

The Memory Hierarchy

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



1

SRAM vs DRAM Summary

	Trans. per bit	Access time	Needs refresh?	Needs EDC?	Cost	Applications
SRAM	4 or 6	1X	No	Maybe	100x	Cache memories
DRAM	1	10X	Yes	Yes	1X	Main memories, frame buffers

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



4

Today

- Storage technologies and trends
- Locality of reference
- Caching in the memory hierarchy

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



2

Nonvolatile Memories

- DRAM and SRAM are volatile memories
 - Lose information if powered off.
- Nonvolatile memories retain value even if powered off
 - Read-only memory (ROM): programmed during production
 - Programmable ROM (PROM): can be programmed once
 - Erasable PROM (EPROM): can be bulk erased (UV, X-Ray)
 - Electrically erasable PROM (EEPROM): electronic erase capability
 - Flash memory: EEPROMs. with partial (block-level) erase capability
 - Wears out after about 100,000 erasures
- Uses for Nonvolatile Memories
 - Firmware programs stored in a ROM (BIOS, controllers for disks, network cards, graphics accelerators, security subsystems,...)
 - Solid state disks (replace rotating disks in thumb drives, smart phones, mp3 players, tablets, laptops,...)
 - Disk caches

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



5

Random-Access Memory (RAM)

- Key features
 - RAM is traditionally packaged as a chip.
 - Basic storage unit is normally a cell (one bit per cell).
 - Multiple RAM chips form a memory.
- RAM comes in two varieties:
 - SRAM (Static RAM)
 - DRAM (Dynamic RAM)

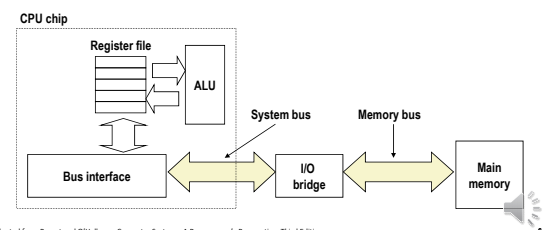
Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



3

Traditional Bus Structure Connecting CPU and Memory

- A bus is a collection of parallel wires that carry address, data, and control signals.
- Buses are typically shared by multiple devices.



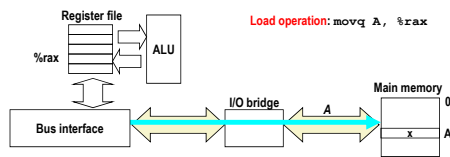
Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



6

Memory Read Transaction (1)

- ⌚ CPU places address A on the memory bus.



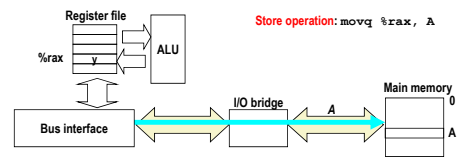
Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



7

Memory Write Transaction (1)

- ⌚ CPU places address A on bus. Main memory reads it and waits for the corresponding data word to arrive.



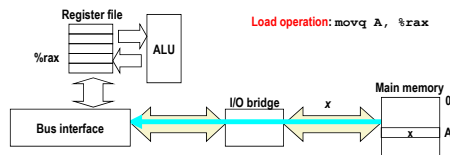
Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



10

Memory Read Transaction (2)

- ⌚ Main memory reads A from the memory bus, retrieves word x, and places it on the bus.



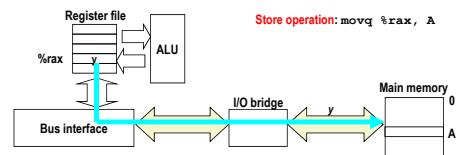
Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



8

Memory Write Transaction (2)

- ⌚ CPU places data word y on the bus.



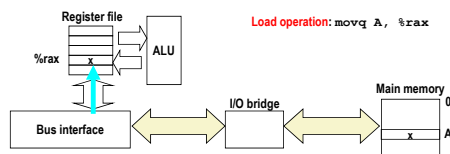
Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



11

Memory Read Transaction (3)

- ⌚ CPU read word x from the bus and copies it into register %rax.



