Announcements

• HW 4: Malloc
  • Final deadline Thursday
  • You should have received comments on your design document
  • Please seek a meeting or feedback from course staff if needed
  • MMTMOH: Mammoth Multi-TF Malloc Office Hours
    • Wednesday evening. See website for details

• Midterm exam
  • Thursday Oct 27
  • Practice exams posted on iSites
    (both College and Extension)
Mid-course evaluation

- 43 responses (122 enrolled students)
- Pace seems about right
  - Some thought too slow, some thought too fast
- Making sections effective
  - Not compulsory (no in-section quizzes)
  - Section notes are available on website before section
    - Generally on Friday
    - Encouraged to look at notes before section, figure out where you need to focus
- Lecture videos available to all students
  - Link from the schedule page
- Feedback
  - HW1 feedback took time; assignments available for pickup in MD 143
  - HW2 and 3 (binary bomb and buffer bomb) feedback was automatic
  - Will endeavor to give you timely feedback on remaining assignments
Dennis Ritchie ’63

• Co-creator of C programming language
• Co-developer (with Ken Thompson) of UNIX operating system
  • C is the foundation of UNIX
• Undergrad (’63) and PhD (’68) at Harvard
• Worked at Bell Labs for 40 years
• Profound impact on computer science

1941-2011
Topics for today

• Cache performance metrics
• Discovering your cache's size and performance
• The “Memory Mountain”
• Matrix multiply, six ways
• Blocked matrix multiplication
• Exploiting locality in your programs
Cache Performance Metrics

- **Miss Rate**
  - Fraction of memory references not found in cache (# misses / # references)
  - Typical numbers:
    - 3-10% for L1
    - Can be quite small (e.g., < 1%) for L2, depending on size and locality.

- **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 cycles for L1; 5-20 clock cycles for L2

- **Miss Penalty**
  - Additional time required because of a miss
    - Typically 50-200 cycles for main memory
  - Average access time = hit time + (miss rate × miss penalty)
Wait, what do those numbers mean?

- 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

- Repeated references to variables are good (**temporal locality**)
- Stride-1 reference patterns are good (**spatial locality**)
- Examples:
  - cold cache, 4-byte words, 4-word cache blocks

```c
int sum_array_rows(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 sum_array_cols(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 = \(1/4 = 25\%\)

Miss rate = 100\%
Determining cache characteristics

- Say you have a machine but don’t know its cache size or speeds.
- How would you figure these values out?
- Idea: Write a program to measure the cache’s behavior and performance.
  - Program needs to perform memory accesses with different locality patterns.
- Simple approach:
  - Allocate array of size $W$ words
  - Loop over the array repeatedly with stride $S$ and measure memory access time
  - Vary $W$ and $S$ to estimate cache characteristics

$S = 4$ words

$W = 32$ words
Determining cache characteristics

- What happens as you vary \( W \) and \( S \)?
- Changing \( W \) varies total amount of memory accessed by program
  - As \( W \) gets larger than one cache level, performance of program will drop
- Changing \( S \) varies the spatial locality of each access.
  - If \( S \) is less than the size of a cache line, sequential accesses will be fast.
  - If \( S \) is greater than the size of a cache line, sequential accesses will be slower.
- See end of lecture notes for example C program to do this.
As stride increases, the spatial locality of program gets worse.

As working set size increases, no longer fits entirely in a cache level, temporal locality worsens.
Varying Working Set

- Keep stride constant at $S = 16$ words, and vary $W$ from 1KB to 64MB
- Shows size and read throughputs of different cache levels and memory
Varying stride

- Keep working set constant at $W = 256$ KB, vary stride from 1-32 words
Core i7

Core i7
2.67 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache

Ridges of temporal locality
Slopes of spatial locality
Pentium Xeon

Pentium Nocona Xeon x86-64
3.2 GHz
12 Kuop on-chip L1 trace cache
16 KB on-chip L1 d-cache
1 MB off-chip unified L2 cache
Opteron Memory Mountain

Read throughput (MB/s)

Stride (words)

L1

L2

Mem

Working set (bytes)

AMD Opteron 2 GHz
Topics for today

- Cache performance metrics
- Discovering your cache's size and performance
- The “Memory Mountain”
- Matrix multiply, six ways
- Blocked matrix multiplication
- Exploiting locality in your programs
Matrix Multiplication Example

- Matrix multiplication is heavily used in numeric and scientific applications.
  - It's also a nice example of a program that is highly sensitive to cache effects.
- Multiply two N x N matrices
  - $O(N^3)$ total operations
  - Read N values for each source element
  - Sum up N values for each destination

