CS420 – Lecture 5

Raghavendra Kanakagiri
Slides: Marc Snir

Fall 2022

Case study: transpose

```
for(i=0;i<N;i++)
  for(j=0;j<N;j++)
     b[i][j]=a[j][i];
```

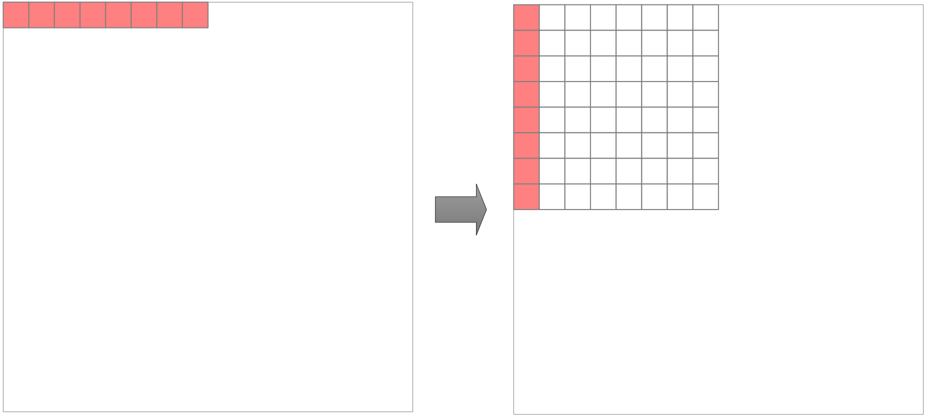
a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p

a

a	e	i	m
b	f	j	n
c	g	k	o
d	h	l	p

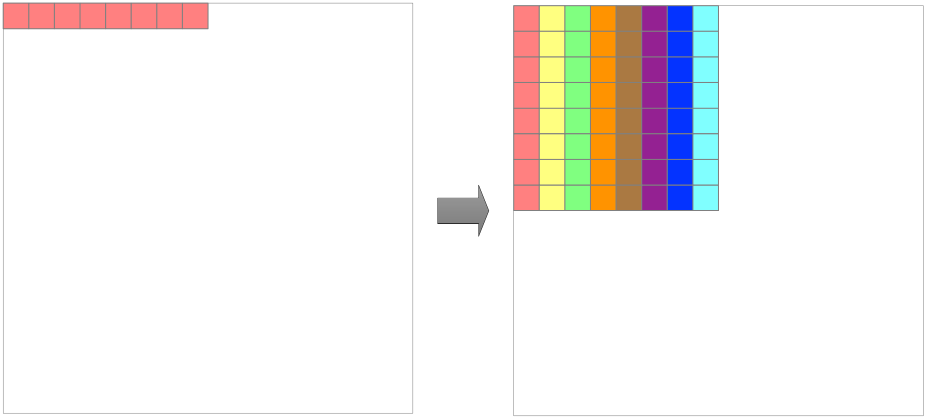
b

- If `a[]` is traversed by rows, then `b[]` is traversed by columns; and vice-versa!
- If matrix is large, have $\sim N^2/16 + N^2 = \frac{17}{16}N^2$ cache misses (assuming 16 words per cache line)

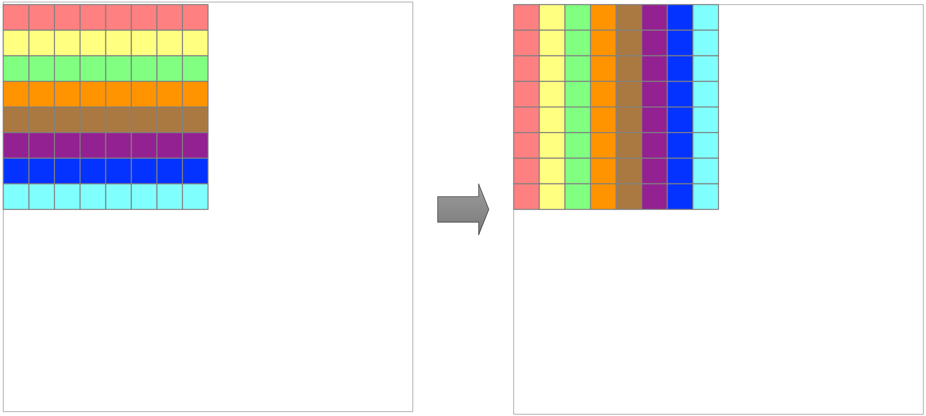


Read cache line (good), write 16 cache lines (bad)

