15-213

Memory System Performance

October 29, 1998

Topics
  • Impact of cache parameters
  • Impact of memory reference patterns
    – matrix multiply
    – transpose
    – memory mountain range
Basic Cache Organization

Address space \((N = 2^n \text{ bytes})\)  

Cache \((C = S \times E \times B \text{ bytes})\)

Address space
\((n = t + s + b \text{ bits})\)

Valid bit | tag | data
---|---|---
1 bit | \(t\) bits | \(B = 2^b\) bytes (block size)

Cache block
(cache line)

\(E\) blocks/set

\(S = 2^s\) sets

Cache block

\(B = 2^b\) bytes (block size)
Multi-Level Caches

Can have separate Icache and Dcache or *unified* Icache/Dcache

- Processor
  - TLB
  - regs
  - L1 Dcache
  - L1 Icache
- L2 Dcache
- L2 Icache
- Memory
  - 128 MB DRAM
  - 70 ns
  - $1.50/MB
  - 10 GB
- disk

<table>
<thead>
<tr>
<th>size</th>
<th>speed</th>
<th>$/Mbyte</th>
<th>block size</th>
<th>larger, slower, cheaper</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 B</td>
<td>5 ns</td>
<td>$200/MB</td>
<td>4 B</td>
<td></td>
</tr>
<tr>
<td>8 KB</td>
<td>5 ns</td>
<td>$1.50/MB</td>
<td>16 B</td>
<td></td>
</tr>
<tr>
<td>1M SRAM</td>
<td>6 ns</td>
<td></td>
<td>32 B</td>
<td></td>
</tr>
<tr>
<td>128 MB DRAM</td>
<td>70 ns</td>
<td>$0.06/MB</td>
<td>4 KB</td>
<td></td>
</tr>
<tr>
<td>10 GB</td>
<td>10 ms</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

larger block size, higher associativity, more likely to write back
Cache Performance Metrics

Miss Rate

- fraction of memory references not found in cache (misses/references)
- Typical numbers:
  5-10% for L1
  1-2% for L2

Hit Time

- time to deliver a block in the cache to the processor (includes time to determine whether the block is in the cache)
- Typical numbers
  1 clock cycle for L1
  3-8 clock cycles for L2

Miss Penalty

- additional time required because of a miss
  - Typically 10-30 cycles for main memory
Impact of Cache and Block Size

Cache Size

• Effect on miss rate
  – Larger is better

• Effect on hit time
  – Smaller is faster

Block Size

• Effect on miss rate
  – Big blocks help exploit spatial locality
  – For given cache size, can hold fewer big blocks than little ones, though

• Effect on miss penalty
  – Longer transfer time
Impact of Associativity

• Direct-mapped, set associative, or fully associative?

Total Cache Size (tags+data)
• Higher associativity requires more tag bits, LRU state machine bits
• Additional read/write logic, multiplexors

Miss rate
• Higher associativity decreases miss rate

Hit time
• Higher associativity increases hit time
  – Direct mapped allows test and data transfer at the same time for read hits.

Miss Penalty
• Higher associativity requires additional delays to select victim
Impact of Write Strategy

• Write-through or write-back?

Advantages of Write Through

• Read misses are cheaper. Why?
• Simpler to implement.
• Requires a write buffer to pipeline writes

Advantages of Write Back

• Reduced traffic to memory
  – Especially if bus used to connect multiple processors or I/O devices
• Individual writes performed at the processor rate
Qualitative Cache Performance Model

Compulsory Misses
- First access to line not in cache
- Also called “Cold start” misses

Capacity Misses
- Active portion of memory exceeds cache size

Conflict Misses
- Active portion of address space fits in cache, but too many lines map to same cache entry
- Direct mapped and set associative placement only
Miss Rate Analysis

Assume

- Block size = 32B (big enough for 4 32-bit words)
- \( n \) is very large
  - Approximate \( 1/n \) as 0.0
- Cache not even big enough to hold multiple rows

Analysis Method

- Look at access pattern by inner loop
Interactions Between Program & Cache

Major Cache Effects to Consider

• Total cache size
  – Try to keep heavily used data in highest level cache
• Block size (sometimes referred to “line size”)
  – Exploit spatial locality

Example Application

• Multiply n X n matrices
• \( O(n^3) \) total operations
• Accesses
  – \( n \) 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 Performance (Sparc20)

- As matrices grow in size, exceed cache capacity
- Different loop orderings give different performance
  - Cache effects
  - Whether or not can accumulate in register
Layout of Arrays in Memory

C Arrays Allocated in Row-Major Order

- Each row in contiguous memory locations

Stepping Through Columns in One Row

```c
for (i = 0; i < n; i++)
    sum += a[0][i];
```
- Accesses successive elements
- For block size > 8, get spatial locality
  - Cold Start Miss Rate = 8/B

