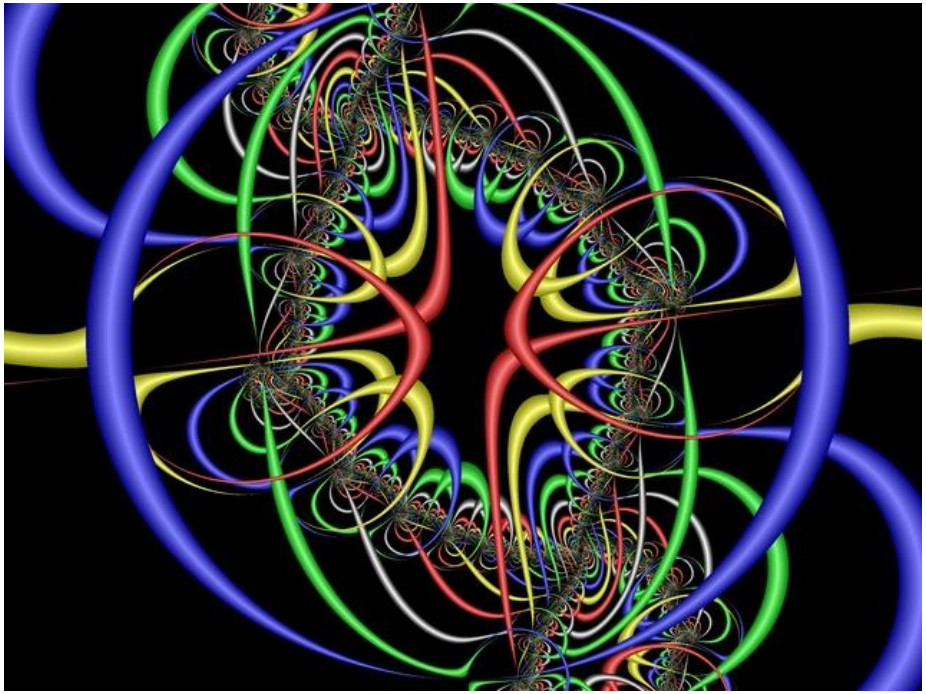
cse5441 - parallel computing

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) (let M = N = 1024)
- 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) (let M = N = 1024)
- 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) (let M = N = 1024)
- 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")

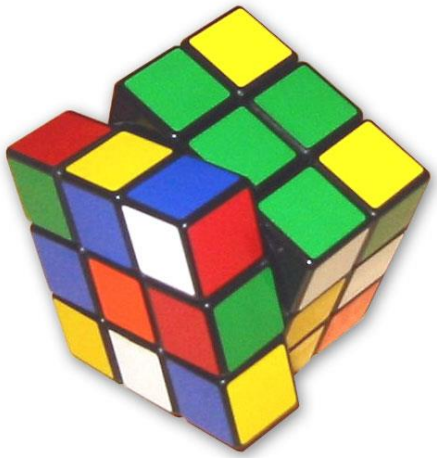
- P % 8 = 0, P < M < NP/8
- assume N, M, P > trivial

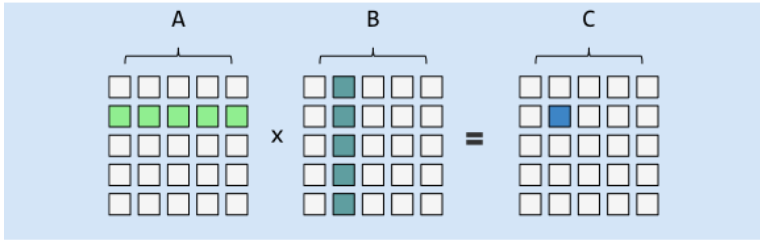
```
int summat_1(int a[M][N][P])
{
    int sum = 0;        // register

    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;
}
```

