Sequential performance is important

- Per-node performance is important.
- Cache and prefetch effects are an important way to gather per-node performance.

Loop unrolling

```
do i = 1, n
   a[i] = b[i] + c[i]
end do
```

- Processors typically have several add and store units
- Without additional hardware, processors cannot move operations around conditional branches
- Thus, in this loop, only one add and one store can be started at a time

```
do i = 1, n, 3
   a[i] = b[i] + c[i]
   a[i+1] = b[i+1] + c[i+1]
   a[i+2] = b[i+2] + c[i+2]
end do

do i = n-((n-1) mod 3), n
   a[i] = b[i] + c[i]
end do
```

- In the right hand version, three floating point adds can be issued at once
- Blue fixup code necessary because n not always divisible by the unroll factor

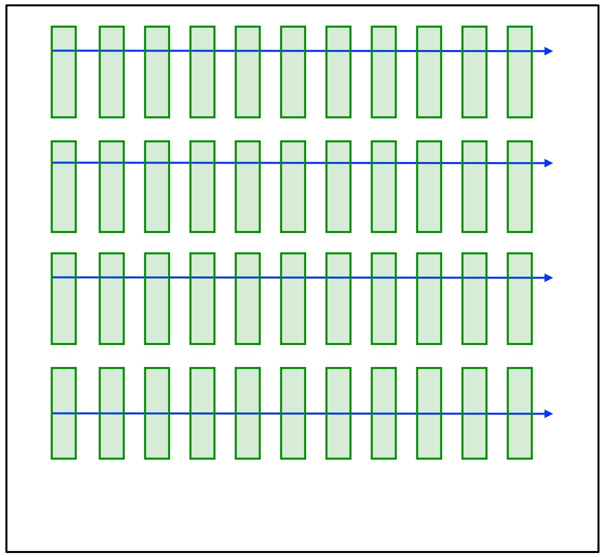
Cache optimizations

- Desire is to maximizes locality and to exploit temporal and spatial locality via the cache.
- Accesses from cache are tens to hundreds of times faster than accesses from memory, and typically the same speed or 2 times slower than accesses from registers
- The more regular the code, the more compilers can do in terms of cache optimizations

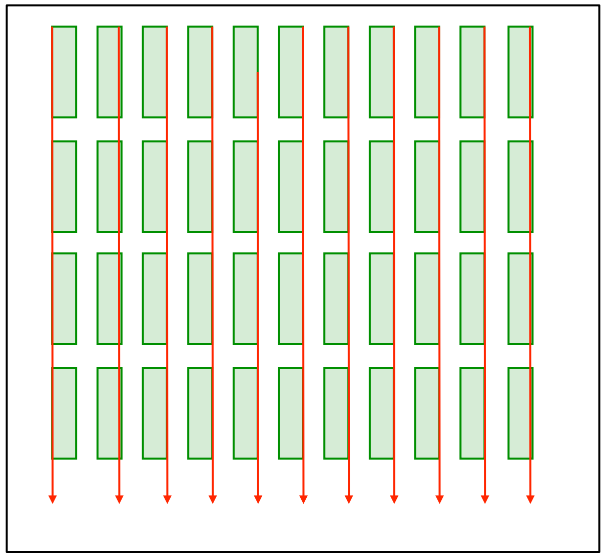
Cache optimizations

- Loop interchange attempts to make accesses coincide with the way data is laid out in memory

```
do i = 1, n
  do j = 1, n
    a[i,j] = …
  end do
end do
```



```
do j = 1, n
  do i = 1, n
    a[i,j] = …
  end do
end do
```



A[col,row], array stored in column major order (row in C)

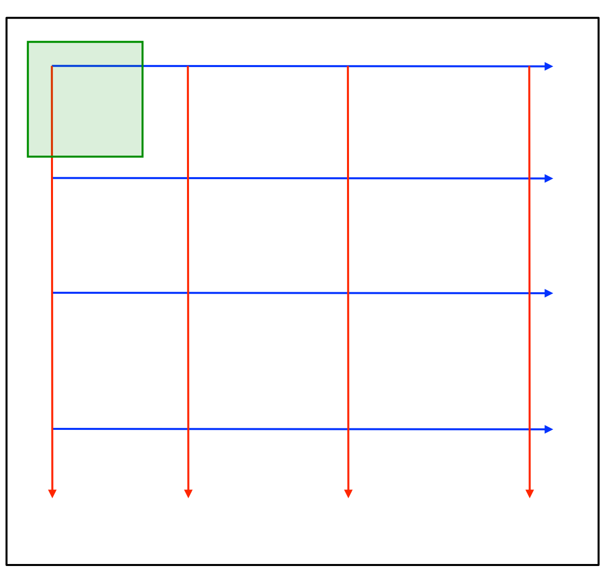
Interchange the loops, and data within a text box is reused.

Loop tiling (page 347, Bacon, Eggers paper on the class web page)

```
Do i = 1, n
   Do j = 1, n
      a[i,j] = b[j,i]
   end do
end do
```

a,b[col,row], array stored in row major order

```
do TI = 1, n, 64
  do TJ = 1, n, 64
    do i = TI, min(TI+63,n)
      do j = TJ, min(TJ+63, n)
        a[i,j] = b[j,i]
      end do
    …
end do
```



What fits in cache

b[i,j] accesses

a[I,j] accesses

This pattern is present in matrix multiply, transpose, etc.

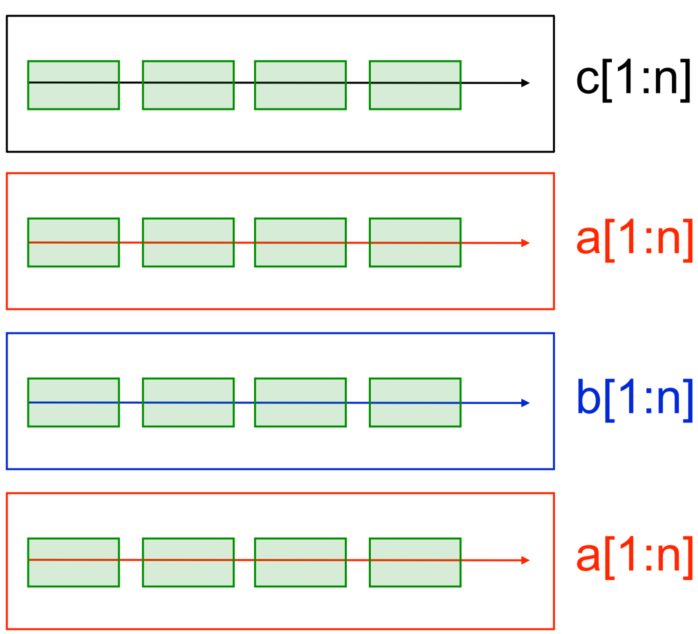
Cache optimizations

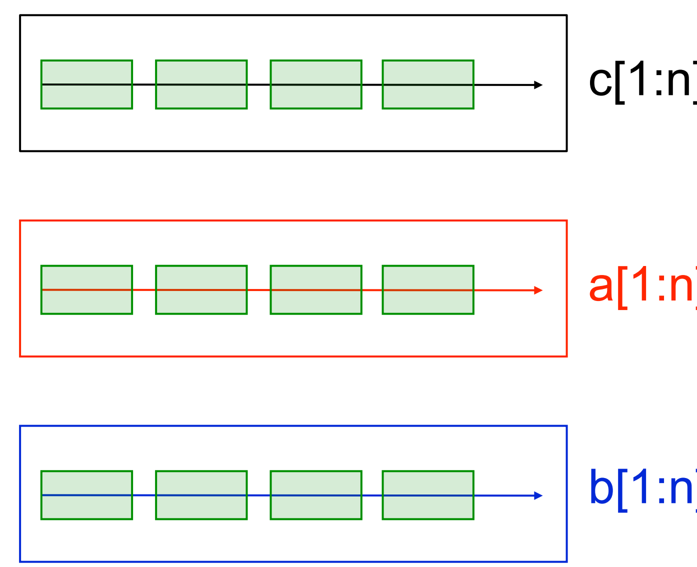
- Loop fusion increases temporal locality

```
do I = 1, n
  c[i] = a[i]
end do
do I = 1, n
  b[i] = a[i]
end do
```

If n is large enough that all of c[1:n], a[1:n] and b[1:n] won’t fit in cache, some of a[1:n] fetched in the first loop will be evicted by the time we need it in the second loop, and need to be refetched.

```
do I = 1, n
  c[i] = a[i]
  b[i] = a[i]
end do
```





Cache optimizations

- Loop fusion, like all transformations that change the access order of storage, need to be checked for legality

```
do i = 1, n
  c[i] = a[i]
end do
do i = 1, n
  b[i] = c[i+1]
end do
```

For example, iteration 4 of the second loop reads c[5], which is not written until iteration 5 of the first loop. The fused loop (on the right) reads stale values for c.

```
do i = 1, n
  c[i] = a[i]
  b[i] = c[i+1]
end do
```

By shifting the iteration spaces we get a correct execution, but the transformation gets more complicated.

```
c[1] = a[1]
do i = 2, n
  c[i] = a[i]
  b[i-1] = c[i]
end do
b[n] = c[n+1]
```

Cache optimizations

- Prefetching attempts to get values in cache before they are used

```
do i = 1, n
  do j = 1, n
    a[i,j] = b[i,j]
  end do
end do
```

```
load a[1,1], b[1,1]
…
do TI = 1, n, 64
  do TJ = 1, n, 64
    load a[TI,TJ], b[TI,TJ]
    do i = TI, min(TI+63,n)
      do j = TJ, min(TJ+63, n)
        a[i,j] = b[j,i]
      end do
    …
end do
```

Problems with prefetching

- Need to fetch early enough to make a difference
- What about page faults?
- What if the value is changed between prefetch and use?
- What if we ***evict*** a needed value from the cache?

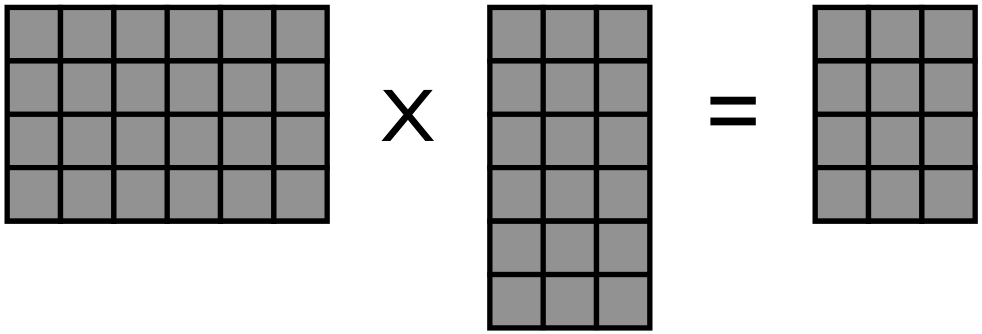
Matrix multiplication and per-node performance

