Data Dependences

- Essential constraints:

```
S1:   a = b + c
S2:   d = a * 2
S3:   a = c + 2
S4:   e = d + c + 2
```

Data Dependences

- Essential constraints:

```
S1:   a = b + c
S2:   d = a * 2
S3:   a = c + 2
S4:   e = d + c + 2
```

- S2 must execute after S1

Data Dependences

- Essential constraints:

```
S1:   a = b + c
S2:   d = a * 2
S3:   a = c + 2
S4:   e = d + c + 2
```

- S3 must execute after S2

Data Dependences

- Essential constraints:

```
S1:   a = b + c
S2:   d = a * 2
S3:   a = c + 2
S4:   e = d + c + 2
```

- S3 must execute after S1

Data Dependences

- Essential constraints:

```
S1:   a = b + c
S2:   d = a * 2
S3:   a = c + 2
S4:   e = d + c + 2
```

- But S3 and S4 can execute in either order, or concurrently

Data Dependences

- Essential constraints:

```
S1:   a = b + c
S2:   d = a * 2
S3:   a = c + 2
S4:   e = d + c + 2
```

- S1 and S2 cannot execute concurrently
- S2 and S3 cannot execute concurrently
- S1 and S3 cannot execute concurrently
- But S3 and S4 can execute concurrently

- Execution conditions due to Bernstein (1966)

Types of Dependences

- Flow-dependence occurs when a variable which is assigned a value in one statement say S1 is read in another statement, say S2 later.

```
S1:   a = b + c
S2:   d = a * 3
```

Types of Dependences

- Anti-dependence occurs when a variable which is read in one statement say S1 is assigned a value in another statement, say S2, later.

```
S1:   d = a * 3
S2:   a = b + c
```

Types of Dependences

- Output-dependence occurs when a variable which is assigned a value in one statement say S1 is later reassigned in another statement, say S2.

```
S1:   a = b + c
S2:   a = d * 3
```

Types of Dependences

- Input-dependence occurs when a variable is read in two different statements say S1 and S2. Relative ordering of S1 and S2 is not important for input dependence.

```
S1:   a = b + c
S2:   d = b * 3
```

Data Dependences in Loops

- Associate a dynamic instance to each statement. For example

```
     For i = 1 to 50
S1:    A(i) = B(i-1) + C(i)
S2:    B(i) = A(i+2) + C(i)
     EndFor
```

- Statements S1 and S2 are executed 50 times. We say S2(10) to mean the execution of S2 when i = 10.
- Dependences are based on dynamic instances of statements.

Data Dependences in Loops

- Unrolling loops can help one figure out dependences:

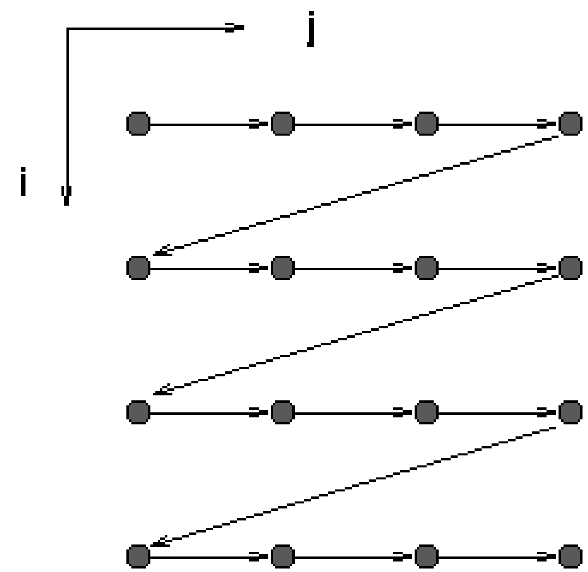
```
S1(1):      A(1) = B(0) + C(1)
S2(1):      B(1) = A(3) + C(1)
S1(2):      A(2) = B(1) + C(2)
S2(2):      B(2) = A(4) + C(2)
S1(3):      A(3) = B(2) + C(3)
S2(3):      B(3) = A(5) + C(3)
.....................

S1(50):    A(50) = B(49) + C(50)
S2(50):    B(50) = A(52) + C(50)
```