Should fill up the remainder of the 16 cache lines soon after filling the first column



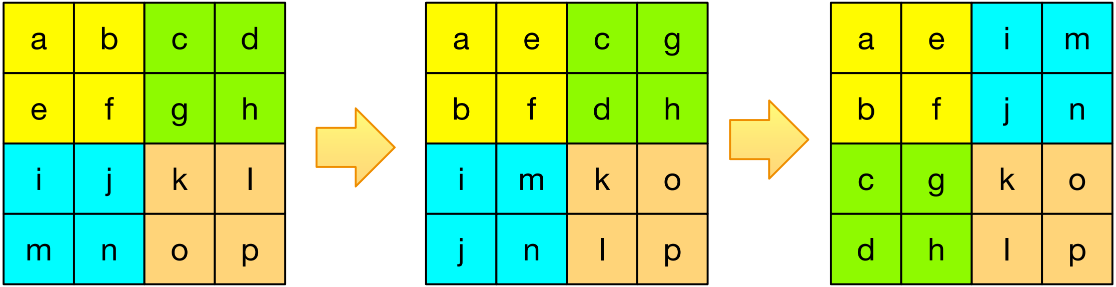
Should read the first cache lines in the next 15 rows soon after reading the first cache line in the first row



Better to read & write larger tile, as long as it fits in cache (prefetch)

Tiled transpose

In effect, tiles are read to cache, transposed within cache and written back to destination tile



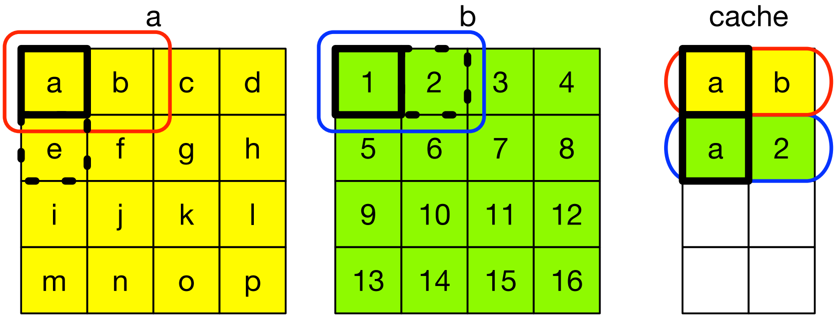
Tiled transpose

```
// T divides N
for(ii=0;ii<N;ii+=T)
 for(jj=0;jj<N;jj+=T)
  for(i=ii;i<ii+T; i++)
   for(j=jj;j<jj+T;j++)
    b[i][j] = a[j][i];
```

- 2 outer loops iterate over 2D tiles
- 2 inner loops permute a tile and store it to its final location (one tile read and one tile written)
- Choose tile size so that 2 tiles fit in cache
- Have $\sim 2N^2/16 = N^2/8$ cache misses (close to $\times 8$ improvement)

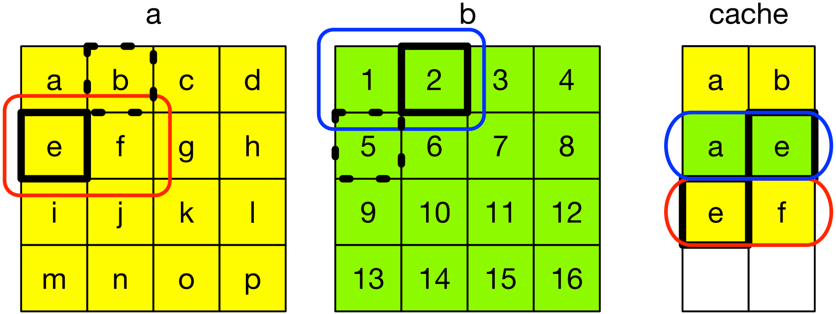
Transpose illustrated, assuming two words per cache line, and 4 lines per cache

Transpose illustrated 1



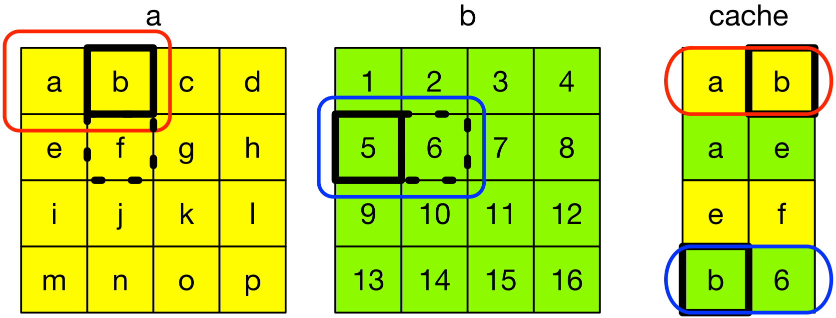
b[0][0]=a[0][0]

