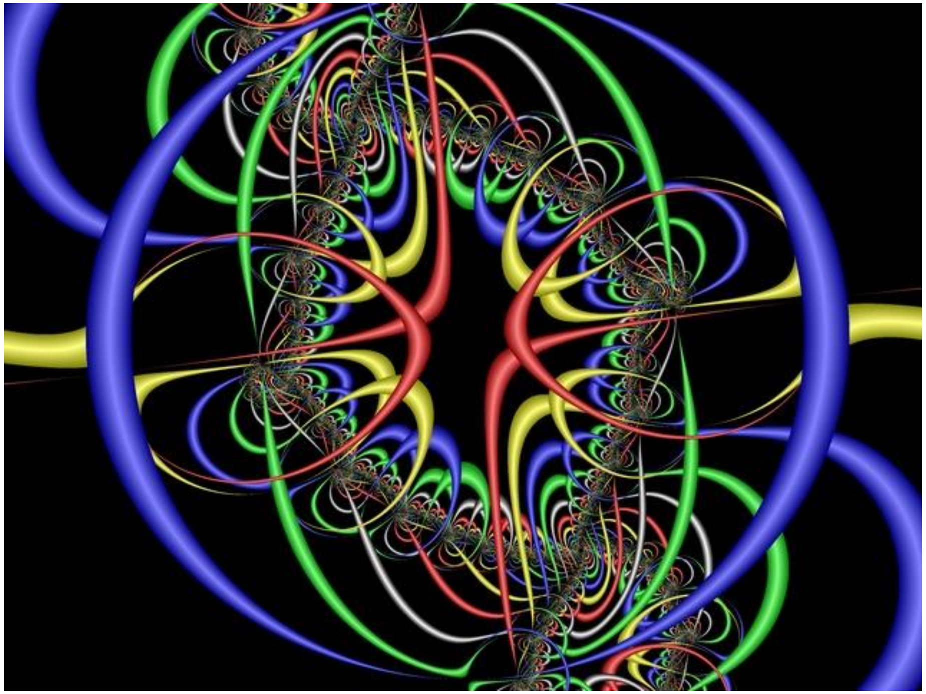
cse5441 - parallel computing

loop analysis and transformation



http://www.fractaldomains.com/site/wp-content/uploads/2011/08/complex-loops.jpg

locality

review

- clustering references to individual variables is good
  (temporal locality)

- stride-1 reference patterns are good
  (spatial locality)

matrix summation

let:

- cold cache, sizeof(int) = 4 bytes, 16-byte cache blocks
- fully associative LRU cache
- C = sizeof(int)(M)(N) (assume M, N $\geq$ 64)
- sum, i, j in registers

```
int  sumarrayrows(int a[M][N])
{
    int  sum = 0;

    for (int  i = 0;  i < M;  i++)
        for (int  j = 0; j < N; j++)

            sum += a[ i ][ j ]

    return sum;
}
```

```
int  sumarraycols(int a[M][N])
{
    int  sum = 0;

    for (int  j = 0;  j < N;  j++)
        for (int  i = 0; i < M; i++)

            sum += a[ i ][ j ]

    return sum;
}
```

matrix summation

let:

- cold cache, sizeof(int) = 4 bytes, 16-byte cache blocks
- fully associative LRU cache
- $C = (.5)(M)(N)$ (assume $M, N \geq 64$)
- sum in register

```
int  sumarrayrows(int a[M][N])
{
   int  sum = 0;

   for (int  i = 0;  i < M;  i++)
      for (int  j = 0; j < N; j++)

         sum += a[ i ][ j ]

   return sum;
}
```

```
int  sumarraycols(int a[M][N])
{
   int  sum = 0;

   for (int  j = 0;  j < N;  j++)
      for (int  i = 0; i < M; i++)

         sum += a[ i ][ j ]

   return sum;
}
```

matrix summation

let:

- cold cache, sizeof(int) = 4 bytes, 16-byte cache blocks
- fully associative LRU cache
- C < (.5)(M)(B) (assume M, N ≥ 64)
- sum in register

```
int  sumarrayrows(int a[M][N])
{
    int  sum = 0;

    for (int  i = 0;  i < M;  i++)
        for (int  j = 0; j < N; j++)

            sum += a[ i ][ j ]

    return sum;
}
```

```
int  sumarraycols(int a[M][N])
{
    int  sum = 0;

    for (int  j = 0;  j < N;  j++)
        for (int  i = 0; i < M; i++)

            sum += a[ i ][ j ]

    return sum;
}
```

it's your turn . . .

let:

- cold cache, sizeof(int) = 4 bytes, 32-byte cache blocks
- fully associative LRU cache
- C = 4(N)(P) (in bytes)
- matrix allocated as contiguous memory (one "new")
- P % 8 = 0, P < M < NP/8
- assume N, M, P > trivial

```
int  summat_1(int  a[M][N][P])
{
   int  sum = 0;        // register

   for (int  i = 0;  i < M;  i++)
      for (int  j = 0; j < N; j++)
         for (int k = 0; k < P; k++)

            sum += a[ i ][ j ][ k ]

   return sum;
}
```

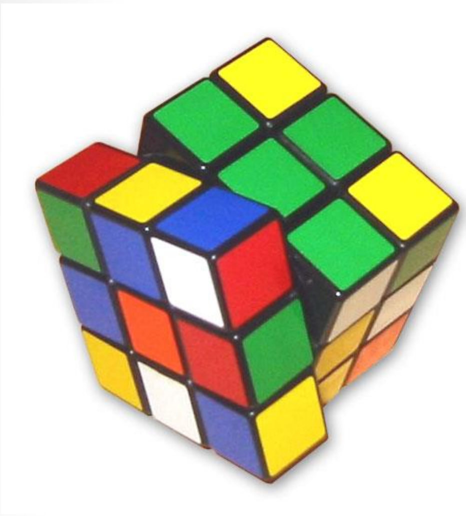
```
int  summat_2(int  a[M][N][P])
{
   int  sum = 0;        // register

   for (int k = 0; k < P; k++)
      for (int  j = 0;  j < N;  j++)
         for (int  i = 0; i < M; i++)

            sum += a[ i ][ j ][ k ]

   return sum;
}
```

