UNM SCHOOL of ENGINEERING
Department of Computer Science

Lecture #20
Ch. 6 The Memory Hierarchy (end)

Prof. Soraya Abad-Mota, PhD

Last topics in the Memory Hierarchy

- Chapter 6: The Memory Hierarchy (end)
  - Cache memory organization and operation (Sec. 6.4)
    - Writes
    - Cache metrics
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality: matrix multiplication example
  - Using blocking to improve temporal locality (on your own, good for the cachelab project)
- Chapter 7: Linking (short)

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

Variable `sum` *held 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;
  }
}
```

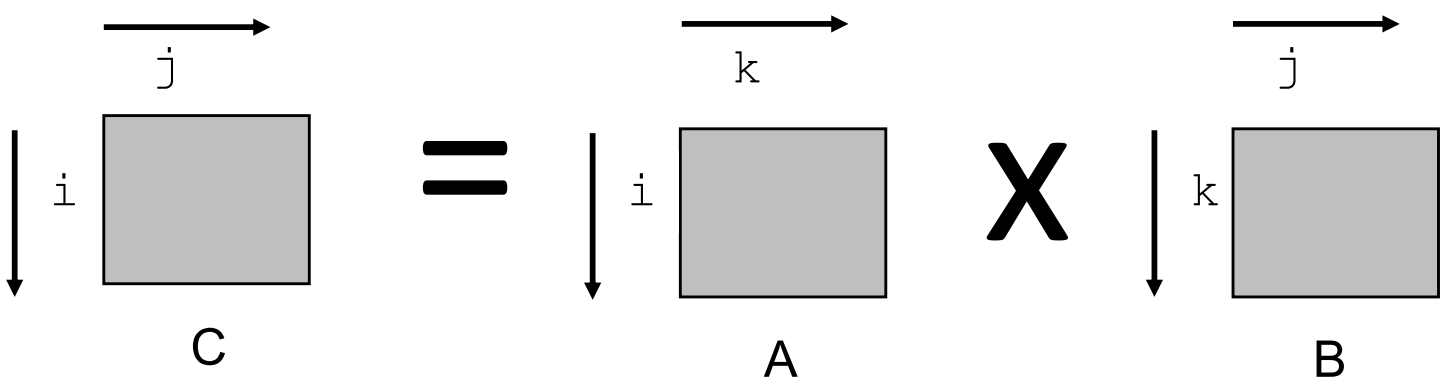
matmult/mm.c

Assumptions of page 644

what is the consequence of having `sum` in a register?

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: (on 6 versions of matrix mult.)
  - Look at access pattern of inner loop



Layout of C Arrays in Memory (review)

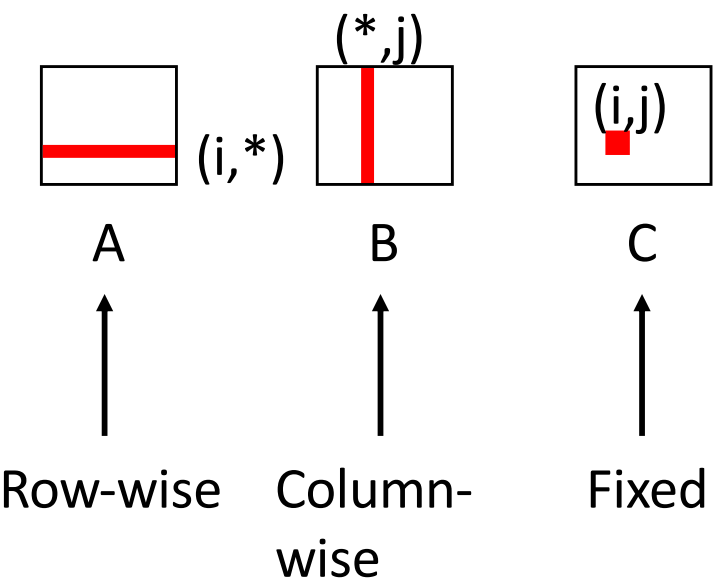
- C arrays allocated in row-major order
  - each row in contiguous memory locations
- Stepping through columns in one row:
  - ```
    for (i = 0; i < N; i++)
    sum += a[0][i];
    ```
  - accesses successive elements
  - if block size (B) > sizeof(a_{ij}) bytes, exploit spatial locality
    - miss rate = sizeof(a_{ij}) / B (our e.g. uses doubles ➔ 8/32 = 0.25)
- 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) version a (class AB)

```
/* 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: (1-stride to access A, 8/32=0.25)

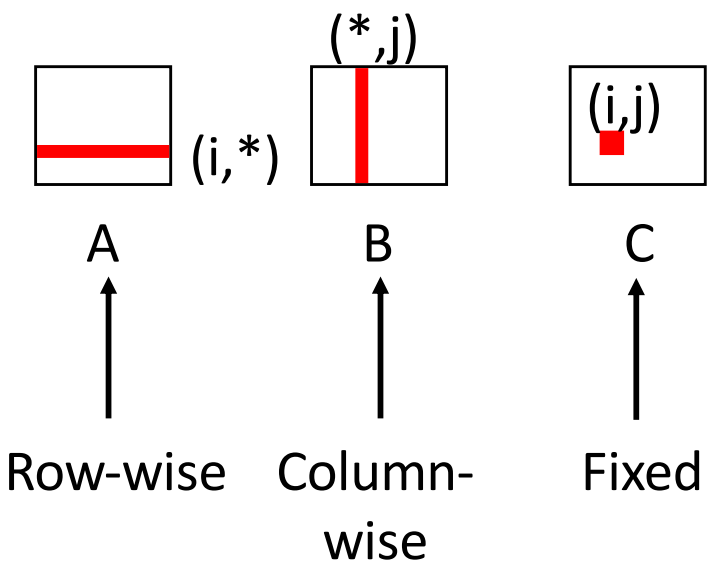
A	B	C
0.25	1.0	0.0

Matrix Multiplication (jik) version b (class AB)