- From Michael J. Quinn, *Parallel Programming in C with MPI and OpenMP*

McGraw Hill

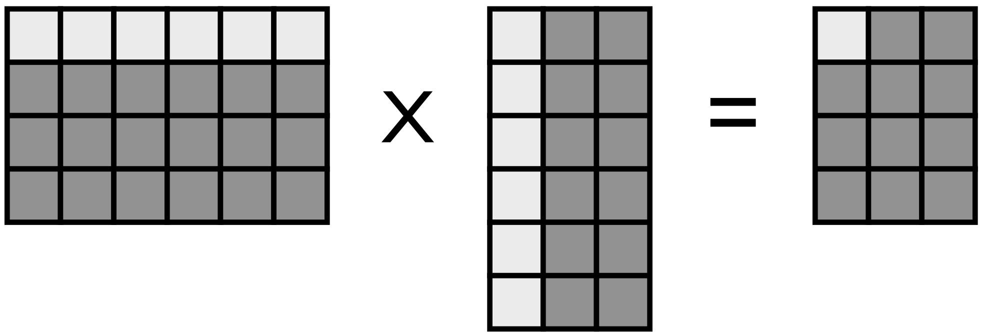
Iterative, Row-oriented Algorithm

Series of inner product (dot product) operations



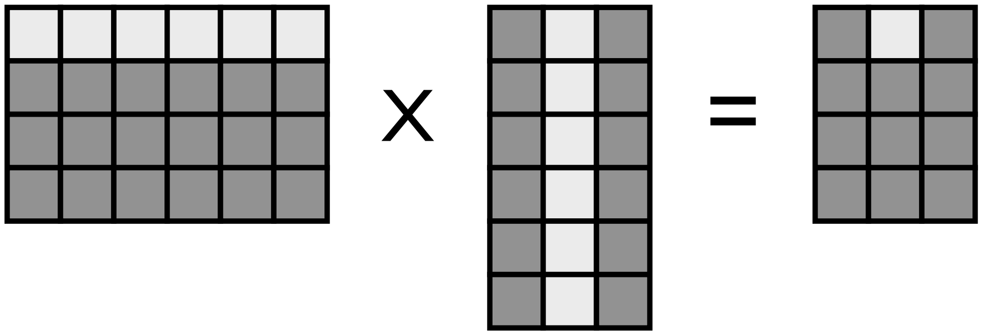
Iterative, Row-oriented Algorithm

Series of inner product (dot product) operations



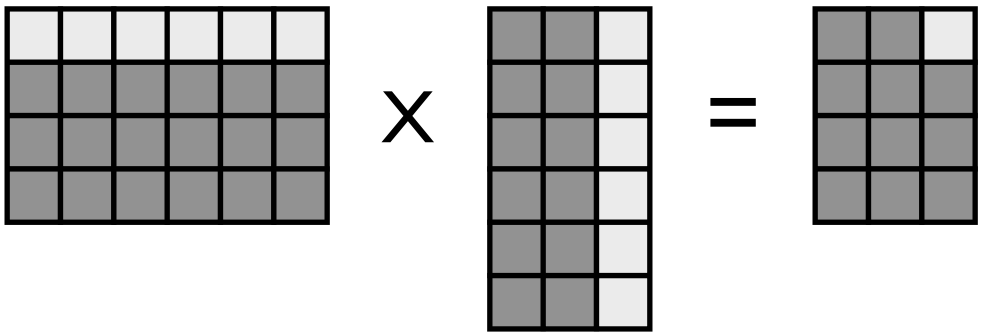
Iterative, Row-oriented Algorithm

Series of inner product (dot product) operations



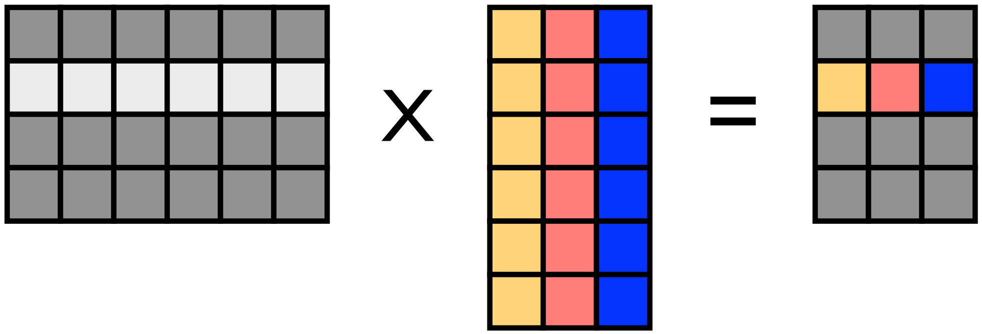
Iterative, Row-oriented Algorithm

Series of inner product (dot product) operations



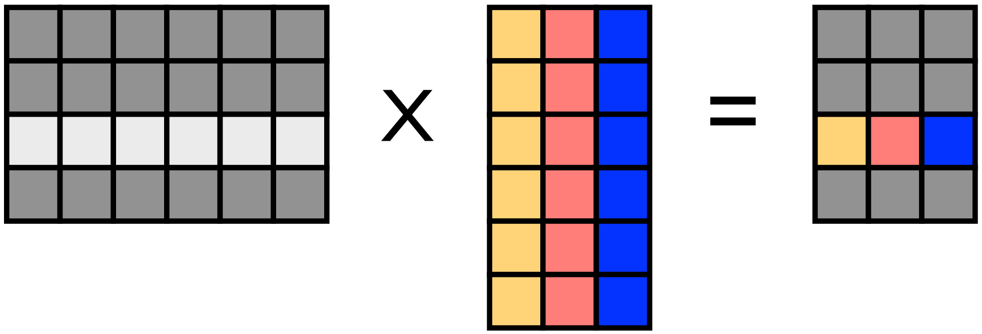
Iterative, Row-oriented Algorithm

Series of inner product (dot product) operations



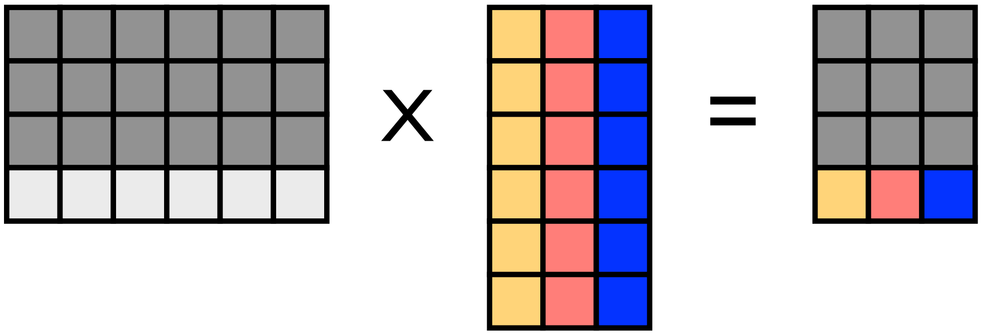
Iterative, Row-oriented Algorithm

Series of inner product (dot product) operations

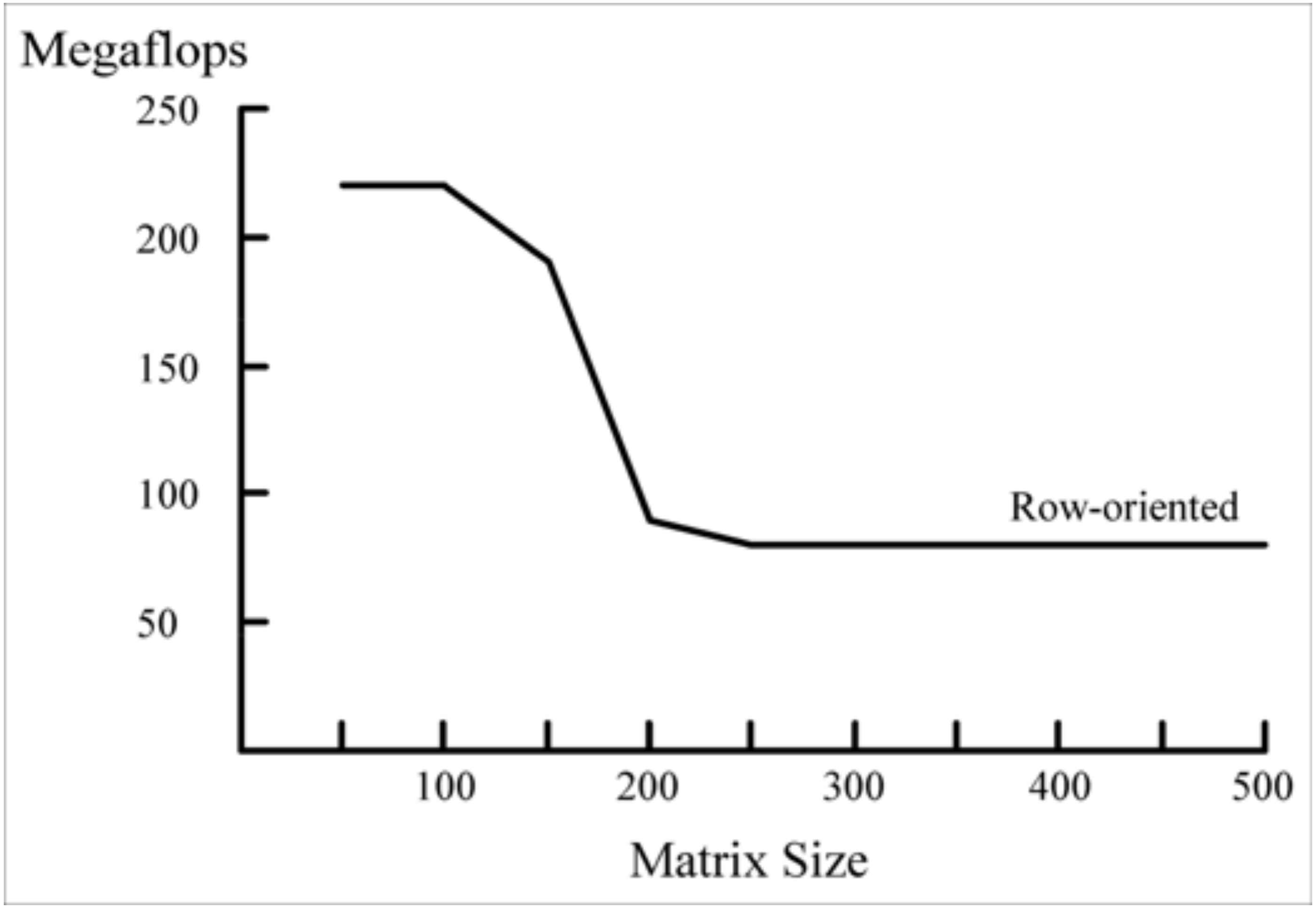


Iterative, Row-oriented Algorithm

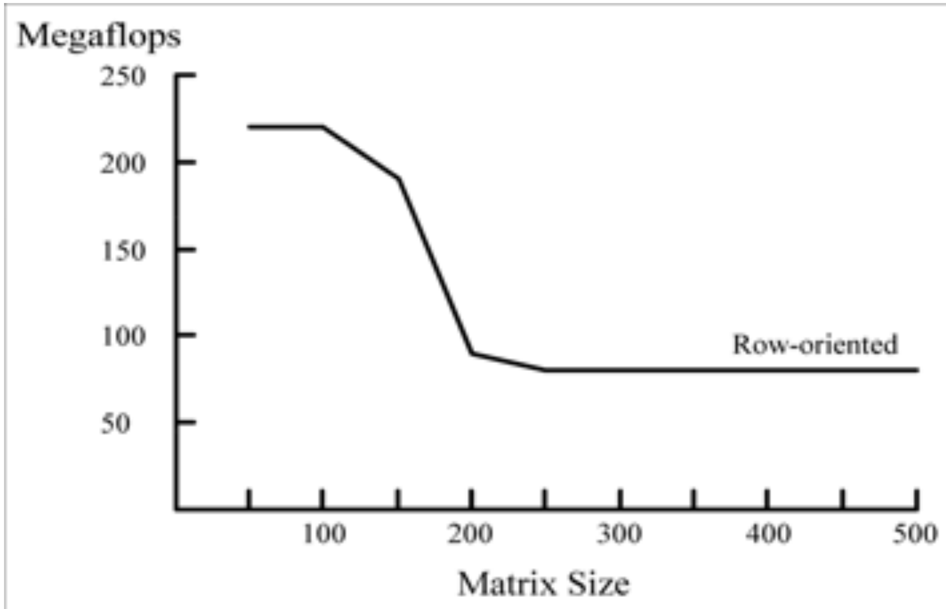
Series of inner product (dot product) operations



Performance as n Increases



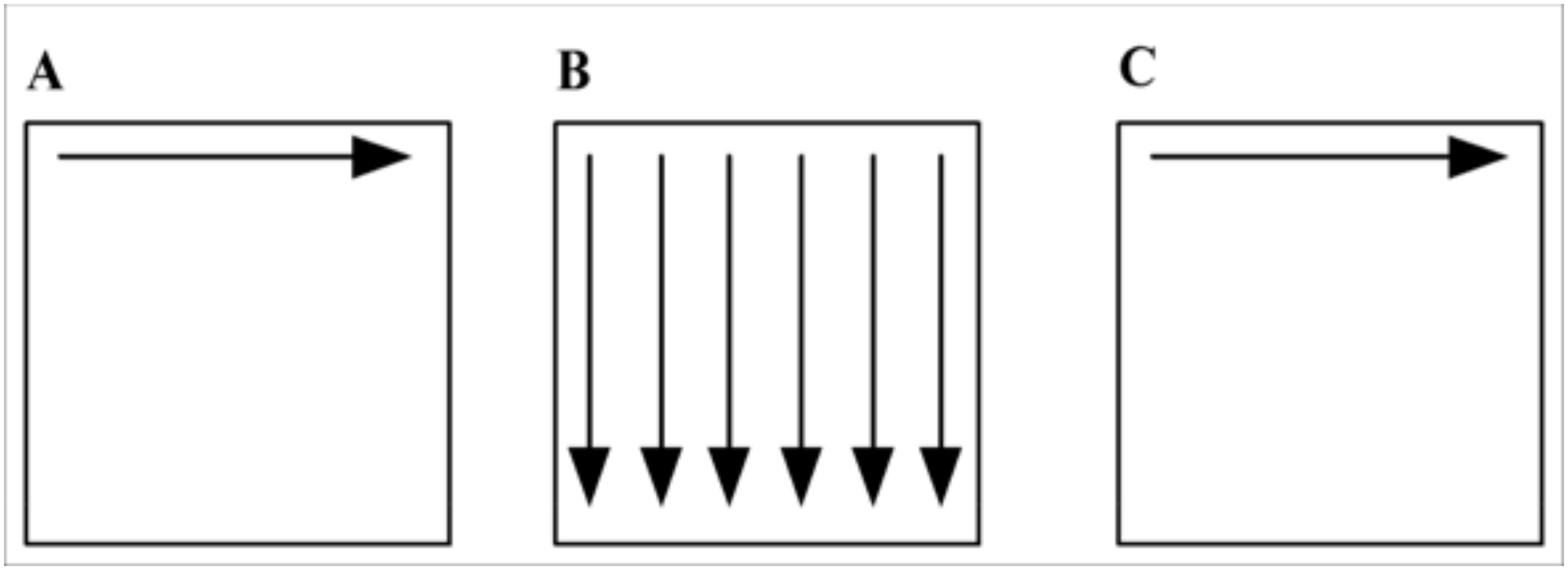
Small matrix issues



Note that even this is non-trivial to achieve. Well-tuned matrix multiplies often have a great deal of code because of *blocking*, etc. and the overhead of traversing this code can lead to slow-downs on smaller matrix multiplies.

A good matrix multiply library routine needs to be good from small (4x4) to big matrix multiplies. DGEMM is 388 lines of code

Reason for bad large matrix performance: Matrix B Gets Too Big for Cache



Computing a row of C requires accessing every element of B

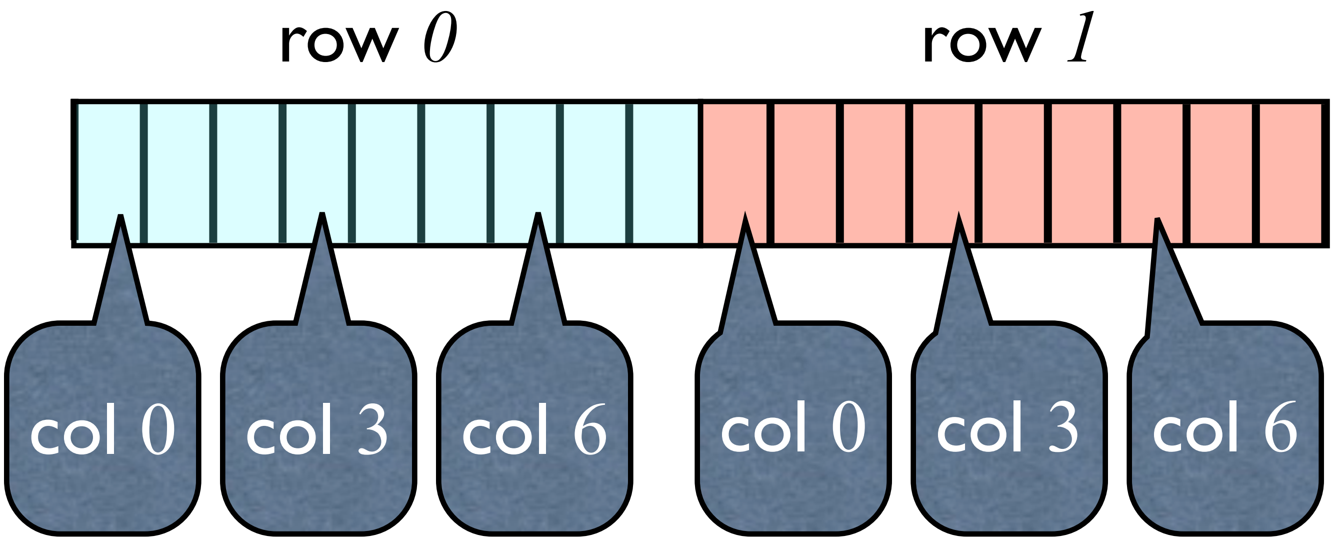
Really two problems

1. Arrays are too large to fit into cache
2. Layout of arrays combined with access order is not cache friendly

Fixing either of these would solve the problem

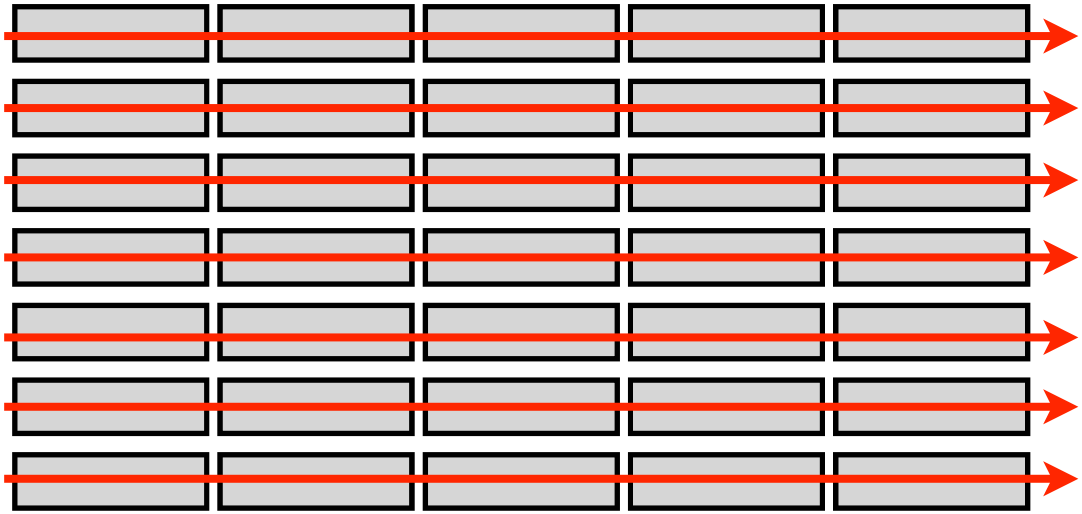
Row order layout with C

Arrays laid out in memory with row *0* first, then row *1*, then row, . . ., then row *n-1*



Row order layout with C

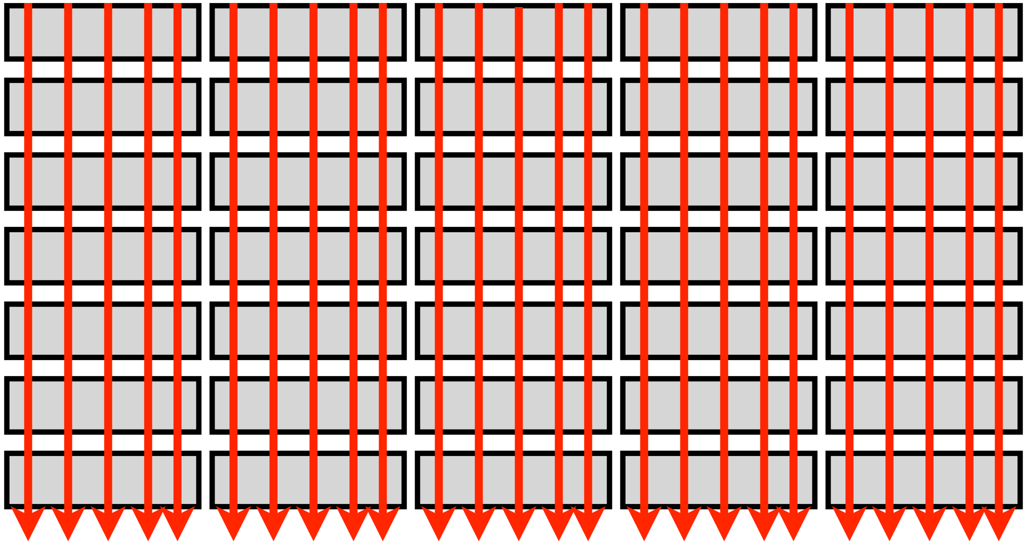
Accessing rows in order stored in memory leads to cache friendly behavior



Each cache line contains several array elements

Cache line is accessed repeatedly after being brought into the cache. This corresponds to the "A" access in the previous slide

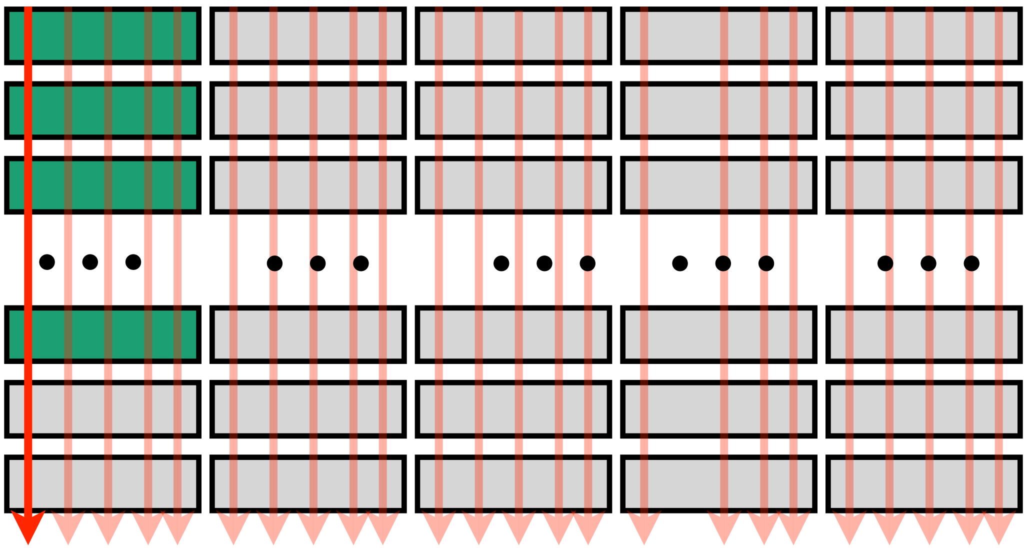
Accessing columns with row order layout



Accessing data against order stored in memory can lead to cache-unfriendly access.

