

Writing cache friendly code

Repeated references to variables are good (temporal locality)

Stride-1 reference patterns are good (spatial locality)

Example

- cold cache, 4-byte words, 4-word cache blocks

```
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;
}
```

Miss rate = 25%

```
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 = ?

CompOrg - Cache and Programs

1

Practice Problem 6.15

- 1024 byte (data) direct mapped cache
- block size 16 bytes

```
struct algae_position {
    int x,y;
}
```

```
struct algae_position grid[16][16];
int total_x=0, total_y=0;
int i,j;
```

- Assume grid starts at address 0, cold cache.
- Everything except grid is in registers.

CompOrg - Cache and Programs

2

Practice Problem 6.15 (cont.)

```
for (i=0;i<16;i++)
    for (j=0;j<16;j++)
        total_x += grid[i][j].x;
```

```
for (i=0;i<16;i++)
    for (j=0;j<16;j++)
        total_y += grid[i][j].y;
```

- What is the total # of memory reads?
- What is the total # of reads that are misses?
- What is the miss rate?

CompOrg - Cache and Programs

3

6.15 Answer

Cache is 1024 bytes (2^{10} bytes)

Each slot is 16 bytes (2^4 bytes)

Cache has 64 slots (2^6 bytes)

Each element of grid is a 2-word struct (2^3 bytes)

Grid has $16 \cdot 16 = 2^8$ elements.

Total size of grid is $2^8 \cdot 2^3 = 2^{11}$ (2048) bytes.

Only $\frac{1}{2}$ of grid will fit in the cache.

Each time a word is placed in the cache, the entire block (4 words) is put in the cache.

- The block # is based on the MS bits of the word address
- cache slot is determined by LS 6 bits

CompOrg - Cache and Programs

4

PP 6.15 memory accesses: loop1

address of grid is 0.

address of `grid[0][0].x` is 0

- this is a miss, puts bytes 0-15 in the cache
- address of `grid[0][1].x` is 8
 - this is already in the cache.
- address of `grid[0][2].x` is 16
 - this is a miss, now bytes 16-31 are in the cache also.
- First loop repeats this pattern until we hit `grid[8][0]`.
- `grid[8][0].x` will be at address 1024
 - maps to the same slot as `grid[0][0]`!
- now cache is filled with 2nd half of the array grid.
 - same pattern: hit, miss, hit, miss, ...

CompOrg - Cache and Programs

5

PP 6.15 memory accesses: loop2

address of grid is 0.

address of `grid[0][0].y` is 4

- this is a miss, puts bytes 0-15 in the cache
- address of `grid[0][1].x` is 12
 - this is already in the cache.
- address of `grid[0][2].x` is 20
 - this is a miss, now bytes 16-31 are in the cache also.
- First loop repeats this pattern until we hit `grid[8][0]`.
- `grid[8][0].y` will be at address 1028
 - maps to the same slot as `grid[0][0]`!
- now cache is filled with 2nd half of the array grid.
 - same pattern: hit, miss, hit, miss, ...

CompOrg - Cache and Programs

6

6.15 summary

- first loop has 16x16 ($2^4 * 2^4 = 2^8$) memory accesses
 - 1/2 of these are hits, 1/2 are misses
- second loop also has 256 memory accesses
 - but nothing left in the cache from the first loop is used
 - same pattern: 1/2 are hits, 1/2 are misses

Total memory access = 512

Total misses = 256

Hit rate=50%

More practice (6.16)

What if the cache was twice as big?

- the entire grid array would fit in the cache
 - no two elements map to the same slot.
- The second loop would be all hits.
- Total hit rate would be 25%

Even more practice (6.17)

New loop (original cache size of 1024 bytes)

```
for (i=0;i<16;i++) {  
  for (j=0;j<16;j++) {  
    total_x += grid[i][j].x;  
    total_y += grid[i][j].y;  
  }  
}
```

All access to `grid[i][j].y` will be hits!

Hit rate will be 25%

What if we double the cache size for this code?

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.

CompOrg - Cache and Programs

10

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 = fcy2(test, elems, stride, 0); /* call test(elems, stride) */
    return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}
```