```
/* 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:

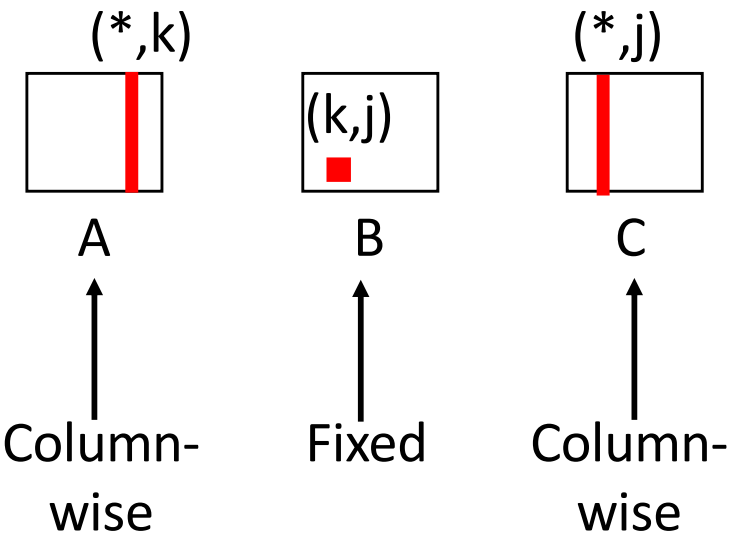
A	B	C
0.25	1.0	0.0

Matrix Multiplication (jki) version c (class AC)

```
/* 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:

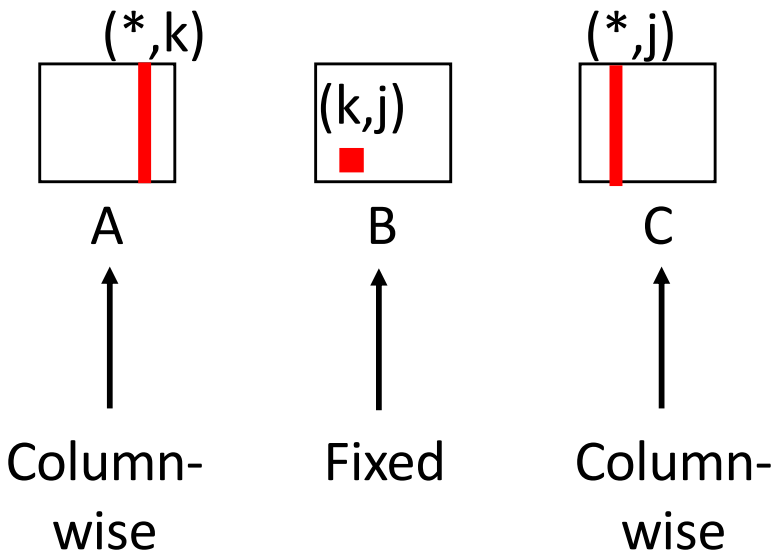
A	B	C
1.0	0.0	1.0

Matrix Multiplication (kji) version d (class AC)