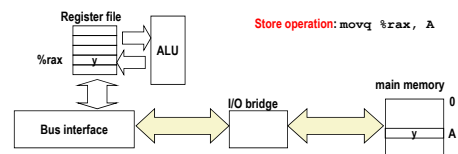
Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



9

Memory Write Transaction (3)

- ⌚ Main memory reads data word y from the bus and stores it at address A.



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



12

What's Inside A Disk Drive?

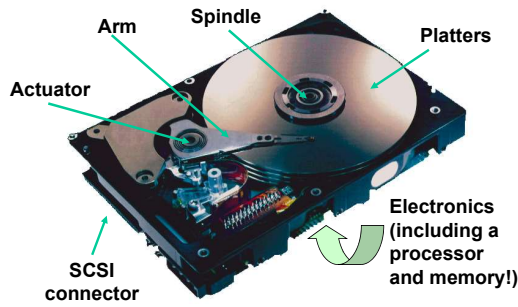
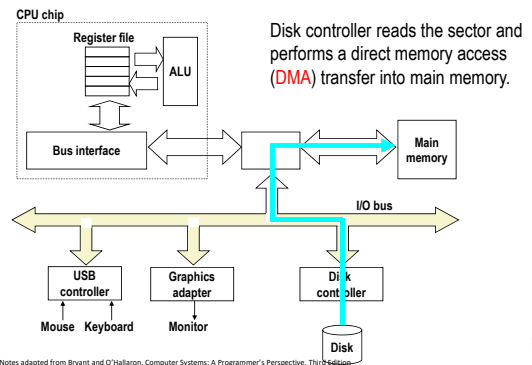


Image courtesy of Seagate Technology

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

13

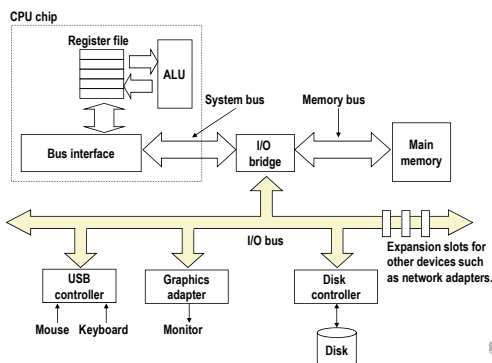
Reading a Disk Sector (2)



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

29

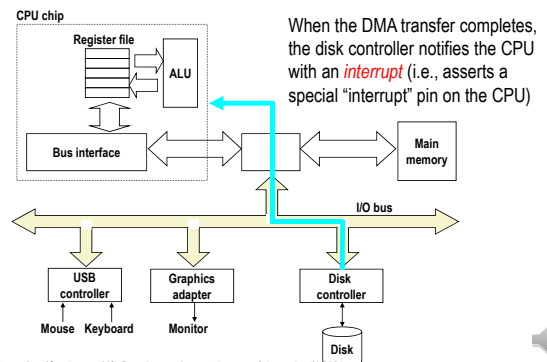
I/O Bus



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

27

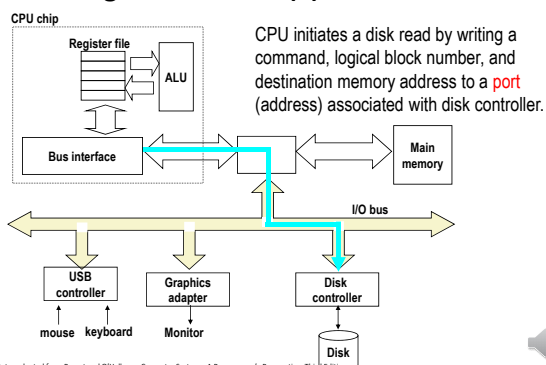
Reading a Disk Sector (3)



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

30

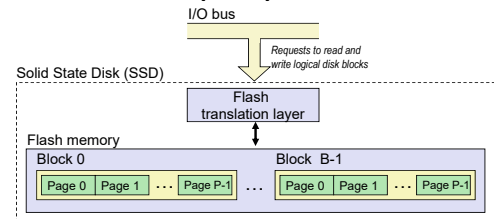
Reading a Disk Sector (1)



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

28

Solid State Disks (SSDs)



- Pages: 512KB to 4KB, Blocks: 32 to 128 pages
- Data read/written in units of pages.
- Page can be written only after its block has been erased
- A block wears out after about 100,000 repeated writes.

