Memory Hierarchy

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
The CPU-Memory Gap

- The increasing gap between DRAM, disk, and CPU speeds

![Graph showing the increasing gap between disk seek time, DRAM access time, SRAM access time, and CPU cycle time over the years 1980 to 2000. The x-axis represents the years, and the y-axis represents the time in nanoseconds (ns). The graph shows a decrease in time across the years, indicating improvements in technology.]
Principle of Locality (1)

- **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 (2)

- Locality example:

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

- Data
  - Reference array elements in succession  \textbf{Spatial locality}
  - Reference sum each iteration  \textbf{Temporal locality}

- Instructions
  - Reference instructions in sequence  \textbf{Spatial locality}
  - Cycle through loop repeatedly  \textbf{Temporal locality}
Principle of Locality (3)

- **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.
Memory Hierarchy

L0: CPU registers
- Smaller, faster, and costlier (per byte) storage devices
- CPU registers hold words retrieved from L1 cache.

L1: on-chip L1 cache (SRAM)
- Smaller, faster, and costlier (per byte) storage devices
- L1 cache holds cache lines retrieved from the L2 cache memory.

L2: off-chip L2 cache (SRAM)
- Larger, slower, and cheaper (per byte) storage devices
- L2 cache holds cache lines retrieved from main memory.

L3: main memory (DRAM)
- Larger, slower, and cheaper (per byte) storage devices
- Main memory holds disk blocks retrieved from local disks.

L4: local secondary storage (local disks)
- Local disks hold files retrieved from disks on remote network servers.

L5: remote secondary storage (distributed file systems, Web servers)
- Larger, slower, and cheaper (per byte) storage devices

Caching (1)

- **Cache**
  - A smaller, faster storage
  - Improves the *average* access time
  - Exploits both temporal and spatial locality
Caching (2)

- Cache performance metrics
  - 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
    - (misses/references)
    - 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
Caching (3)

- **Cache design issues**
  - Cache size
    - 8KB ~ 64KB for L1
  - Cache line size
    - Typically, 32B or 64B 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), etc.
Matrix Multiplication (1)

- **Description**
  - Multiply N x N matrices
  - $O(N^3)$ total operations

- **Assumptions**
  - Line size = 32B (big enough for 4 64-bit words)
  - Matrix dimension (N) is very large

```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;
  }
}
```
Matrix Multiplication (2)

- 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 (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></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 (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];
    }
}
```

**Misses per Inner Loop Iteration:**

<table>
<thead>
<tr>
<th></th>
<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>
<td></td>
</tr>
</tbody>
</table>

Inner loop:

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

- 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>

Inner loop:
- Column-wise
- Fixed
- Column-wise
Matrix Multiplication (8)

### Summary

- **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

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

```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];
    }
}
```

```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;
    }
}
```
Matrix Multiplication (9)

- Performance in Pentium

![Graph showing performance in Pentium for different array sizes and matrix multiplication methods.](image-url)
Blocked Matrix Multiplication (1)

- **Improving temporal locality by blocking**
  - “Block” means a sub-block within the matrix.
  - Example: N = 8, sub-block size = 4

\[
\begin{bmatrix}
  A_{11} & A_{12} \\
  A_{21} & A_{22}
\end{bmatrix}
\begin{bmatrix}
  B_{11} & B_{12} \\
  B_{21} & B_{22}
\end{bmatrix}
= \begin{bmatrix}
  C_{11} & C_{12} \\
  C_{21} & C_{22}
\end{bmatrix}
\]

- Key idea: Sub-blocks (i.e., \( A_{xy} \)) can be treated just like scalars.

\[
C_{11} = A_{11}B_{11} + A_{12}B_{21} \quad C_{12} = A_{11}B_{12} + A_{12}B_{22}
\]
\[
C_{21} = A_{21}B_{11} + A_{22}B_{21} \quad C_{22} = A_{21}B_{12} + A_{22}B_{22}
\]
Blocked Matrix Multiplication (2)

- **Performance in Pentium**
  - Blocking improves performance by a factor of two over unblocked version (ijk and jik)
Calculating Pi (1)

- \[ \tan \frac{\pi}{4} = 1 \quad \rightarrow \quad \arctan(1) = \frac{\pi}{4} \]

- \[ 4 \int_{0}^{1} \frac{1}{1+x^2} \, dx \]
  \[ = 4 \left( \arctan(1) - \arctan(0) \right) \]
  \[ = 4 \left( \frac{\pi}{4} - 0 \right) = \pi \]
Calculating Pi (2)

- **Pi: Sequential version**

```c
#define N 20000000
#define STEP (1.0 / (double) N)

double compute ()
{
    int i;
    double x;
    double sum = 0.0;

    for (i = 0; i < N; i++)
    {
        x = (i + 0.5) * STEP;
        sum += 4.0 / (1.0 + x * x);
    }

    return sum * STEP;
}
```
Calculating Pi (3)

- False sharing

```c
struct thread_stat {
    int count;
    char pad[PADS];
} stat[8];

double compute () {
    int i, id;
    double x, sum = 0.0;

    #pragma omp parallel for
    private(x) reduction(+:sum)
    for (i = 0; i < N; i++) {
        int id=omp_get_thread_num();
        stat[id].count++;
        x = (i + 0.5) * STEP;
        sum += 4.0 / (1.0 + x * x);
    }
    return sum * STEP;
}
```

![Graph showing speedup with different pad values](attachment://graph.png)
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)