```
/* 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:

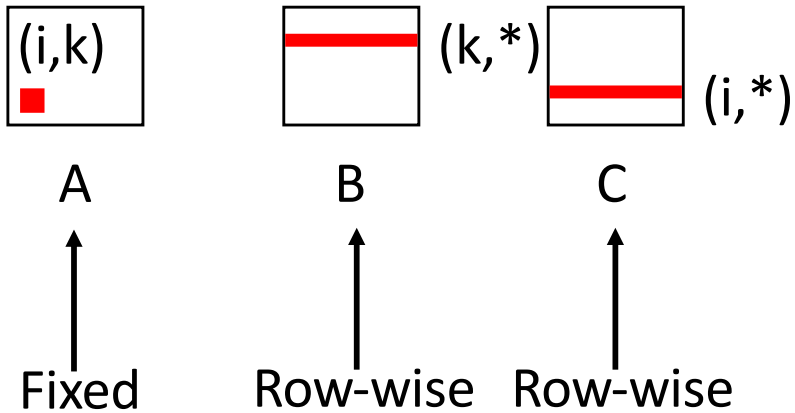
A	B	C
1.0	0.0	1.0

Matrix Multiplication (kij) version e (class BC)

```
/* 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:

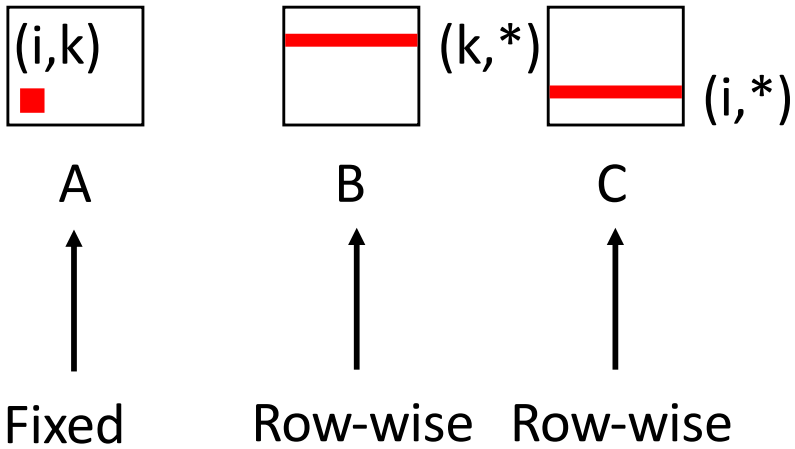
A	B	C
0.0	0.25	0.25

Matrix Multiplication (ikj) version f (class BC)