Let the cache line hold n array elements, and let c be the length of a column. Then $n*c > size\ of\ the\ cache$ will result in cache unfriendly accesses

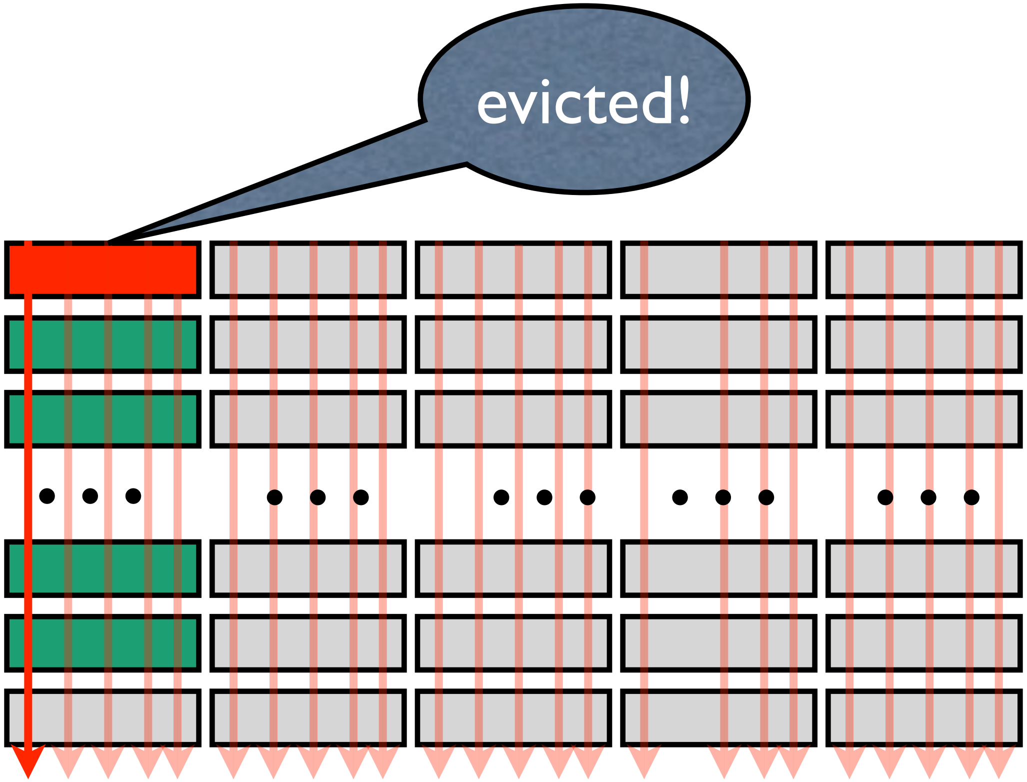
Things are initially ok



Let the green cache lines be lines that have been placed into cache

Let *#lines = (size of cache) / line size*. Then the *#lines+1* access will likely cause a cache line to be evicted.

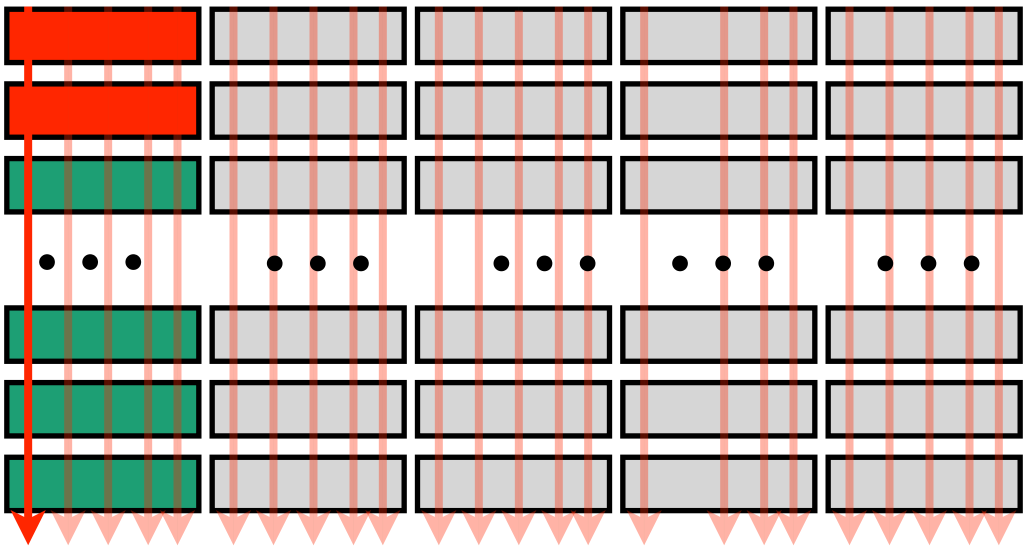
But lines will be evicted



Let the green cache lines be lines that have been placed into cache

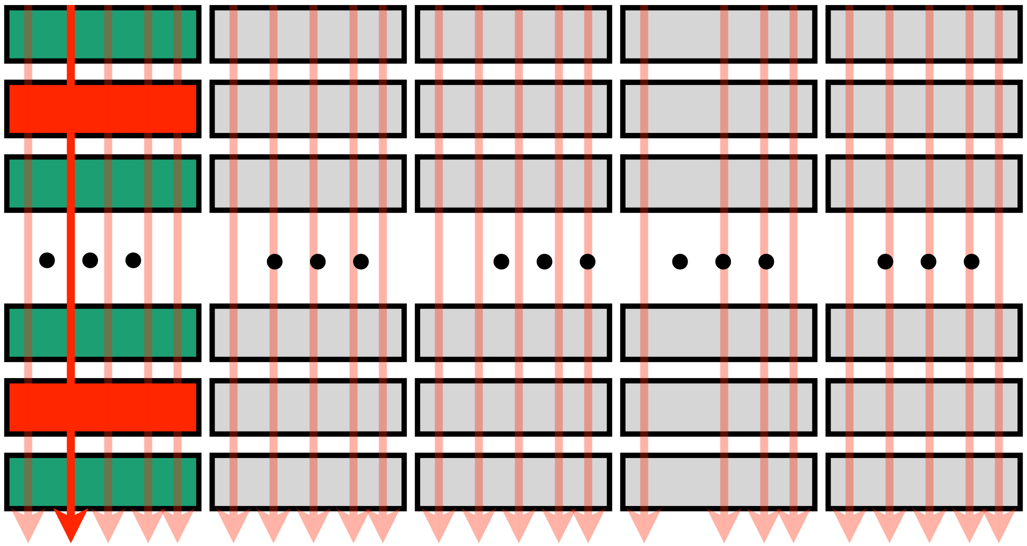
Let *#accesses = (size of cache) / line size*. Then the *#accesses+1* access will likely cause a cache line to be evicted.

Many evictions (and *misses*) occur



Each miss causes an access to take tens to hundreds of cycles

Each new column reloads data



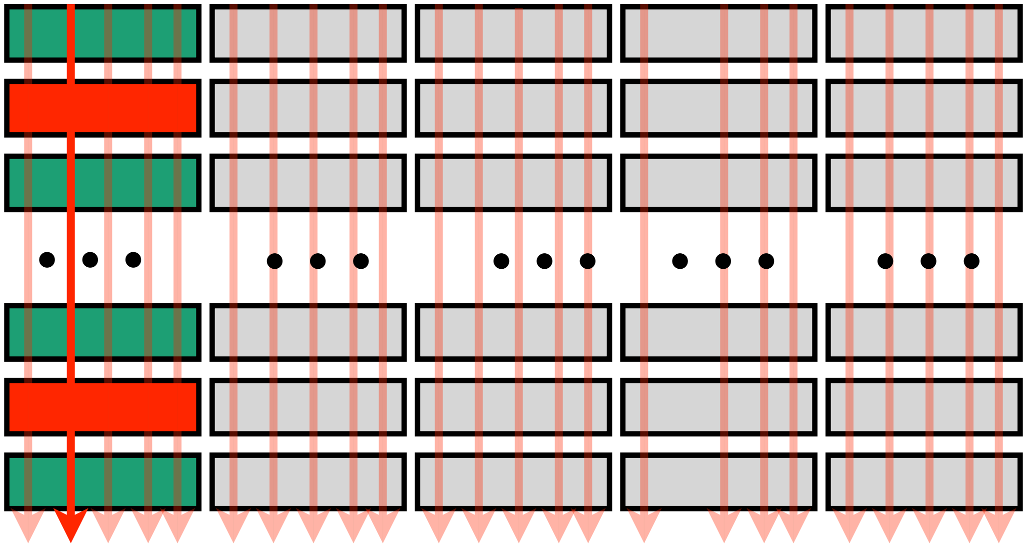
When the next column is fetched, grabbing the first elements will evict other cache lines.

This results in ~1 miss per element accessed. For matrix multiply, n^3 operations mean n^3 fetches overall. Would prefer to have n^2 (size of the data) fetches, and $n^2/line\ size$ misses

Cache optimizations

- Desire is to maximizes locality and to exploit temporal and spatial locality via the cache.
- Accesses from cache are tens to hundreds of times faster than accesses from memory, and typically the same speed or 2 times slower than accesses from registers
- The more regular the code and subscript functions, the more compilers can do in terms of cache optimizations

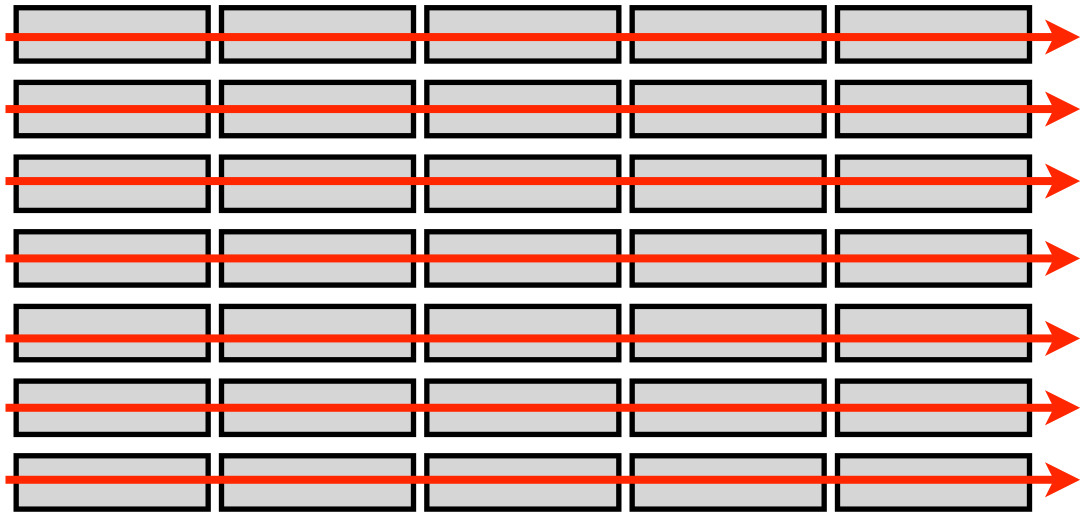
Loop interchange can help



```
for (i = ... ) {
   for (j = ... ) {
      a[j,i] = ...
   }
}
```

Becomes ...

Loop interchange can help

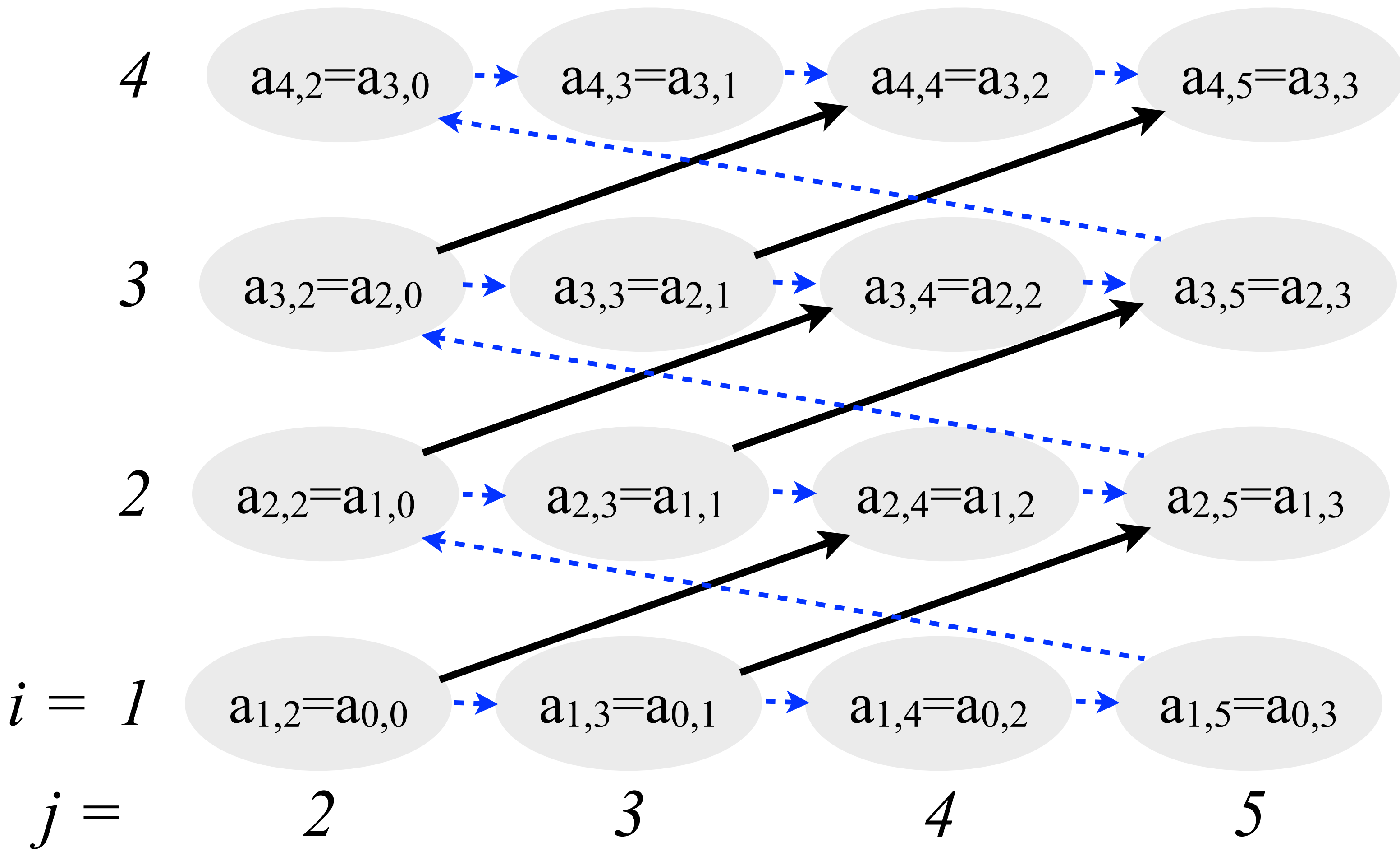


```
for (j = ... ) {
    for (i = ... ) {
        a[j,i] = ...
    }
}
```