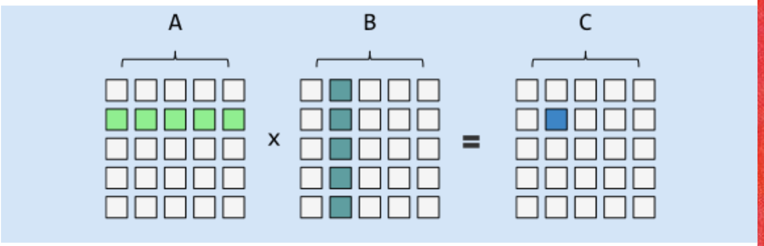
what are the access stride, hit rate and cache contents at end of loop nest?

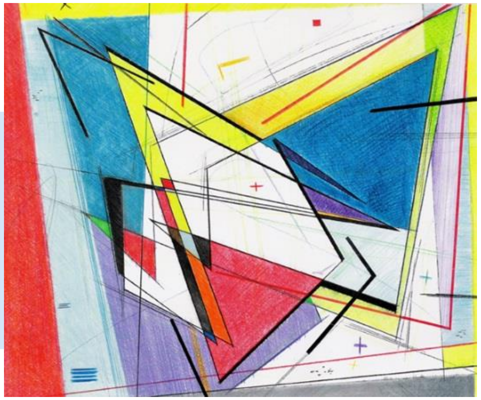
OSU CSE 5441

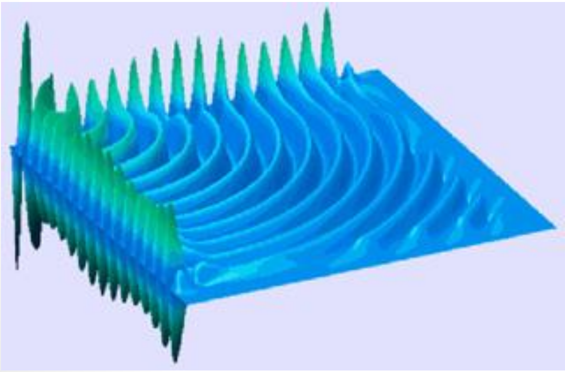
J. S. Jones

matrix multiplication









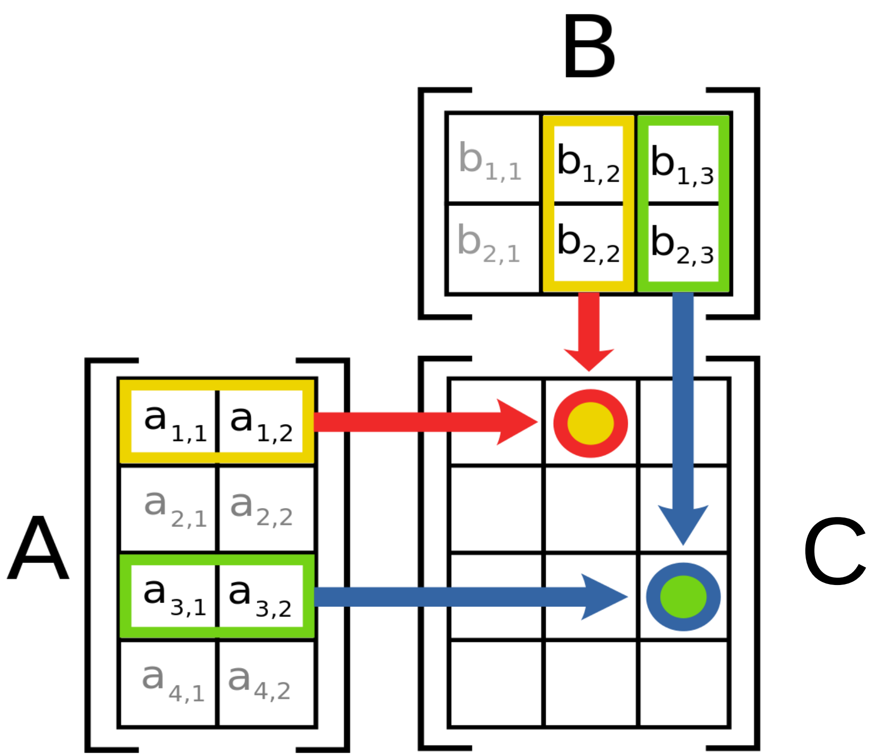
```
{{-0.4 X02_MPS - 0.32 X14_MPS - 0.6 X23_MPS - 0.48 X36_MPS + 10. X39_MPS ,
  -1. X01_MPS + 1. X02_MPS + 1. X03_MPS == 0. && -1.06 X01_MPS + 1. X04_MPS == 0. &&
   1. X01_MPS ≤ 80. && -1. X02_MPS + 1.4 X14_MPS ≤ 0. &&
   -1. X06_MPS - 1. X07_MPS - 1. X08_MPS - 1. X09_MPS + 1. X14_MPS + 1. X15_MPS == 0. &&
   -1.06 X06_MPS - 1.06 X07_MPS - 0.96 X08_MPS - 0.86 X09_MPS + 1. X16_MPS == 0. &&
   1. X06_MPS - 1. X10_MPS ≤ 80. && 1. X07_MPS - 1. X11_MPS ≤ 0. && 1. X08_MPS - 1. X12_MPS ≤ 0. &&
   1. X09_MPS - 1. X13_MPS ≤ 0. && -1. X22_MPS + 1. X23_MPS + 1. X24_MPS + 1. X25_MPS == 0. &&
   -0.43 X22_MPS + 1. X26_MPS == 0. && 1. X22_MPS ≤ 500. && -1. X23_MPS + 1.4 X36_MPS ≤ 0. &&
   -0.43 X28_MPS - 0.43 X29_MPS - 0.39 X30_MPS - 0.37 X31_MPS + 1. X38_MPS == 0. &&
   1. X28_MPS + 1. X29_MPS + 1. X30_MPS + 1. X31_MPS - 1. X36_MPS + 1. X37_MPS + 1. X39_MPS == 44. &&
   1. X28_MPS - 1. X32_MPS ≤ 500. && 1. X29_MPS - 1. X33_MPS ≤ 0. &&
   1. X30_MPS - 1. X34_MPS ≤ 0. && 1. X31_MPS - 1. X35_MPS ≤ 0. &&