```
/* 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

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

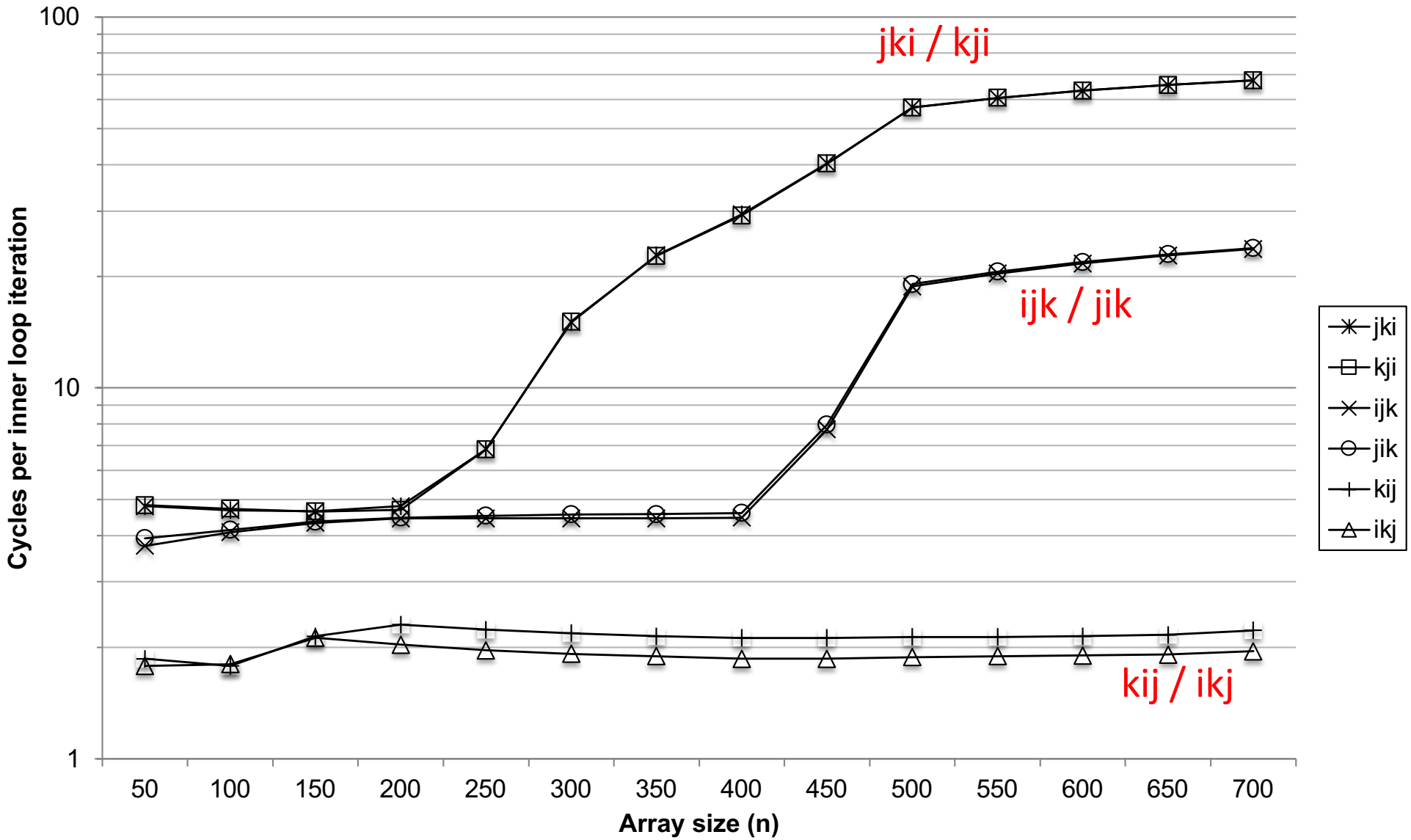
```
for (j=0; j<n; j++) {
 for (k=0; k<n; k++) {
   r = b[k][j];
   for (i=0; i<n; i++)
    c[i][j] += a[i][k] * r;
 }
}
```

jki (& kji):

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

Core i7 Matrix Multiply Performance

(analysis on pp. 646-647, please read)



Exploiting locality in your programs

- Main principle:

  "Programs with good locality access most of their data from fast cache memories. Programs with poor locality access most of their data from the relatively slow DRAM main memory." p.647

Recommendations

1. Focus the attention on the inner loops because they have the bulk of the computation and memory access.
2. Maximize spatial locality ➔ reading data objects sequentially, with stride 1, in the order they are stored.
3. Maximize temporal locality by using a data object as often as possible once it has been read from memory.

Conclusions about caches

Have seen:

- Feasible to transfer data between devices which are very different in speed and capacity avoiding the wait-for-the-slower-device syndrome.

Evaluated the approach experimentally ➔ memory mountain (quantitative measures)

- Performance of the caches from three dimensions:
  - read throughput (a measure)
  - analysis of a technique to improve access to multidim. arrays
  - blocking: another technique to improve this access (next 8 slides)

Last topics in the memory hierarchy

- Performance impact of caches
  - 6.6.1 The Memory Mountain
  - 6.6.2 Rearranging loops to improve spatial locality
    - Using blocking to improve temporal locality:

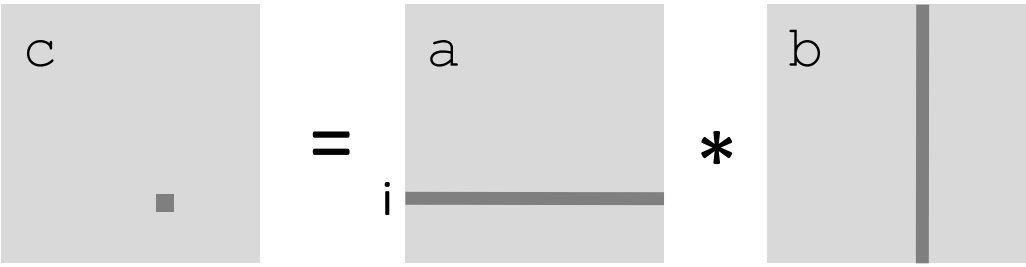
    Seven (7) slides here describe an example of how to use "blocking" to improve the performance of matrix multiplication by increasing temporal locality.

    (Study on your own, useful for the Cachelab Part B project.)

Go to slide 167 to conclude this chapter and see what's next.

Example: Matrix Multiplication

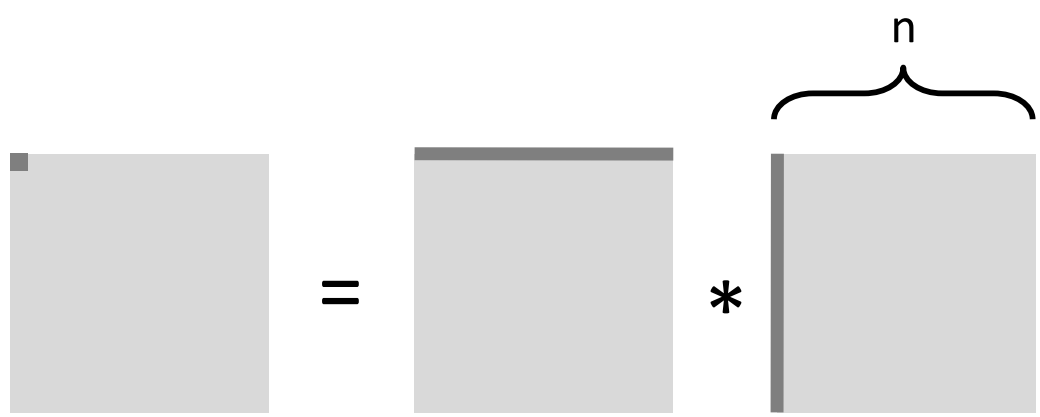
```
c = (double *) calloc(sizeof(double), n*n);