This is not always legal and doesn't always help

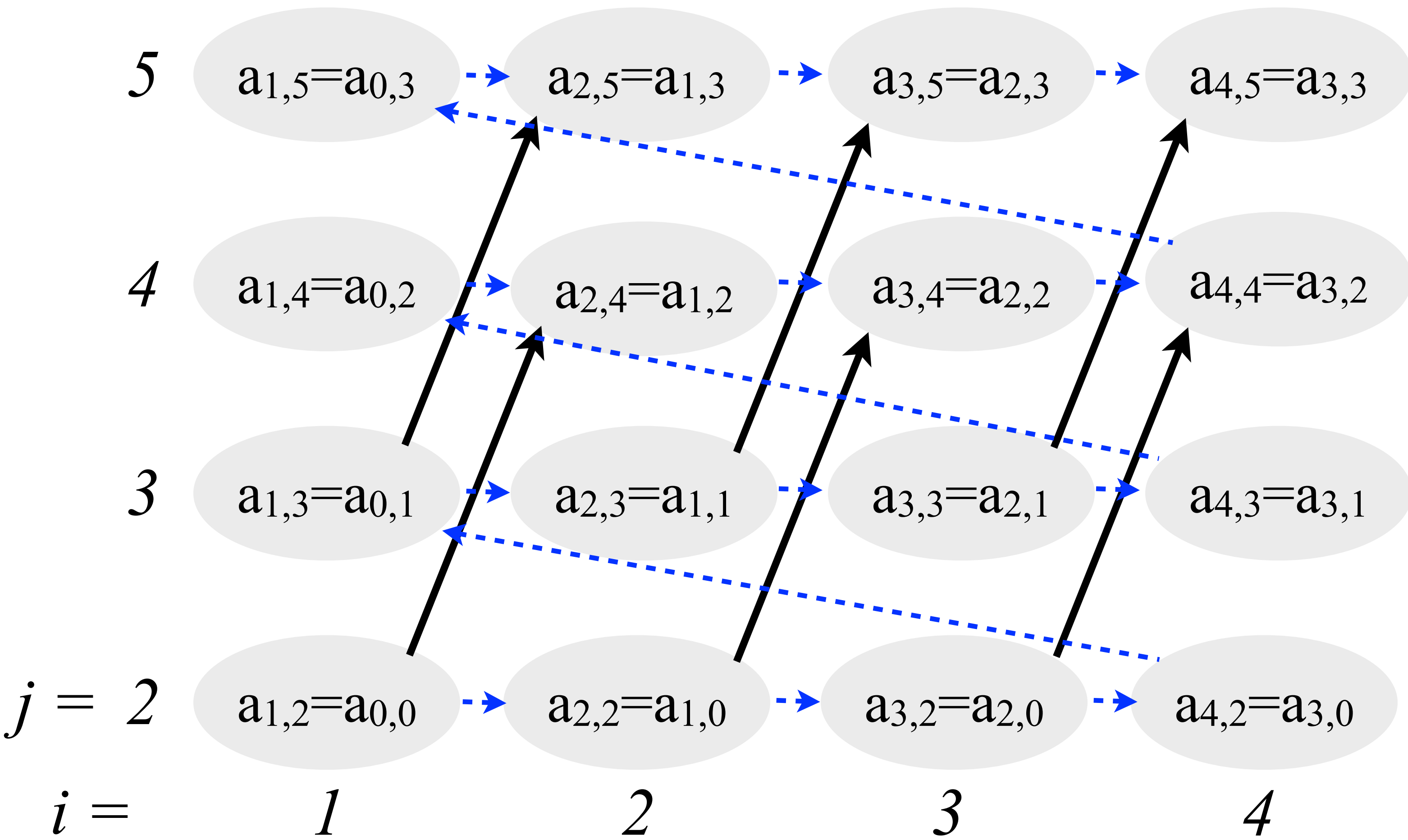
```
for (i=1; i < n; i++)
  for (j=2; j < n; j++)
    a[i][j] = a[i-1][j-2]
```

When is it legal? Tail is a write, and head is the read of the written array element. Read of data follows write in the iteration space. blue lines are iteration orders.



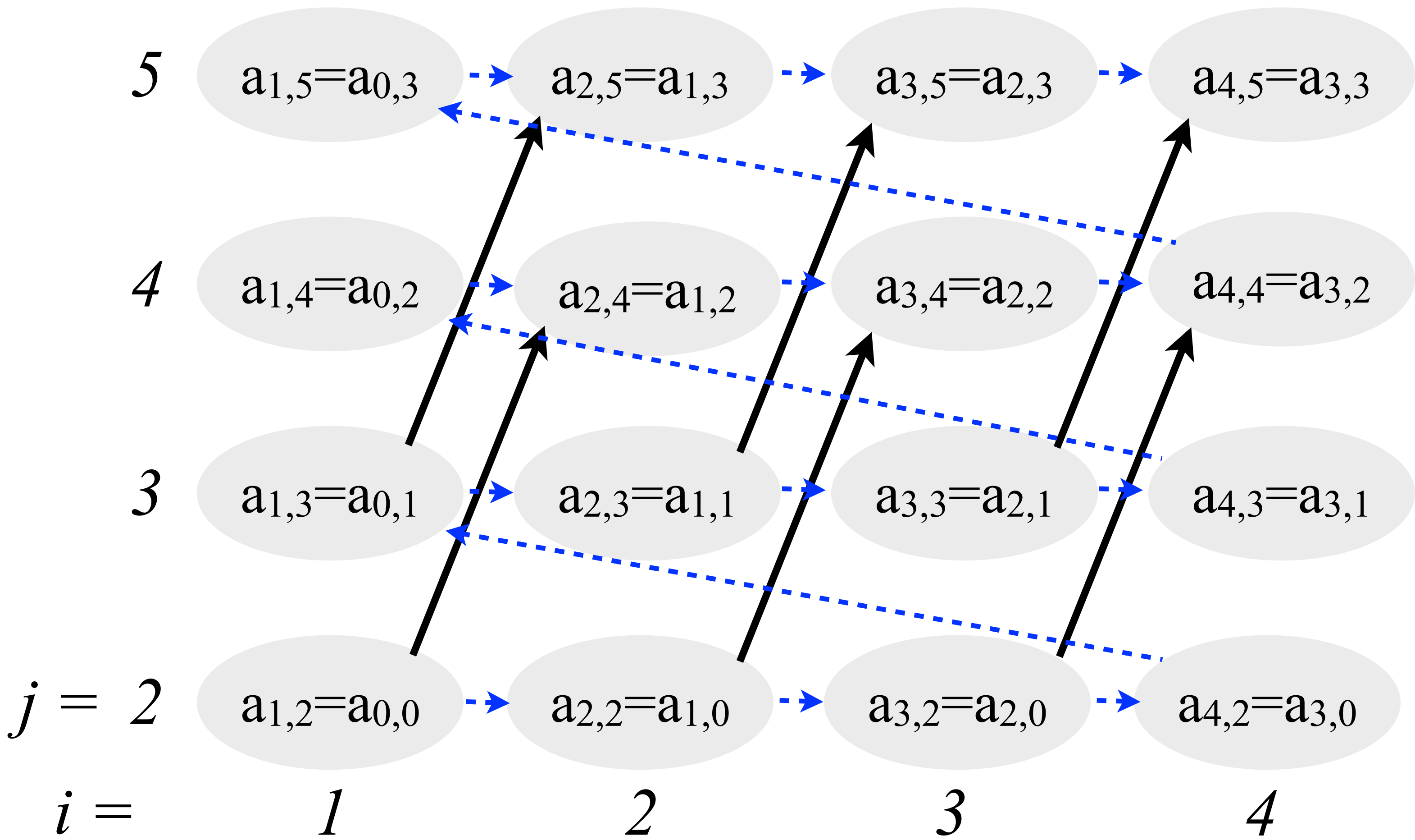
```
for (j=2; i < n; i++)
  for (i=1; j < n; j++)
    a[i][j] = a[i-1][j-2]
```

After interchange, i and j axes switched Tail is a write, and head is the read of the written array element. Read of data follows write in the iteration space.



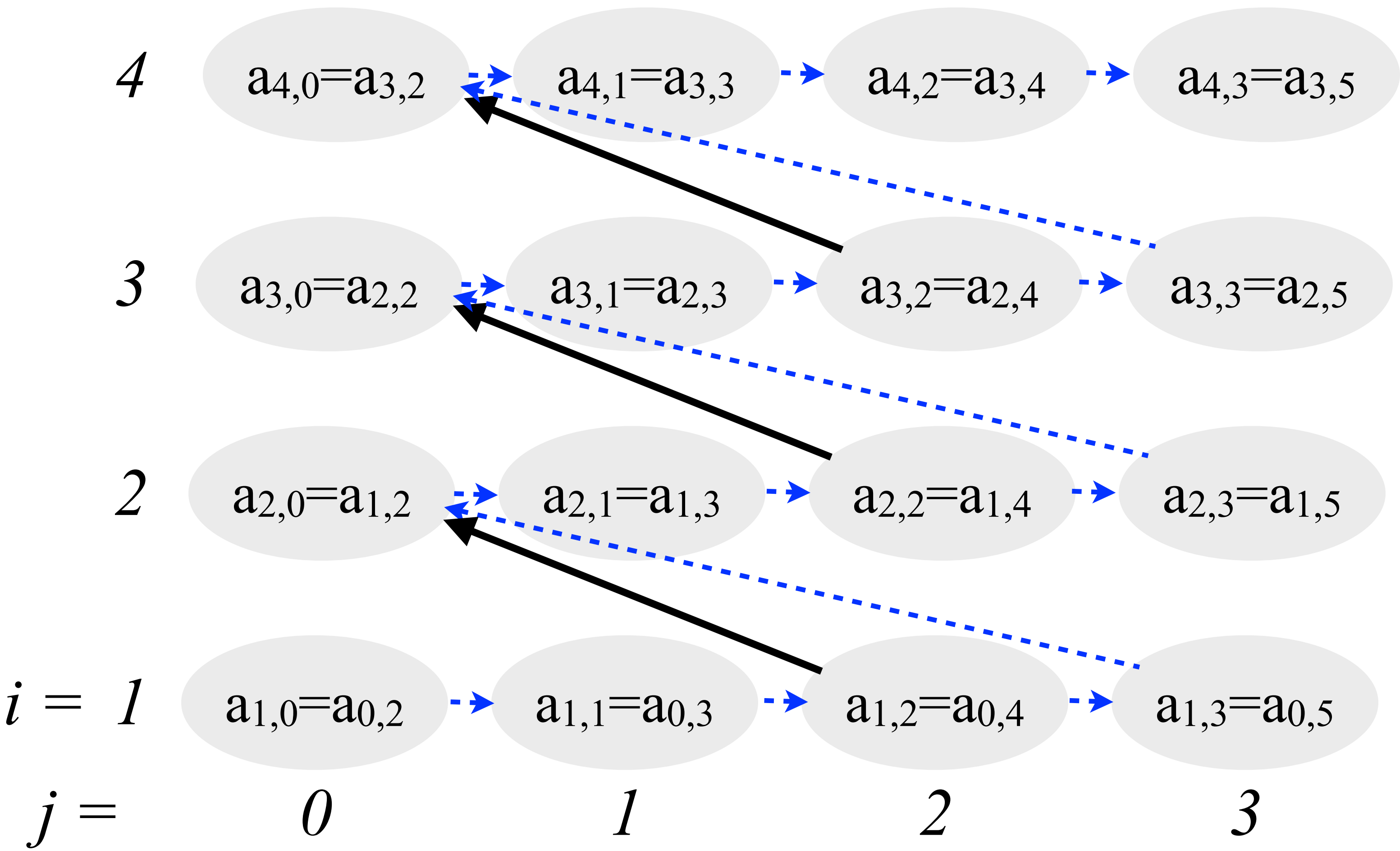
```
for (j=2; i < n; i++)
  for (i=1; j < n; j++)
    a[i][j] = a[i-1][j-2]
```

Legal because the read that happened after the write in the original loop still happens after the write.



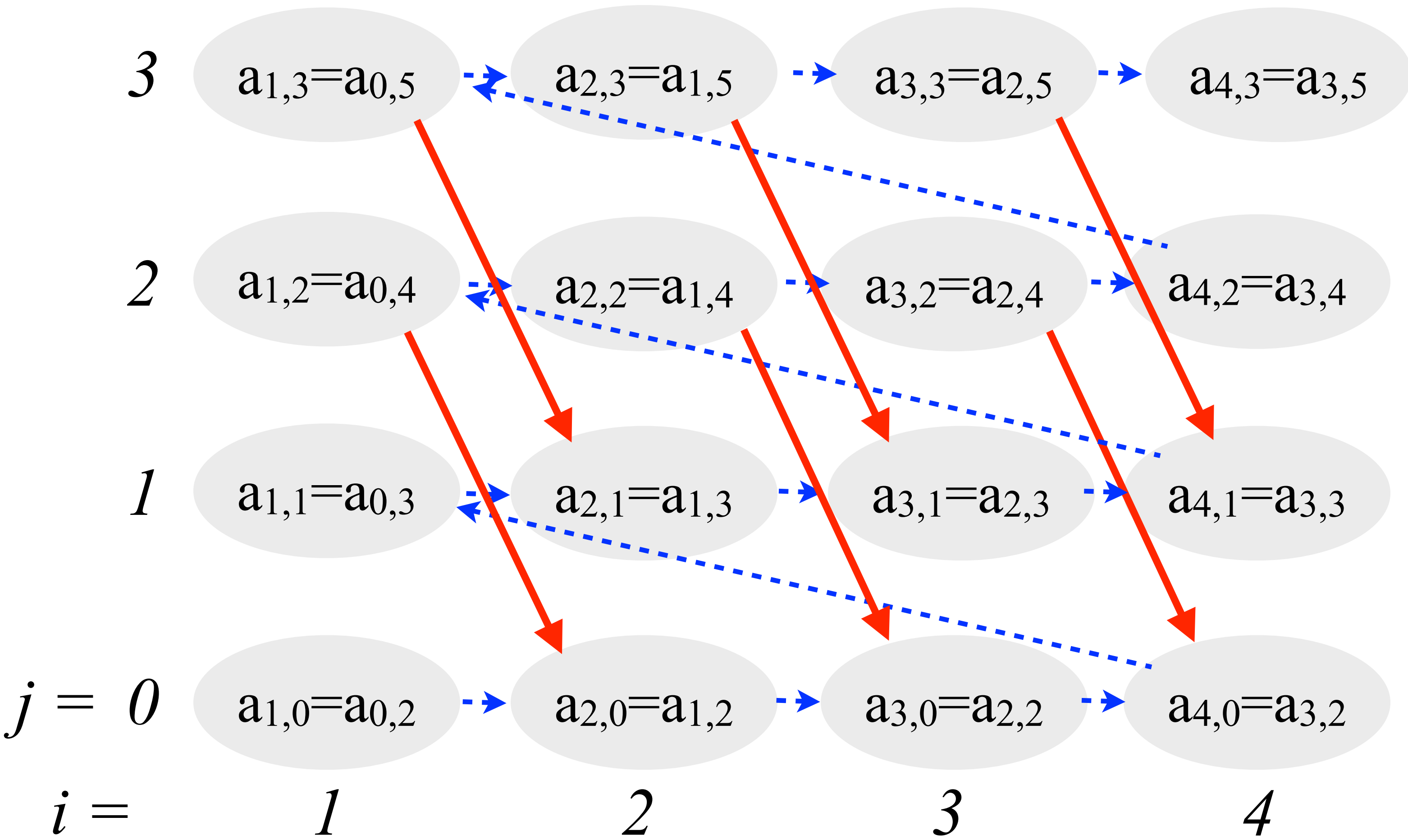
```
for (i=1; i < n; i++)
   for (j=0; j < n; j++)
      a[i][j] = a[i-1][j+2]
```

When is it illegal? Tail is a write, and head is the read of the written array element. Read of data follows write in the iteration space. blue lines are iteration orders.