   2.364 X10_MPS + 2.386 X11_MPS + 2.408 X12_MPS + 2.429 X13_MPS - 1. X25_MPS + 2.191 X32_MPS +
      2.219 X33_MPS + 2.249 X34_MPS + 2.279 X35_MPS ≤ 0. && -1. X03_MPS + 0.109 X22_MPS ≤ 0. &&
   -1. X15_MPS + 0.109 X28_MPS + 0.108 X29_MPS + 0.108 X30_MPS + 0.107 X31_MPS ≤ 0. &&
   0.301 X01_MPS - 1. X24_MPS ≤ 0. &&
   0.301 X06_MPS + 0.313 X07_MPS + 0.313 X08_MPS + 0.326 X09_MPS - 1. X37_MPS ≤ 0. &&
   1. X04_MPS + 1. X26_MPS ≤ 310. && 1. X16_MPS + 1. X38_MPS ≤ 300. && X01_MPS ≥ 0 && X02_MPS ≥ 0 &&
   X03_MPS ≥ 0 && X04_MPS ≥ 0 && X06_MPS ≥ 0 && X07_MPS ≥ 0 && X08_MPS ≥ 0 && X09_MPS ≥ 0 &&
   X10_MPS ≥ 0 && X11_MPS ≥ 0 && X12_MPS ≥ 0 && X13_MPS ≥ 0 && X14_MPS ≥ 0 && X15_MPS ≥ 0 &&
   X16_MPS ≥ 0 && X22_MPS ≥ 0 && X23_MPS ≥ 0 && X24_MPS ≥ 0 && X25_MPS ≥ 0 && X26_MPS ≥ 0 &&
   X28_MPS ≥ 0 && X29_MPS ≥ 0 && X30_MPS ≥ 0 && X31_MPS ≥ 0 && X32_MPS ≥ 0 && X33_MPS ≥ 0 &&
   X34_MPS ≥ 0 && X35_MPS ≥ 0 && X36_MPS ≥ 0 && X37_MPS ≥ 0 && X38_MPS ≥ 0 && X39_MPS ≥ 0},
 {X01_MPS , X02_MPS , X03_MPS , X04_MPS , X06_MPS , X07_MPS , X08_MPS , X09_MPS , X10_MPS ,
  X11_MPS , X12_MPS , X13_MPS , X14_MPS , X15_MPS , X16_MPS , X22_MPS , X23_MPS ,
  X24_MPS , X25_MPS , X26_MPS , X28_MPS , X29_MPS , X30_MPS , X31_MPS , X32_MPS ,
  X33_MPS , X34_MPS , X35_MPS , X36_MPS , X37_MPS , X38_MPS , X39_MPS }}
```

