-Out), RANDOM, etc.
Intel Core i7 Cache Hierarchy

Processor package

Core 0

- Registers
- L1 d-cache
- L2 unified cache

Core 3

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

... (repeated for other cores)

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
Cache Performance

• Average memory access time \(= T_{hit} + R_{miss} \times T_{miss} \)

• Hit time \(T_{hit}\)
  – Time to deliver a line in the cache to the processor
  – Includes time to determine whether the line is in the cache
  – 1 clock cycle for L1, 3 ~ 8 clock cycles for L2

• Miss rate \(R_{miss}\)
  – Fraction of memory references not found in cache
  – 3 ~ 10% for L1, < 1% for L2

• Miss penalty \(T_{miss}\)
  – Additional time required because of a miss
  – Typically 25 ~ 100 cycles for main memory
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)
  – Sequential reference patterns are good (spatial locality)
Matrix Multiplication (1)

• Description
  – Multiply N x N matrices
  – $O(N^3)$ total operations

```
/* 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;
    }
}
```

• Assumptions
  – Line size = 32 bytes (big enough for 4 64-bit words)
  – Matrix dimension (N) is very large
Matrix Multiplication (2)

• 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>0.25</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Matrix Multiplication (3)

• 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>Misses</td>
<td>0.25</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Matrix Multiplication (4)

- 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];
    }
}
```

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>

**Misses per Inner Loop Iteration:**

- Fixed
- Row-wise
- Row-wise
Matrix Multiplication (5)

- Matrix multiplication (ikj)

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

**Misses per Inner Loop Iteration:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>
Matrix Multiplication (6)

- 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>Misses</td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Matrix Multiplication (7)

• 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>1.0</td>
<td>0.0</td>
<td>1.0</td>
<td></td>
</tr>
</tbody>
</table>
## Matrix Multiplication (8)

### Summary

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

```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;
    }
}
```

**kij (& ikj):**
- 2 loads, 1 store
- misses/iter = 0.5

```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];
    }
}
```

**jki (& kji):**
- 2 loads, 1 store
- misses/iter = 2.0

```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;
    }
}
```

### Matrices

$$
\begin{array}{ccc}
A & B & C \\
0.25 & 1.0 & 0.0 \\
\end{array}
\begin{array}{ccc}
A & B & C \\
0.0 & 0.25 & 0.25 \\
\end{array}
\begin{array}{ccc}
A & B & C \\
1.0 & 0.0 & 1.0 \\
\end{array}
$$
Matrix Multiplication (9)

- Performance in Core i7

![Graph showing performance in Core i7 with different array sizes and cycle counts for jki / kji, ijk / jik, and kij / ikj.]
Summary

• 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)