

Performance Optimizations (Cache)

CENG331 - Computer Organization

Instructor:

Murat Manguoglu (Sections 1-2)

Adapted from: <http://csapp.cs.cmu.edu/>

Today

- Cache organization and operation
- **Performance impact of caches**
 - The memory mountain
 - Rearranging loops to improve spatial locality
 - Using blocking to improve temporal locality

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

```
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));  
}
```

mountain/mountain.c

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

For each `elems` and `stride`:

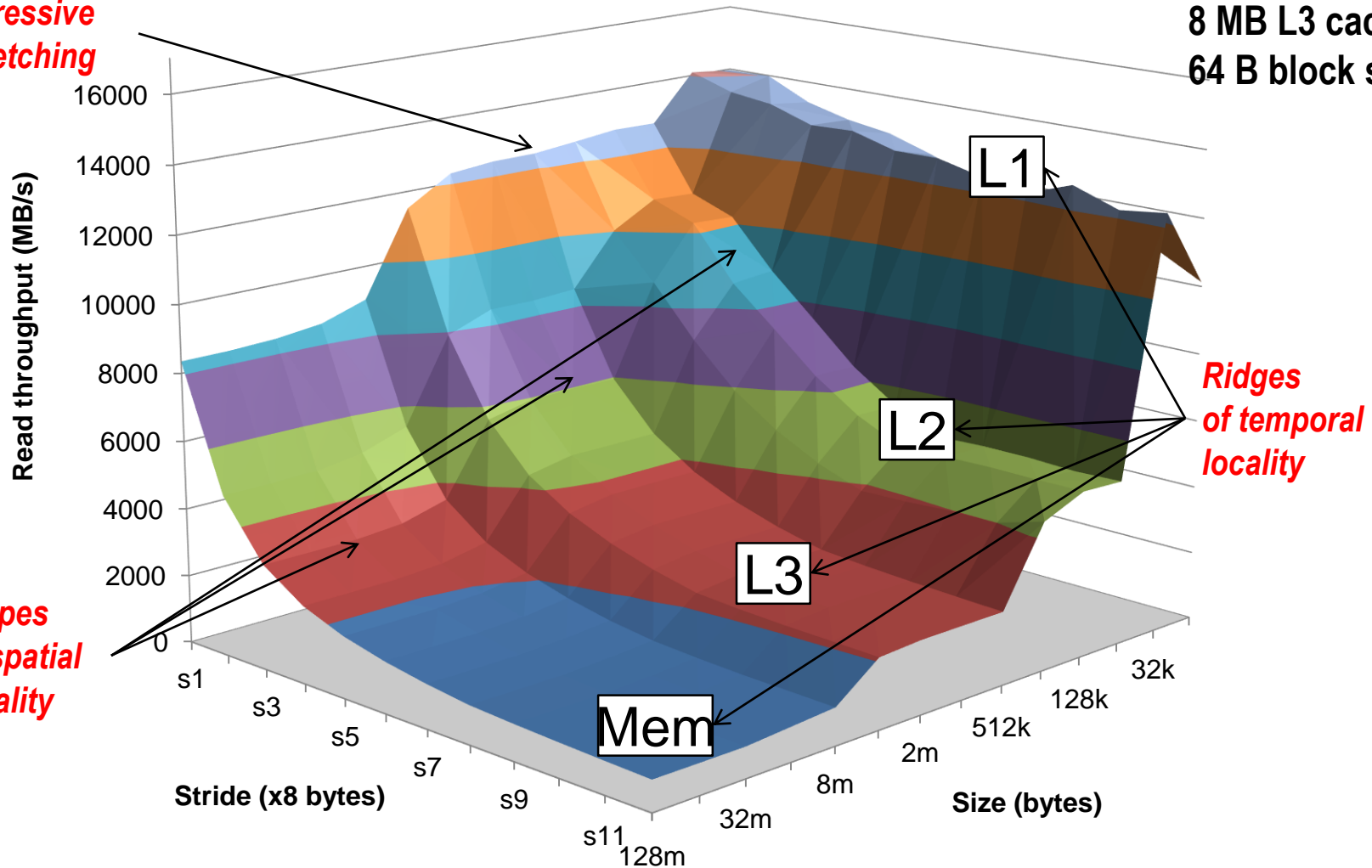
1. Call `test()` once to warm up the caches.