http://www.stoimen.com/blog/wp-content/uploads/2012/11/3.-Matrix-Multiplication.png
https://reference.wolfram.com/language/tutorial/Files/ConstrainedOptimizationLinearProgramming.en/O_19.gif
http://chemwiki.ucdavis.edu/@api/deki/files/9875/Rubiks_cube.jpg?size=bestfit&width=255&height=220&revision=1
http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-728-applied-quantum-and-statistical-physics-fall-2006/6-728f06.jpg
https://www.google.com/search?q=illustration&tbm=isch&tbo=u&source=univ&sa=X&ved=0CFsQsARqFQoTCI-y2Jvm-8YCFRZ9iAodxTgNWQ&biw=1920&bih=969#tbm=isch&q=linerar+transformation&imgrc=TlIPHaBdabGJoM%3A

matrix multiplication

C = A x B

Cij = $\sum_i \sum_j \sum_k \; Aik * Bkj$



matrix multiplication

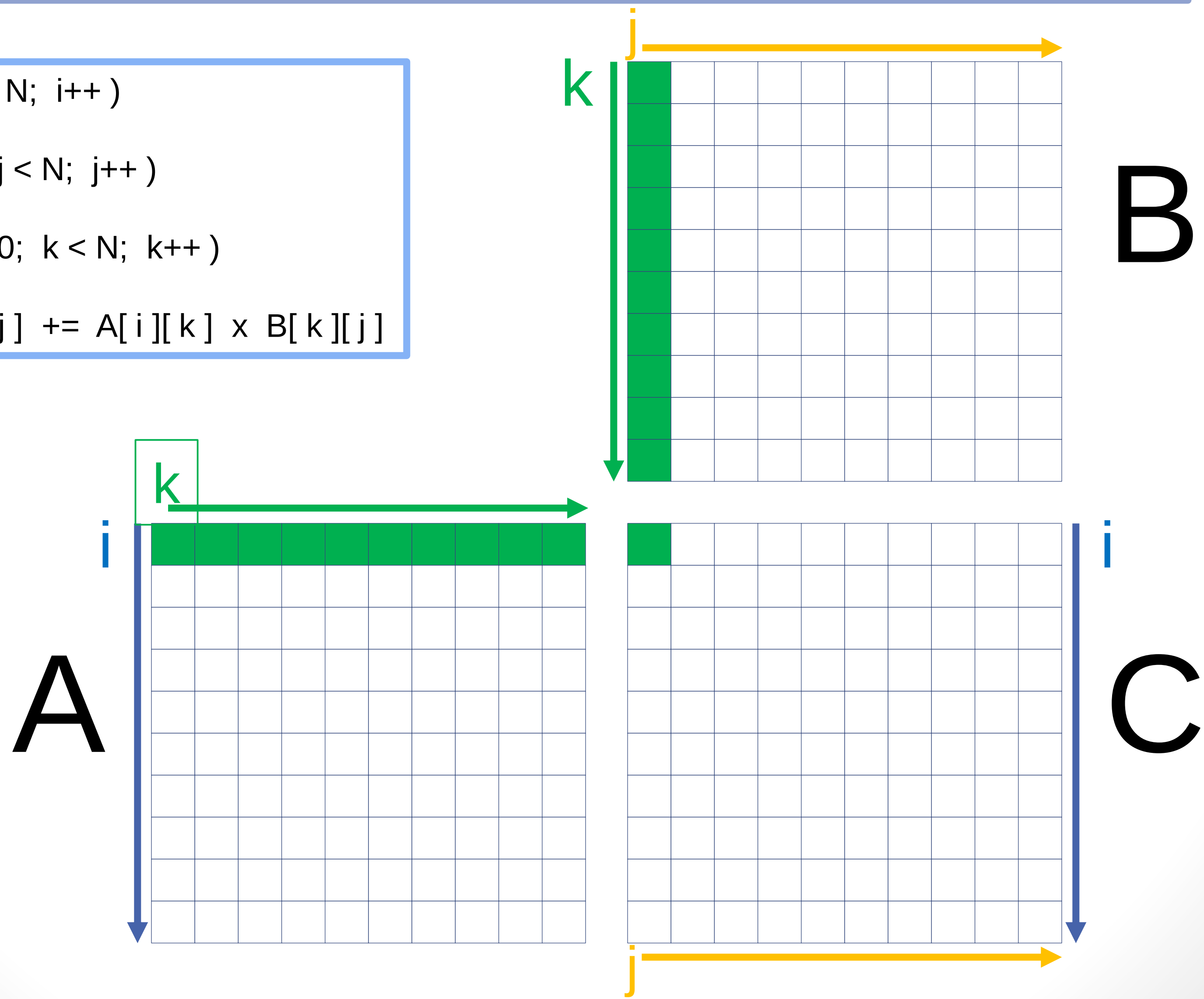
C = A x B

Cij = $\sum_i \sum_j \sum_k \; Aik \; * Bkj$

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

matrix multiplication

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



m-mult cache performance

inner-loop analysis method

I J K

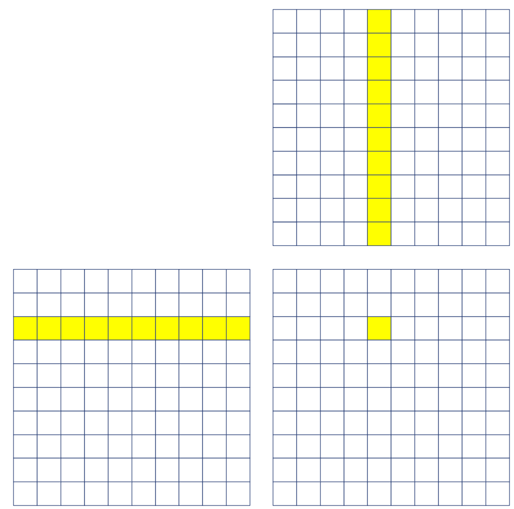
```
for ( i = 0;  i < N;  i++ )
    for ( j = 0;  j < N;  j++ )
        for ( k = 0;  k < N;  k++ )
            C[ i ][ j ]  +=  A[ i ][ k ]  x  B[ k ][ j ]
```

let: N is very large, $1/N \approx 0$
$N >> E$
fully associative cache
LRU
B = 4 * sizeof(int)

consider access pattern of inner loop:

- **k** is iterating
- what is stride of A?
- what is stride of B?
- what is stride of C?

A		B		C		
0.25	+	1.00	+	0.00	=	1.25



inner-loop iteration pattern

this is the “misses per iteration” of the inner loop

matrix multiplication

C = A x B

$$Cij = \sum_i \sum_j \sum_k \; Aik \, * Bkj$$

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

matrix multiplication

disclaimer

WLOG, all examples are integer

m-mult cache performance

JKI

J
K
I

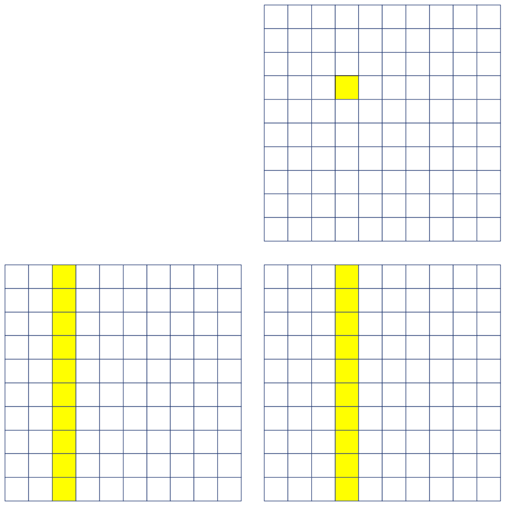
```
for ( j = 0;  j < N;  j++ )
   for ( k = 0;  k < N;  k++ )
      for ( i = 0;  i < N;  i++ )
         C[ i ][ j ]  +=  A[ i ][ k ]  x  B[ k ][ j ]
```

let: N is very large, $1/N \approx 0$
N >> E
fully associative cache
LRU
B = 4 * sizeof(int)

consider access pattern of inner loop:

- i is iterating
- what is stride of A?
- what is stride of B?
- what is stride of C?