```c
void mmm(double *a, double *b, double *c, int n) {
  int i, j, k;
  /* 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;
    }
  }
}
```
Matrix Multiplication Example

```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;
    }
}
```

\[
4 \times 3 + 2 \times 2 + 7 \times 5 = 51
\]

\[
\begin{array}{ccc}
4 & 2 & 7 \\
1 & 8 & 2 \\
6 & 0 & 1 \\
\end{array}
\times
\begin{array}{ccc}
3 & 0 & 1 \\
2 & 4 & 5 \\
5 & 9 & 1 \\
\end{array}
=
\begin{array}{c}
51 \\
\end{array}
\]
Miss Rate Analysis for Matrix Multiply

• Assume:
  • Line size = 32B (big enough for four 64-bit “double” values)
  • Matrix dimension N is very large
  • Cache is not big enough to hold multiple rows

• Analysis Method:
  • Look at access pattern of inner loop

\[
\begin{align*}
A \times B &= C \\
i &\quad \downarrow \quad \quad \quad \downarrow \\
\quad \quad \quad k &\quad \quad \quad \quad \quad j \\
\quad \quad \quad i &\quad \quad \quad \quad \quad j
\end{align*}
\]
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
  - Compulsory miss rate: \((8 \text{ bytes per double}) / \text{(block size of cache)}\)

- 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 = 100%
Matrix Multiplication (ijk)

- 2 loads, 0 stores per iteration
- Assume cache line size of 32 bytes, so 4 doubles per line
- Misses per iteration:

\[
\begin{align*}
A &= 0.25 \\
B &= 1 \\
C &= 0 \\
\text{Total} &= 1.25
\end{align*}
\]
/* 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;
    }
}

First iteration:
\[ A \] \[ B \] \[ C \]

After first iteration in cache (schematic):
\[ A \] \[ B \] \[ C \]
\[ 4 \text{ doubles wide} \]
Cache miss analysis

```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;
    }
}
```

First iteration:
- Row-wise: 4 doubles wide

After first iteration in cache (schematic):
Matrix Multiplication (jik)

- Same as ijk, just swapped order of outer loops
- 2 loads, 0 stores per iteration
- Assume cache line size of 32 bytes, so 4 doubles per line
- Misses per iteration:

  A = 0.25  
  B = 1  
  C = 0  
  Total: 1.25

```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;
    }
}
```
Matrix Multiplication (kij)

- 2 load, 1 store per iteration
- Assume cache line size of 32 bytes, so 4 doubles per line
- Misses per iteration:

  \[
  A = 0 \\
  B = 0.25 \\
  C = 0.25 \\
  \text{Total: } 0.5
  \]
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];
    }
}
```

- Same as kij, just swapped order of outer loops
- 2 load, 1 store per iteration
- Assume cache line size of 32 bytes, so 4 doubles per line
- Misses per iteration:
  
  \[
  \begin{align*}
  A &= 0 \\
  B &= 0.25 \\
  C &= 0.25 \\
  \text{Total: } &= 0.5
  \end{align*}
  \]
Matrix Multiplication (jki)

- 2 load, 1 store per iteration
- Assume cache line size of 32 bytes, so 4 doubles per line
- Misses per iteration:

```
A = 1
B = 0
C = 1
Total: 2
```
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;
    }
}
```

Inner loop:
- \((*,k)\)
- \((k,j)\)
- \((*,j)\)

- Same as kji, just swapped order of outer loops
- 2 load, 1 store per iteration
- Assume cache line size of 32 bytes, so 4 doubles per line
- Misses per iteration:
  - \(A = 1\)
  - \(B = 0\)
  - \(C = 1\)
  - Total: 2
Summary of Matrix Multiplication

**ijk or 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 or 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 or 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;
    }
}
```
Matrix Multiply Performance