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

31

SSD Performance Characteristics

Sequential read tput	550 MB/s	Sequential write tput	470 MB/s
Random read tput	365 MB/s	Random write tput	303 MB/s
Avg seq read time	50 us	Avg seq write time	60 us

- Sequential access faster than random access
 - Common theme in the memory hierarchy
- Random writes are somewhat slower
 - Erasing a block takes a long time (~1 ms)
 - Modifying a block page requires all other pages to be copied to new block
 - In earlier SSDs, the read/write gap was much larger.

Source: Intel SSD 730 product specification.

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



32

Locality to the Rescue!

The key to bridging this CPU-Memory gap is a fundamental property of computer programs known as **locality**



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

35

SSD Tradeoffs vs Rotating Disks

- Advantages
 - No moving parts → faster, less power, more rugged
- Disadvantages
 - Have the potential to wear out
 - Mitigated by "wear leveling logic" in flash translation layer
 - E.g. Intel SSD 730 guarantees 128 petabyte (128×10^{15} bytes) of writes before they wear out
 - In 2015, about 30 times more expensive per byte
- Applications
 - MP3 players, smart phones, laptops
 - Beginning to appear in desktops and servers

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



33

Today

- Storage technologies and trends
- Locality of reference
- Caching in the memory hierarchy

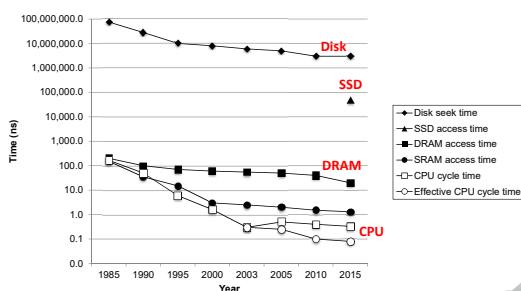


Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

36

The CPU-Memory Gap

The gap widens between DRAM, disk, and CPU speeds.



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

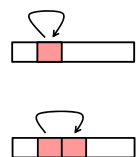


34

Locality

- 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



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

37

Locality Example

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

Data references

- Reference array elements in succession (stride-1 reference pattern).
- Reference variable `sum` each iteration.

Spatial locality

Temporal locality

Instruction references

- Reference instructions in sequence.
- Cycle through loop repeatedly.

Spatial locality

Temporal locality

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



Memory Hierarchies

Some fundamental and enduring properties of hardware and software:

- Fast storage technologies cost more per byte, have less capacity, and require more power (heat!).
- The gap between CPU and main memory speed is widening.
- Well-written programs tend to exhibit good locality.

These fundamental properties complement each other beautifully.

They suggest an approach for organizing memory and storage systems known as a **memory hierarchy**.

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



Qualitative Estimates of Locality

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 with respect to array `a`?

```
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;
}
```

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



Today

- Storage technologies and trends
- Locality of reference
- Caching in the memory hierarchy

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



Locality Example

Question: Does this function have good locality with respect to array `a`?

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

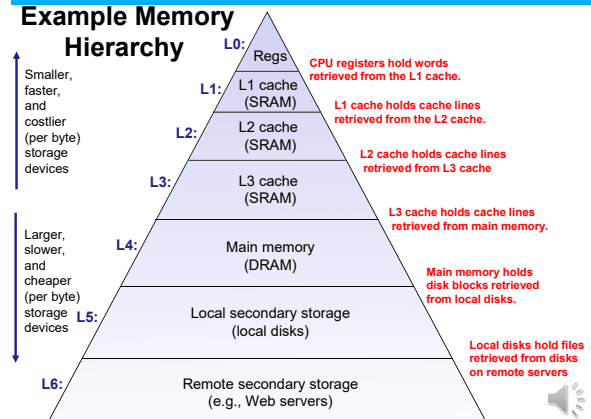
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];