/* 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 << n (much smaller than n)

- First iteration:
  - n/8 + n = 9n/8 misses

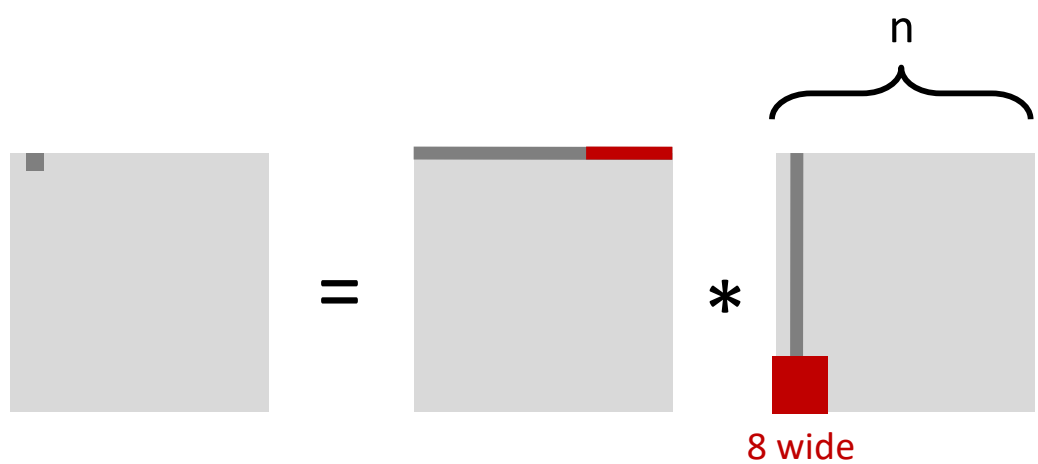


  - Afterwards in cache: (schematic)

Cache Miss Analysis

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

- Second iteration:
  - Again:
    n/8 + n = 9n/8 misses

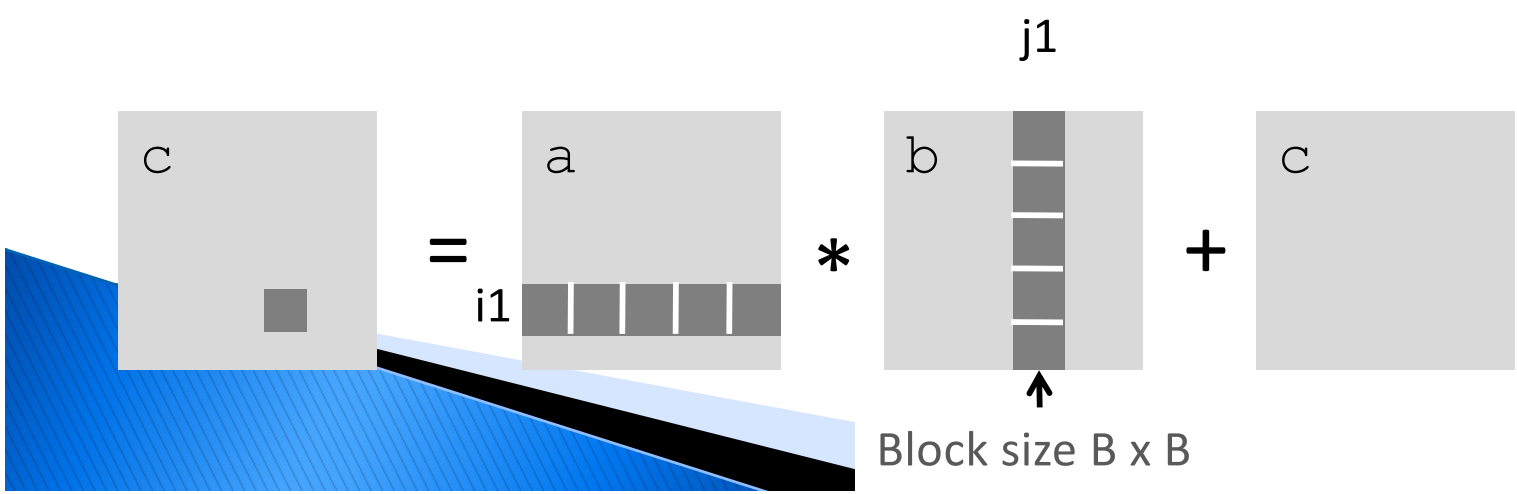


- Total misses:
  - $9n/8 * n^2 = (9/8) * n^3$

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];
}
```