CompOrg - Cache and Programs

11

Memory mountain main routine

```
/* 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)

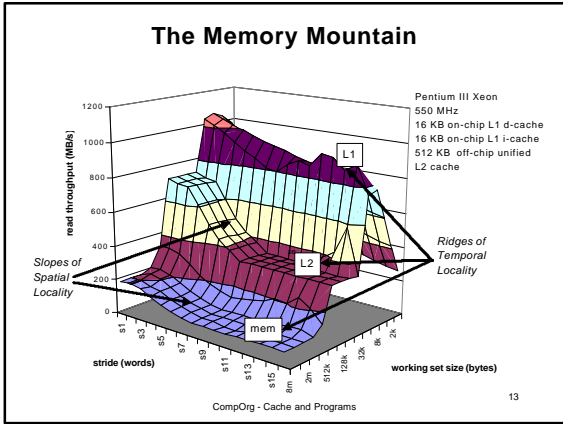
int data[MAXElems]; /* The array we'll be traversing */

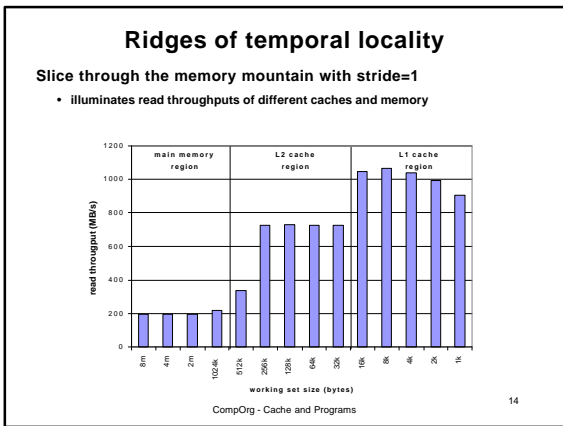
int main()
{
    int size; /* Working set size (in bytes) */
    int stride; /* Stride (in array elements) */
    double Mhz; /* Clock frequency */

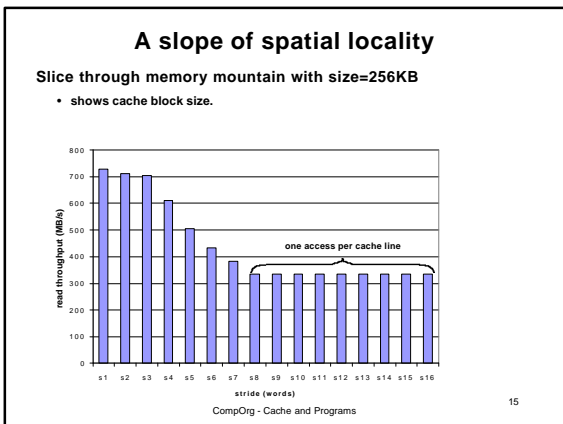
    init_data(data, MAXElems); /* Initialize each element in data to 1 */
    Mhz = mhz(0); /* Estimate the clock frequency */
    for (size = MAXBYTES; size >= MINBYTES; size >= 1) {
        for (stride = 1; stride <= MAXSTRIDE; stride++)
            printf("%.1f\t", run(size, stride, Mhz));
        printf("\n");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

CompOrg - Cache and Programs

12







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

```

/* ijk */
for (i=0; i<n; i++) { Variable sum held in register
  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;
  }
}
    
```

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

CompOrig - Cache and Programs

16

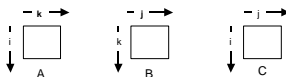
Miss rate analysis for matrix multiply

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 even big enough to hold multiple rows

Analysis Method:

- Look at access pattern of inner loop



CompOrig - Cache and Programs

17

Layout of arrays in memory

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%)

CompOrig - Cache and Programs

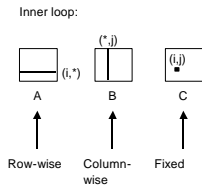
18

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:

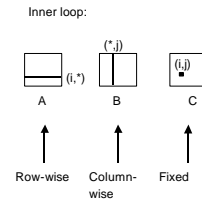
| A | B | C |
|------|-----|-----|
| 0.25 | 1.0 | 0.0 |

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;
  }
}