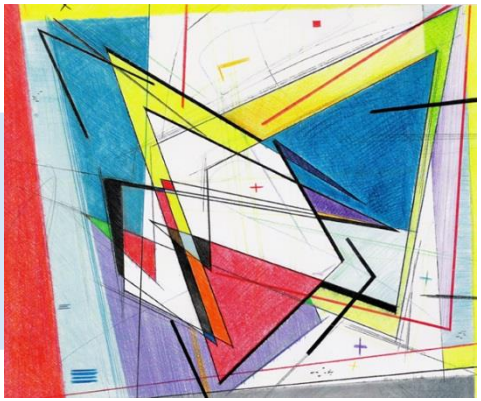
```
int summat_2(int a[M][N][P])
{
    int sum = 0;        // register

    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;
}
```

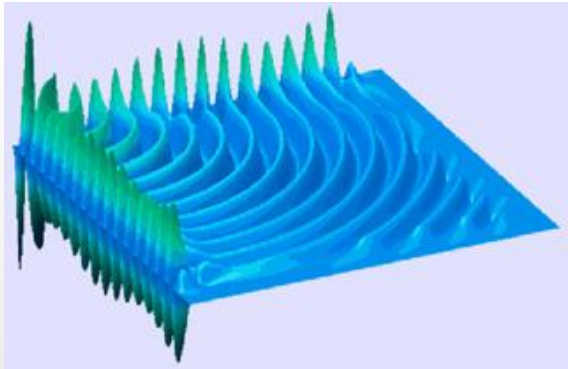
what are the access stride, hit rate and cache contents at end of loop nest?