Transpose illustrated 2



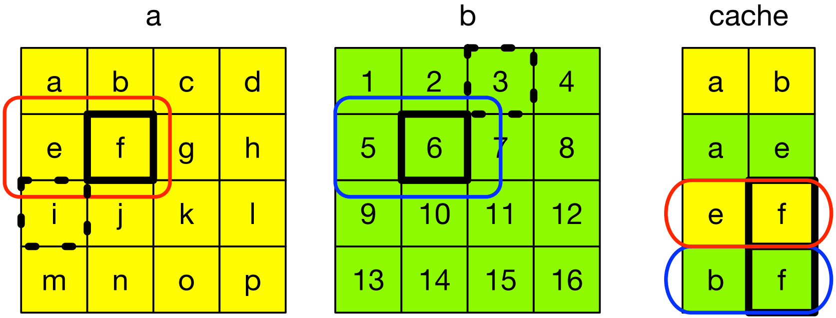
b[0][1]=a[1][0]

Transpose illustrated 3



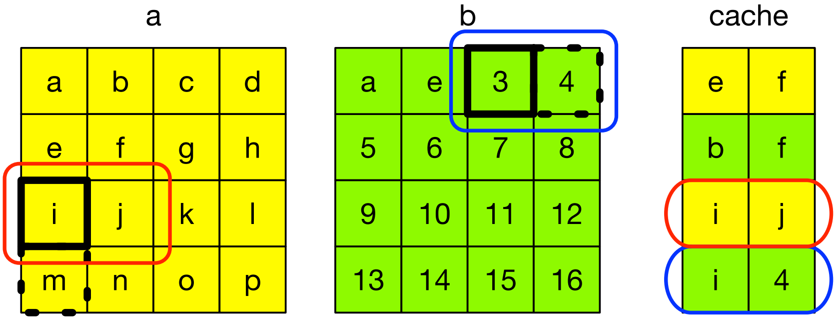
b[1][0]=a[0][1]

Transpose illustrated 4



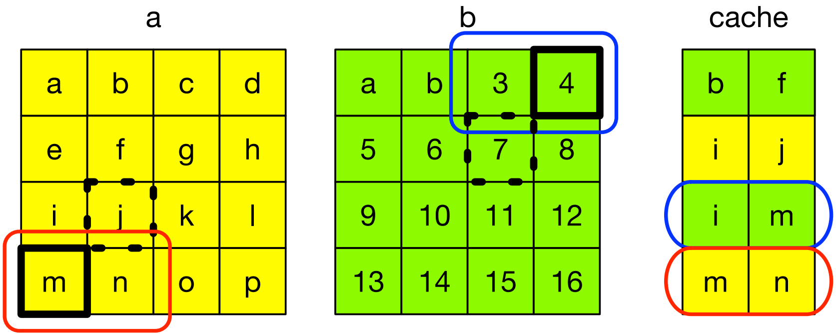
b[1][1]=a[1][1]

Transpose illustrated 5



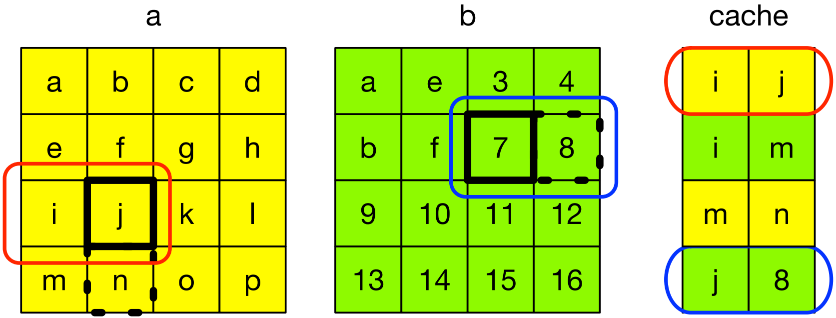
b[2][0]=a[0][2]

Transpose illustrated 6



b[3][0]=a[0][3]

Transpose illustrated 7



b[3][1]=a[1][3]

Experiment

- Transpose of $10{,}000 \times 10{,}000$ matrix, naive code: running time $=$811ms $\pm$ 17
- Transpose of $10{,}000 \times 10{,}000$ matrix, tiled code, tile size $=$ 50: running time $=$ 182 $\pm$ 10

Reuse distance – measure of temporal locality

- *Reuse distance* for an access to cache line v: How long since v was last accessed.
- More precisely: how many different lines were accessed since last access to v.