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

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*



Today

- Cache organization and operation
- Performance impact of caches
 - The memory mountain
 - Rearranging loops to improve spatial locality
 - Using blocking to improve temporal locality

Matrix Multiplication Example

■ Description:

- Multiply $N \times N$ 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;
    }
}
```

*Variable sum
held in register*

matmult/mm.c

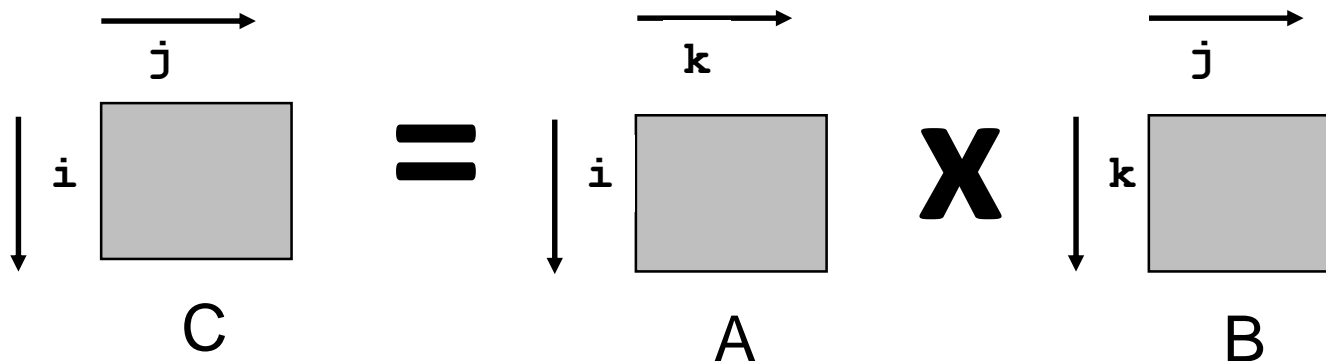
Miss Rate Analysis for Matrix Multiply

■ 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



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<sub>ij</sub>) bytes, exploit spatial locality
  - miss rate = sizeof(a<sub>ij</sub>) / B

- **Stepping through rows in one column:**

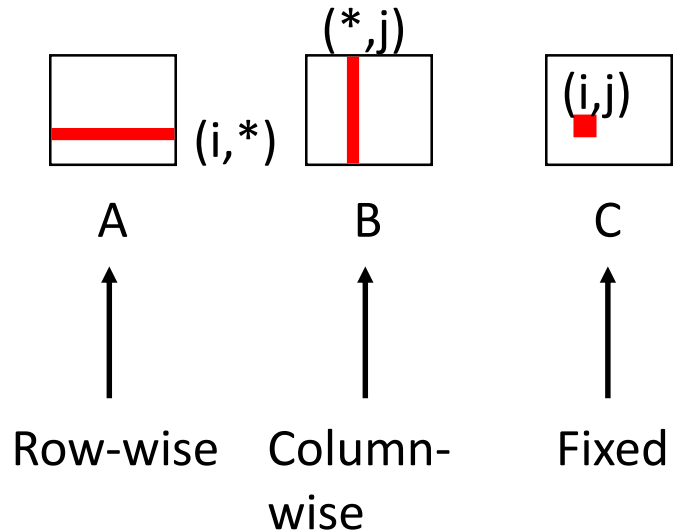
- ```
for (i = 0; i < n; i++)  
    sum += a[i][0];
```
- accesses distant elements
- no spatial locality!
 - miss rate = 1 (i.e. 100%)

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:

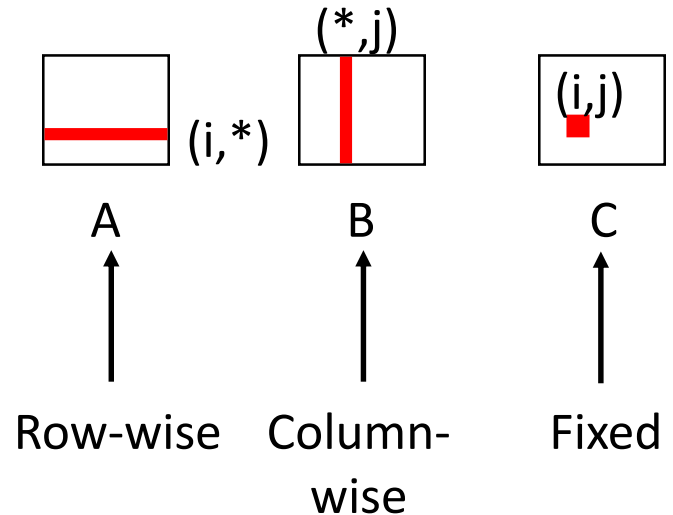
<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

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  
    }  
}
```

matmult/mm.c

Inner loop:



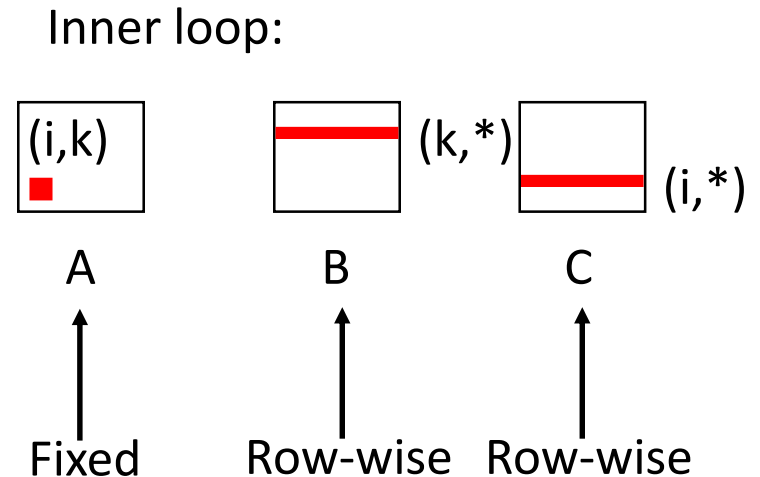
Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
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];
    }
}
```

matmult/mm.c



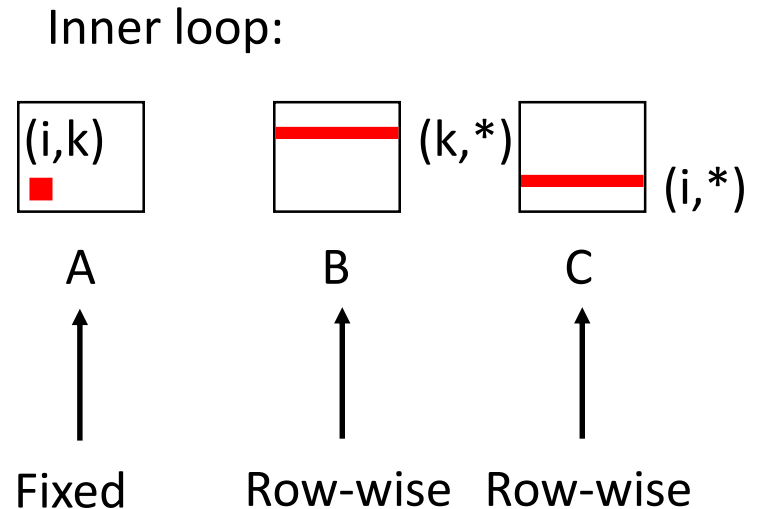
Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
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];
    }
}
```

matmult/mm.c



Misses per inner loop iteration:

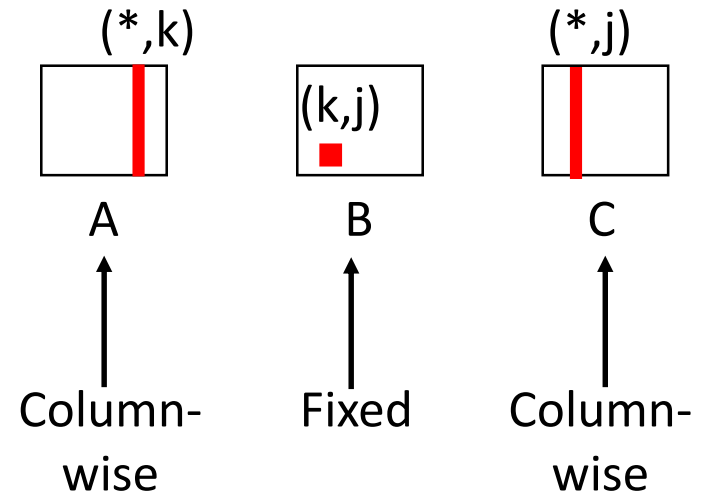
<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25

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;
    }
}
```

matmult/mm.c

Inner loop:



Misses per inner loop iteration:

A
1.0

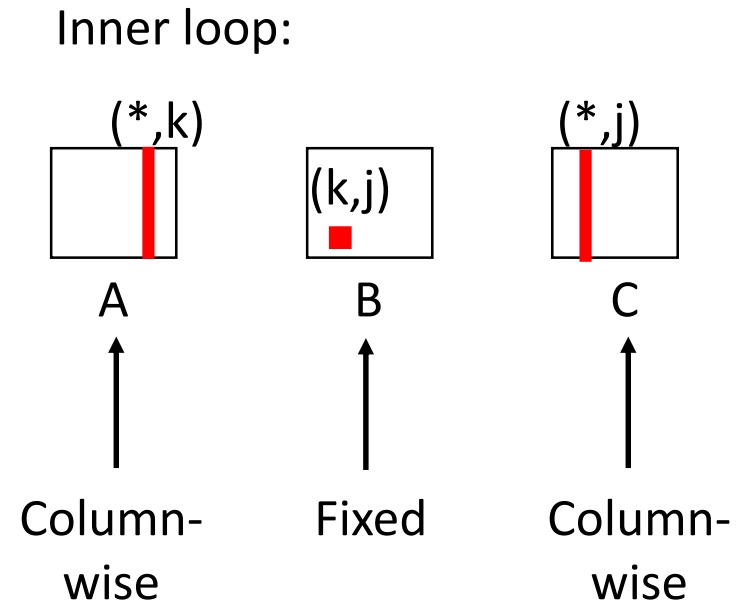
B
0.0

C
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;  
    }  
}
```

matmult/mm.c



Misses per inner loop iteration:

A
1.0

B
0.0

C
1.0

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):

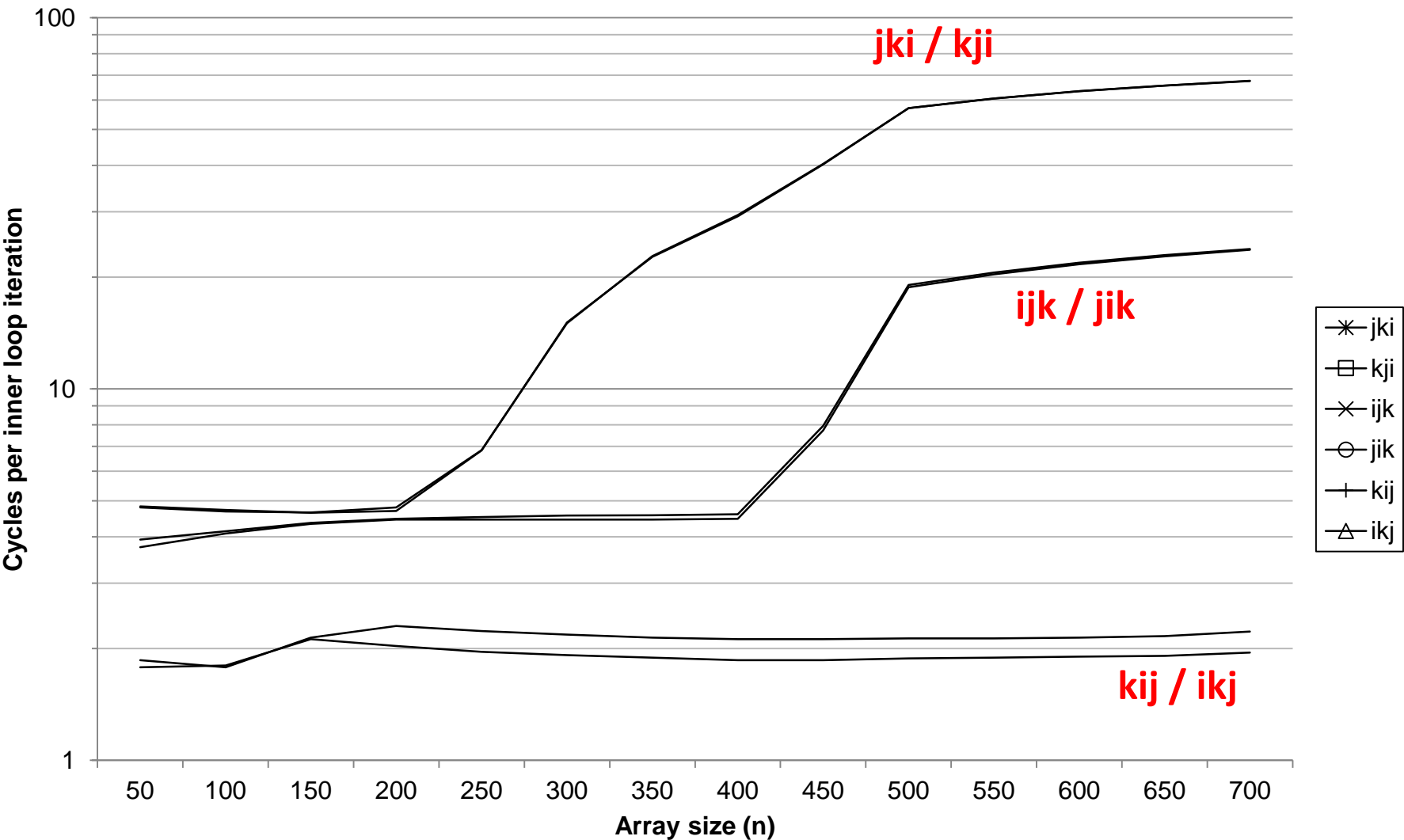
- 2 loads, 1 store
- misses/iter = **0.5**

```
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**

Core i7 Matrix Multiply Performance

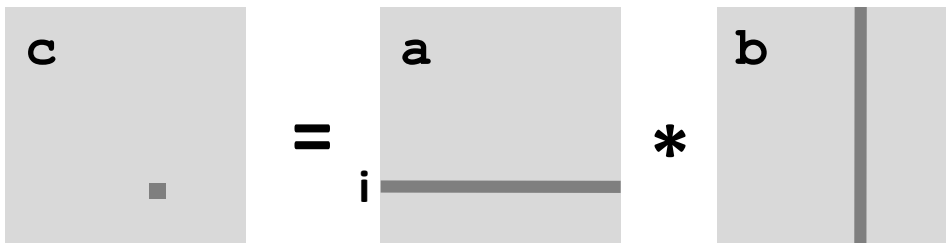


Today

- Cache organization and operation
- Performance impact of caches
 - The memory mountain
 - Rearranging loops to improve spatial locality
 - Using blocking to improve temporal locality

Example: Matrix Multiplication