matrix multiplication









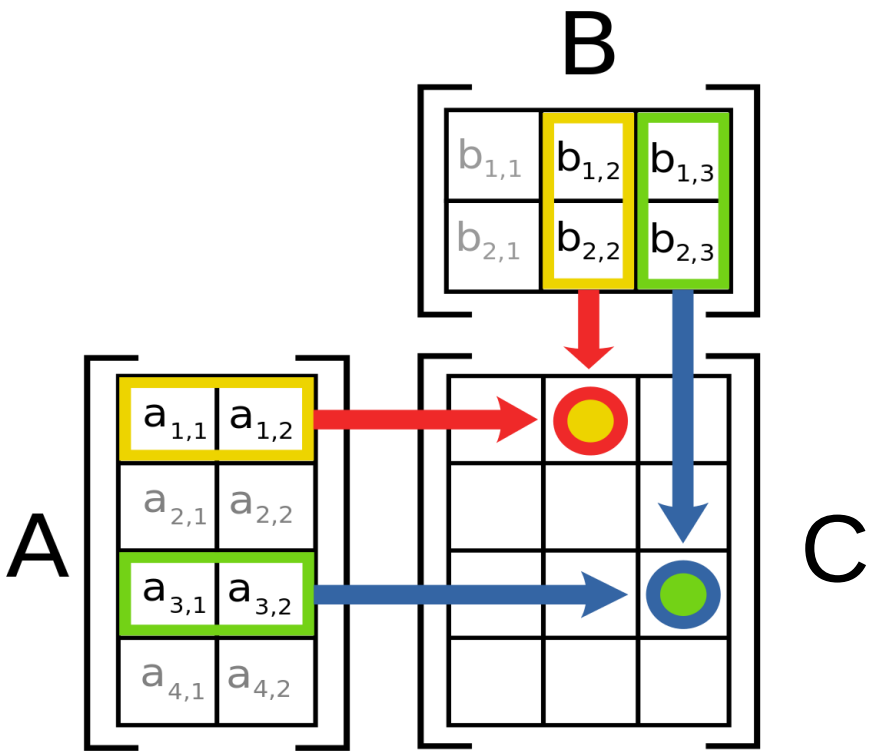
$$
\begin{aligned}
&\{\{-0.4\,X02_{MPS} - 0.32\,X14_{MPS} - 0.6\,X23_{MPS} - 0.48\,X36_{MPS} + 10.\,X39_{MPS},\\
&\quad -1.\,X01_{MPS} + 1.\,X02_{MPS} + 1.\,X03_{MPS} == 0.\ \&\&\ -1.06\,X01_{MPS} + 1.\,X04_{MPS} == 0.\ \&\&\\
&\quad 1.\,X01_{MPS} \le 80.\ \&\&\ -1.\,X02_{MPS} + 1.4\,X14_{MPS} \le 0.\ \&\&\\
&\quad -1.\,X06_{MPS} - 1.\,X07_{MPS} - 1.\,X08_{MPS} - 1.\,X09_{MPS} + 1.\,X14_{MPS} + 1.\,X15_{MPS} == 0.\ \&\&\\
&\quad -1.06\,X06_{MPS} - 1.06\,X07_{MPS} - 0.96\,X08_{MPS} - 0.86\,X09_{MPS} + 1.\,X16_{MPS} == 0.\ \&\&\\
&\quad 1.\,X06_{MPS} - 1.\,X10_{MPS} \le 80.\ \&\&\ 1.\,X07_{MPS} - 1.\,X11_{MPS} \le 0.\ \&\&\ 1.\,X08_{MPS} - 1.\,X12_{MPS} \le 0.\ \&\&\\
&\quad 1.\,X09_{MPS} - 1.\,X13_{MPS} \le 0.\ \&\&\ -1.\,X22_{MPS} + 1.\,X23_{MPS} + 1.\,X24_{MPS} + 1.\,X25_{MPS} == 0.\ \&\&\\
&\quad -0.43\,X22_{MPS} + 1.\,X26_{MPS} == 0.\ \&\&\ 1.\,X22_{MPS} \le 500.\ \&\&\ -1.\,X23_{MPS} + 1.4\,X36_{MPS} \le 0.\ \&\&\\
&\quad -0.43\,X28_{MPS} - 0.43\,X29_{MPS} - 0.39\,X30_{MPS} - 0.37\,X31_{MPS} + 1.\,X38_{MPS} == 0.\ \&\&\\
&\quad 1.\,X28_{MPS} + 1.\,X29_{MPS} + 1.\,X30_{MPS} + 1.\,X31_{MPS} - 1.\,X36_{MPS} + 1.\,X37_{MPS} + 1.\,X39_{MPS} == 44.\ \&\&\\
&\quad 1.\,X28_{MPS} - 1.\,X32_{MPS} \le 500.\ \&\&\ 1.\,X29_{MPS} - 1.\,X33_{MPS} \le 0.\ \&\&\\
&\quad 1.\,X30_{MPS} - 1.\,X34_{MPS} \le 0.\ \&\&\ 1.\,X31_{MPS} - 1.\,X35_{MPS} \le 0.\ \&\&\\
&\quad 2.364\,X10_{MPS} + 2.386\,X11_{MPS} + 2.408\,X12_{MPS} + 2.429\,X13_{MPS} - 1.\,X25_{MPS} + 2.191\,X32_{MPS} +\\