    return sum;
}
```

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



Example Memory Hierarchy



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



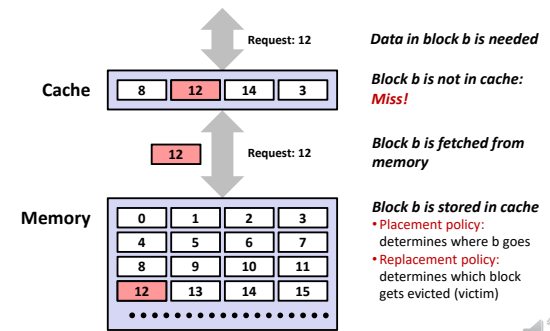
Caches

- 🔊 **Cache:** A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.
- 🔊 **Fundamental idea of a memory hierarchy:**
 - 🔊 For each k , the faster, smaller device at level k serves as a cache for the larger, slower device at level $k+1$.
- 🔊 **Why do memory hierarchies work?**
 - 🔊 Because of locality, programs tend to access the data at level k more often than they access the data at level $k+1$.
 - 🔊 Thus, the storage at level $k+1$ can be slower, and thus larger and cheaper per bit.
- 🔊 **Big Idea:** The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

45

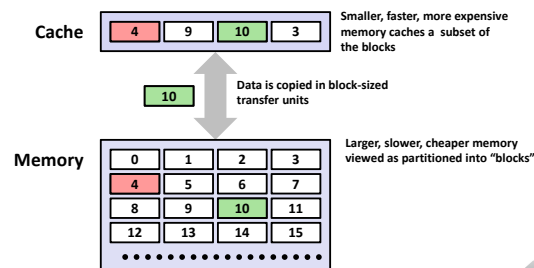
General Cache Concepts: Miss



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

46

General Cache Concepts



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

47

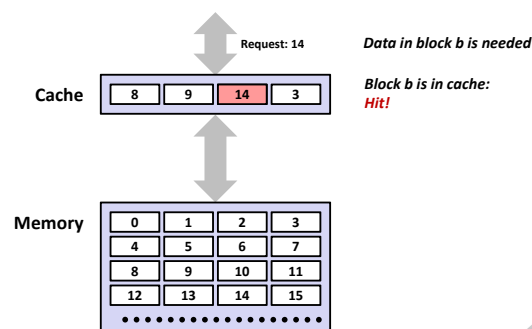
Examples of Caching in the Mem. Hierarchy

Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 bytes words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware MMU
L1 cache	64-byte blocks	On-Chip L1	4	Hardware
L2 cache	64-byte blocks	On-Chip L2	10	Hardware
Virtual Memory	4-KB pages	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer cache	Parts of files	Local disk	10,000,000	NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

48

General Cache Concepts: Hit



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

49

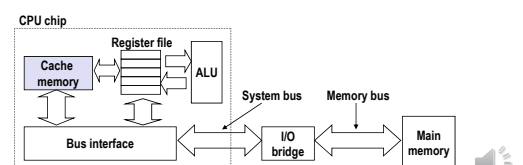
Cache Memories

- 🔊 **Cache memories** are small, fast SRAM-based memories managed automatically in hardware

- 🔊 Hold frequently accessed blocks of main memory

- 🔊 CPU looks first for data in cache

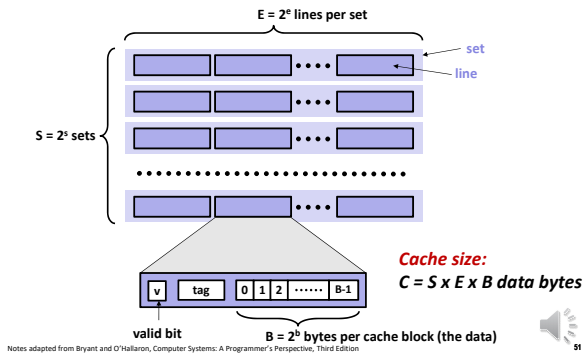
- 🔊 Typical system structure:



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

50

General Cache Organization (S, E, B)

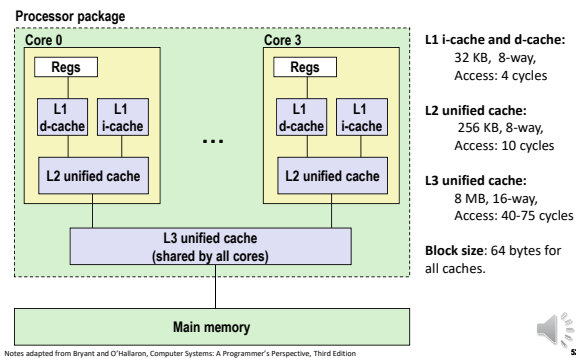


Let's think about those numbers

- 🕒 **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"**

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Intel Core i7 Cache Hierarchy



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

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Cache Performance Metrics

- 🕒 **Miss Rate**
 - 🕒 Fraction of memory references not found in cache (misses / accesses)
= 1 - 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:
 - 🕒 4 clock cycle 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!)

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

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.

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Memory Mountain Test Function