A		B		C		
1.00	+	0.00	+	1.00	=	2.00



inner-loop iteration pattern

m-mult cache performance

IKJ

I
K
J

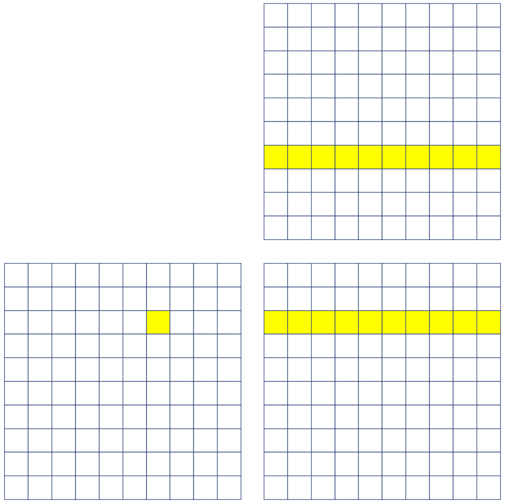
```
for ( i = 0;  i < N;  i++ )
    for ( k = 0;  k < N;  k++ )
        for ( j = 0;  j < N;  j++ )
            C[ i ][ j ]  +=  A[ i ][ k ]  x  B[ k ][ j ]
```

let: N is very large, 1/N ≈ 0
N >> E
fully associative cache
LRU
B = 4 * sizeof(int)

consider access pattern of inner loop:

- j is iterating
- what is stride of A?
- what is stride of B?
- what is stride of C?

A		B		C		
0.00	+	0.25	+	0.25	=	0.50



inner-loop iteration pattern

m-mult cache performance

summary

let: N is very large, 1/N ≈ 0
N >> E
fully associative cache
LRU
B = 4 * sizeof(float)

IJK

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

A		B		C		
0.25	+	1.00	+	0.00	=	1.25

JKI

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

A		B		C		
1.00	+	0.00	+	1.00	=	2.00

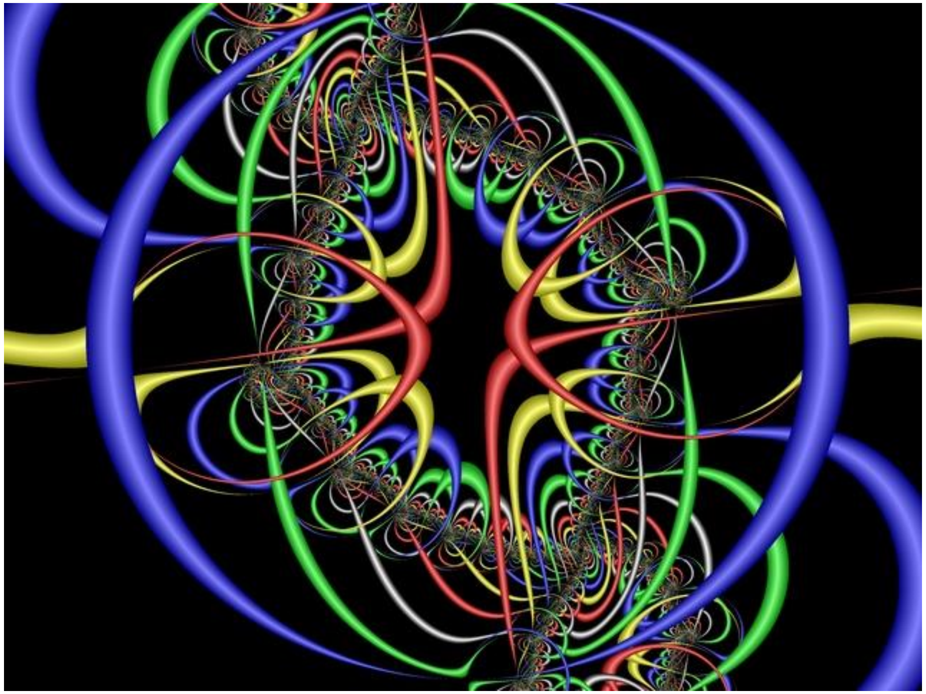
IKJ

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

A		B		C		
0.00	+	0.25	+	0.25	=	0.50

cse5441 - parallel computing

loop analysis and transformation



http://www.fractaldomains.com/site/wp-content/uploads/2011/08/complex-loops.jpg

m-mult cache performance

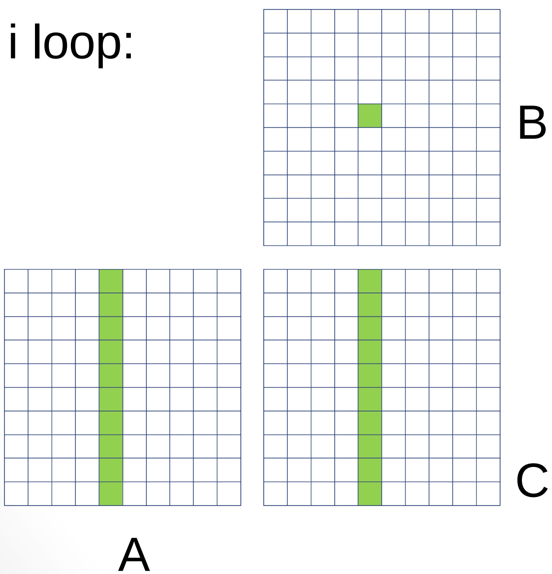
total miss analysis method

I J K

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

let: N is very large, $1/N \approx 0$
$N >> 4E$
fully associative cache
LRU
$B = 4 * \text{sizeof(int)}$
all register variables are held in cache

i loop:



	A	B	C
I	N	N	N
J	N	N	$\frac{N}{B}$
K	$\frac{N}{B}$	N	1
	$\frac{N^3}{B}$	N^3	$\frac{N^2}{B}$

m-mult cache performance

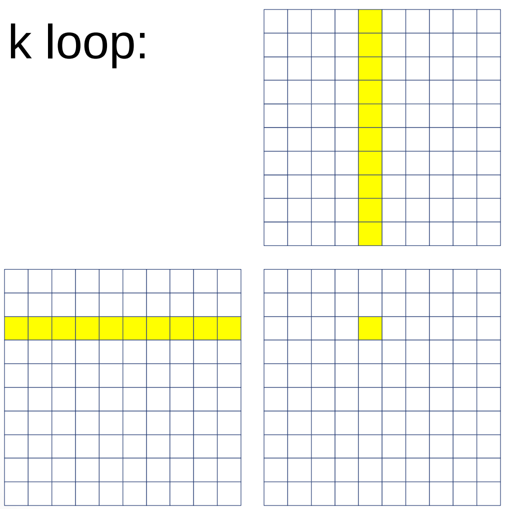
total miss analysis method

J I K

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

let: N is very large, $1/N \approx 0$
N >> 4E
fully associative cache
LRU
B = 4 * sizeof(int)
all register variables are held in cache

k loop:



	A	B	C
J	N	N	N
I	N	N	N
K	$\frac{N}{B}$	N	1
	$\frac{N^3}{B}$	N^3	N^2

m-mult cache performance

total miss analysis method, example 2 (IJK)

```
I  for ( i = 0;  i < N;  i++ )
J     for ( j = 0;  j < N;  j++ )
K        for ( k = 0;  k < N;  k++ )
            C[ i ][ j ]  +=  A[ i ][ k ]  x  B[ k ][ j ]
```

let: $C = (N^2 + 4N + \mathcal{E}) *$ sizeof(element)

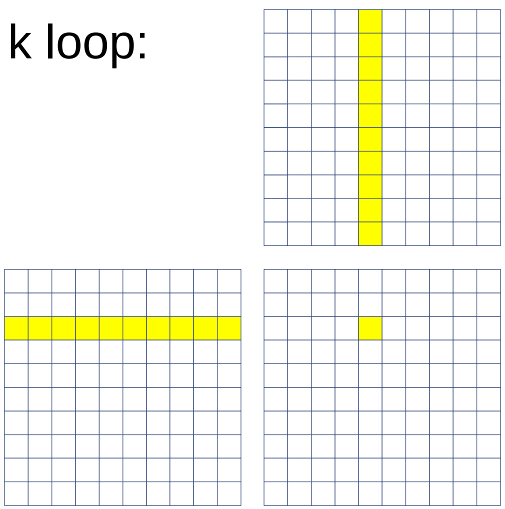
fully associative cache

LRU

$B = 4 *$ sizeof(element)

all register variables are held in cache

k loop:



	A	B	C
I	N	1	N
J	1	$\frac{N}{B}$	$\frac{N}{B}$
K	$\frac{N}{B}$	N	1
	$\frac{N^2}{B}$	$\frac{N^2}{B}$	$\frac{N^2}{B}$

it's your turn ...

```
for ( ? = 0;  ? < N;  ?++ )

    for ( ? = 0;  ? < N;  ?++ )

        for ( ? = 0;  ? < N;  ?++ )

            C[ i ][ j ]  +=  A[ i ][ k ]  x  B[ k ][ j ]
```

let: $C = 3N * _{\text{sizeof(element)}} + \varepsilon$

fully associative cache

LRU

$B = 8 * _{\text{sizeof(element)}}$

- what is an optimal loop ordering?
- what is the hit rate of your solution?
- what analysis method did you use?

loop permutation

summary

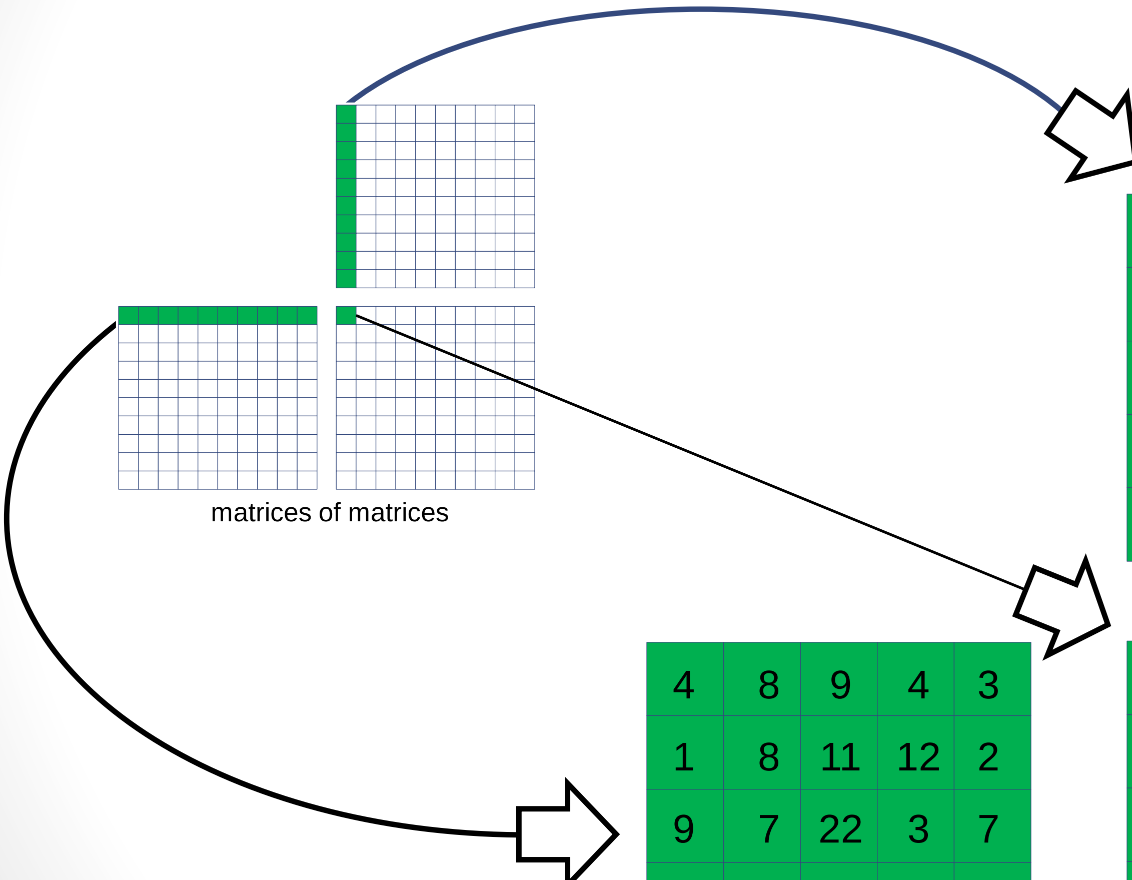
cache-friendly code will:

- consider loop permutations
- strive for stride-1 access
  (or at least <B)
- minimize long strides

variation in loop permutation:

- affect the cache miss ratio
- affect the minimum cache needed
  for optimal performance

blocking (tiling)



B[0][0]

9	7	22	13	37
5	7	27	6	8
7	6	5	0	9
4	8	12	0	3
2	5	5	7	8

C[0][0]

1	8	11	12	32
9	7	22	3	7
5	7	27	6	8
7	6	5	0	9
4	8	12	0	3

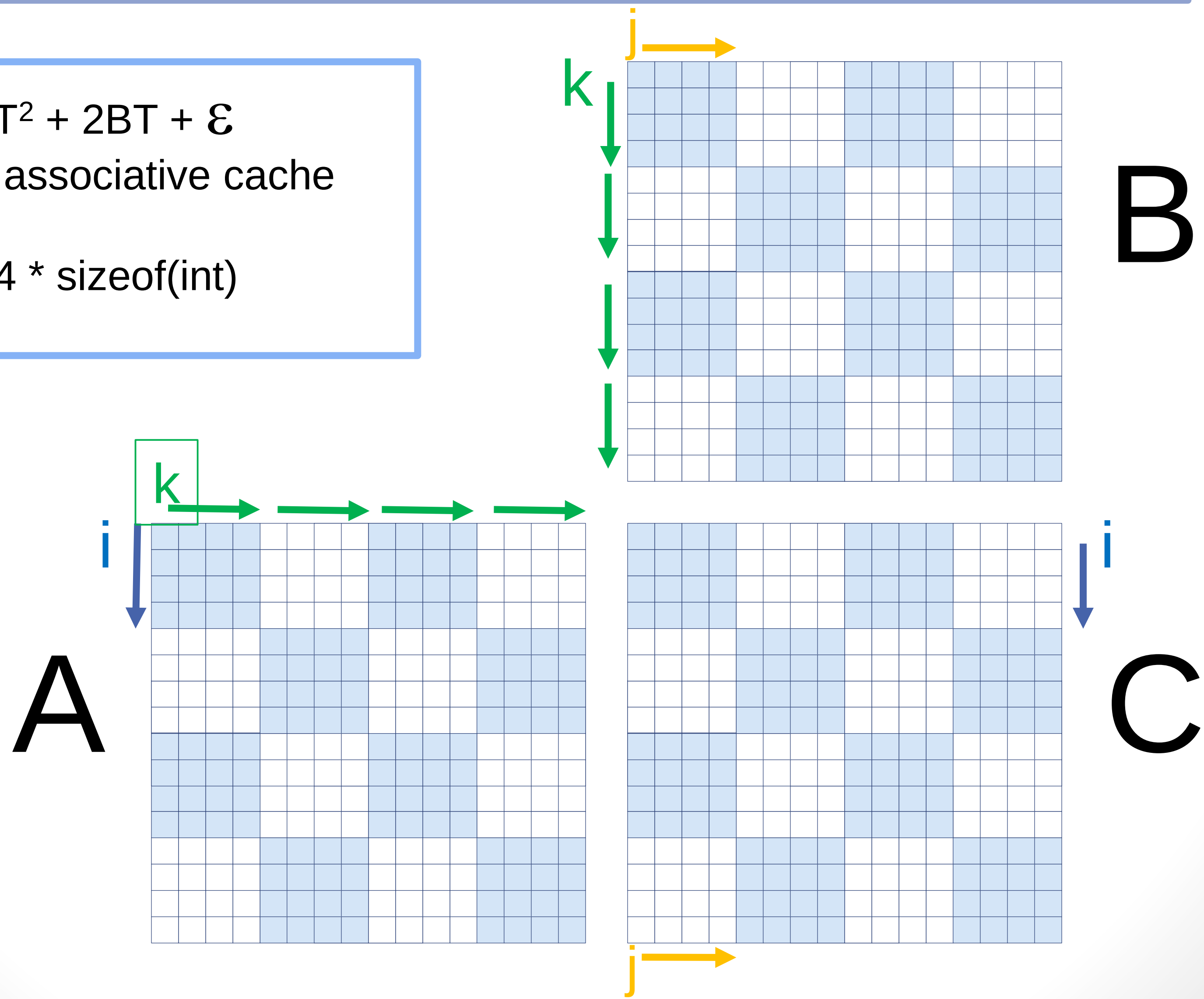
A[0][0]

4	8	9	4	3
1	8	11	12	2
9	7	22	3	7
5	7	27	6	8
7	6	5	0	9

blocking (tiling)

let: $C = T^2 + 2BT + \varepsilon$
fully associative cache
LRU
B = 4 * sizeof(int)

IJK loop



it's your turn ...