Iteration Spaces

- Nested loops define an iteration space:

```
For i = 1 to 4
   for j = 1 to 4
      A(i,j) = A(i,j) + C(j)
   Endfor
Endfor
```



- Sequential execution (traversal order):
- Dimensionality of iteration space = loop nest level; arbitrary convex shapes are allowed
- Change in order of execution is valid if no dependences are violated

Single Processor Performance Enhancement

- Two fundamental issues:
  - Adequate fine-grained parallelism
    - Exploit vector instructions sets (SSE, AVX, AVX-512, ...)
    - Multiple pipelined functional units in each core
  - Minimize memory-access costs (about an order of magnitude higher than clock cycle)
- Useful loop transformations:
  - Loop Permutation
  - Loop Unrolling
  - Loop Blocking (tiling)
  - Loop Fusion/Distribution

Access Stride and Spatial Locality

- Access stride: Separation between successively accessed memory locations
- Unit access stride maximizes spatial locality (only one miss per cache line)
- 2-D arrays have different linearized representations in Fortran and C

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

a	b	c	d	e	f		o	p

Row Major Order (C)

a	e	i	m	b	f		l	p

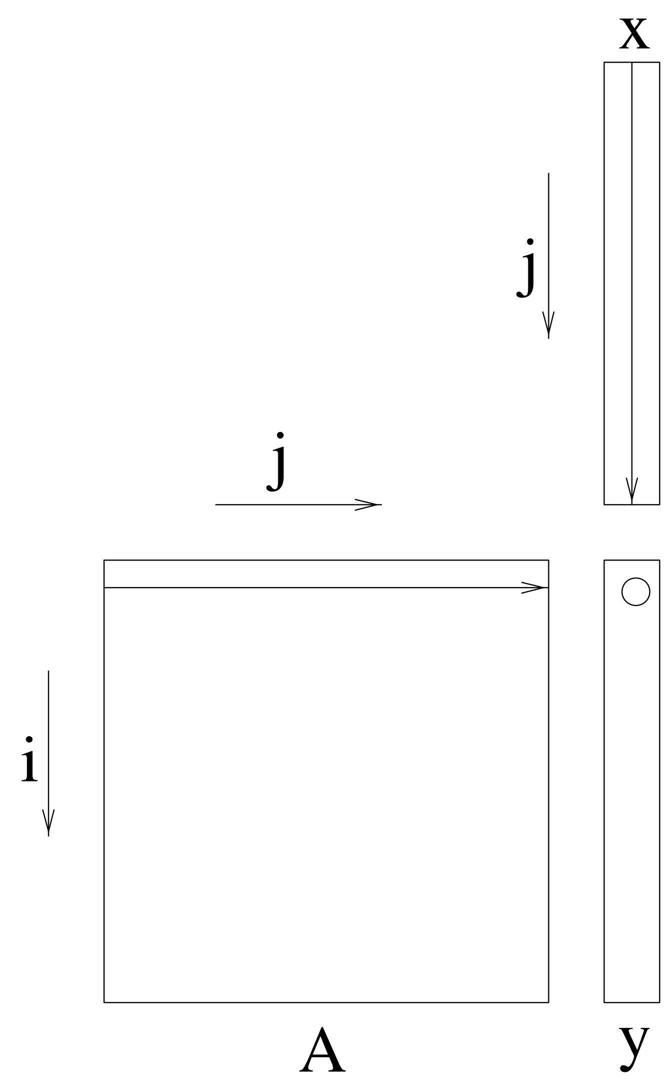
Column-major Order (Fortran)

Matrix-Vector Multiplication: Dot-Product

```
For I = 1, N
 For J = 1, N
  y(I)=y(I)+A(I,J)*x(J)
 EndFor
EndFor
```