```
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

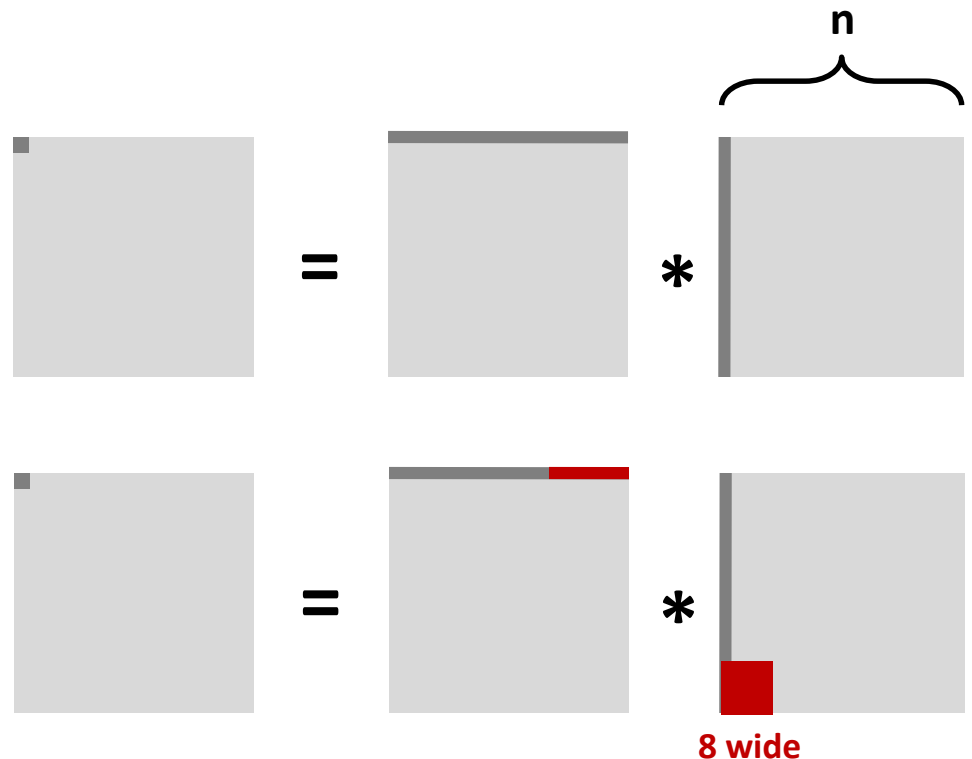
■ Assume:

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

■ First iteration:

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

- Afterwards **in cache**:
(schematic)



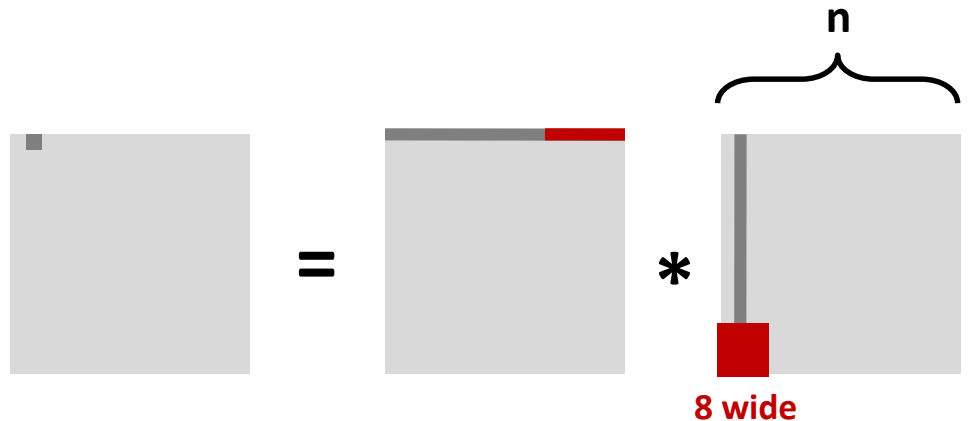
Cache Miss Analysis

■ Assume:

