The Memory Hierarchy (4)

(Slides adapted from CSAPP)
Announcements

- Midterm exam: next Monday
  - Midterm topics: topics form the beginning up to end (including this class topics)

- HW 3 hand-out is posted
  - Due: Fri., Oct 21, 11:59 pm
Summary of previous class

- Cache organization and addressing
  - Direct mapped
  - Two associative cache

- Cache replacement policies

- Cache write policies

- This class: (more on caches)
  - Cache performance metrics (review)
  - Cache performance impact
  - Cache friendly programming
Agenda

- Intel Core Cache Hierarchy
- Cache performance metrics
- Cache performance impact
  - Memory mountain
- Cache friendly programming
  - Rearranging the loops to improve locality
  - Using blocking to improve temporal locality
- Midterm overview
- Project requirement overview
- HW 3 overview
Intel Core i7 Cache Hierarchy

Processor package

Core 0
- Regs
- L1 d-cache
- L1 i-cache
- L2 unified cache

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

... (additional cores)

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

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.

Main memory
Cache Performance Metrics

- Miss Rate
  - Fraction of memory references not found in cache (misses / accesses)
    \[ = 1 – \text{hit rate} \]
  - Typical numbers (in percentages):
    - 3-10\% for L1
    - can be quite small (e.g., < 1\%) for L2, depending on size, etc.

- 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)

Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories.
Performance impact of caches
The Memory Mountain

- **Read throughput** (read bandwidth)
  - Number of bytes read from memory per second (MB/s)

- **Memory mountain**: Measured read throughput as a function of spatial and temporal locality.
  - Compact way to characterize memory system performance.
Memory Mountain Test Function

/* The test function */
void test(int elems, int stride) {
    int i, result = 0;
    volatile int sink;

    for (i = 0; i < elems; i += stride)
        result += data[i];
    sink = result; /* So compiler doesn't optimize away the loop */
}

/* Run test(elems, stride) and return read throughput (MB/s) */
double run(int size, int stride, double Mhz) {
    double cycles;
    int elems = size / sizeof(int);

    test(elems, stride); /* warm up the cache */
cycles = fcyc2(test, elems, stride, 0); /* call test(elems,stride) */
return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}
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

Read throughput (MB/s)

Stride (x8 bytes)

Working set size (bytes)

Slopes of spatial locality

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

Slopes of spatial locality
Ridges of Temporal locality

Read throughput (MB/s)

Stride (x8 bytes)

Working set size (bytes)
Rearranging loops to improve 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

\[
\begin{align*}
A & : i \rightarrow k \\
B & : j \rightarrow k \\
C & : i \rightarrow j
\end{align*}
\]
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;
    }
}
```
Layout of C Arrays in Memory (review)

- C arrays are 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>Misses</td>
<td>0.25</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Inner loop:

- **Row-wise**
- **Column-wise**
- **Fixed**

Diagram:

```
  A
  ↑
(i,*)
  ↑
  B
  ↑
(*,j)
  ↑
  C
```

CEN591 Fall 2011
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>
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];
    }
}
```

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 (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 (jki)

\[
\text{Inner loop:} \quad (*) , k \\
\text{Column-wise} \\
A \\
\text{Fixed} \\
B \\
\text{Column-wise} \\
C \\
\]

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 (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>miss rate</td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Inner loop:

- (*,k) Column-wise
- (k,j) Fixed
- (*,j) Column-wise
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;
    }
}

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
Core i7 Matrix Multiply Performance

Cycles per inner loop iteration vs. Array size (n)

- jki / kji
- ijk / jik
- kij / ikj
Using blocking to improve temporal locality
Example: 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++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n+j] += a[i*n + k]*b[k*n + j];
}
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 $C << n$ (much smaller than $n$)

- Second iteration:
  - Again:
    - $n/8 + n = 9n/8$ misses

- Total misses:
  - $9n/8 * n^2 = (9/8) * n^3$
Blocked Matrix Multiplication

\[ c = (\text{double *} \text{calloc})(\text{sizeof(\text{double})}, n \times n); \]

/* 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; i1++)
                    for (j1 = j; j1 < j + B; j1++)
                        for (k1 = k; k1 < k + B; k1++)
                            \[ c[i1 \times n + j1] += a[i1 \times n + k1] \times b[k1 \times n + j1]; \]
}

Block size B x B
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)

n/B blocks

Block size $B \times B$
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)
Midterm overview
Midterm topics

- Ch 1 of CAQA book (sections 1.1 up to 1.10): Fundamentals of computer system designs (e.g., class of computers, quantitative comparisons of design principals).

- Ch 1 of CSAPP book: Computer system organization overview.

- Ch 2 of CSAPP book (sections 1 to 5): Representing and manipulating information e.g., integer representation and floating points.

- Ch 3 of CSAPP book (all sections excluding 3.6.7, 3.8.4, 3.8.5, 3.13.5, 3.13.6 and 3.14): Machine level Representation of programs e.g., operations, procedures, homogenous and heterogeneous data structures.

- Ch 6 of CSAPP book (all sections): Memory hierarchy.

- All other topics that are covered in the class.
Term project overview
Project overview

- Group: 3-4 people per group
- Phase one:
  - Phase one due: Oct 17
    - Report
    - 7 minutes presentation per each group
- Phase two
  - Phase two due: Nov 14 (tentatively)
    - 7 minutes presentation per each group
    - Report
- Phase 3
  - Report due: Last day of the class
  - Class presentation or a poster session

- Project phase requirements will be posted
- Send your slides before 11 am on the corresponding due date to TA
HW 3 overview

Code optimization
Hand-out and Hand-in

- What does handout include?
  - Instruction
  - kernels .c (The file you have to modify)
  - driver program
    - To evaluate the performance of your code
    - You have to compile it for every change in the kernels.c

- Hand-in
  - Submit your modified kernels.c to TA
    (zahracen591@gmail.com)
What does the homework about

- Developing different versions of rotating and smoothing an image (represented by a two dimensional matrix)
  - The baseline implementation model is given
Grading

- Work as a group
  - Up to three people at each group

- The grading is based on the performance improvement
  - Tentative grading scheme:
    - Full credit of 50 pts.: Improvement up to 20%
    - bonus credit of 20 pts: more than 30%