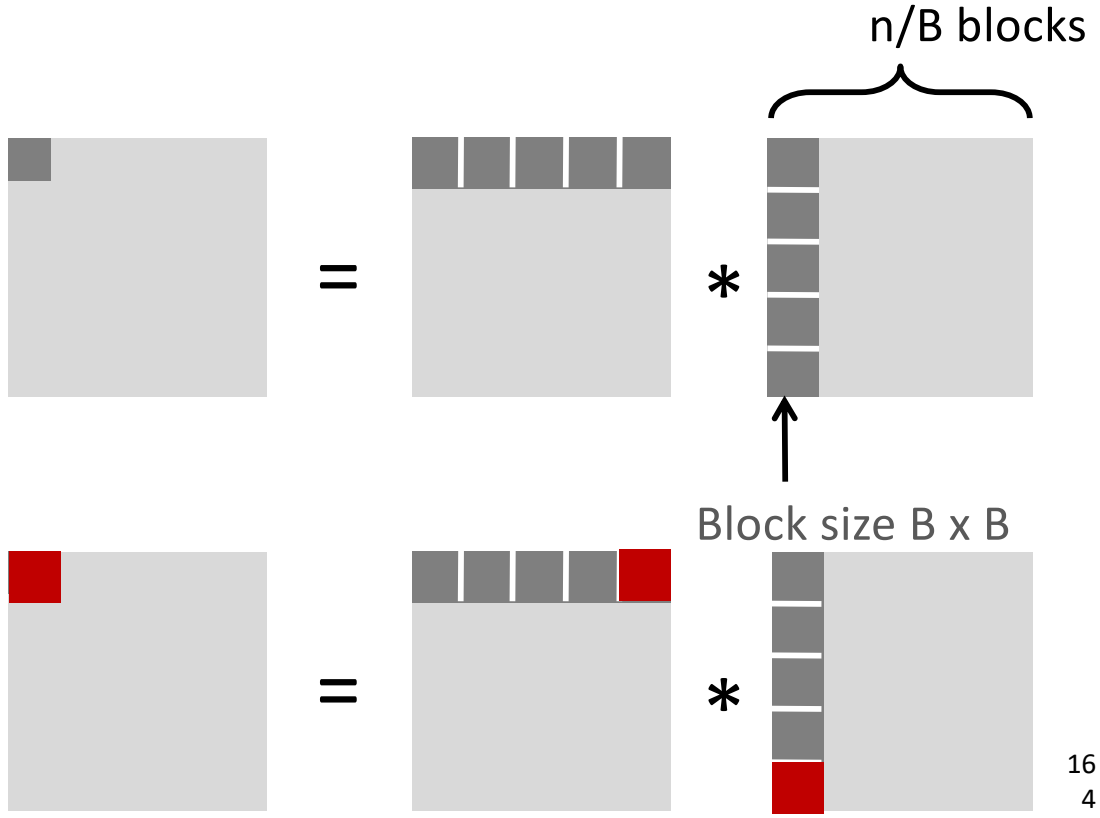
matmult/bmm.c



Cache Miss Analysis

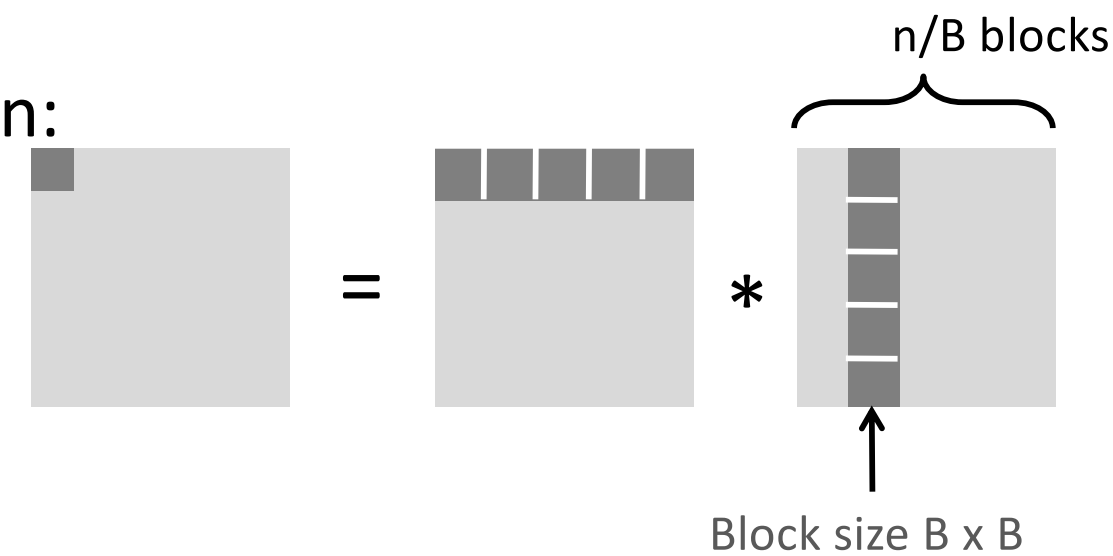
- Assume:
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks fit into cache: $3B^2 < C$

- First (block) iteration:
  - $B^2/8$ misses for each block
  - $2n/B * B^2/8 = nB/4$ (omitting matrix c)
  - Afterwards in cache (schematic)



Cache Miss Analysis

- Assume:
  - Cache block = 8 doubles
  - Cache size C << 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 (and it is more complicated to understand)

End of Ch. 6 - Memory Hierarchy

- This was the end of this chapter, main theme was:
  - bridging the gap between main memory and the CPU
- What follows? (a parenthesis)
  - (Ch. 7 - Linking (important to know how is it that programs may end up being executed in the CPU with more than one source + libraries all together in the "same executable file")
  - Ch. 8 - Exceptional Control Flow (ECF)
    - flow of control of the processor
    - need to understand the *exceptions* to this flow and how they work
    - to interesting applications, OS concepts, concurrency)
- Ch. 9 - Virtual Memory (HW-mainMemory-diskFiles-OS)

Go to linking pptx file

- Lecture #21 - Ch. 7 Linking