&\qquad 2.219\,X33_{MPS} + 2.249\,X34_{MPS} + 2.279\,X35_{MPS} \le 0.\ \&\&\ -1.\,X03_{MPS} + 0.109\,X22_{MPS} \le 0.\ \&\&\\
&\quad -1.\,X15_{MPS} + 0.109\,X28_{MPS} + 0.108\,X29_{MPS} + 0.108\,X30_{MPS} + 0.107\,X31_{MPS} \le 0.\ \&\&\\
&\quad 0.301\,X01_{MPS} - 1.\,X24_{MPS} \le 0.\ \&\&\\
&\quad 0.301\,X06_{MPS} + 0.313\,X07_{MPS} + 0.313\,X08_{MPS} + 0.326\,X09_{MPS} - 1.\,X37_{MPS} \le 0.\ \&\&\\
&\quad 1.\,X04_{MPS} + 1.\,X26_{MPS} \le 310.\ \&\&\ 1.\,X16_{MPS} + 1.\,X38_{MPS} \le 300.\ \&\&\ X01_{MPS} \ge 0\ \&\&\ X02_{MPS} \ge 0\ \&\&\\
&\quad X03_{MPS} \ge 0\ \&\&\ X04_{MPS} \ge 0\ \&\&\ X06_{MPS} \ge 0\ \&\&\ X07_{MPS} \ge 0\ \&\&\ X08_{MPS} \ge 0\ \&\&\ X09_{MPS} \ge 0\ \&\&\\
&\quad X10_{MPS} \ge 0\ \&\&\ X11_{MPS} \ge 0\ \&\&\ X12_{MPS} \ge 0\ \&\&\ X13_{MPS} \ge 0\ \&\&\ X14_{MPS} \ge 0\ \&\&\ X15_{MPS} \ge 0\ \&\&\\
&\quad X16_{MPS} \ge 0\ \&\&\ X22_{MPS} \ge 0\ \&\&\ X23_{MPS} \ge 0\ \&\&\ X24_{MPS} \ge 0\ \&\&\ X25_{MPS} \ge 0\ \&\&\ X26_{MPS} \ge 0\ \&\&\\
&\quad X28_{MPS} \ge 0\ \&\&\ X29_{MPS} \ge 0\ \&\&\ X30_{MPS} \ge 0\ \&\&\ X31_{MPS} \ge 0\ \&\&\ X32_{MPS} \ge 0\ \&\&\ X33_{MPS} \ge 0\ \&\&\\
&\quad X34_{MPS} \ge 0\ \&\&\ X35_{MPS} \ge 0\ \&\&\ X36_{MPS} \ge 0\ \&\&\ X37_{MPS} \ge 0\ \&\&\ X38_{MPS} \ge 0\ \&\&\ X39_{MPS} \ge 0\},\\
&\{X01_{MPS}, X02_{MPS}, X03_{MPS}, X04_{MPS}, X06_{MPS}, X07_{MPS}, X08_{MPS}, X09_{MPS}, X10_{MPS},\\
&\quad X11_{MPS}, X12_{MPS}, X13_{MPS}, X14_{MPS}, X15_{MPS}, X16_{MPS}, X22_{MPS}, X23_{MPS},\\
&\quad X24_{MPS}, X25_{MPS}, X26_{MPS}, X28_{MPS}, X29_{MPS}, X30_{MPS}, X31_{MPS}, X32_{MPS},\\
&\quad X33_{MPS}, X34_{MPS}, X35_{MPS}, X36_{MPS}, X37_{MPS}, X38_{MPS}, X39_{MPS}\}\}
\end{aligned}
$$