Stepping Through Rows in One Column

```c
for (i = 0; i < n; i++)
    sum += a[i][0];
```
- Accesses distant elements
- No spatial locality
  - Cold Start Miss rate = 1
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;
    }
}
```

Approx. Miss Rates

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Inner loop:

- Row-wise
- Column-wise
- Fixed
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
    }
}

Approx. Miss Rates

\[
\begin{array}{ccc}
  a & b & c \\
  0.25 & 1.0 & 0.0 \\
\end{array}
\]
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];
    }
}

Approx. Miss Rates

<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 (ikj)

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

Fixed Row-wise Row-wise

Approx. Miss Rates

<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>
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;
    }
}
```

Approx. Miss Rates

<table>
<thead>
<tr>
<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>
</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;
    }
}

Approx. Miss Rates

<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:

A
---
(*)

B
---
(k, j)

C
---
(*, j)

Column-wise
---

Fixed
---

Column-wise
---

class20.ppt
Summary of Matrix Multiplication

**ijk (L=2, S=0, MR=1.25)**
```
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

**jik (L=2, S=0, MR=1.25)**
```
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;
    }
}
```

**kij (L=2, S=1, MR=0.5)**
```
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];
    }
}
```

**ijk (L=2, S=1, MR=0.5)**
```
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];
    }
}
```

**jki (L=2, S=1, MR=2.0)**
```
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;
    }
}
```

**kji (L=2, S=1, MR=2.0)**
```
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;
    }
}
```
Matmult performance (DEC5000)

![Graph showing matrix multiplication performance](image-url)

- (L=2, S=1, MR=0.5)
- (L=2, S=0, MR=1.25)
- (L=2, S=1, MR=2.0)
Matmult Performance (Sparc20)

Multiple columns of B fit in cache?

- ikj
- kij
- ijk
- jik
- jki
- kji

(L=2, S=1, MR=0.5)
(L=2, S=0, MR=1.25)
(L=2, S=1, MR=2.0)
Matmult Performance (Alpha 21164)

Too big for L1 Cache

Too big for L2 Cache

(L=2, S=1, MR=0.5)
(L=2, S=0, MR=1.25)
(L=2, S=1, MR=2.0)
Block Matrix Multiplication

Example $n=8$, $B = 4$:

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

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

\[
\begin{align*}
C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\
C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\
C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\
C_{22} &= A_{21}B_{12} + A_{22}B_{22}
\end{align*}
\]
Blocked Matrix Multiply (bijk)

```c
for (jj=0; jj<n; jj+=bsize) {
    for (i=0; i<n; i++)
        for (j=jj; j < min(jj+bsize,n); j++)
            c[i][j] = 0.0;
    for (kk=0; kk<n; kk+=bsize) {
        for (i=0; i<n; i++) {
            for (j=jj; j < min(jj+bsize,n); j++) {
                sum = 0.0
                for (k=kk; k < min(kk+bsize,n); k++) {
                    sum += a[i][k] * b[k][j];
                }
                c[i][j] += sum;
            }
        }
    }
}
```
Blocked Matrix Multiply Analysis

- Innermost loop pair multiplies 1 X bsize sliver of A times bsize X bsize block of B and accumulates into 1 X bsize sliver of C
- Loop over i steps through n row slivers of A & C, using same B

\[
\text{for (i=0; i<n; i++)} \{ \\
  \text{for (j=jj; j < min(jj+bsize,n); j++)} \{ \\
    \text{sum = 0.0} \\
    \text{for (k=kk; k < min(kk+bsize,n); k++)} \{ \\
      \text{sum += a[i][k] * b[k][j];} \\
    \}
    \text{c[i][j] += sum;} \\
  \}
\}
\]

- Innermost Loop Pair

A

\text{row sliver accessed bsize times}

B

\text{block reused n times in succession}

C

\text{Update successive elements of sliver}
Blocked matmult perf (DEC5000)

- ■ bik
- ● bikj
- △ ikj
- ◆ ijk

mflops (d.p.)

matrix size (n)

50 75 100 125 150 175 200
Blocked matmult perf (Sparc20)
Blocked matmul perf (Alpha 21164)

The graph shows the performance of different matrix multiplication algorithms as a function of matrix size (n). The algorithms represented are:

- `bijk`
- `bikj`
- `ijk`
- `ikj`

The performance is measured in millions of floating-point operations per second (MFLOPS). The x-axis represents the matrix size (n), and the y-axis represents the MFLOPS.
Matrix transpose

Row-wise transpose:

```
for (i=0; i < N; i++)
    for (j=0; j < M; j++)
        dst[j][i] = src[i][j]
```

Column-wise transpose:

```
for (j=0; j < M; j++)
    for (i=0; i < N; i++)
        dst[j][i] = src[i][j]
```
Row-Wise Transposition

MB/s

Columns

Rows

11 MB/s

CS 213 F'98
Improved Transposition

Columns

Rows

MB/s

32  64  128  256  512  1024  2048

32

128

512

2048

0

100

200

300

400

500

600

45 MB/s
The Memory Mountain Range

DEC Alpha 8400 (21164)
300 MHz
8 KB (L1) 96 KB (L2) 4 M (L3)
Effects Seen in Mountain Range

Cache Capacity
  • See sudden drops as increase working set size

Cache Block Effects
  • Performance degrades as increase stride
    - Less spatial locality
  • Levels off
    - When reach single access per block