```



Misses per Inner Loop Iteration:

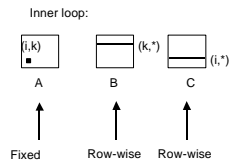
| A | B | C |
|------|-----|-----|
| 0.25 | 1.0 | 0.0 |

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:

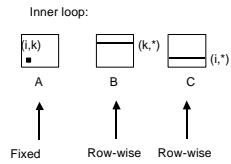
| A | B | C |
|-----|------|------|
| 0.0 | 0.25 | 0.25 |

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];
  }
}

```



Misses per Inner Loop Iteration:

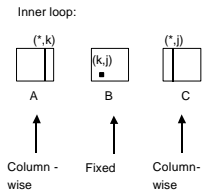
| A | B | C |
|-----|------|------|
| 0.0 | 0.25 | 0.25 |

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:

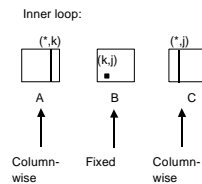
| A | B | C |
|-----|-----|-----|
| 1.0 | 0.0 | 1.0 |

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:

| A | B | C |
|-----|-----|-----|
| 1.0 | 0.0 | 1.0 |

Summary of matrix multiplication

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

```
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;
  }
}
```

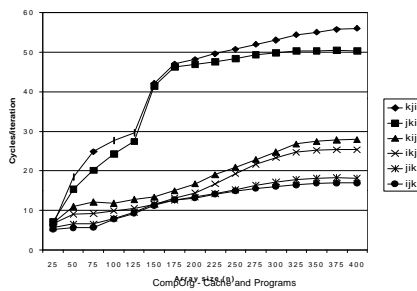
CompOrg - Cache and Programs

25

Pentium matrix multiply performance

Notice that miss rates are helpful but not perfect predictors.

- Code scheduling matters, too.



CompOrg - Cache and Programs

26

Improving temporal locality by blocking

Example: Blocked matrix multiplication

- "block" (in this context) does not mean "cache block".
- Instead, it means 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.

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} \quad C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21} \quad C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

CompOrg - Cache and Programs

27

Blocked matrix multiply (bijk)

```

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

```

28

CompOrg - Cache and Programs

Blocked matrix multiply analysis

- Innermost loop pair multiplies a 1 X *bysize* sliver of A by a *bysize* X *bysize* block of B and accumulates into 1 X *bysize* 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+bysize,n); j++) {
    sum = 0.0
    for (k=kk; k < min(kk+bysize,n); k++) {
      sum += a[i][k] * b[k][j];
    }
    c[i][j] += sum;
  }
}

```

Innermost Loop Pair

row sliver accessed
bysize times

block reused *n* times
in succession

Update successive
elements of sliver

29

CompOrg - Cache and Programs

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.

| Array Size (n) | kji | jki | kj | ikj | ijk | ijk | bijk (size 25) | bikj (size 25) |
|----------------|-----|-----|----|-----|-----|-----|----------------|----------------|
| 25 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| 50 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 |
| 75 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 |
| 100 | 45 | 45 | 45 | 45 | 45 | 45 | 45 | 45 |
| 125 | 50 | 50 | 50 | 50 | 50 | 50 | 50 | 50 |
| 150 | 55 | 55 | 55 | 55 | 55 | 55 | 55 | 55 |
| 175 | 58 | 58 | 58 | 58 | 58 | 58 | 58 | 58 |
| 200 | 60 | 60 | 60 | 60 | 60 | 60 | 60 | 60 |

30

CompOrg - Cache and Programs

Concluding observations

Programmer can optimize for cache performance

- How data structures are organized
- How data accessed
 - Nested loop structure
 - Blocking (see text) is a general technique

All machines like "cache friendly code"

- Getting absolute optimum performance 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)
