Caching and Performance

Kai Shen

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 caches (e.g., L1, L2, and L3), then in main memory.
- Typical system structure:

Cache Set Association

- E-way set-associative cache.
- Special cases:
  - Direct-mapped: One line per set.
  - Fully-mapped: One set for the whole cache.

Cache Read

- Locate set
- Check if any line in set has matching tag
- Yes + line valid: hit
  - Locate data starting at offset
**Cache Helps Programs with Locality**

- Cache improves the performance of programs with temporal locality
  - How?
- Cache improves the performance of programs with spatial locality
  - Cacheline-grained memory access (implicit prefetching)
  - Explicit prefetching

**What about writes?**

- Write-through
  - write immediately to memory
- Write-back
  - write to cache only (most of time)
  - defer write to memory until replacement of line

**Intel Core i7 Cache Hierarchy**

- Core 0
  - L1 d-cache
  - L1 i-cache
  - L2 unified cache
- ... Core 3
  - L1 d-cache
  - L1 i-cache
  - L2 unified cache
  - L3 unified cache (shared by all cores)

**Cache Performance**

- Huge difference between a hit and a miss
  - Could be 100x, if just L1 and main memory
- Could you believe 99% hits (fraction of memory accesses in cache) 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
- “Miss rate” is more illustrative than “hit rate”
The Memory Mountain

Intel Core i7
32 KB L1 i-cache
32 KB L1 d-cache
256 KB unified L2 cache
8M unified L3 cache
All caches on-chip

The Memory Mountain

Intel Core i7
32 KB L1 i-cache
32 KB L1 d-cache
256 KB unified L2 cache
8M unified L3 cache
All caches on-chip

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)

- Case studies
  - Rearrange loops to improve spatial locality
  - Use blocking to improve temporal locality
Matrix Multiplication Performance

```
/* 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;
    }
}
```

- Multiply n x n matrices
- \(O(n^3)\) total operations
- 3n² data accesses
- Cache misses dominate the performance
  - Some data accesses go to registers
  - Some memory accesses hit in cache

```
Variable sum held in register
```

```
/* 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 (ijk)

Assume matrix dimension (n) is so large that a single row can’t fit into cache.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.125</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Other implementations?

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 += \(a[i][0]\);
  - accesses successive elements
  - recall that we load a cacheline at a time
  - if cache line size \((64) > 8\) bytes, exploit spatial locality
    - miss rate = \(8 / 64\)
- Stepping through rows in one column:
  - for \(i = 0; i < n; i++\)
    - sum += \(a[0][i]\);
  - accesses distant elements
  - no spatial locality!
    - miss rate = 1 (i.e. 100%)

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];
    }
}
```

```
/* 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>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>

Inner loop:

Fixed

Row-wise

Row-wise

Column-wise

Fixed

\((i,k)\)

\((k,*)\)

\((i,*)\)
**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;
    }
}
```

**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;
    }
}
```

**Misses per inner loop iteration:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column-wise</td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

**Blocked Matrix Multiplication**

```
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)
            c[i*n+j] += a[i*n + k] * b[k*n + j];
```

---

**Core i7 Matrix Multiply Performance**

---

**Summary of Matrix Multiplication**

- **ijk (jki):**
  - misses/iter = 1.125

- **kij (ikj):**
  - misses/iter = 0.25

- **jki (kji):**
  - misses/iter = 2.0
Cache Miss Analysis

Assume:
- Cache line = 8 doubles (elements)
- Three blocks fit into cache
- B is the block size in number of elements

First (block) iteration:
- $B^2/8$ misses for each block
- $2n/B \times B^2/8 = nB/4$ (omitting matrix C)

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

Summary

- Blocking: $n^3/(4B)$
- No blocking: $n^3 \times 0.25$
- Suggest largest possible block size B, but limit it so three blocks fit into cache
  - I once used block size B=25 elements (doubles), what was the cache size?
- Foundation for performance enhancement:
  - 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

Disclaimer

These slides were adapted from the CMU course slides provided along with the textbook of “Computer Systems: A programmer’s Perspective” by Bryant and O’Hallaron. The slides are intended for the sole purpose of teaching the computer organization course at the University of Rochester.