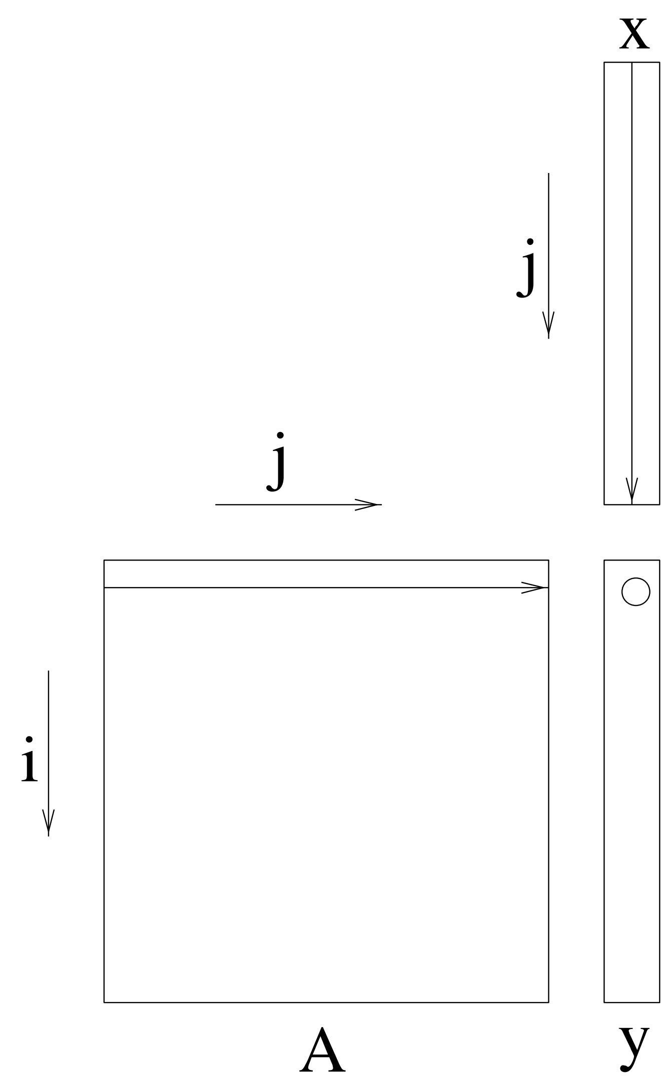
	A	x	y
C	1	1	0
Fortran	n	1	0

Access Stride for Arrays



Matrix-Vector Multiplication: SAXPY

```
For J = 1, N
 For I = 1, N
  y(I)=y(I)+A(I,J)*x(J)
 EndFor
EndFor
```



	A	x	y
C	n	0	1
Fortran	1	0	1

Access Stride for Arrays

Loop Permutation: Matrix Multiplication

```
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];
```

Reference	ikj	kij	jik	ijk	jki	kji
C(i,j)	1	1	0	0	n	n
A(i,k)	0	0	1	1	n	n
B(k,j)	1	1	n	n	0	0
	Best	Best			Worst	Worst

Access Stride for Arrays (C: Row-Major)

Loop Permutation: Matrix Multiplication

Reference	ikj	kij	jik	ijk	jki	kji
C(i,j)	1	1	0	0	n	n
A(i,k)	0	0	1	1	n	n
B(k,j)	1	1	n	n	0	0
	Best	Best			Worst	Worst

Access Stride for Arrays (C: Row-Major)

Compiler/Opt	ikj	kij	jik	ijk	jki	kji
icc -fast	17.0	17.0	17.0	17.0	17.0	17.0
icc -O3	5.0	5.0	5.0	5.0	5.0	5.0
icc -O2	7.8	7.8	7.8	7.8	7.8	7.8
icc -O1	2.0	2.0	.95	1.0	.29	.29
gcc -O3	6.1	7.6	.94	1.0	.29	.29
gcc -O2	2.0	2.0	.94	1.0	.29	.29
gcc -O1	1.9	1.9	.94	1.0	.29	.29