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

■ Second iteration:

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



■ Total misses:

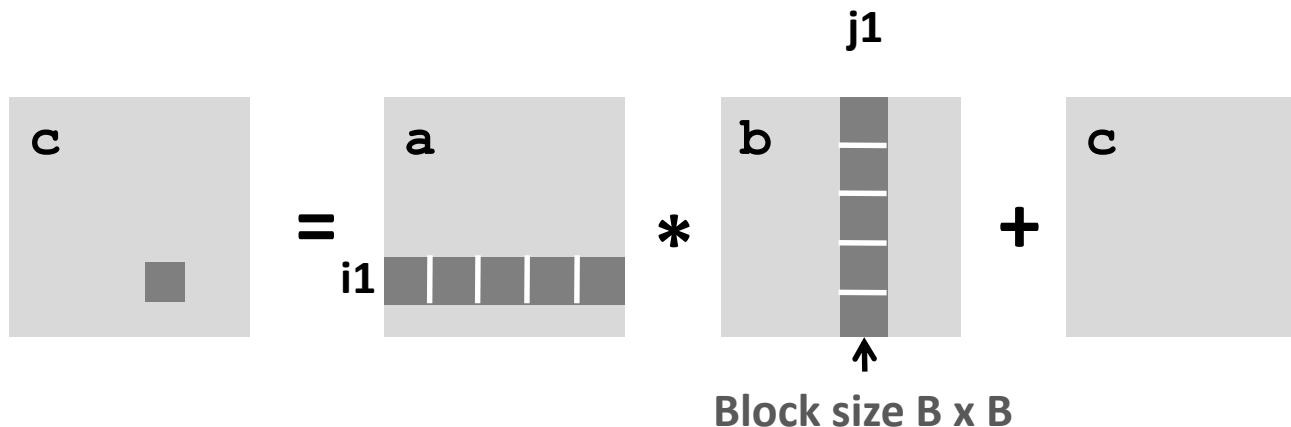
- $9n/8 * n^2 = (9/8) * n^3$

Blocked Matrix Multiplication

```
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+=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];
}
```

matmult/bmm.c



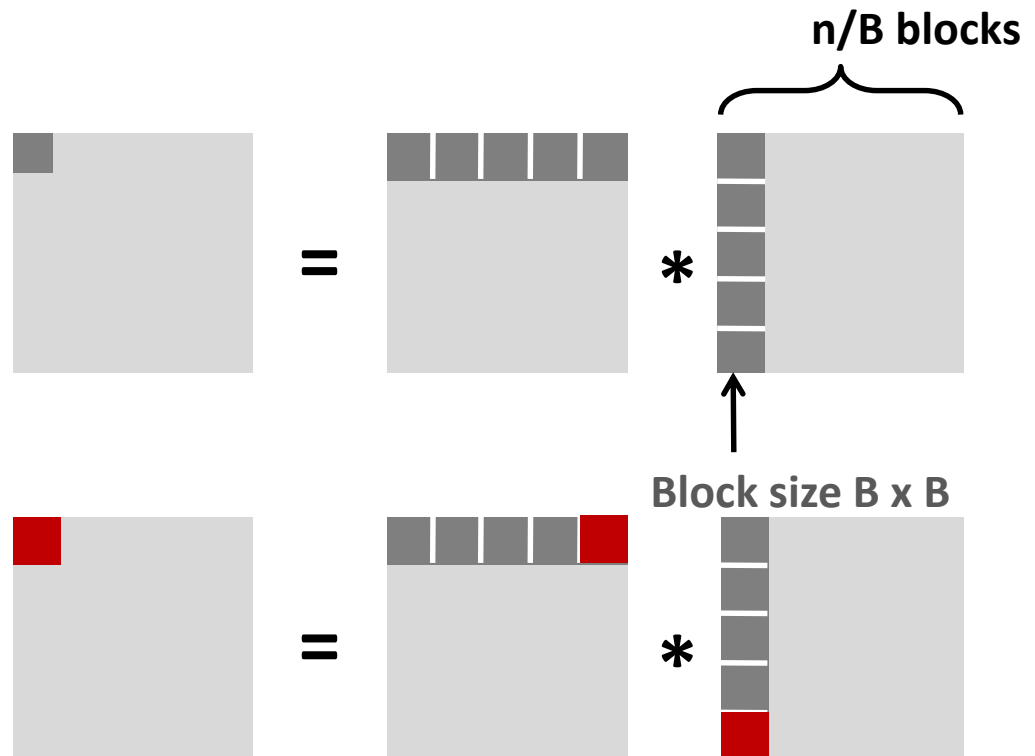
Cache Miss Analysis

■ Assume:

- Cache block = 8 doubles
- Cache size $C \ll n$ (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)



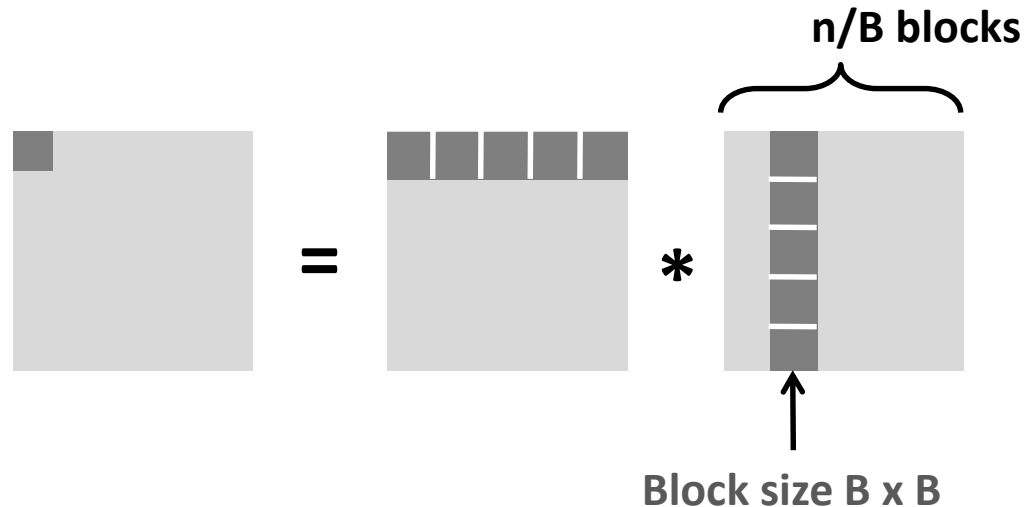
Cache Miss Analysis

■ Assume:

- Cache block = 8 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
- $2n/B * B^2/8 = nB/4$



■ Total misses:

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

Blocking Summary

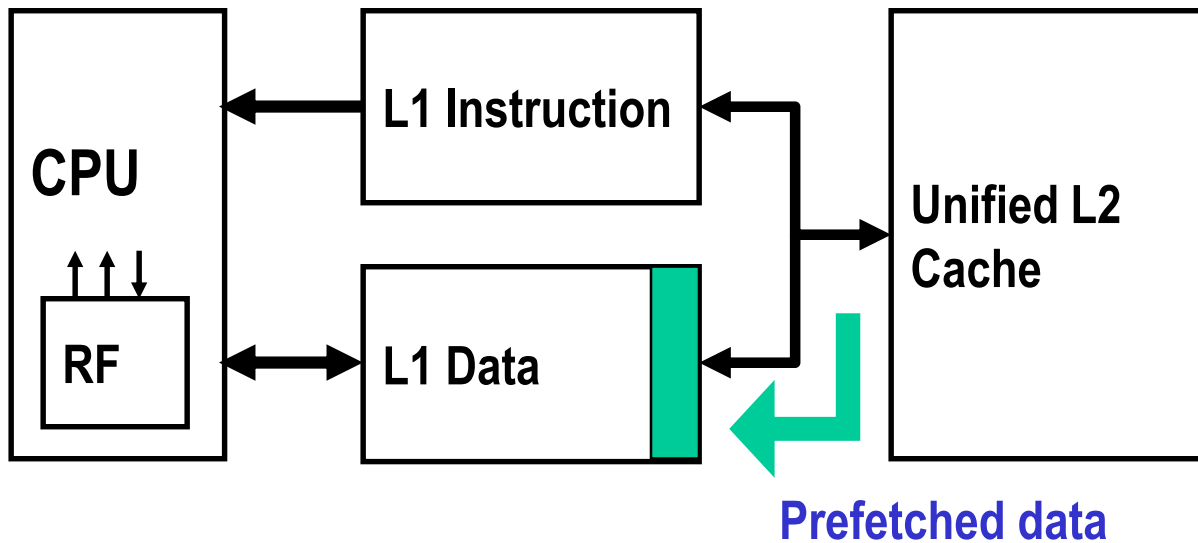
- No blocking: $(9/8) * n^3$
- Blocking: $1/(4B) * n^3$
- Suggest largest possible block size B , but limit $3B^2 < C$!
- Reason for dramatic difference:
 - Matrix multiplication has inherent temporal locality:
 - Input data: $3n^2$, computation $2n^3$
 - Every array elements used $O(n)$ times!
 - But program has to be written properly