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=TIIPHaBdabGJoM%3A

matrix multiplication

C = A x B

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



matrix multiplication

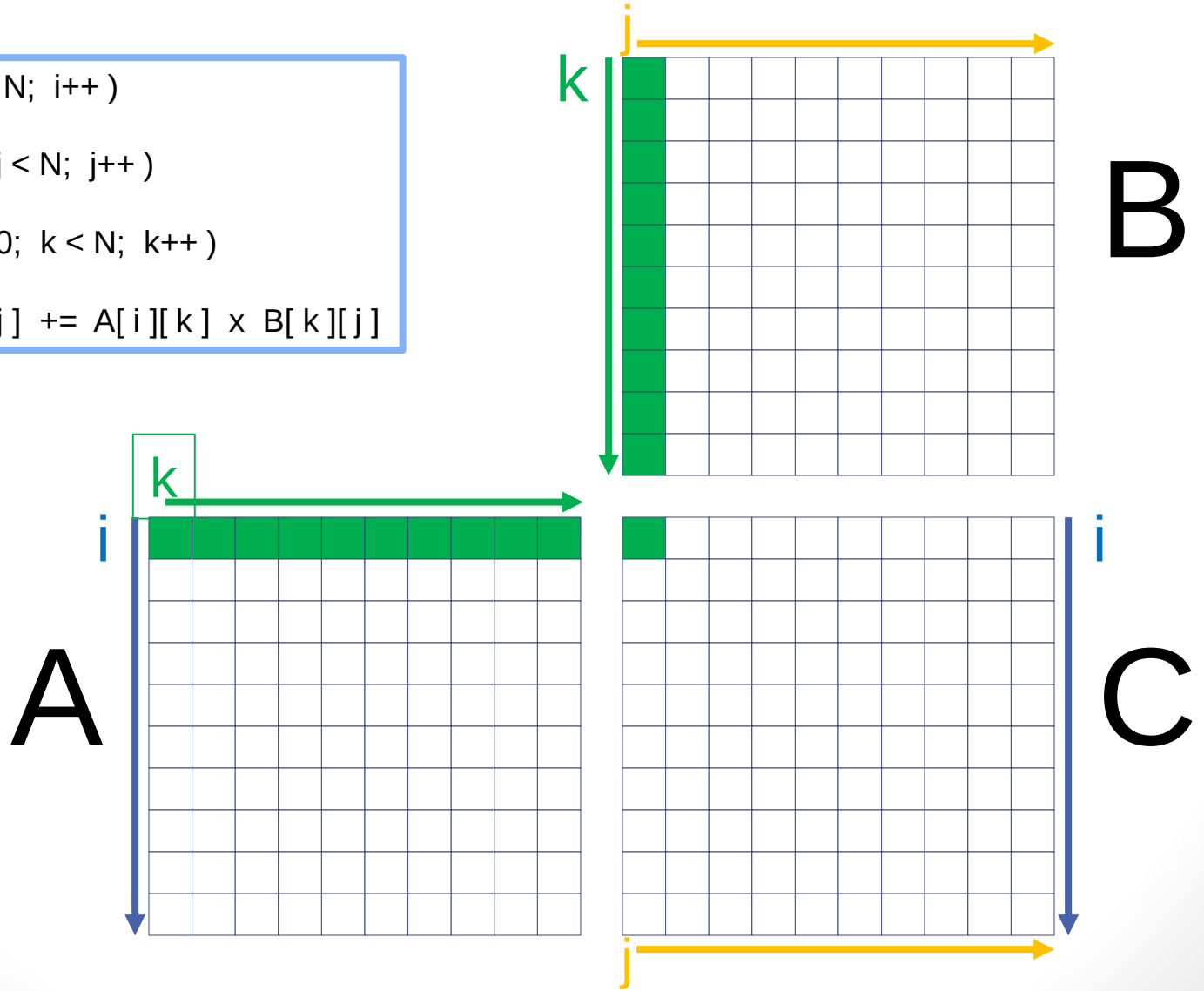
C = A x B

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

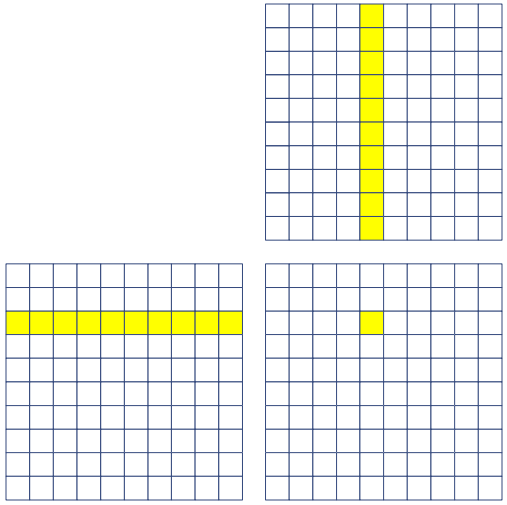
```
for ( i = 0;  i < N;  i++ )
    for ( j = 0;  j < N;  j++ )
        for ( k = 0;  k < N;  k++ )
            C[ i ][ j ]  +=  A[ i ][ k ]  x  B[ k ][ j ]
```

let: N is very large, 1/N ≈ 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

