Lecture 14: Optimization with Caches

CS 105

March 9, 2020
Review: Memory Hierarchy

- **L0:** CPU registers hold words retrieved from the L1 cache.
- **L1:** L1 cache holds cache lines retrieved from the L2 cache.
- **L2:** L2 cache holds cache lines retrieved from the L3 cache.
- **L3:** L3 cache holds cache lines retrieved from main memory.
- **L4:** Main memory holds disk blocks retrieved from local disks.
- **L5:** Local secondary storage (local disks)
- **L6:** Remote secondary storage (e.g., cloud, web servers)

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

Larger, slower, and cheaper (per byte) storage devices
Review: Principle of Locality

Programs tend to use data and instructions with addresses near or equal to those they have used recently

- **Temporal locality:**
  - Recently referenced items are likely to be referenced again in the near future

- **Spatial locality:**
  - Items with nearby addresses tend to be referenced close together in time
Review: An Example Cache

E = 2: Two lines per set
Assume: cache block size 8 bytes

Address of data:

- **tag**: the rest of the bits
- **index**: \( \log(\# \text{ sets}) \) bits
- **offset**: \( \log(\text{block size}) \) bits
Typical Intel Core i7 Hierarchy

Processor package

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

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

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 Metrics

- **Miss Rate**
  - Fraction of memory references not found in cache (misses / accesses)
  - Typically 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
  - Typically 4 clock cycles for L1, 10 clock cycles for L2

- **Miss Penalty**
  - Additional time required because of a miss
    - typically 50-200 cycles for main memory (Trend: increasing!)
Memory Performance with Caching

- **Read throughput (aka 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

Call test() with many combinations of elems and stride.

For each elems and stride:

1. Call test() once to warm up the caches.

2. Call test() again and measure the read throughput (MB/s)

```c
long data[MAXELEMS];  /* Global array to traverse */

/* test - Iterate over first "elems" elements of array "data" with stride of "stride", using
using 4x4 loop unrolling. */
int test(int elems, int stride) {
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;

    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}
```
The Memory Mountain

**Core i7 Haswell**
- 2.1 GHz
- 32 KB L1 d-cache
- 256 KB L2 cache
- 8 MB L3 cache
- 64 B block size

**Aggressive prefetching**

**Slopes of spatial locality**

**Ridges of temporal locality**

**L1**

**L2**

**L3**

**Mem**

**Read throughput (MB/s)**

**Stride (x8 bytes)**

**Size (bytes)**
Locality Example

Which of the following functions is better in terms of locality with respect to array src?

```c
void copyji(int src[2048][2048],
           int dst[2048][2048])
{
    int i,j;
    for (i = 0; i < 2048; i++)
        for (j = 0; j < 2048; j++)
            dst[i][j] = src[i][j];
}
```

```c
void copyij(int src[2048][2048],
            int dst[2048][2048])
{
    int i,j;
    for (j = 0; j < 2048; j++)
        for (i = 0; i < 2048; i++)
            dst[i][j] = src[i][j];
}
```

4.3ms 81.8ms

2.0 GHz Intel Core i7 Haswell
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)
Example: Matrix Multiplication

- Multiply N x N matrices
- Matrix elements are doubles (8 bytes)
- $O(N^3)$ total operations
- N reads per source element
- N values summed per destination

```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;
    }
}
```
Miss Rate Analysis for Matrix Multiply

• Assume:
  • Block size = 32B (big enough for four doubles)
  • 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

\[
C_{i,j} = A_{i,k} \times B_{k,j}
\]
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:**
  - accesses successive elements
  - if data block size \((B) > \text{sizeof}(a_{ij})\) bytes, exploit spatial locality
    - miss rate = \(\frac{\text{sizeof}(a_{ij})}{B}\)

- **Stepping through rows in one column:**
  - accesses distant elements
  - no spatial locality!
    - miss rate = 1 (i.e. 100%)
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;
    }
}

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>0.25</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

2 loads, no stores per inner loop iteration
Matrix Multiplication (kij/ikj, kij/ikj)

/* 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];
    }
}

/* 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;
    }
}

Inner loop: (i,k)
2 loads, 1 store per inner loop iteration

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>

Inner loop: (k,*) (i,*)
2 loads, 1 store per inner loop iteration

Misses per inner loop iteration:
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
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
Matrix Multiply Performance

Core i7

- Cycles per inner loop iteration
- Array size (n)

Pentium III Xeon

- Cycles/iteration
- Array size (n)
Can we do better?

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 = 4 doubles
  - Cache size $C \ll n$ (much smaller than $n$)

- **First iteration:**
  - $\frac{n}{4} + n = \frac{5n}{4}$ misses

- **Afterwards in cache:**
  (schematic)
Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache block = 4 doubles
  - Cache size \( C \ll n \) (much smaller than \( n \))

- **Second iteration:**
  - \( \frac{n}{4} + n = \frac{5n}{4} \) misses

- **Total misses:**
  - \( \frac{5n}{4} \times n^2 = \left(\frac{5}{4}\right) \times n^3 \)
Blocked Matrix Multiplication

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

/* 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+=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; i++)
                    for (j1 = j; j1 < j+B; j++)
                        for (k1 = k; k1 < k+B; k++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
Cache Miss Analysis

• Assume:
  • Cache block = 4 doubles
  • Cache size C << n (much smaller than n)
  • Three blocks fit into cache: $3B^2 < C$

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

• Afterwards in cache
  (schematic)
Cache Miss Analysis

• Assume:
  • Cache block = 4 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
  • $2n/B \times B^2/4 = nB/2$

• Total misses:
  • $nB/2 \times (n/B)^2 = n^3/(2B)$
Blocking Summary

• No blocking: \( (5/4) * n^3 \)
• Blocking: \( n^3 / (4B) \)

• 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
A reality check

- This analysis only holds on some machines!

- Intel Core i7 does aggressive pre-fetching for one-stride programs, so blocking doesn't actually improve performance

- But on a Pentium III Xeon:
And that's the end of Part 1