I
K
J

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

- N = 2000
- sizeof(int) = 4
- b = 5
- E = 1
- C = 16k (4096 elements)
- A[i][k] moved to register (temporal locality)

what tile size would you use?

inner loop unrolling

ORIGINAL

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

UNROLLED (for simplicity, let j % 4 = 0)

```
for ( i = 0;  i < n;  i++ )
    for ( j = 0;  j < n;  j += 4)
        y[i] = y[i] + a[ i ][ j ] * x[ j ];
        y[i] = y[i] + a[ i ][ j+1 ] * x[ j+1 ];
        y[i] = y[i] + a[ i ][ j+2 ] * x[ j+2 ];
        y[i] = y[i] + a[ i ][ j+3 ] * x[ j+3 ];
```

```
for ( i = 0;  i < n;  i++ )
    for ( j = 0;  j < n;  j += 4)
        y[i] = y[i] + a[ i ][ j ] * x[ j ];
                    + a[ i ][ j+1 ] * x[ j+1 ];
                    + a[ i ][ j+2 ] * x[ j+2 ];
                    + a[ i ][ j+3 ] * x[ j+3 ];
```

outer loop unrolling (unroll / jam)

ORIGINAL

```
for ( i = 0;  i < 2n;  i++ )
    for ( j = 0;  j < m;  j++)
        loop-body(j, i)
```

UNROLLED

```
for ( i = 0;  i < 2n;  i += 2 )
    for ( j = 0;  j < m;  j++)
        loop-body(j, i)
    for ( j = 0;  j < m;  j++)
        loop-body(j, i+1)
```

preserves execution order

but doesn't accomplish anything ...

UNROLLED / JAMMED

```
for ( i = 0;  i < 2n;  i += 2 )
    for ( j = 0;  j < m;  j++)
        loop-body(j, i)
        loop-body(j, i+1)
```

unrolling benefits

but ... does not preserve access order

outer loop unrolling (unroll / jam)

example

ORIGINAL

```
for ( i = 0;  i < 4n;  i++ )
    for ( j = 0;  j < 4n;  j++)
        y[ i ] = y[ i ] + a[i ][ j ] * x[ j ]
```

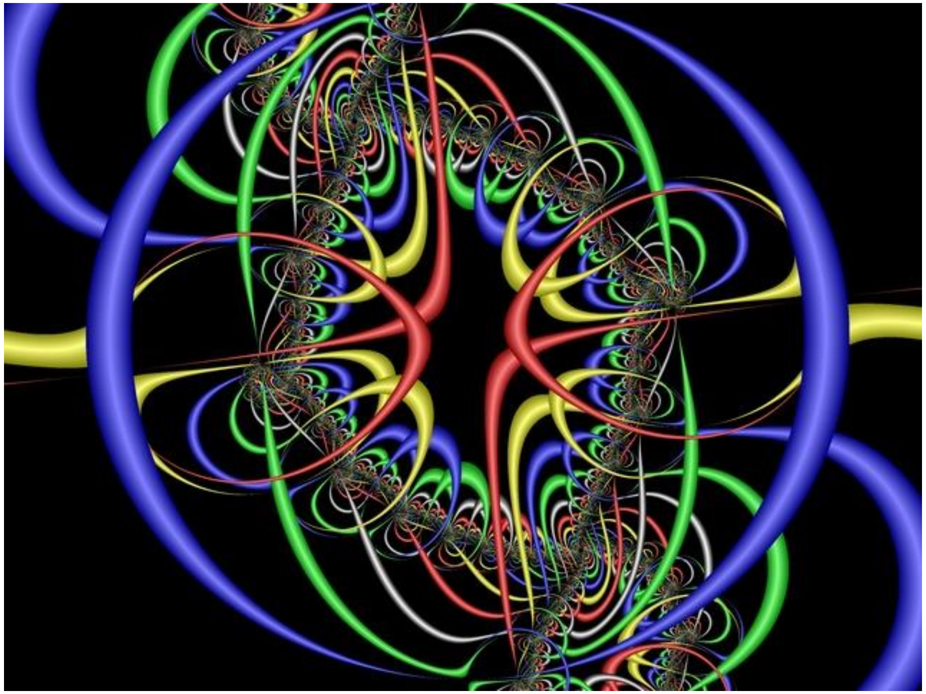
UNROLLED / JAMMED

```
for ( i = 0;  i < 4n;  i += 4 )
    for ( j = 0;  j < 4n;  j++)
        y[ i ]     =  y[ i ]     +  a[ i ][ j ]     * x[ j ]
        y[ i+1 ]  =  y[ i+1 ]  +  a[ i+1 ][ j ] * x[ j ]
        y[ i+2 ]  =  y[ i+1 ]  +  a[ i+2 ][ j ] * x[ j ]
        y[ i+3 ]  =  y[ i+1 ]  +  a[ i+3 ][ j ] * x[ j ]
```

does not preserve access order

cse5441 - parallel computing

loop analysis and transformation



http://www.fractaldomains.com/site/wp-content/uploads/2011/08/complex-loops.jpg

loops - answers

slide 6: summat_1

- access stride is 1 element, or 4 bytes
- hit rate is 7/8 = 87.5%
- a[M-1][*][*] will be in cache

summat_2

- stride is N*P*|element|
- hit rate is 0%
- a[*][(N-(NP/*M)) ... N-1][(P-8)...(P-1)] in cache

loops - answers

slide 21: IKJ will be best, need total miss analysis to differentiate between IKJ and KIJ

inner loop analysis method:

**K: A= .125, B= 1.000, C= 0, total 1.125
**J: A= 0, B= .125, C= .125, total 0.250
**I: A= 1.000, B= 0, C= 1.000, total 2.000

	A	B	C
I	N	N	N
K	$\frac{N}{B}$	N	1
J	1	$\frac{N}{B}$	$\frac{N}{B}$
	$\frac{N^2}{B}$	$\frac{N^3}{B}$	$\frac{N^2}{B}$

	A	B	C
K	N	N	N
I	N	1	N
J	1	$\frac{N}{B}$	$\frac{N}{B}$
	N^2	$\frac{N^2}{B}$	$\frac{N^3}{B}$

loops - answers

slide 25: the loop is already set up optimally $3N^2/B$)

if we allow K and J to iterate, we would need:
A – N elements
B – N^2 elements
C – N elements

so, we pick a tile size such that $T^2 + 2T < C$, also with the restriction that the tile size is a multiple of B / |element| so that we align with the cache blocks (over- or under-running would waste space and could introduce contention)

solving for T can be done numerically or by "guess and test"

$24^2 + 2(24) = 624 < 4096/4$

$32^2 + 2(32) = 1088 \not< 4096/4$