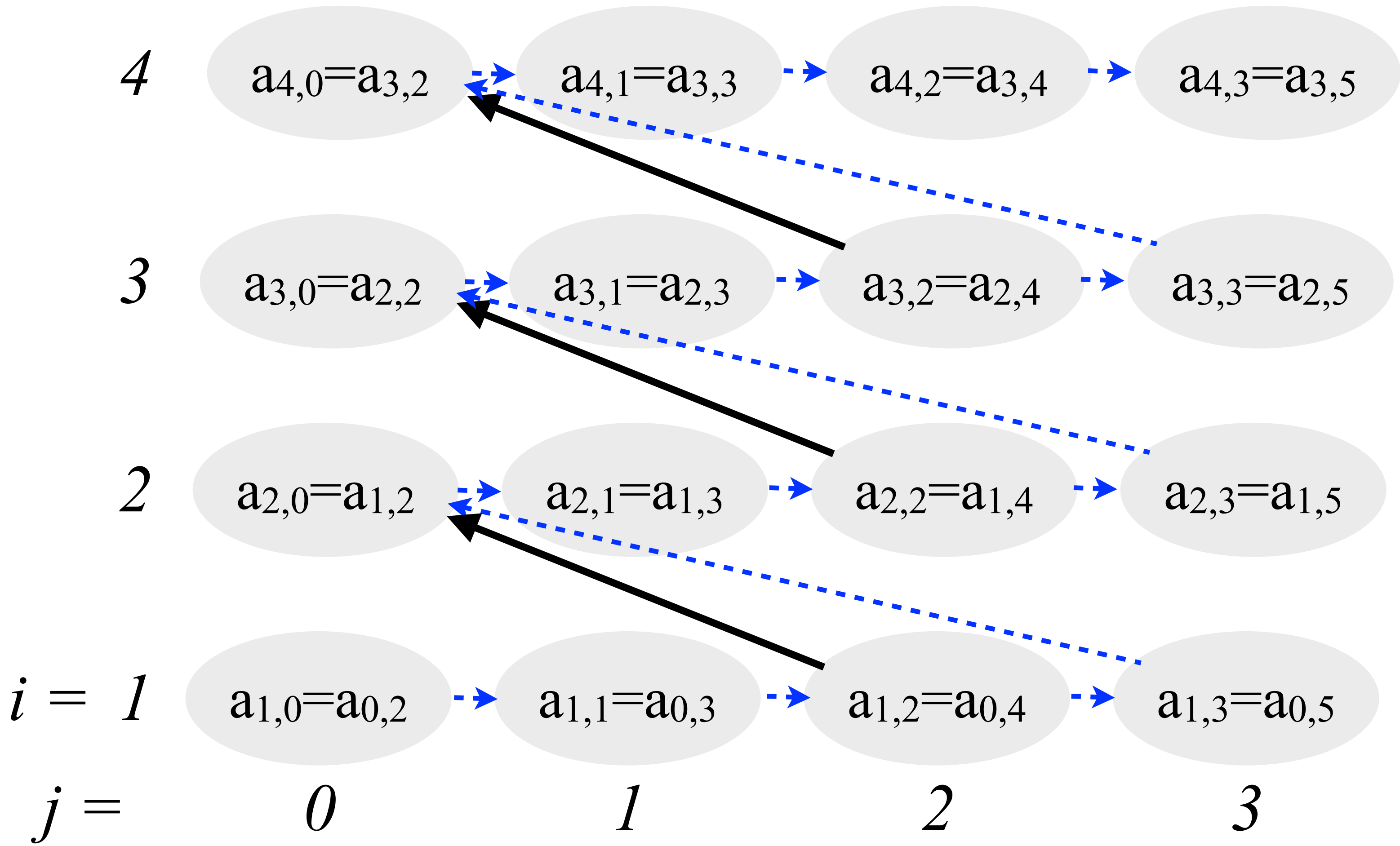
```
for (i=1; i < n; i++)
  for (j=0; j < n; j++)
    a[i][j] = a[i-1][j+2]
```

When is it illegal? After the interchange, the write, which used to come before the read, now comes after the read. blue lines are iteration orders. Red edges are what *should* be the execution order.

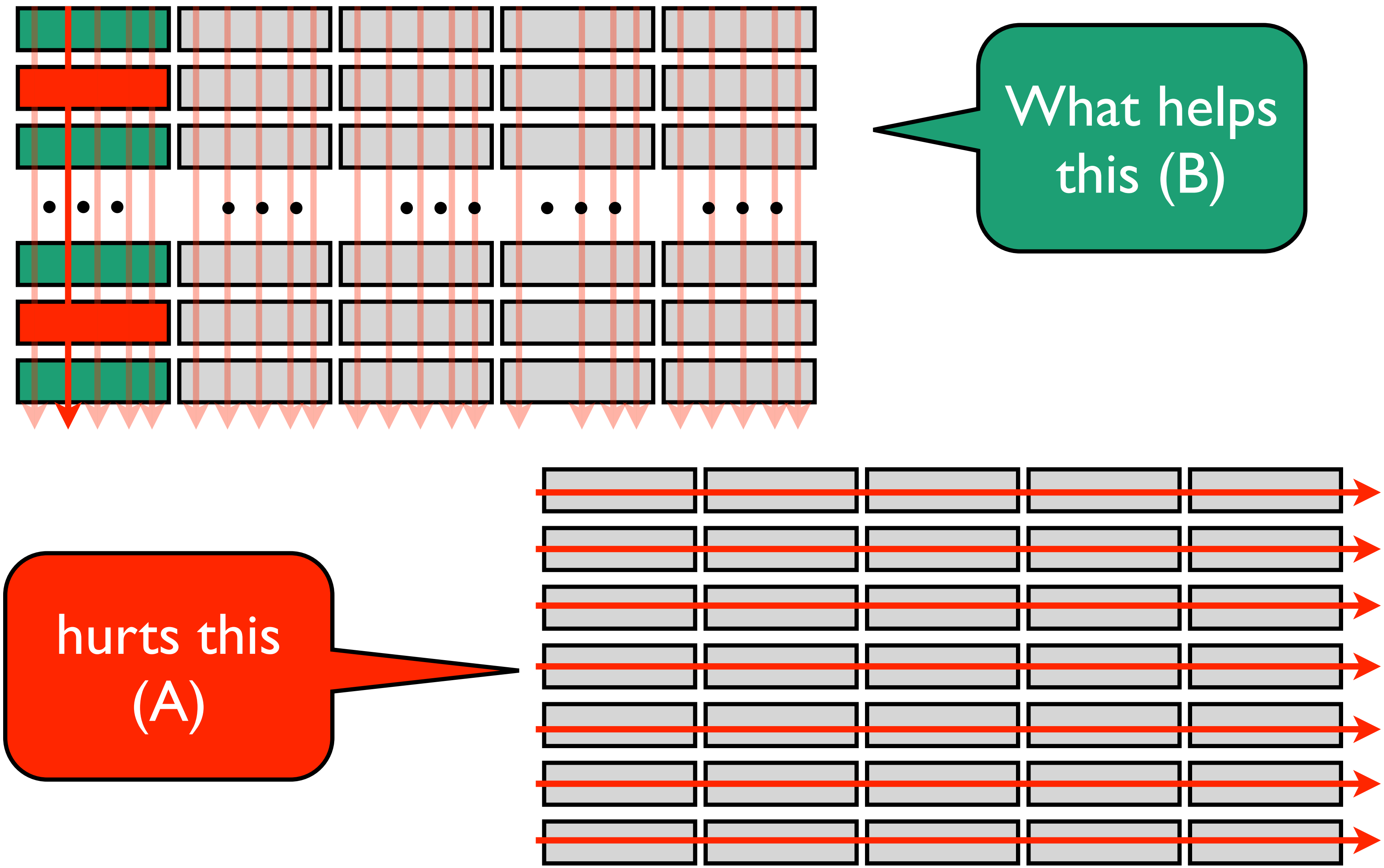


```
for (i=1; i < n; i++)
   for (j=0; j < n; j++)
      a[i][j] = a[i-1][j+2]
```

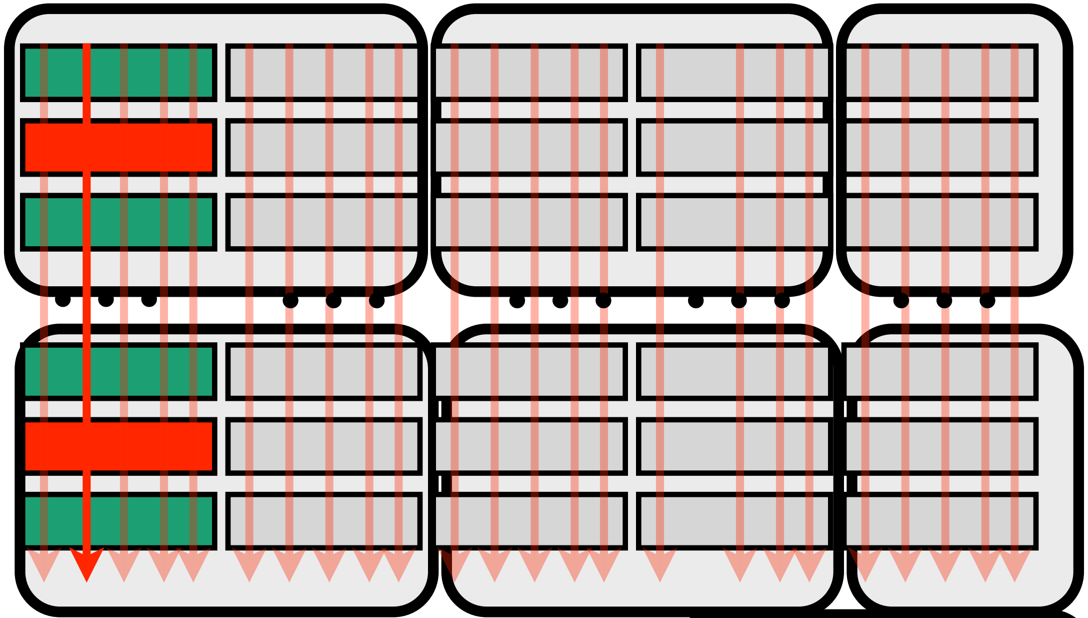
What causes the illegal behavior? It is that the distance traveled from read to write on the i and j loop iteration spaces is *(1,-2)*. After interchange, it is (-2,1), or backwards in the interchanged iteration space.



When it doesn't help

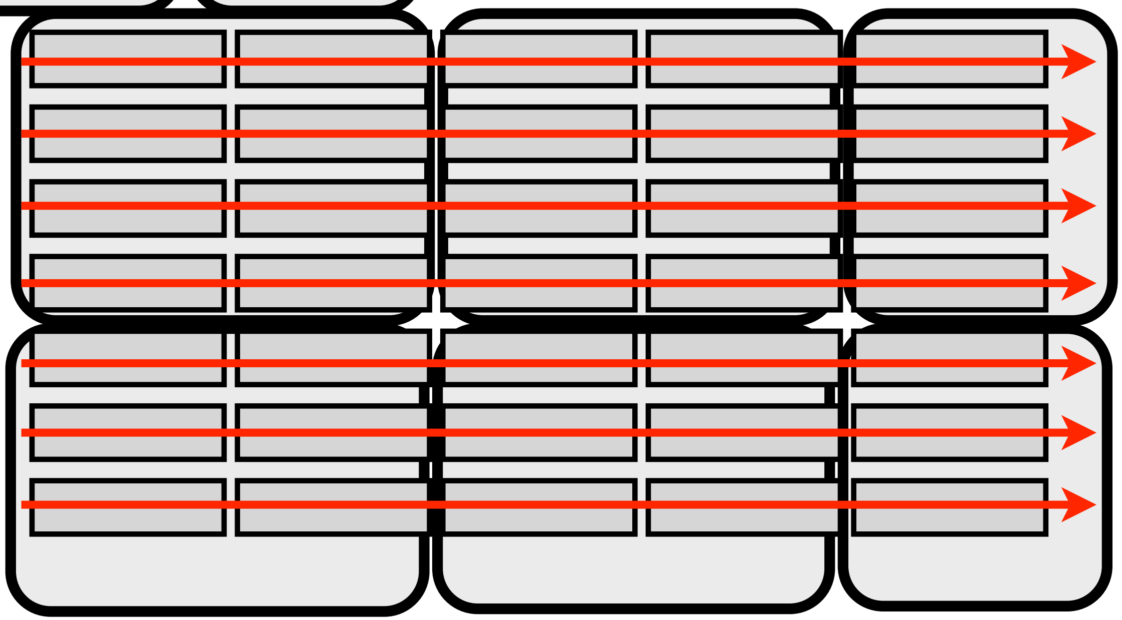


Do the computation in chunks

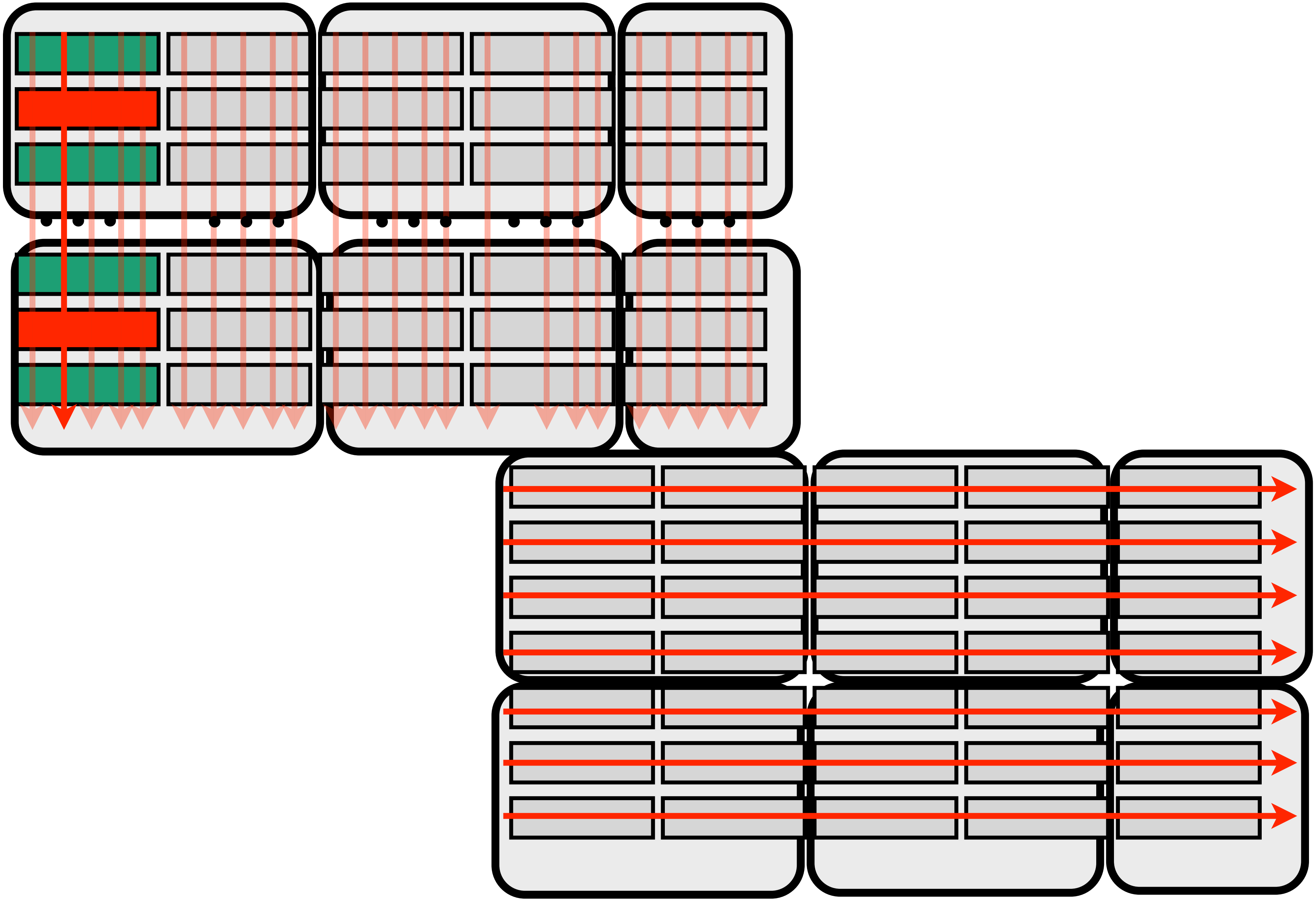


In a compiler perform the *tiling* transformation

or can program a recursively blocked algorithm (shown later)



Two common ways to do this

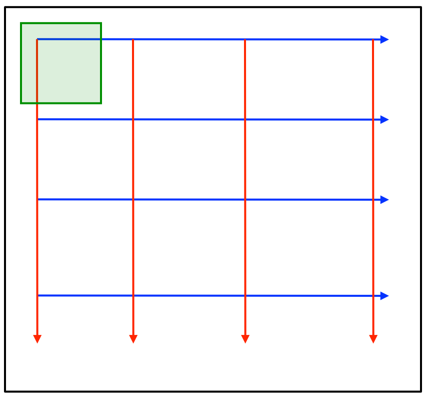


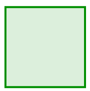
Loop tiling (page 347, Bacon, Graham paper)

```
Do i = 1, n
   Do j = 1, n
      a[i,j] = b[j,i]
   end do
end do
```

a,b[col,row], array stored in row major order

```
do TI = 1, n, 64
   do TJ = 1, n, 64
      do i = TI, min(TI+63,n)
         do j = TJ, min(TJ+63, n)
            a[i,j] = b[j,i]
         end do
      ...
end do
```