- Each implementation doing same number of arithmetic operations, but $\sim20\times$ difference!
- Pairs with same number of mem. references and misses per iteration almost identical
- Miss rate better predictor or performance than number of mem. accesses!
- For large N, kij and ikj performance almost constant. Due to **hardware prefetching**, able to recognize stride-1 patterns.
Topics for today

• Cache performance metrics
• Discovering your cache's size and performance
• The “Memory Mountain”
• Matrix multiply, six ways
• Blocked matrix multiplication
• Exploiting locality in your programs
Using blocking to improve locality

• Blocked matrix multiplication
  • Break matrix into smaller blocks and perform independent multiplications on each block.
  • Improves locality by operating on one block at a time.
  • Best if each block can fit in the cache!

• Example: Break each matrix into four sub-blocks

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

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

\[
C_{11} = A_{11}B_{11} + A_{12}B_{21} \quad C_{12} = A_{11}B_{12} + A_{12}B_{22} \\
C_{21} = A_{21}B_{11} + A_{22}B_{21} \quad C_{22} = A_{21}B_{12} + A_{22}B_{22}
\]
Blocked Matrix Multiply

void bmmm(int n, double a[n][n], double b[n][n], double c[n][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][j1] += a[i1][k1] * b[k1][j1];
}

• Partition arrays into \textit{bsize} \times \textit{bsize} chunks
• Innermost \textit{(i1, j1, k1)} loop pair multiplies an \textit{A} chunk by a \textit{B} chunk and accumulates result in a \textit{C} chunk

Code becomes harder to read! Is it worth it?
Tradeoff between performance and maintainability...
Blocked matrix multiply

- Assume 3 chunks can fit into the cache, i.e., $3bsize^2 < C$

- First block iteration

  ![Diagram of the first block iteration]

- After first iteration in cache (schematic)

  ![Diagram after first iteration]
Cache miss analysis

- Assume 3 chunks can fit into the cache
- Assume $bsize$ is a multiple of 4
- $bsize^2/4$ misses per chunk, so $3/4 \times bsize^2$ misses per chunk iteration
- $(n/bsize)^3$ chunk iterations
- Total of $(n/bsize)^3 \times 3/4 \times bsize^2$ misses = $n^3 \times 3/(4 \times bsize)$
- Compare with $n^3 \times 1/2$ total misses for kij algorithm
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.
Blocked Matrix Multiply Performance

Intel Core i7
2.7 GHz
32 KB L1 d-cache
256 KB L2 cache
8MB L3 cache
CAVEAT: Tested on a VM

Cycles per loop iteration

n

ijk
jik
jki
kji
kij
ikj
bijk
bikj
Blocked Matrix Multiply Performance

Intel Core i7
2.7 GHz
32 KB L1 d-cache
256 KB L2 cache
8MB L3 cache

CAVEAT: Tested on a VM

Cycles per loop iteration

n
Exploiting locality in your programs

• Focus attention on inner loops
  • This is where most computation and memory accesses in your program occurs

• Try to maximize spatial locality
  • Read data objects sequentially, with stride 1, in the order they are stored in memory

• Try to maximize temporal locality
  • Use a data object as often as possible once it has been read from memory
Next lecture

- Virtual memory
  - Using memory as a cache for disk
Cache performance test program

/* 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)
{
    uint64_t start_cycles, end_cycles, diff;
    int elems = size / sizeof(int);

    test(elems, stride); /* warm up the cache */
    start_cycles = get_cpu_cycle_counter(); /* Read CPU cycle counter */
    test(elems, stride); /* Run test */
    end_cycles = get_cpu_cycle_counter(); /* Read CPU cycle counter again */
    diff = end_cycles - start_cycles; /* Compute time */
    return (size / stride) / (diff / CPU_MHZ); /* convert cycles to MB/s */
}
# Cache performance main routine

```c
#define CPU_MHZ 2.8 * 1024.0 * 1024.0; /* e.g., 2.8 GHz */
#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) */

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