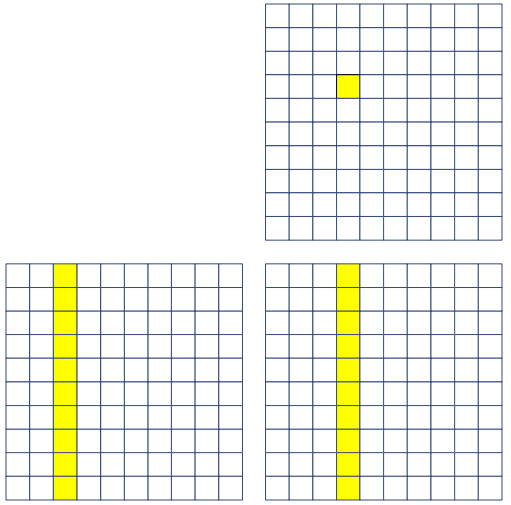
```
for ( j = 0;  j < N;  j++ )

    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 ≈ 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

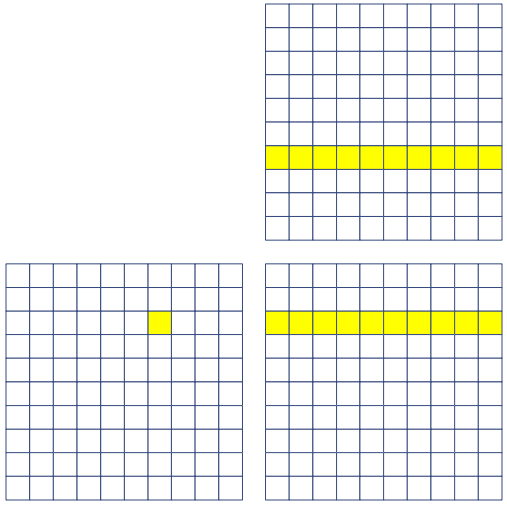
```
I   for ( i = 0;  i < N;  i++ )
K       for ( k = 0;  k < N;  k++ )
J           for ( j = 0;  j < N;  j++ )
                C[ i ][ j ]  +=  A[ i ][ k ]  x  B[ k ][ j ]
```

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)

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 ]
```

A		B		C		
0.25	+	1.00	+	0.00	=	1.25

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 ]
```

A		B		C		
1.00	+	0.00	+	1.00	=	2.00

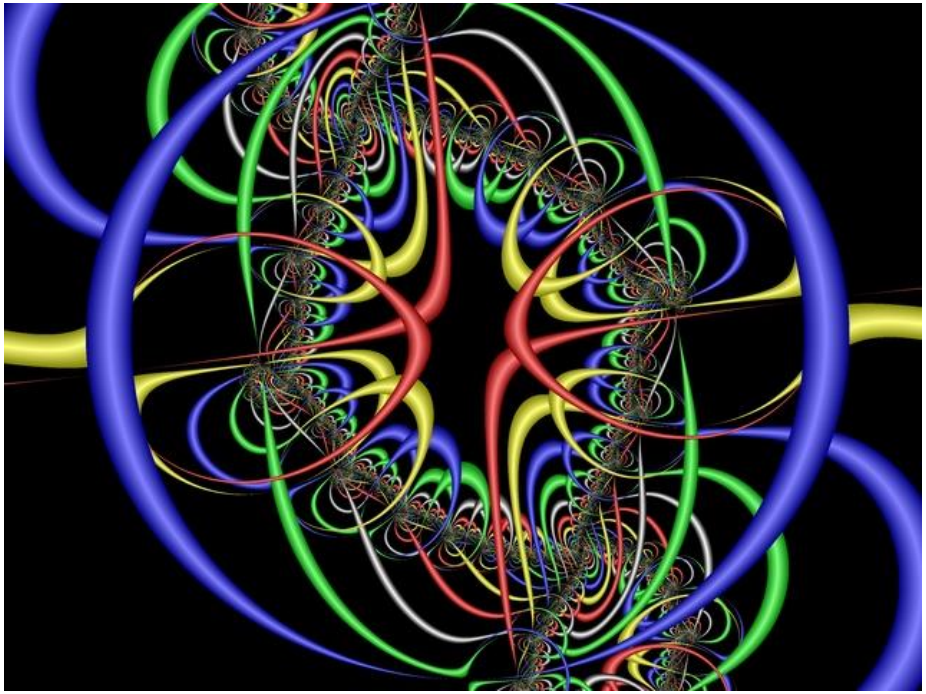
I K J

