CS 201

Writing Cache-Friendly Code

Gerson Robboy
Portland State University
An Example Memory Hierarchy

L0: CPU registers
- Hold words retrieved from L1 cache.

L1: on-chip L1 cache (SRAM)
- Hold cache lines retrieved from the L2 cache memory.

L2: off-chip L2 cache (SRAM)
- Hold cache lines retrieved from main memory.

L3: main memory (DRAM)
- Hold disk blocks retrieved from local disks.

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

L5: remote secondary storage (distributed file systems, Web servers)

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

- Larger, slower, and cheaper (per byte) storage devices
Why Cache-Friendly Code is Important

<table>
<thead>
<tr>
<th>Cache type</th>
<th>Size of item (bytes)</th>
<th>Latency (cpu cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>4 bytes</td>
<td>0</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>32 bytes</td>
<td>1</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>32 bytes</td>
<td>10</td>
</tr>
<tr>
<td>Main Memory</td>
<td>4-KB pages</td>
<td>100</td>
</tr>
<tr>
<td>Disk</td>
<td></td>
<td>millions</td>
</tr>
</tbody>
</table>

On ia32 processor, with few registers, even local variables are likely to spill to memory. We want them in cache!
Just what does a cache do?

The cache stores memory in units or *cache lines*

- Fixed length chunks, hardware dependent
- For our example, let’s say cache lines are 32 bytes
- Aligned on a cache-line (32 byte) boundary

When the CPU accesses a memory address (store or load), the cache line containing that address is pulled into the cache
Examples

Suppose a certain processor has a 32-byte cache line size.

You access address 0x3a40. What addresses are pulled into the cache?

You access address 0x3a94. What addresses are pulled into the cache?

Next you access 0x3a48. What happens?

You access 4 32-bit words sequentially, from 0x8000 to 0x801c

- How many cache misses and how many cache hits?
Locality

Principle of Locality:

- Programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves.
- **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.

Locality Example:

- **Data**
  - Reference array elements in succession (stride-1 reference pattern): **Spatial locality**
  - Reference `sum` each iteration: **Temporal locality**

- **Instructions**
  - Reference instructions in sequence: **Spatial locality**
  - Cycle through loop repeatedly: **Temporal locality**
Locality Example

Claim: Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.

Question: Does this function have good locality?
  - Spatial, temporal, both, or neither?

```c
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];

    return sum
}
```
Locality Example

Question: Does this function have good locality?

- Spatial, temporal, both, or neither?

```c
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];

    return sum
}
```
Locality Example

Question: Can you permute the loops so that the function scans the 3-d array $a[]$ with a stride-1 reference pattern (and thus has good spatial locality)?

```c
int sumarray3d(int a[M][N][N])
{
    int i, j, k, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                sum += a[k][i][j];
    return sum
}
```
Why does traversing a matrix with stride 1 give you good spatial locality?

Why do strides other than 1 give you bad spatial locality?
Writing Cache Friendly Code

Repeated references to variables are good (temporal locality)

Stride-1 reference patterns are good (spatial locality)

Examples:

- Cold cache, 4-byte words, 8-word cache blocks

```c
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```c
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Miss rate = \( \frac{1}{8} = 12.5\% \)

Miss rate = 100\%
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