What fits in cache

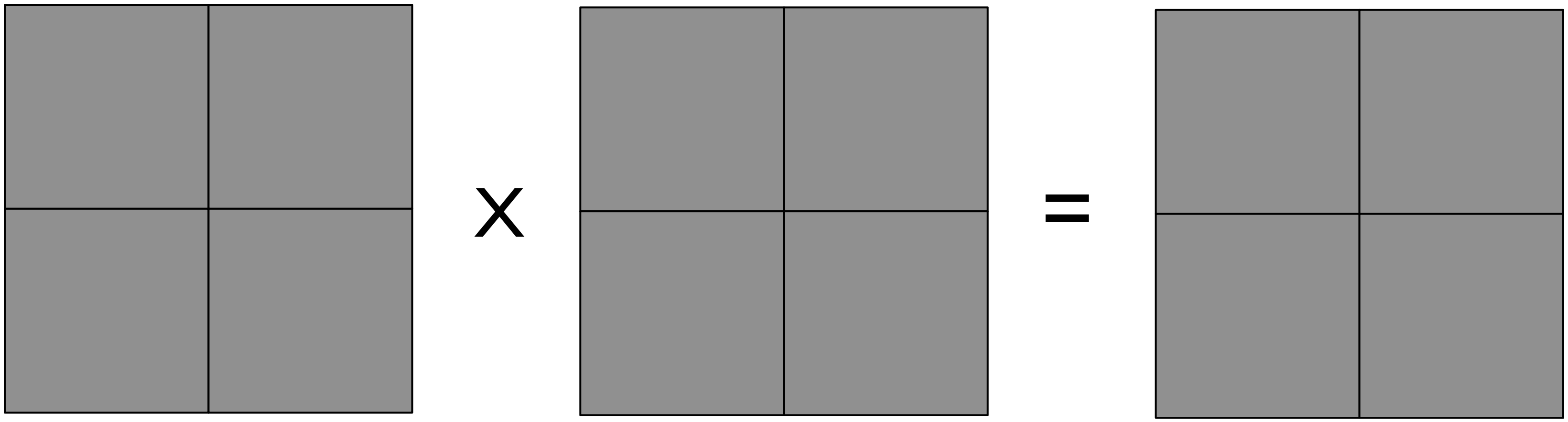
b[i,j] accesses

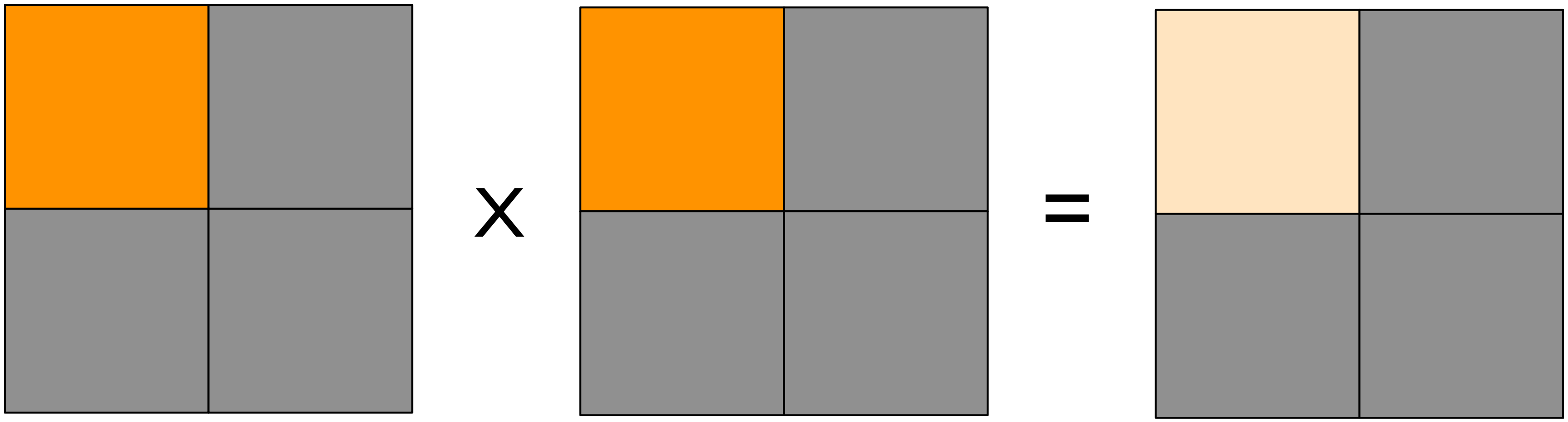
a[I,j] accesses

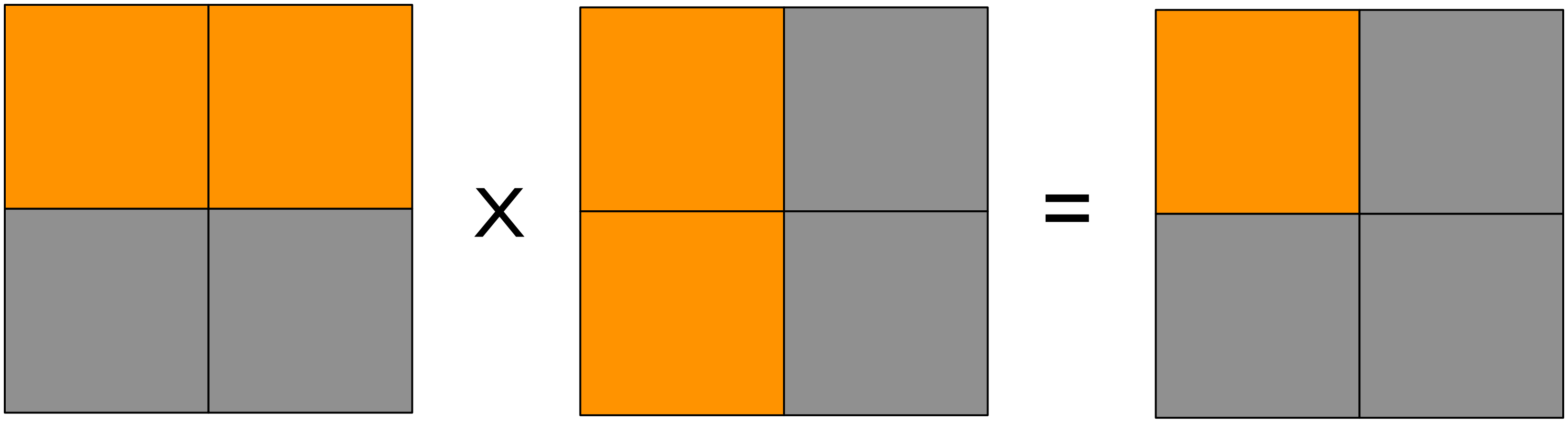
This pattern is present in matrix multiply, transpose, etc.

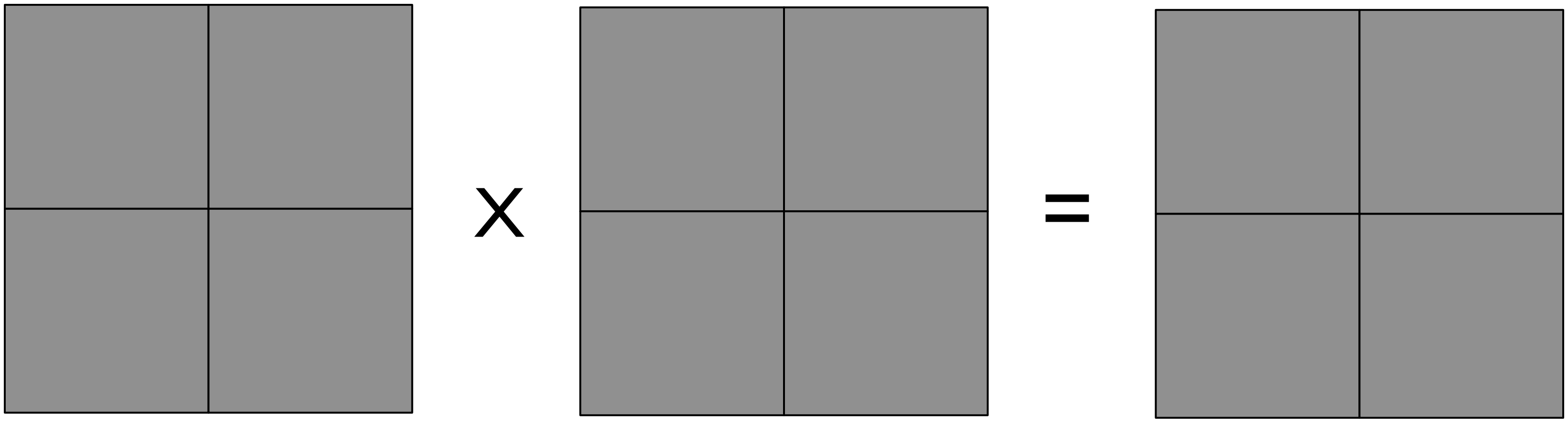
Blocked matrix multiply

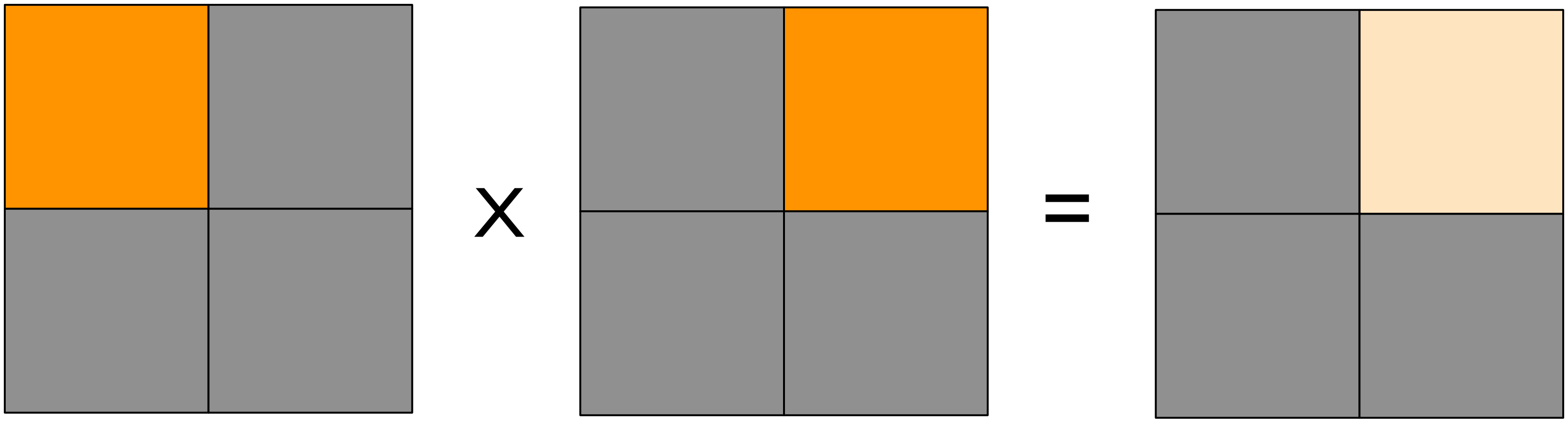
- Replace scalar multiplication with matrix multiplication
- Replace scalar addition with matrix addition