```
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 ]
```

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

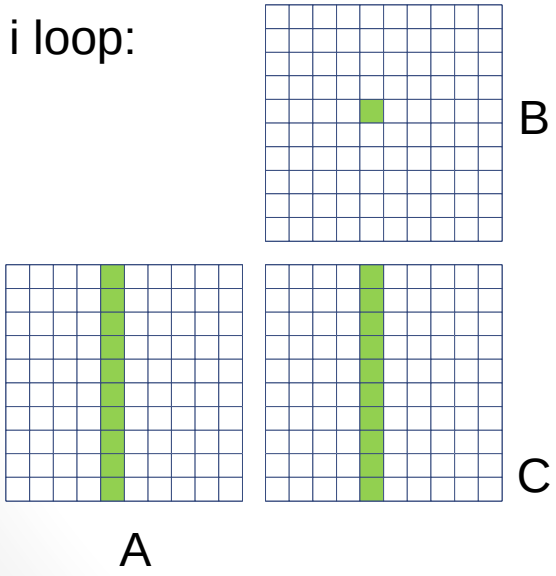
m-mult cache performance

total miss analysis method

```
I   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: N is very large, 1/N ≈ 0
N >> 4E
fully associative cache
LRU
B = 4 * sizeof(int)
i,j,k register variables

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	$\frac{N^2}{B}$

$$M = \frac{N^3}{B} + N^3 + \frac{N^2}{B}$$

m-mult cache performance

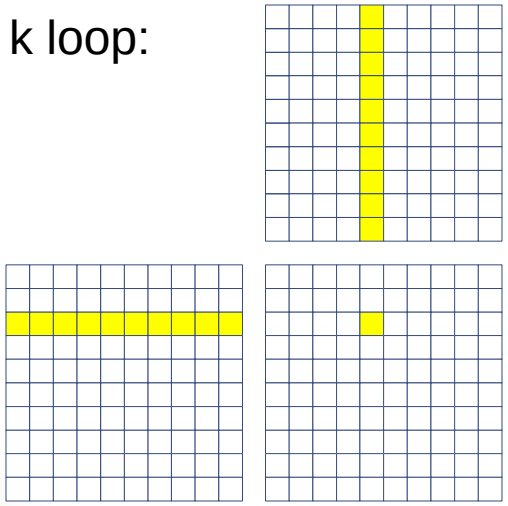
total miss analysis method

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 ≈ 0
N >> 4E
fully associative cache
LRU
B = 4 * sizeof(int)
i,j,k register variables

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

$$M = \frac{N^3}{B} + N^3 + N^2$$

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)