```
long data[MAXELEM]; /* 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+sx2];
        acc2 = acc2 + data[i+sx3];
        acc3 = acc3 + data[i+sx4];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }

    return ((acc0 + acc1) + (acc2 + acc3));
}
mountain/mountain.c
```

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)



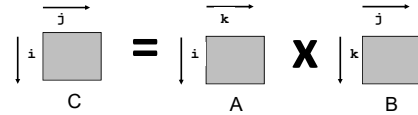
Miss Rate Analysis for Matrix Multiply

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:

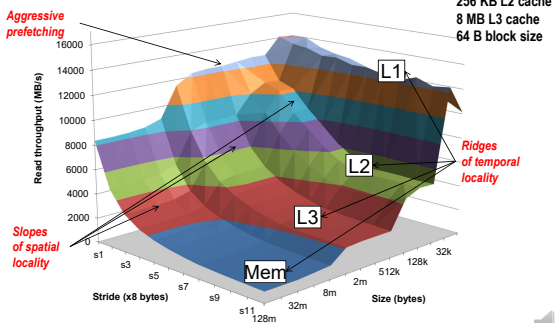
- Look at access pattern of inner loop



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



The Memory Mountain



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



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) > sizeof(a_{ij}) bytes, exploit spatial locality
 - miss rate = sizeof(a_{ij}) / B

Stepping through rows in one column:

- for (i = 0; i < n; i++)
 - sum += a[i][0];
- accesses distant elements
- no spatial locality!
 - miss rate = 1 (i.e. 100%)

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



Matrix Multiplication Example

Description:

- 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
 - but may be able to hold in register

```
/* 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;
    }
}
matmult/mm.c
```

Variable sum held in register

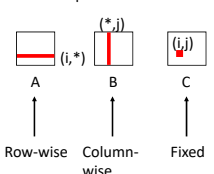


Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

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;
    }
}
matmult/mm.c
```

Inner loop:



Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

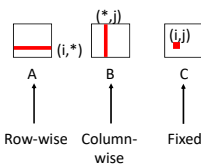


Matrix Multiplication (jik)

```
/* 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;
  }
}
```

matmult/mm.c

Inner loop:



Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

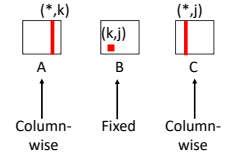
Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

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;
  }
}
```

matmult/mm.c

Inner loop:



Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

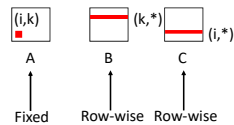
Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

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];
  }
}
```

matmult/mm.c

Inner loop:



Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

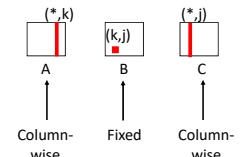
Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

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;
  }
}
```

matmult/mm.c

Inner loop:



Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

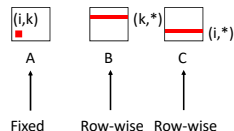
Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

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];
  }
}
```

matmult/mm.c

Inner loop:



Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

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;
  }
}
```

ijk (& jik):

- 2 loads, 0 stores
- misses/iter = 1.25