```c
/* 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 */
}
```
/* mountain.c - Generate the memory mountain. */
#define MINBYTES (1 << 10) /* Working set size ranges from 1 KB */
#define MAXBYTES (1 << 23) /* ... up to 8 MB */
#define MAXSTRIDE 16 /* Strides range from 1 to 16 */
#define MAXELEMS MAXBYTES/sizeof(int)

int data[MAXELEMS];         /* The array we'll be traversing */

int main()
{
    int size;        /* Working set size (in bytes) */
    int stride;      /* Stride (in array elements) */
    double Mhz;      /* Clock frequency */

    init_data(data, MAXELEMS); /* Initialize each element in data to 1 */
    Mhz = mhz(0);              /* Estimate the clock frequency */
    for (size = MAXBYTES; size >= MINBYTES; size >>= 1) {
        for (stride = 1; stride <= MAXSTRIDE; stride++)
            printf("%.1f\t", run(size, stride, Mhz));
        printf("\n");
    }
    exit(0);
}
The Memory Mountain

Pentium III Xeon
550 MHz
16 KB on-chip L1 d-cache
16 KB on-chip L1 i-cache
512 KB off-chip unified
L2 cache

Slopes of Spatial Locality
Ridges of Temporal Locality

read throughput (MB/s)

stride (words)

working set size (bytes)
Ridges of Temporal Locality

Slice through the memory mountain with stride=1

- illuminates read throughputs of different caches and memory

![Graph showing read throughput (MB/s) vs. working set size (bytes) for main memory region, L2 cache region, and L1 cache region. The x-axis represents working set size in bytes (8m, 4m, 2m, 1024k, 512k, 256k, 128k, 64k, 32k, 16k, 8k, 4k, 2k, 1k), and the y-axis represents read throughput in MB/s (0 to 1200). The graph displays bars for each working set size, indicating the throughput values for each region.]
A Slope of Spatial Locality

Slice through memory mountain with size=256KB

- shows cache block size.

![Bar chart showing read throughput (MB/s) vs. stride (words) for different strides. The chart indicates one access per cache line.](chart.png)
Matrix Multiplication Example

Major Cache Effects to Consider

- Total cache size
  - Exploit temporal locality and keep the working set small (e.g., by using blocking)
- Block size
  - Exploit spatial locality

Description:

- 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
Miss Rate Analysis for Matrix Multiply

Assume:

- Line size = 32B (big enough for 4 64-bit words)
- Matrix dimension (N) is very large
  - Approximate 1/N as 0.0
- Cache is not big enough to hold multiple rows

Analysis Method:

- Look at access pattern of inner loop
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[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)

/* 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>
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>Misses</td>
<td>0.25</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
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];
    }
}
```

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)

/* 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>1.0</td>
<td>0.0</td>
<td>1.0</td>
<td></td>
</tr>
</tbody>
</table>
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>
Summary of Matrix Multiplication

**ijk (\& jik):**
- 2 loads, 0 stores
- misses/iter = 1.25

```c
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;
    }
}
```

**kij (\& ikj):**
- 2 loads, 1 store
- misses/iter = 0.5

```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];
    }
}
```

**jki (\& kji):**
- 2 loads, 1 store
- misses/iter = 2.0

```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;
    }
}
```
Pentium Matrix Multiply Performance

Miss rates are helpful but not perfect predictors.

- Code scheduling matters, too.

![Graph showing performance trends for different array sizes and permutations.]
Improving Temporal Locality by Blocking

Example: Blocked matrix multiplication

- “block” (in this context) does not mean “cache block”.
- Instead, it mean a sub-block within the matrix.
- Example: N = 8; sub-block size = 4

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

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

\[
C_{11} = A_{11}B_{11} + A_{12}B_{21}
\]
\[
C_{12} = A_{11}B_{12} + A_{12}B_{22}
\]
\[
C_{21} = A_{21}B_{11} + A_{22}B_{21}
\]
\[
C_{22} = A_{21}B_{12} + A_{22}B_{22}
\]
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 a $1 \times bsize$ sliver of $A$ by a $bsize \times bsize$ block of $B$ and accumulates into $1 \times bsize$ sliver of $C$
- Loop over $i$ steps through $n$ row slivers of $A$ & $C$, using same $B$

```
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;
    }
}
```

Row sliver accessed $bsize$ times
Block reused $n$ times in succession
Update successive elements of sliver
Let’s try to see what this does

```c
for (jj=0; jj<n; jj+=bsize) { // for each bsize block
    // skip zeroing C for now
    for (kk=0; kk<n; kk+=bsize) {
        for (i=0; i<n; i++) { // for each row of A
            for (j=jj; j < min(jj+bsize,n); j++) {
                sum = 0.0
                // For each element of the sliver of A/column of B
                for (k=kk; k < min(kk+bsize,n); k++) {
                    sum += a[i][k] * b[k][j];
                }
                c[i][j] += sum;
            }
        }
    }
}
```

- `kk` points to the start of a block in B.
- `jj` points to the start of a block in A.
- `i` points to a row in A.
- `j` points to a column in B.
- `k` points to an element in A.

This code snippet iterates over blocks of matrices A and B, calculating the product of corresponding elements and storing the result in matrix C.
So here’s the point of blocking

- Use a block size smaller than the size of the CPU cache
- The row sliver and the block in B are re-used many times in a row.
- They are in cache after the first time they are used.
- Then go on to another small block, get it in the cache.
- If you do it in the right order, you multiply all the horizontal slivers in A times one block in B, before going on to another block in B.

![Diagram showing the process of blocking]

- Row sliver accessed \( bsize \) times
- Block reused \( n \) times in succession
- Update successive elements of sliver
Pentium Blocked Matrix Multiply Performance

Blocking (bijk and bikj) improves performance by a factor of two over unblocked versions (ijk and jik)

- relatively insensitive to array size.
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)