Theorem

If the cache is fully associative then the access to v is a miss if and only if the reuse distance of v is larger than the cache size

More convenient to discuss reuse distance for variable accesses (rather than lines).

Matrix-Vector Multiply

$y = y + A * x$

```
for(i=0;i<N;i++)
  for(j=0;j<N;j++)
    y[i]=y[i]+A[i][j]*x[j]
```

Matrix-Vector Multiply

$y = y + A * x$

```
for(i=0;i<N;i++)
  for(j=0;j<N;j++)
    y[i]=y[i]+A[i][j]*x[j]
```

- Number of arithmetic operations: $2N^2$

Matrix-Vector Multiply

$y = y + A * x$

```
// Read vectors y and x into cache
for(i=0;i<N;i++)
  // Read a row i of A into cache
  for(j=0;j<N;j++)
    y[i]=y[i]+A[i][j]*x[j]
// Write back vector y
```

Matrix-Vector Multiply

$y = y + A * x$

```
// Read vectors y and x into cache
for(i=0;i<N;i++)
  // Read a row i of A into cache
  for(j=0;j<N;j++)
    y[i]=y[i]+A[i][j]*x[j]
// Write back vector y
```

- Number of arithmetic operations: $2N^2$

Matrix-Vector Multiply

$y = y + A * x$

```
// Read vectors y and x into cache
for(i=0;i<N;i++)
  // Read a row i of A into cache
  for(j=0;j<N;j++)
    y[i]=y[i]+A[i][j]*x[j]
// Write back vector y
```

- Number of arithmetic operations: $2N^2$
- Number of memory refs:

Matrix-Vector Multiply

$y = y + A * x$

```
// Read vectors y and x into cache
for(i=0;i<N;i++)
  // Read a row i of A into cache
  for(j=0;j<N;j++)
    y[i]=y[i]+A[i][j]*x[j]
// Write back vector y
```

- Number of arithmetic operations: $2N^2$
- Number of memory refs:
  - $3N + N^2$

Matrix-Vector Multiply

$y = y + A * x$

```
// Read vectors y and x into cache
for(i=0;i<N;i++)
  // Read a row i of A into cache
  for(j=0;j<N;j++)
    y[i]=y[i]+A[i][j]*x[j]
// Write back vector y
```

- Number of arithmetic operations: $2N^2$
- Number of memory refs:
  - $3N + N^2$
- CI: 2

Matrix-vector

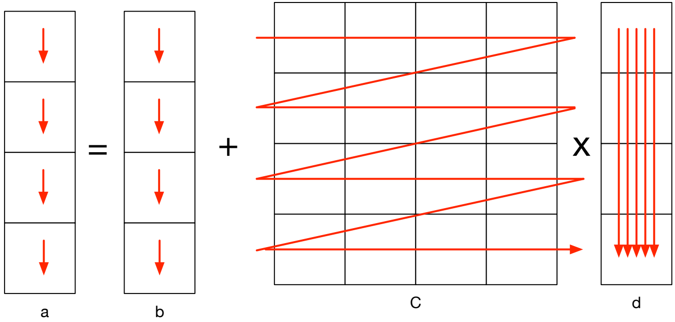
```
for(i=0;i<M;i++) {
 a[i]=b[i];
  for(j=0;j<N;j++)
   a[i]=a[i]+C[i][j]*d[j];
 }
```

Compiler will optimize and keep temp in register

```
for(i=0;i<M;i++) {
 temp=b[i];
  for(j=0;j<N;j++)
   temp+=C[i][j]*d[j];
  a[i]=tmp;
 }
```

Compiler will generate Fused Multiply-Add (FMA) operations (a=b*c+d))

$$a = b + C \times d$$



Naive Matrix Multiply

$C = C + A * B$

```
for(i=0;i<N;i++)
  for(j=0;j<N;j++)
    for(k=0;k<N;k++)
      C[i][j]=C[i][j]+A[i][k]*B[k][j]
```