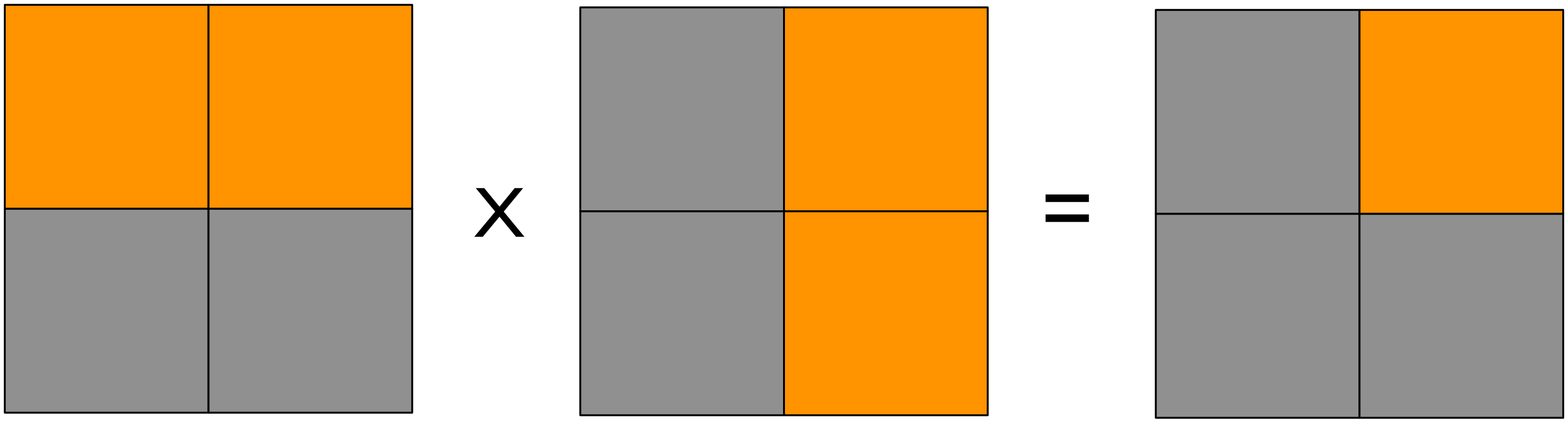


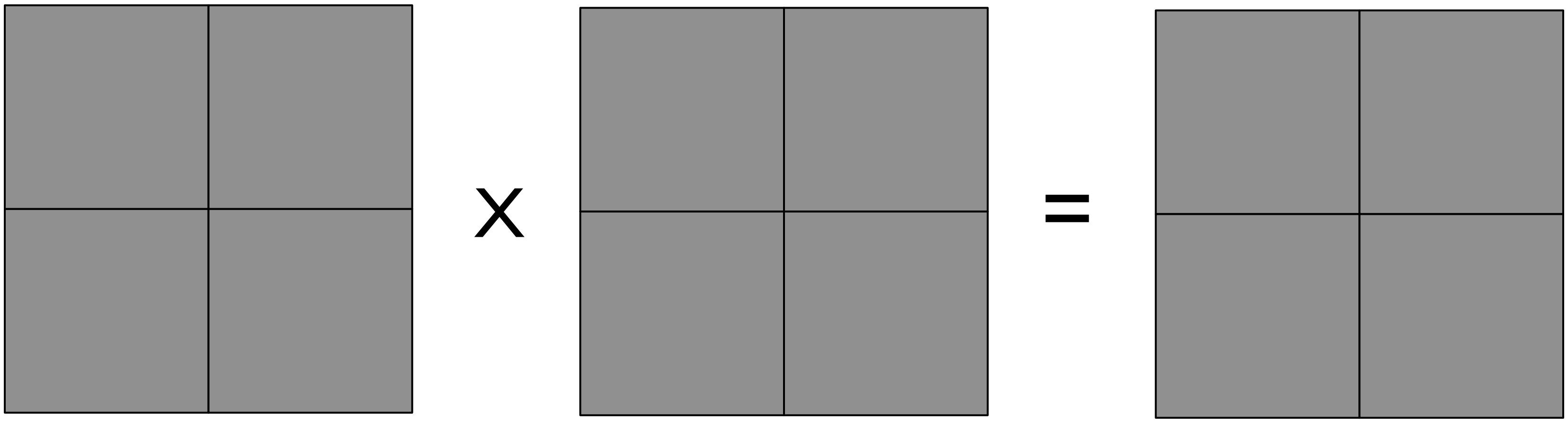

x
=

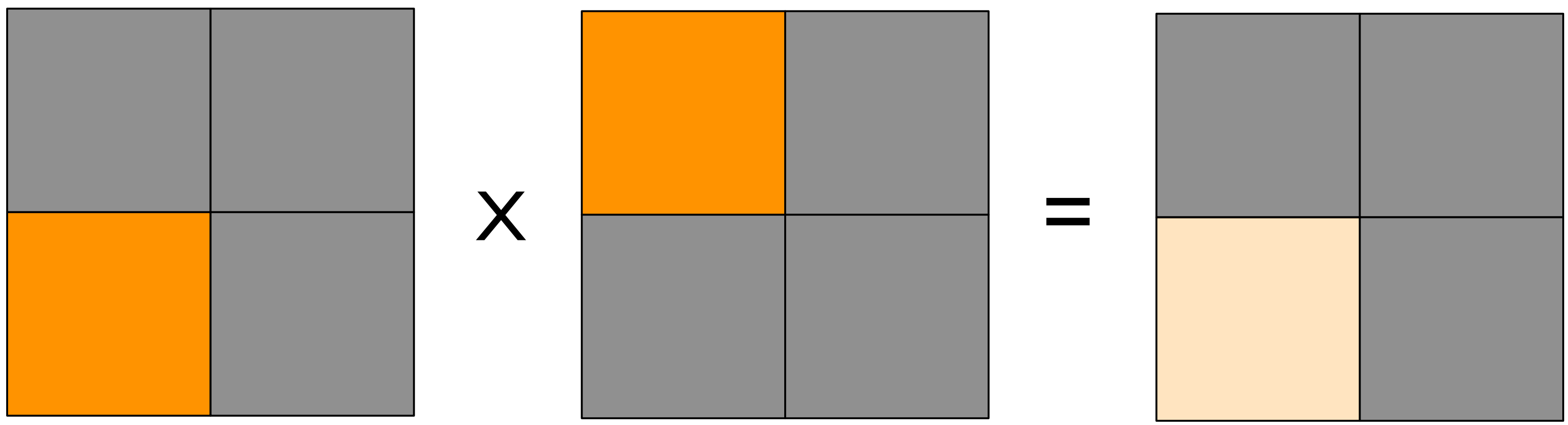


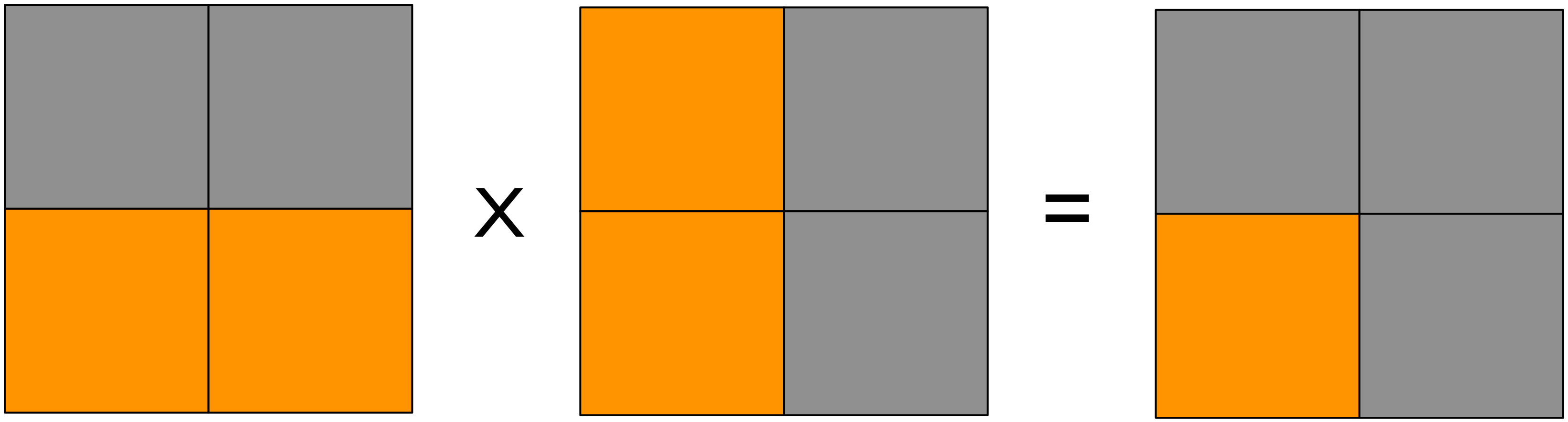

x
=

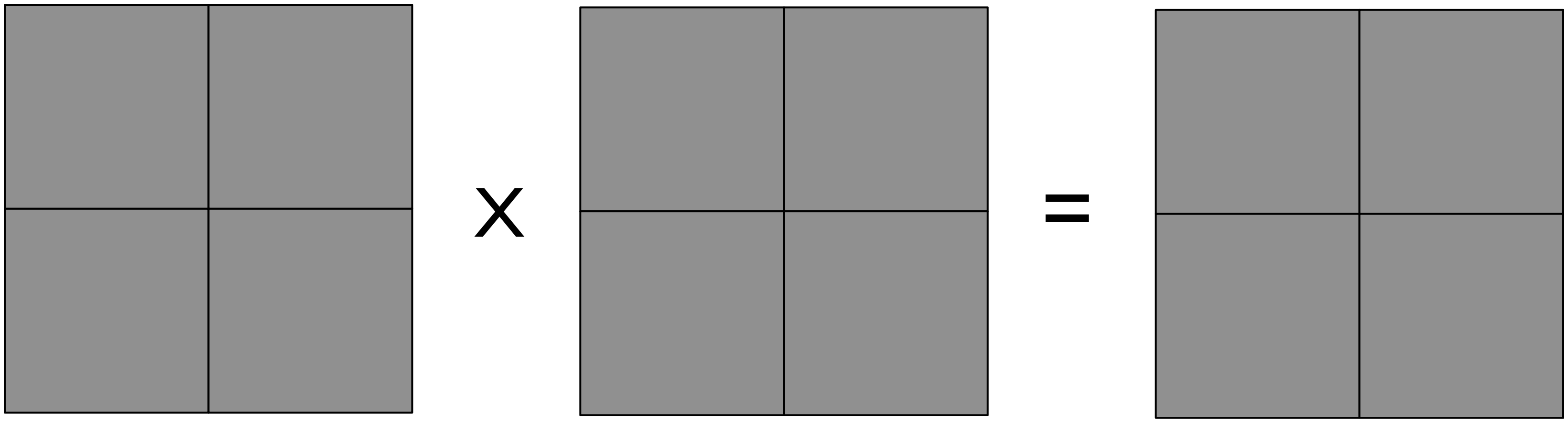


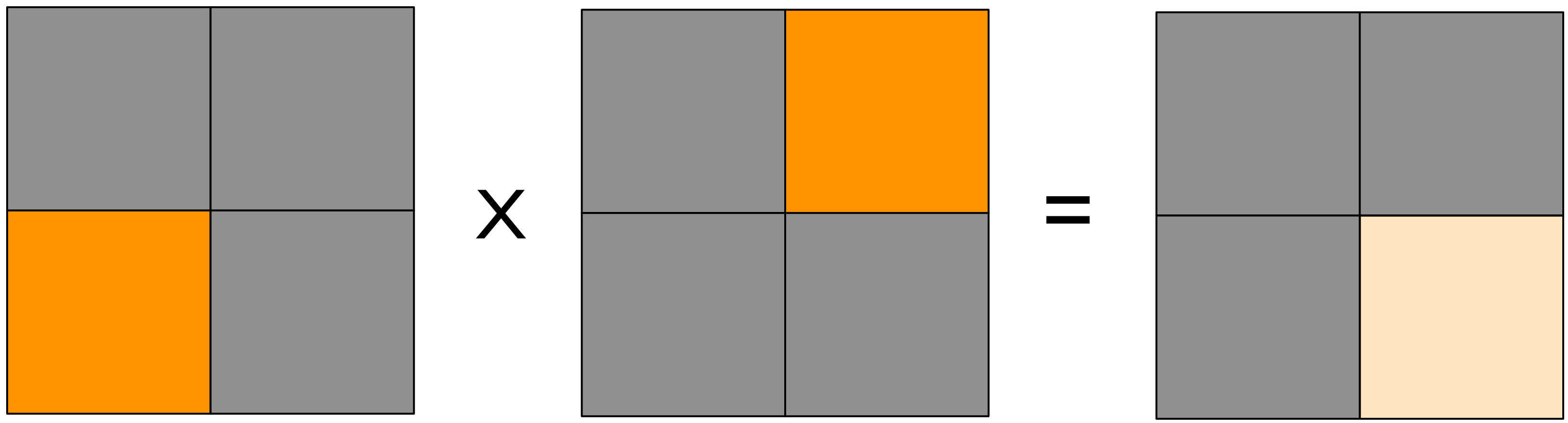

x
=

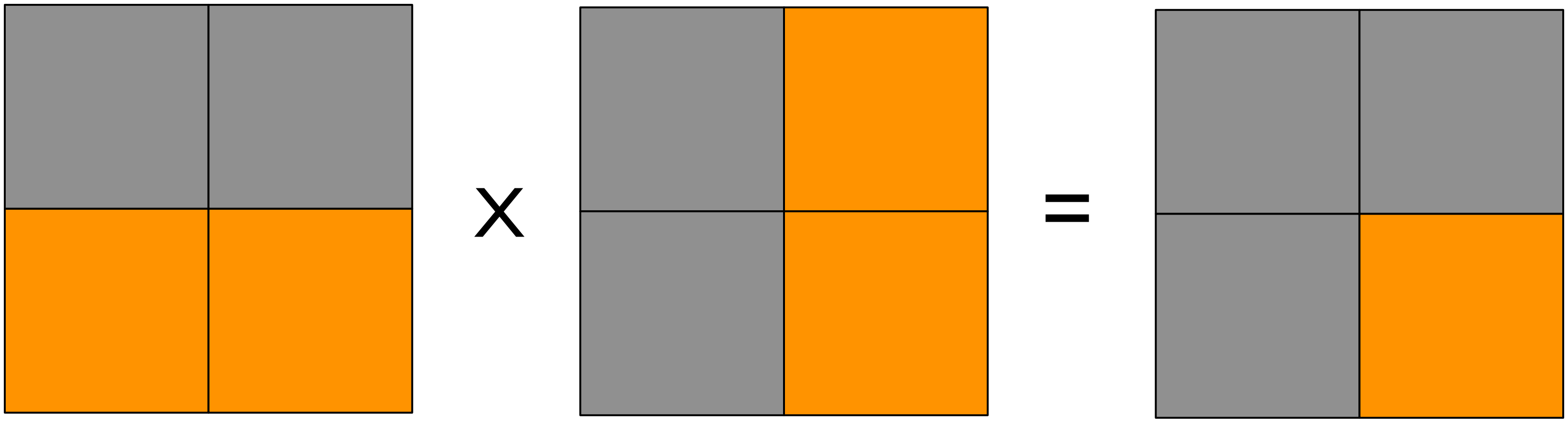

x
=



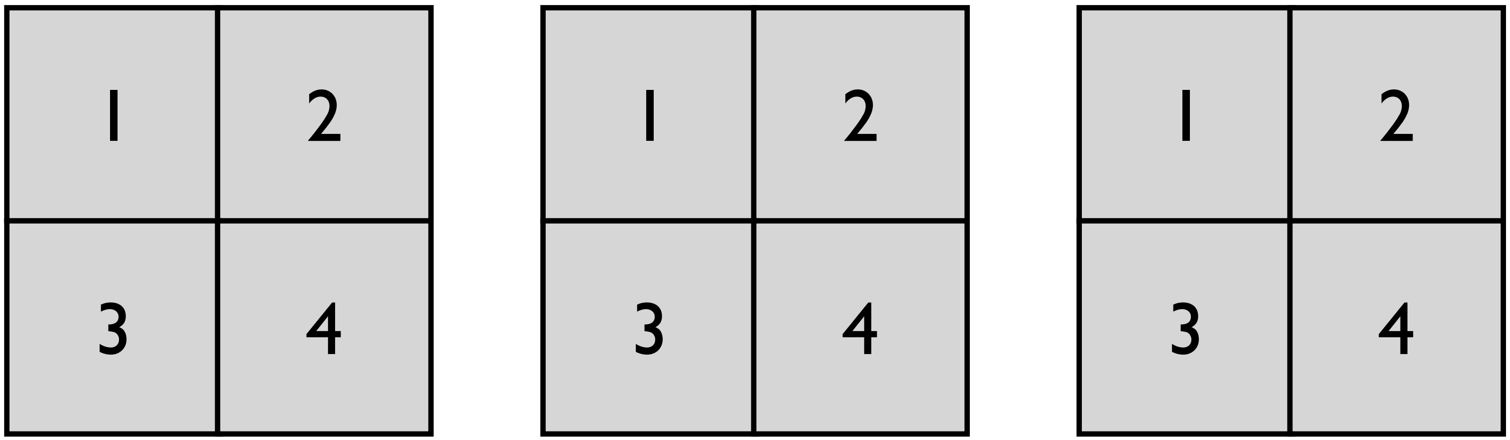




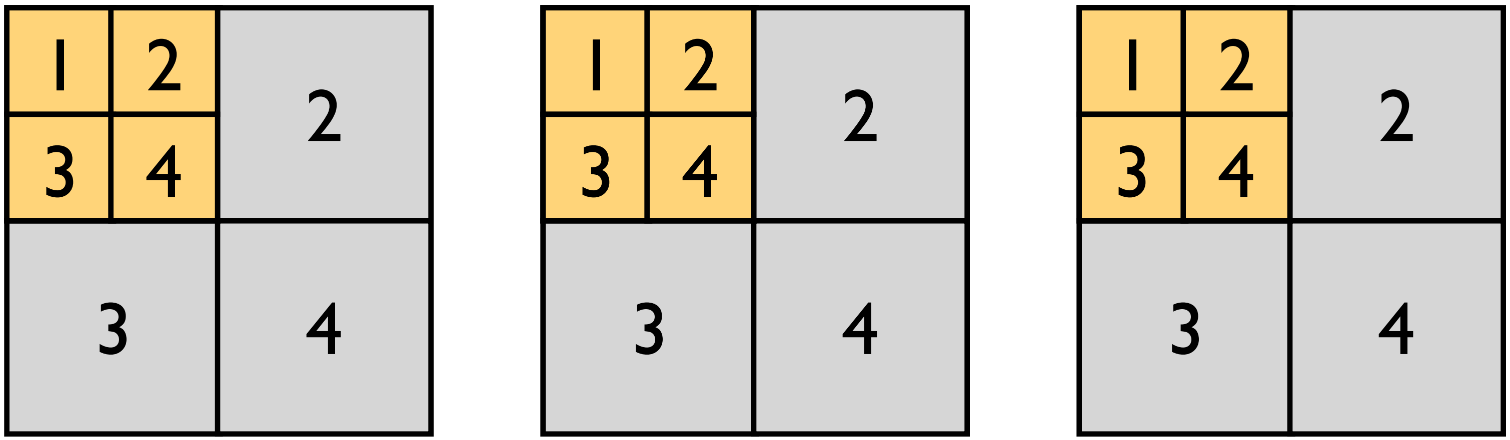



x
=

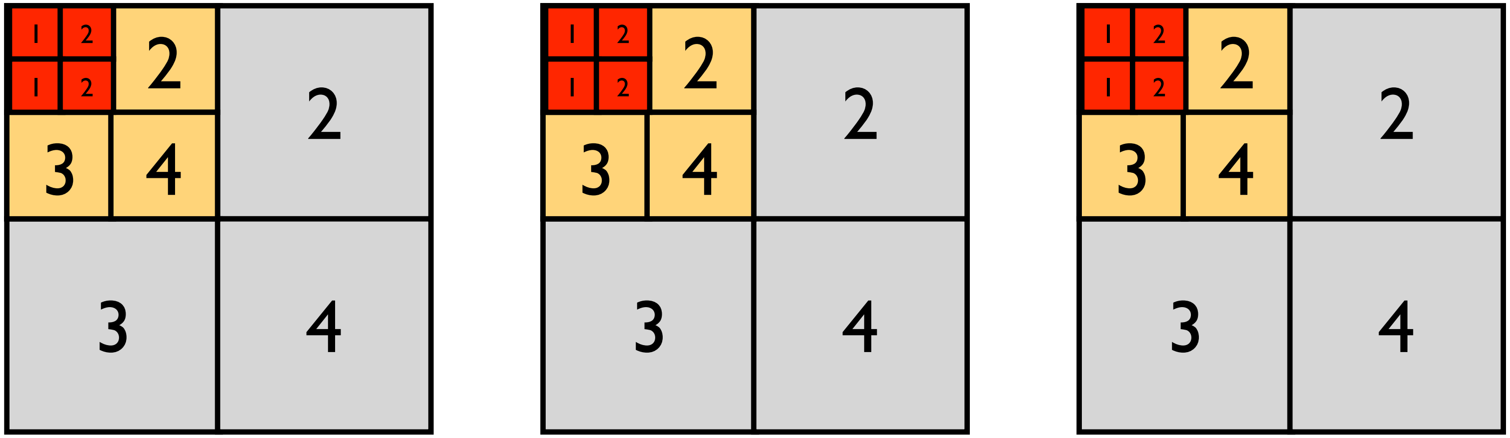
Recursively block until B small enough



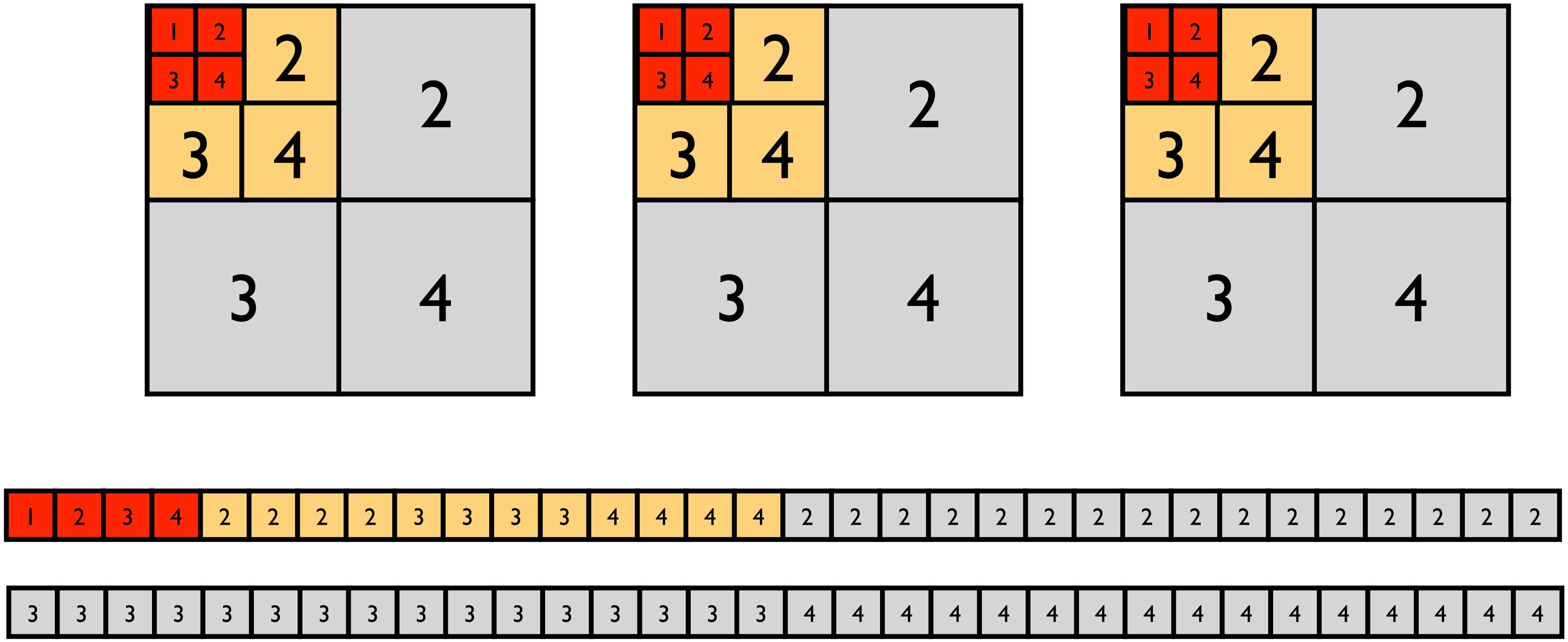
Recursively block until B small enough to fit into cache