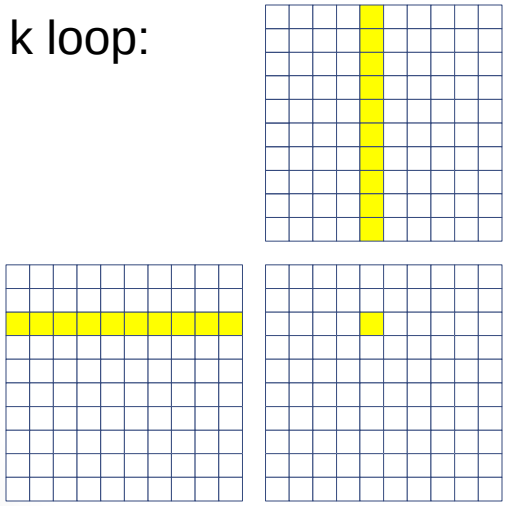
fully associative cache

LRU

$B = 4 *$ sizeof(element)

i,j,k register variables

k loop:



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

$$M = \frac{N^2}{B} + \frac{N^2}{B} + \frac{N^2}{B}$$

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

- compute total misses for all loop permutations, in terms of N and B
- what is the optimal loop ordering?

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

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 ];
        other computations ...
```

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 += 4 )
    for ( j = 0;  j < m;  j++)
        loop-body(j, i)
    for ( j = 0;  j < m;  j++)
        loop-body(j, i+1)
    for ( j = 0;  j < m;  j++)
        loop-body(j, i+2)
    for ( j = 0;  j < m;  j++)
        loop-body(j, i+2)
```

preserves execution order

but doesn't accomplish anything ...

UNROLLED / JAMMED

```
for ( i = 0;  i < 2n;  i += 4 )
    for ( j = 0;  j < m;  j++)
        loop-body(j, i)
        loop-body(j, i+1)
        loop-body(j, i+1)
        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 ]
```

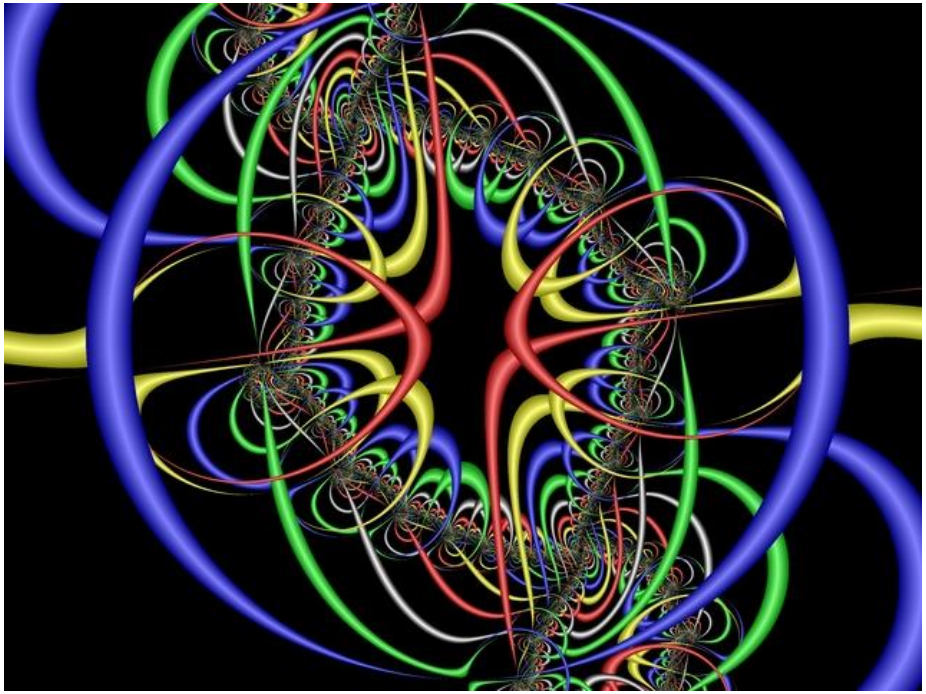
UNROLLED / JAMMED

```
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	N/B	N	1
J	1	N/B	N/B
	N^2/B	N^3/B	N^2/B

	A	B	C
K	N	N	N
I	N	1	N
J	1	N/B	N/B
	N^2	N^2/B	N^3/B