```
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];
  }
}
```

kij (& ikj):

- 2 loads, 1 store
- misses/iter = 0.5

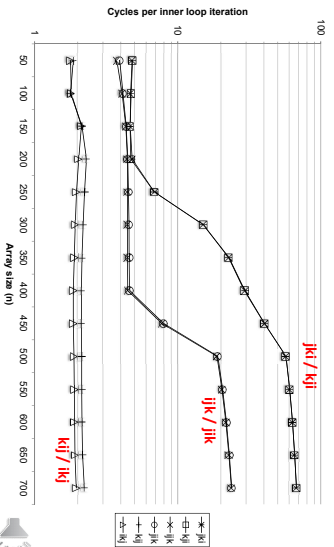
```
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

jki (& kji):

- 2 loads, 1 store
- misses/iter = 2.0

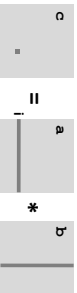
Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Core i7 Matrix Multiplication Performance



Example: Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);
/* Multiply n x n matrices a and b */
void mmul(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];
}
```



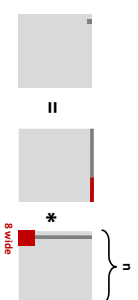
Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

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:

- $9n/8 * n^2 = (9/8) * n^3$



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Blocked Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);
/* Multiply n x n matrices a and b */
void mmul(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)
            /* B x B mini matrix multiplications */
            for (k1 = i; k1 < i+B; k1++)
                for (k2 = j; k2 < j+B; k2++)
                    c[k1*n + k2] += a[k1*n + k1] * b[k1*n + k2];
}
```



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Cache Miss Analysis

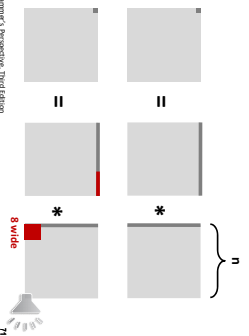
Assume:

- Matrix elements are doubles
- Cache block = 8 doubles
- Cache size $C < n$ (much smaller than n)

First iteration:

- $n/8 + n = 9n/8$ misses

Afterwards in cache: (schematic)



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Cache Miss Analysis

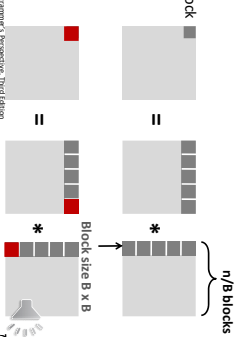
Assume:

- Cache block = 8 doubles
- Cache size $C < n$ (much smaller than n)
- Three blocks fit into cache: $3B^2 < C$

First (block) iteration:

- $B^2/8$ misses for each block
- $2n/B * B^2/8 = nB/4$ (omitting matrix c)

Afterwards in cache (schematic)



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Cache Miss Analysis

Assume:

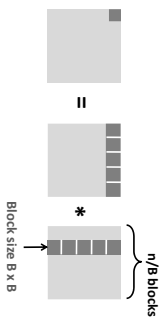
- Cache block = 8 doubles
- Cache size $C \ll n$ (much smaller than n)
- Three blocks \blacksquare fit into cache: $3B^2 < C$

Second (block) iteration:

- Same as first iteration
- $2n/B * B^2/8 = nB/4$

Total misses:

- $nB/4 * (n/B)^2 = n^3/(4B)$



Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



78

Blocking Summary

- No blocking: $(9/8) * n^3$
- Blocking: $1/(4B) * n^3$

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 element's used $O(n)$ times!
- But program has to be written properly

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



79

Summary

- The speed gap between CPU, memory and mass storage continues to widen.
- Well-written programs exhibit a property called *locality*.
- Memory hierarchies based on *caching* close the gap by exploiting *locality*.
 - Cache memories can have significant performance impact
- You can write **your programs to exploit this!**
 - Focus on the inner loops, where bulk of computations and memory accesses occur.
 - Try to maximize spatial locality by reading data objects with sequentially with stride 1.
 - Try to maximize temporal locality by using a data object as often as possible once it's read from memory.

Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



77