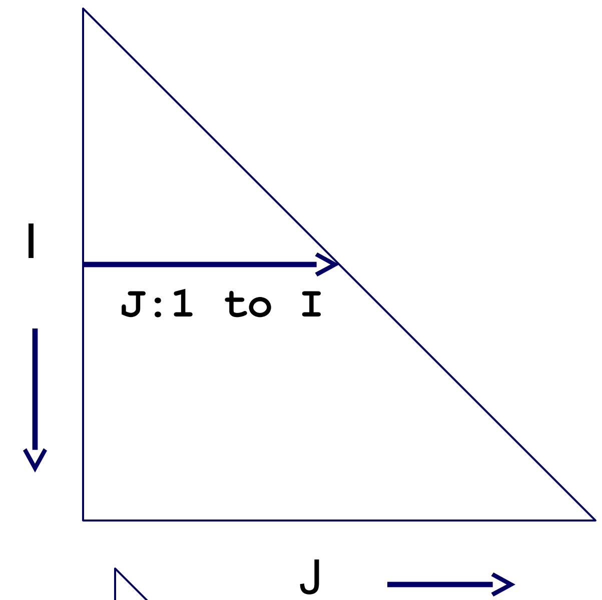
Performance on one core of Intel Xeon x5650 (GFLOPS)

Permutation: Non-Rectangular Loops

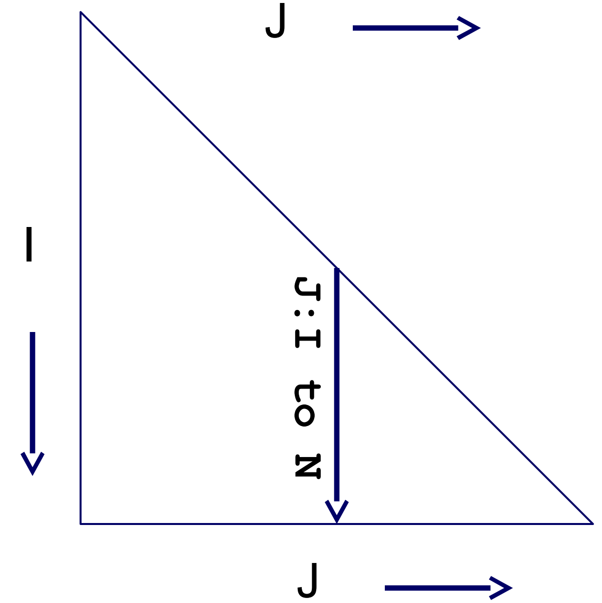
```
For I = 1, N
 For J = 1, I
  y(I)= y(I)+A(I,J)*x(J)
 EndFor
EndFor
```

Permutation: Non-Rectangular Loops

```
For I = 1, N
 For J = 1, I
  y(I)= y(I)+A(I,J)*x(J)
 EndFor
EndFor
```




```
For J = 1, N
 For I = J, N
  y(I)= y(I)+A(I,J)*x(J)
 EndFor
EndFor
```



Transformations: Loop Fusion

- Fusion: Fuses two loops, also known as jamming (useful for locality enhancement). In example below, after fusion, you cannot have dependencies from S2 to S1

```
    For I = 1, N
S1:  A(I) = B(I)+C(I)
    EndFor
    For I = 1,N
S2:  E(I) = A(I)*D(I)
    EndFor
```

```
    = 1, N
    = B(I)+C(I)
E(I) = A(I)*D(I)
```

```
For I =    N
  a = B(  C  )
  E(I) = a*D(I)
EndFor
```

Illegal Loop Fusion Example

```
    For I = 1, N
S1:  A(I) = B(I) + C(I)
    EndFor
    For I = 1,N
S2:  E(I) = A(I+1)* D(I)
    EndFor
```

We have flow dependences from S1 to S2

Illegal Loop Fusion Example

```
    For I = 1, N
S1:  A(I) = B(I) + C(I)
    EndFor
    For I = 1,N
S2:  E(I) = A(I+1)* D(I)
    EndFor
```

We have flow dependences from S1 to S2

On fusion

```
    For I = 1, N
S1:  A(I) = B(I) + C(I)
S2:  E(I) = A(I+1)* D(I)
    EndFor
```

Illegal fusion: On fusing the two loops, we have a violation of original data dependence

Transformations: Loop Distribution

- ***Loop Distribution***: Splits a single loop nest into many, also known as ***loop fission.***

```
    For I = 1, N
S1:  A(I) = B(I)+C(I)
S2:  E(I) = A(I)*D(I)
    EndFor
```

```
For I = 1, N
  A(I) = B(I)+C(I)
EndFor
For I = 1,N
      E(I) = A(I)*D(I)
EndFor
```