Naive Matrix Multiply

$C = C + A * B$

```
for(i=0;i<N;i++)
  // Read row i of A into cache
  for(j=0;j<N;j++)
    // Read column j of B into cache
    // Read element C[i][j] into cache
    for(k=0;k<N;k++)
      C[i][j]=C[i][j]+A[i][k]*B[k][j]
    // Update C[i][j] in memory
```

Naive Matrix Multiply

$C = C + A * B$

```
for(i=0;i<N;i++)
  // Read row i of A into cache
  for(j=0;j<N;j++)
    // Read column j of B into cache
    // Read element C[i][j] into cache
    for(k=0;k<N;k++)
      C[i][j]=C[i][j]+A[i][k]*B[k][j]
    // Update C[i][j] in memory
```

- Number of arithmetic operations:
  - $2N^3$
- Number of memory refs:

Naive Matrix Multiply

$C = C + A * B$

```
for(i=0;i<N;i++)
  // Read row i of A into cache
  for(j=0;j<N;j++)
    // Read column j of B into cache
    // Read element C[i][j] into cache
    for(k=0;k<N;k++)
      C[i][j]=C[i][j]+A[i][k]*B[k][j]
    // Update C[i][j] in memory
```

- Number of arithmetic operations:
  - $2N^3$
- Number of memory refs:
  - Read each row of A once:

Naive Matrix Multiply

$C = C + A * B$

```
for(i=0;i<N;i++)
  // Read row i of A into cache
  for(j=0;j<N;j++)
    // Read column j of B into cache
    // Read element C[i][j] into cache
    for(k=0;k<N;k++)
      C[i][j]=C[i][j]+A[i][k]*B[k][j]
    // Update C[i][j] in memory
```

- Number of arithmetic operations:
  - $2N^3$
- Number of memory refs:
  - Read each row of A once:
    - N^2

Naive Matrix Multiply

$C = C + A * B$

```
for(i=0;i<N;i++)
  // Read row i of A into cache
  for(j=0;j<N;j++)
    // Read column j of B into cache
    // Read element C[i][j] into cache
    for(k=0;k<N;k++)
      C[i][j]=C[i][j]+A[i][k]*B[k][j]
    // Update C[i][j] in memory
```

- Number of arithmetic operations:
  - $2N^3$
- Number of memory refs:
  - Read each row of A once:
    - N^2
  - Read and write each element of C once:

Naive Matrix Multiply

$C = C + A * B$

```
for(i=0;i<N;i++)
  // Read row i of A into cache
  for(j=0;j<N;j++)
    // Read column j of B into cache
    // Read element C[i][j] into cache
    for(k=0;k<N;k++)
      C[i][j]=C[i][j]+A[i][k]*B[k][j]
    // Update C[i][j] in memory
```

- Number of arithmetic operations:
  - $2N^3$
- Number of memory refs:
  - Read each row of A once:
    - N^2
  - Read and write each element of C once:
    - $2N^2$

Naive Matrix Multiply

$C = C + A * B$

```
for(i=0;i<N;i++)
  // Read row i of A into cache
  for(j=0;j<N;j++)
    // Read column j of B into cache
    // Read element C[i][j] into cache
    for(k=0;k<N;k++)
      C[i][j]=C[i][j]+A[i][k]*B[k][j]
    // Update C[i][j] in memory
```

- Number of arithmetic operations:
  - $2N^3$
- Number of memory refs:
  - Read each row of A once:
    - N^2
  - Read and write each element of C once:
    - $2N^2$
  - Read each column of B ? times:

Naive Matrix Multiply

$C = C + A * B$

```
for(i=0;i<N;i++)
  // Read row i of A into cache
  for(j=0;j<N;j++)
    // Read column j of B into cache
    // Read element C[i][j] into cache
    for(k=0;k<N;k++)
      C[i][j]=C[i][j]+A[i][k]*B[k][j]
    // Update C[i][j] in memory
```

- Number of arithmetic operations:
  - $2N^3$
- Number of memory refs:
  - Read each row of A once:
    - N^2
  - Read and write each element of C once:
    - $2N^2$
  - Read each column of B ? times:
    - N^3

Naive Matrix Multiply

$C = C + A * B$

```
for(i=0;i<N;i++)
  // Read row i of A into cache
  for(j=0;j<N;j++)
    // Read column j of B into cache
    // Read element C[i][j] into cache
    for(k=0;k<N;k++)
      C[i][j]=C[i][j]+A[i][k]*B[k][j]
    // Update C[i][j] in memory
```