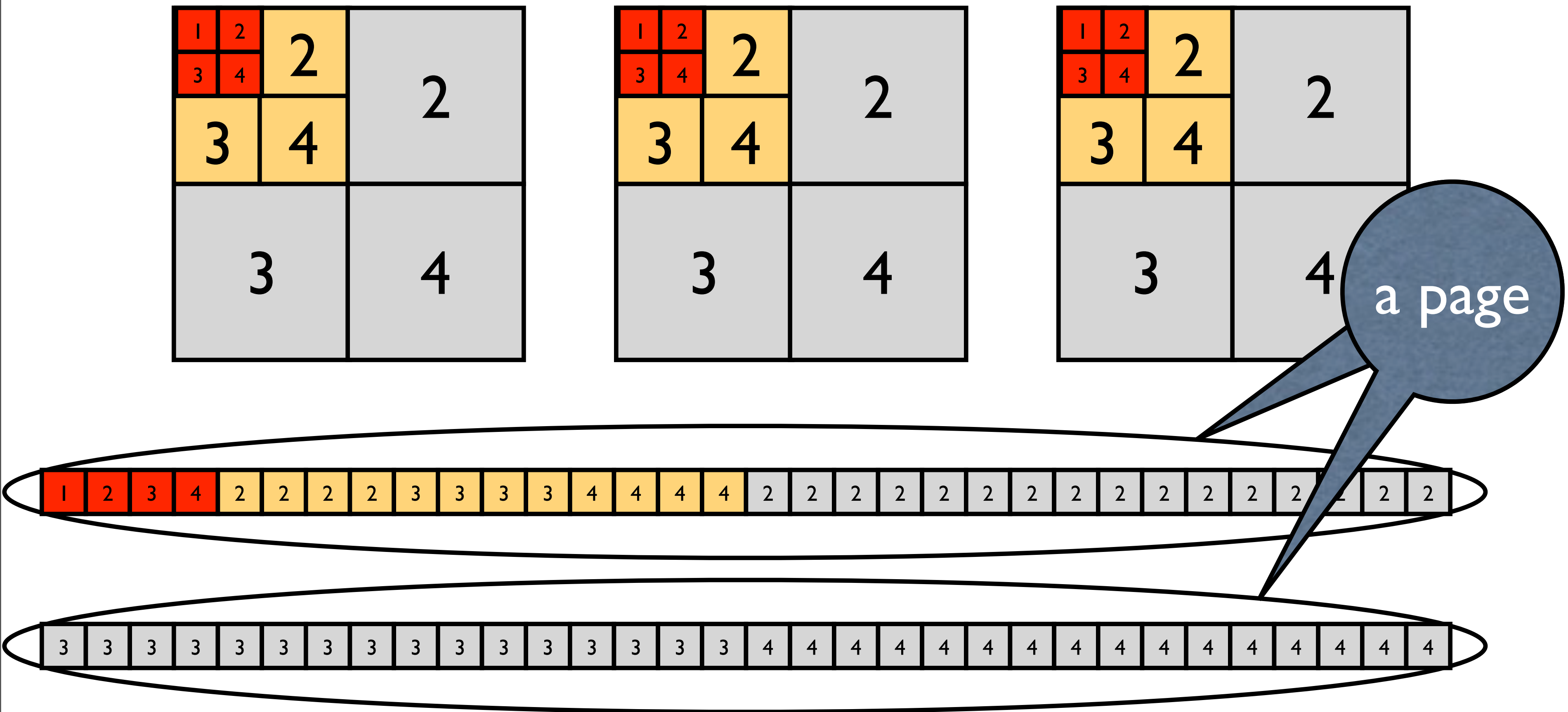
Recursively block until B small enough to fit into cache



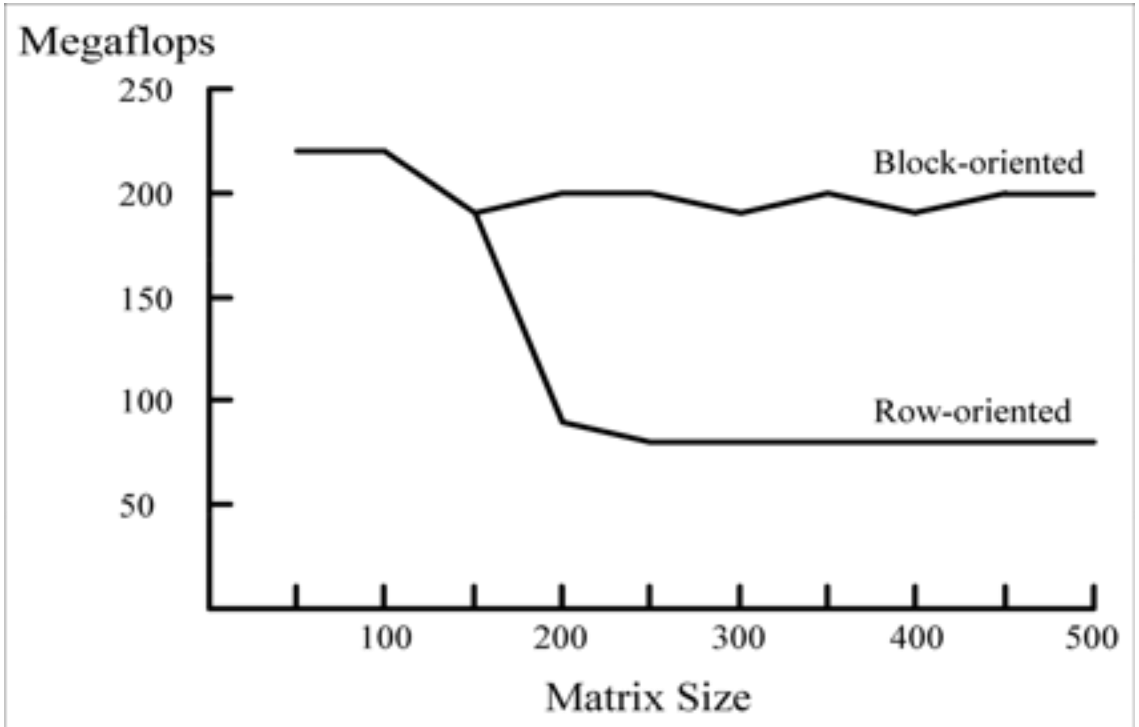
Layout in memory



Layout in memory



Comparing Sequential Performance

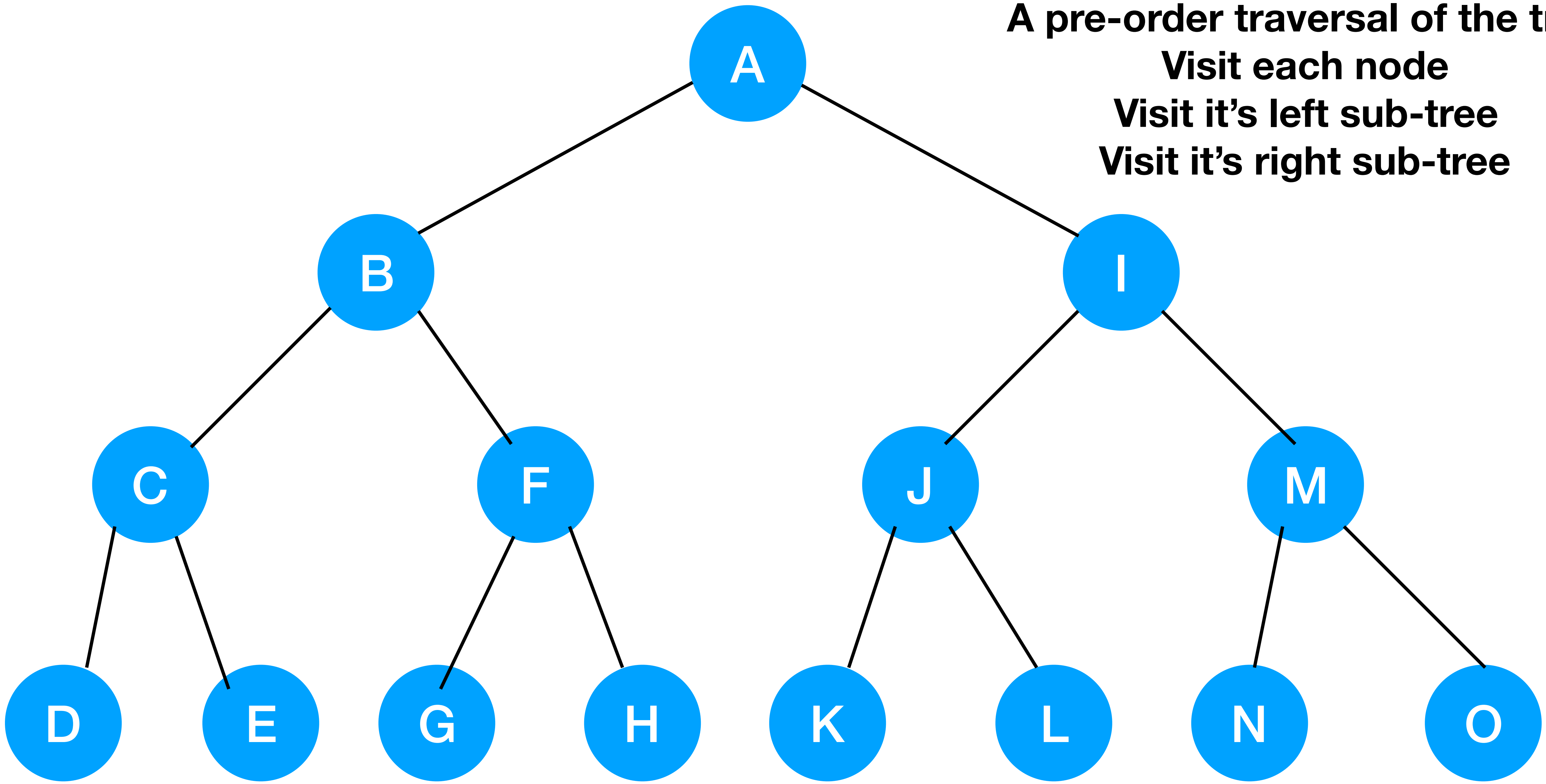


On modern processors, recursively blocked algorithms can achieve 90% of peak performance

The generalized technique is to use *space filling curves*. Often these are Hilbert Curves.

Prefetching often requires rethinking data layout

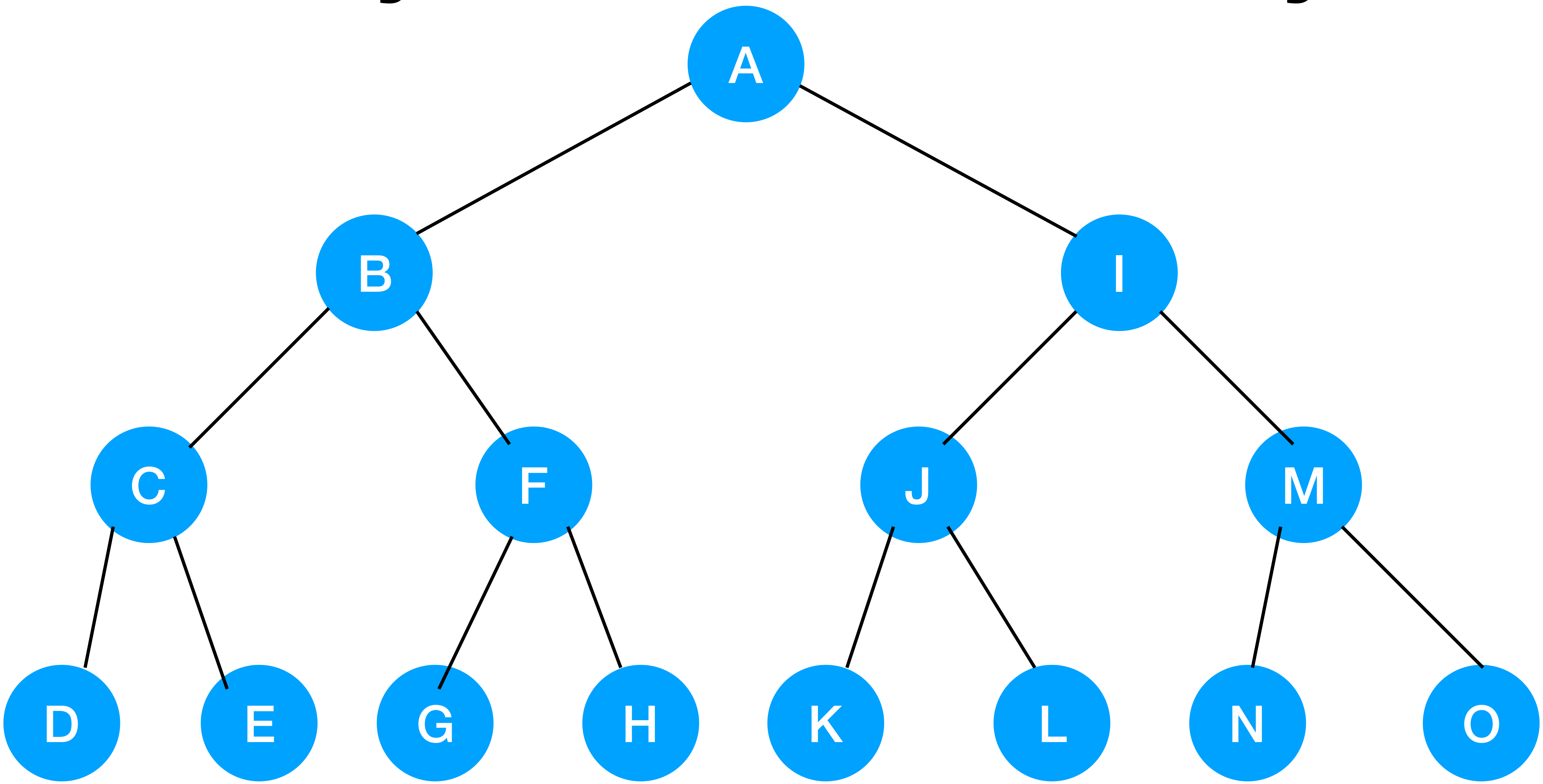
A pre-order traversal of the tree
Visit each node
Visit it's left sub-tree
Visit it's right sub-tree



Heap allocation

- Depending on where free space is, the nodes may be scattered throughout memory
- This is true even if the nodes are allocated in pre-order order.
- Cache behavior and pre-fetching will be poor

Lay it out in an array



A	B	C	D	E	F	G	H	I	J	K	L	M	N	O