- Like loop fusion, distribution is not always legal – must ensure that no data dependences are violated.
- Needed for vectorization

Loop Unrolling

- Reduce number of iterations of loop but add statement(s) to loop body to do work of missing iterations
- Increases amount of instruction-level parallelism in loop body

```
for(j=0; j< 2*m; j++)
{
   Loop-Body(j)
}
```

→

```
for(j=0; j< 2*m; j+=2)
{
   Loop-Body(j)
   Loop-Body(j+1)
}
```

```
for(i=0; i< n; i++)
 for(j=0; j< 2*m; j++)
 {
    Loop-Body(i,j)
 }
```

→

```
for(i=0; i< n; i++)
 for(j=0; j< 2*m; j+=2)
 {
    Loop-Body(i,j)
    Loop-Body(i,j+1)
 }
```

Example: Inner Loop Unrolling

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

```
// Assumes n is a multiple of 4
   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)

- Reduce number of iterations of an outer loop
- Simply replicating inner-loop structures will not increase op-level parallelism; need to fuse ("jam") replicated inner-loops
- Changes memory access order
  - Could reduce cache misses
  - Hence must verify validity of transformation

```
for(i=0;i<2*n;i++)
 for(j=0;j< m;j++)
  Loop-Body(i,j)
```

```
for(i=0;i<2*n;i+=2)
{for(j=0;j<m;j++)
  Loop-Body(i,j)   // 2-way outer-unroll
 for(j=0;j<m;j++)  // does not increase
  Loop-Body(i+1,j) // op-lvl parallelism
}
```

```
for(i=0;i<2*n;i+=2)
{for(j=0;j<m;j++)
 {Loop-Body(i,j)    // unroll-jam increases
  Loop-Body(i+1,j)} // op-lvl parallelism
}
```

Example: Outer Loop Unrolling

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

↓

```
// Assumes n is a multiple of 4
   for(i=0;i<n;i+=4)
    for(j=0;j<n;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+2]+a[i+2][j]*x[j];
      y[i+3]=y[i+3]+a[i+3][j]*x[j];
   }
```

Improving Temporal Locality by Blocking

Example: Blocked matrix multiplication

- **"block" (in this context) does not mean "cache block".**
- **Instead, it means a sub-block within the matrix.**
- **Example: N = 8; sub-block size = 4**

$$\begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix} \times \begin{bmatrix} B11 & B12 \\ B21 & B22 \end{bmatrix} = \begin{bmatrix} C11 & C12 \\ C21 & C22 \end{bmatrix}$$

<u>Key idea:</u> Sub-blocks (i.e., **Axy**) can be treated just like scalars.

C11 = A11B11 + A12B21   C12 = A11B12 + A12B22

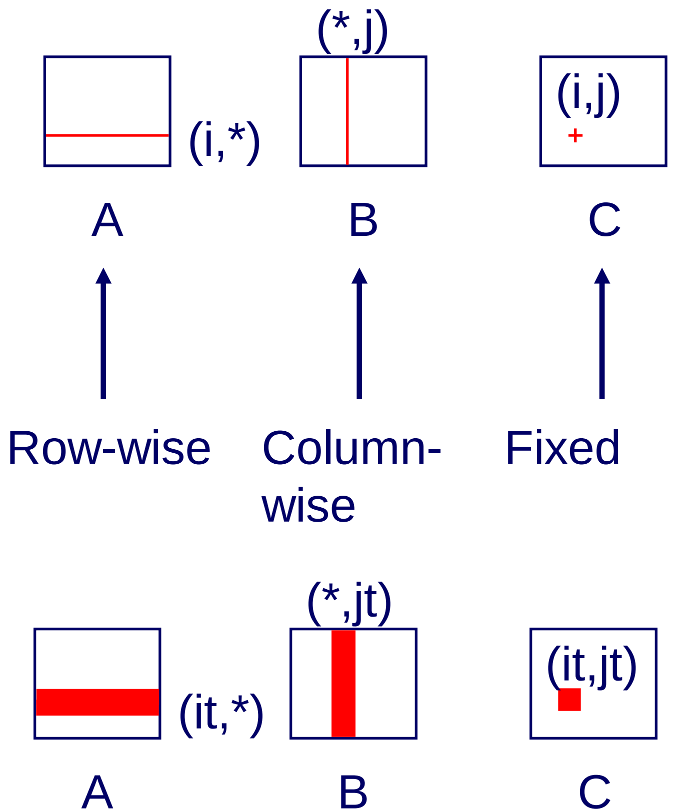
C21 = A21B11 + A22B21   C22 = A21B12 + A22B22

Blocked Matrix Multiplication