Prefetching

- **Speculate on future instruction and data accesses and fetch them into cache(s)**
 - Instruction accesses easier to predict than data accesses
- **Varieties of prefetching**
 - Hardware prefetching
 - Software prefetching
 - Mixed schemes
- ***What types of misses does prefetching affect?***

Issues in Prefetching

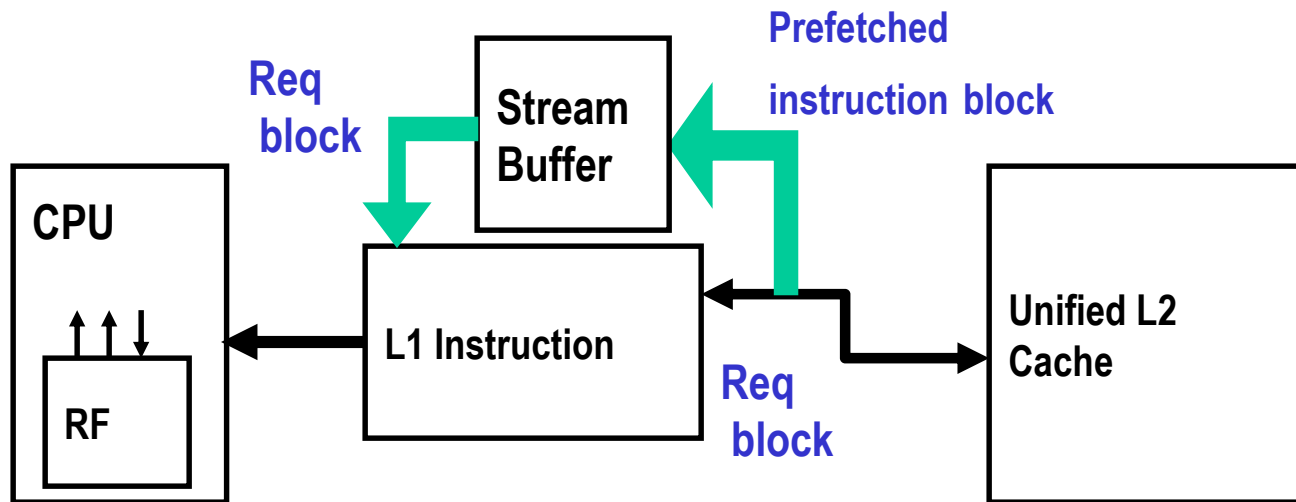
- Usefulness – should produce hits
- Timeliness – not late and not too early
- Cache and bandwidth pollution



Hardware Instruction Prefetching

Instruction prefetch in Alpha AXP 21064

- Fetch two blocks on a miss; the requested block (i) and the next consecutive block (i+1)
- Requested block placed in cache, and next block in instruction stream buffer
- If miss in cache but hit in stream buffer, move stream buffer block into cache and prefetch next block (i+2)



Hardware Data Prefetching

■ Prefetch-on-miss:

- Prefetch $b + 1$ upon miss on b

■ One Block Lookahead (OBL) scheme

- Initiate prefetch for block $b + 1$ when block b is accessed
- *Why is this different from doubling block size?*
- Can extend to N-block lookahead

■ Strided prefetch

- If observe sequence of accesses to block b , $b+N$, $b+2N$, then prefetch $b+3N$ etc.

Example: IBM Power 5 [2003] supports eight independent streams of strided prefetch per processor, prefetching 12 lines ahead of current access

Software Prefetching

```
for (i=0; i < N; i++) {  
    prefetch( &a[i + 1] );  
    prefetch( &b[i + 1] );  
    SUM = SUM + a[i] * b[i];  
}
```

Software Prefetching Issues

■ Timing is the biggest issue, not predictability

- If you prefetch very close to when the data is required, you might be too late
- Prefetch too early, cause pollution
- Estimate how long it will take for the data to come into L1, so we can set P appropriately
- *Why is this hard to do?*

```
for(i=0; i < N; i++) {  
    prefetch( &a[i + P] );  
    prefetch( &b[i + P] );  
    SUM = SUM + a[i] * b[i];  
}
```

Must consider cost of prefetch instructions

Cache Summary

- **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.

Acknowledgements

- **These slides contain material developed and copyright by:**
 - Arvind (MIT)
 - Krste Asanovic (MIT/UCB)
 - Joel Emer (Intel/MIT)
 - James Hoe (CMU)
 - John Kubiatowicz (UCB)
 - David Patterson (UCB)
- **MIT material derived from course 6.823**
- **UCB material derived from course CS252**