- Number of arithmetic operations:
  - $2N^3$
- Number of memory refs:
  - Read each row of A once:
    - N^2
  - Read and write each element of C once:
    - $2N^2$
  - Read each column of B ? times:
    - N^3
- CI: 2

Blocked/Tiled Matrix Multiply

$C = C + A * B$ N-by-N matrices of b-by-b subblocks (b = n / N)

```
for(i=0;i<N;i++)
  for(j=0;j<N;j++)
    // Read block C[i][j] into cache
    for(k=0;k<N;k++)
      // Read block A[i][k] into cache
      // Read block B[k][j] into cache
      // Matrix multiply of blocks
      C[i][j]=C[i][j]+A[i][k]*B[k][j]
// write back block C[i][j]
```

Blocked/Tiled Matrix Multiply

$C = C + A * B$ N-by-N matrices of b-by-b subblocks (b = n / N)

- block of B

Blocked/Tiled Matrix Multiply

$C = C + A * B$ N-by-N matrices of b-by-b subblocks (b = n / N)

- block of B
  - $N * N * N = N^3$

Blocked/Tiled Matrix Multiply

$C = C + A * B$ N-by-N matrices of b-by-b subblocks (b = n / N)

- block of B
  - $N * N * N = N^3$
  - block size: b^2

Blocked/Tiled Matrix Multiply

$C = C + A * B$ N-by-N matrices of b-by-b subblocks (b = n / N)

- block of B
  - $N * N * N = N^3$
  - block size: b^2
  - $N^3 * b^2 = N^3 * (n/N)^2 = N * n^2$

Blocked/Tiled Matrix Multiply

$C = C + A * B$ N-by-N matrices of b-by-b subblocks (b = n / N)

- block of B
  - $N * N * N = N^3$
  - block size: b^2
  - $N^3 * b^2 = N^3 * (n/N)^2 = N * n^2$
- block of A

Blocked/Tiled Matrix Multiply

$C = C + A * B$ N-by-N matrices of b-by-b subblocks (b = n / N)

- block of B
  - $N * N * N = N^3$
  - block size: b^2
  - $N^3 * b^2 = N^3 * (n/N)^2 = N * n^2$
- block of A
  - $N * n^2$

Blocked/Tiled Matrix Multiply

$C = C + A * B$ N-by-N matrices of b-by-b subblocks (b = n / N)

- block of B
  - $N * N * N = N^3$
  - block size: b^2
  - $N^3 * b^2 = N^3 * (n/N)^2 = N * n^2$
- block of A
  - $N * n^2$
- block of C

Blocked/Tiled Matrix Multiply

$C = C + A * B$ N-by-N matrices of b-by-b subblocks (b = n / N)

- block of B
  - $N * N * N = N^3$
  - block size: b^2
  - $N^3 * b^2 = N^3 * (n/N)^2 = N * n^2$
- block of A
  - $N * n^2$
- block of C
  - $2N^2 * b^2 = 2n^2$

Blocked/Tiled Matrix Multiply

$C = C + A * B$ N-by-N matrices of b-by-b subblocks (b = n / N)

- block of B
  - $N * N * N = N^3$
  - block size: b^2
  - $N^3 * b^2 = N^3 * (n/N)^2 = N * n^2$
- block of A
  - $N * n^2$
- block of C
  - $2N^2 * b^2 = 2n^2$
- total: $(2N + 2) * n^2$

Blocked/Tiled Matrix Multiply

$C = C + A * B$ N-by-N matrices of b-by-b subblocks (b = n / N)

- block of B
  - $N * N * N = N^3$
  - block size: b^2
  - $N^3 * b^2 = N^3 * (n/N)^2 = N * n^2$
- block of A
  - $N * n^2$
- block of C
  - $2N^2 * b^2 = 2n^2$
- total: $(2N + 2) * n^2$
- CI $= 2n^3/((2N + 2) * n^2) \approx n/N = b$

- Previous discussion assumed one cache level; how do we deal with multiple cache levels?
- Usually, memory is main bottleneck; tile so as to reduce memory accesses.
- Often sufficient to tile for L2 locality
- But can do hierarchical tiling: E.g., for transpose, can transpose 8×8 in vector registers (load 8 vectors of 8 words; transpose in registres; store 8 vectors)
- For Matrix$\times$Matrix can use recursive algorithm that tiles for successive cache levels