```
/* 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]*b[k][j];
```

```
/* Tiled; assume n multiple of T */
for (it=0; it<n; it+=T)
 for (jt=0; jt<n; jt+=T)
  for (kt=0; kt<n; kt+=T)
   for (i=it; i<it+T; i++)
    for (j=jt; j<jt+T; j++)
     for (k=kt; k<kt+T; k++)
      c[i][j]+= a[i][k]*b[k][j];
```

Inner loop:



Cache Misses: Blocked Mat-Mult

```
/* Tiled; assume n multiple of T */
for (it=0; it<n; it+=T)
 for (jt=0; jt<n; jt+=T)
  for (kt=0; kt<n; kt+=T)
   for (i=it; i<it+T; i++)
    for (j=jt; j<jt+T; j++)
     for (k=kt; k<kt+T; k++)
      c[i][j]+= a[i][k]*b[k][j];
```

Assume fully associative
Cache of size > 3*T*T

sub-mat-mult

- Each sub-mat-mult involves product of two TxT sub-matrices of A,B to contribute to a TxT sub-matrix of C
- Each sub-mat-mult has at most 3*(T2/B) cache misses (no evictions during computation; T2 elements for each array)
- Number of result blocks of C: (N/T)*(N/T) = N2/T2
- Each C-block requires (N/T) sub-mat-mults
- Total cache misses <= 3*(T2/B)*(N/T)*N2/T2 = 3N3/(B*T)
- T can be as large as sqrt(CacheSize/3)

Tiling = Loop-Split+Permutation

```
/* 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]*b[k][j];
```

Strip-mine each loop into a pair of equivalent loops

```
for (it=0; it<n; it+=T)
 for (i=it; i<it+T; i++)
  for (jt=0; jt<n; jt+=T)
   for (j=jt; j<jt+T; j++)
    for (kt=0; kt<n; kt+=T)
     for (k=kt; k<kt+T; k++)
     c[i][j]+= a[i][k]*b[k][j];
```

Loop Permutation

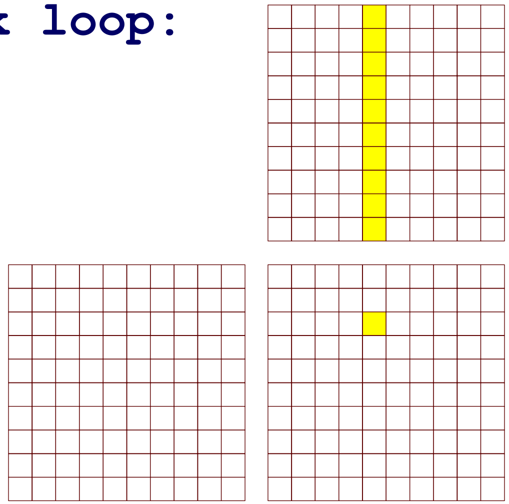
```
for (it=0; it<n; it+=T)
 for (jt=0; jt<n; jt+=T)
  for (kt=0; kt<n; kt+=T)
   for (i=it; i<it+T; i++)
    for (j=jt; j<jt+T; j++)
     for (k=kt; k<kt+T; k++)
     c[i][j]+= a[i][k]*b[k][j];
```

Total Cache Miss Analysis: IJK

```
I  for ( i = 0;  j < 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 < B*N
fully associative cache

k loop:



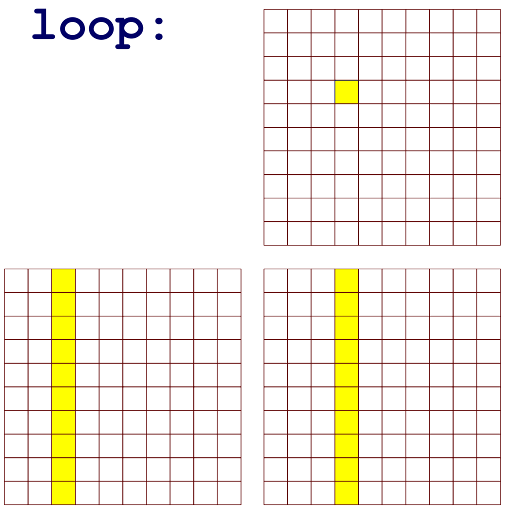
	A	**B**	**C**
I	N	N	N
J	1	N	$\frac{N}{B}$
K	$\frac{N}{B}$	N	1
	$\frac{N2}{B}$	N3	$\frac{N2}{B}$

Total Cache Miss Analysis: JKI

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

let: C < B*N
fully associative cache

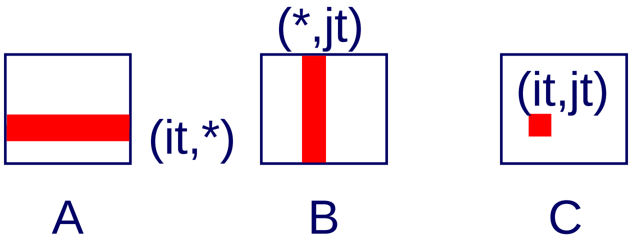
i loop:



	A	B	C
J	N	N	N
K	N	N	N
I	N	1	N
	N3	N2	N3

Blocked Matrix Multiply: Cache Misses

```
/* Tiled; assume n multiple of T */
for (it=0; it<n; it+=T)
 for (jt=0; jt<n; jt+=T)
  for (kt=0; kt<n; kt+=T)
   for (i=it; i<it+T; i++)
    for (j=jt; j<jt+T; j++)
     for (k=kt; k<kt+T; kt++)
      c[i][j]+= a[i][k]*b[k][j];
```

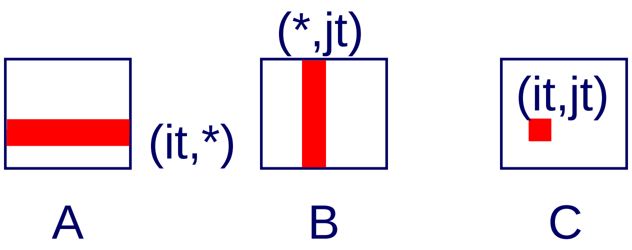


Assume fully associative
Cache: size > 3*T*T
But size < T*N

Loop	A	B	C
it			
jt			
kt			
i			
j			
k			

Blocked Matrix Multiply: Cache Misses

```
/* Tiled; assume n multiple of T */
for (it=0; it<n; it+=T)
 for (jt=0; jt<n; jt+=T)
  for (kt=0; kt<n; kt+=T)
   for (i=it; i<it+T; i++)
    for (j=jt; j<jt+T; j++)
     for (k=kt; k<kt+T; kt++)
      c[i][j]+= a[i][k]*b[k][j];
```



Assume fully associative
Cache: size > 3*T*T
But size < T*N

Loop	A	B	C
it	N/T	N/T	N/T
jt	N/T	N/T	N/T
kt	N/T	N/T	1
i	T	1	T
j	1	T/B	T/B
k	T/B	T	1
Total	N3/(TB)	N3/(TB)	N3/(TB)

Tiling: Arbitrary Bounds and Tilesize

```
/* ijk */
for (i=0; i<m; i++)
  for (j=0; j<n; j++)
    for (k=0; k<p; k++)
      c[i][j]+= a[i][k]*b[k][j];
```

```
for (it=0; it<n; it+=Ti)
 for (jt=0; jt<m; jt+=Tj)
  for (kt=0; kt<p; kt+=Tk)
   for (i=it; i< min(it+Ti,n); i++)
    for (j=jt; j< min(jt+Tj,m); j++)
     for (k=kt; k< min(kt+Tk,p); k++)
      c[i][j]+= a[i][k]*b[k][j];
```