Locality - 1

15-411/15-611 Compiler Design

Seth Copen Goldstein

November 10, 2020

Our Path

- Finding Loops
- Loop Invariant Code Motion (LICM)
- Partial Redundancy Elimination aka Lazy Code Motion (subsumes LICM)
- Understanding Dependencies
- Understanding Locality
- SRP
- Finding Dependencies
- Scheduling

Defining Dependencies

- Flow Dependence $\quad W \rightarrow R \quad \delta^f$ } true
- Anti-Dependence $\quad R \rightarrow W \quad \delta^a$
- Output Dependence $\quad W \rightarrow W \quad \delta^o$

} false

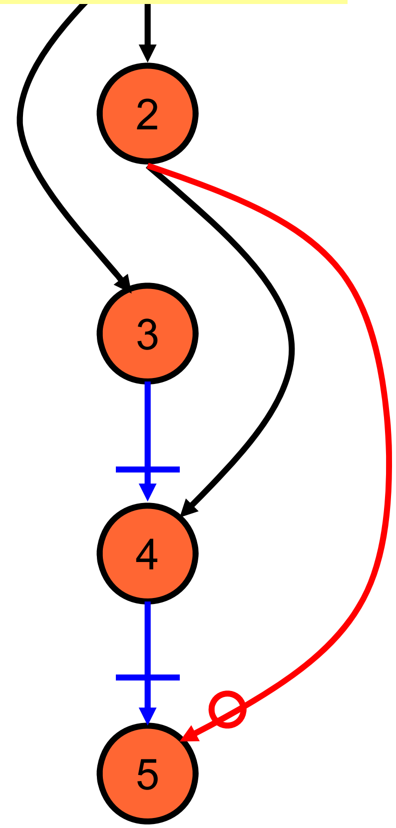
```
S1) a=0;
S2) b=a;
S3) c=a+d+e;
S4) d=b;
S5) b=5+e;
```

Example Dependencies

```
S1) a=0;
S2) b=a;
S3) c=a+d+e;
S4) d=b;
S5) b=5+e;
```

These are scalar dependencies. The same idea holds for memory accesses.

source	type	target	due to
S1	δ^f	S2	a
S1	δ^f	S3	a
S2	δ^f	S4	b
S3	δ^a	S4	d
S4	δ^a	S5	b
S2	δ^o	S5	b



What can we do with this information?
What are anti- and flow- called "false" dependences?

Dependencies in Loops

- Loop independent data dependence occurs between accesses in the same loop iteration.
- Loop-carried data dependence occurs between accesses across different loop iterations.
- There is data dependence between
  access a at iteration i-k and
  access b at iteration i when:
  - a and b access the same memory location
  - There is a path from a to b
  - Either a or b is a write

Data Dependence in Loops

- Dependence can flow across iterations of the loop.
- Dependence information is annotated with iteration information.
- If dependence is across iterations it is loop carried otherwise loop independent.

```
for (i=0; i<n; i++) {
      A[i] = B[i];
      B[i+1] = A[i];
}
```

Data Dependence in Loops

- Dependence can flow across iterations of the loop.
- Dependence information is annotated with iteration information.
- If dependence is across iterations it is loop carried otherwise loop independent.

```
for (i=0; i<n; i++) {
    A[i] = B[i];
    B[i+1] = A[i];
}
```

δ^f loop carried

δ^f loop independent

Unroll Loop to Find Dependencies

δ^f loop carried

```
for (i=0; i<n; i++) {
    A[i] = B[i];
    B[i+1] = A[i];
}
```

δ^f loop independent

```
A[0] = B[0];
B[1] = A[0];
A[1] = B[1];
B[2] = A[1];
A[2] = B[2];
B[3] = A[2];
   ⋮
```

i=0

i=1

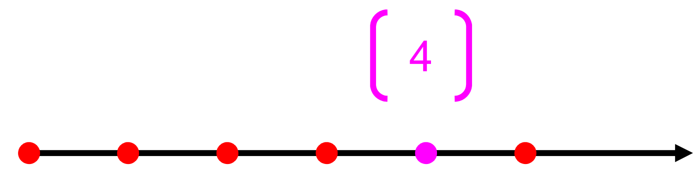
i=2

Distance/Direction of the dependence is also important.

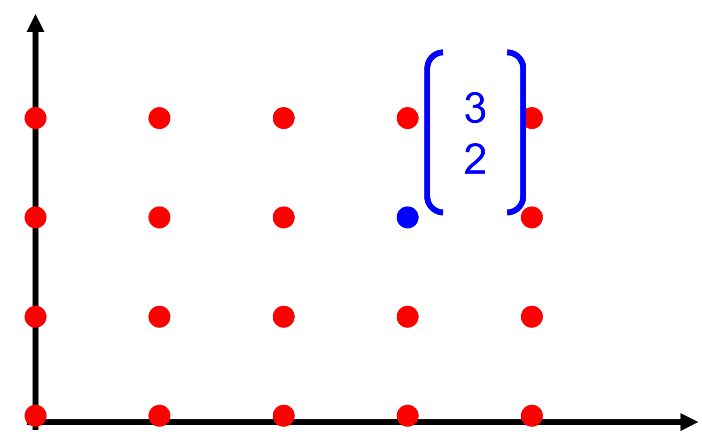
Iteration Space

Every iteration generates a point in an n-dimensional space, where n is the depth of the loop nest.

```
for (i=0; i<n; i++) {
    •••
}
```




```
for (i=0; i<n; i++)
    for (j=0; j<4; j++) {
        •••
}
```



Distance Vector

```
for (i=0; i<n; i++) {
    A[i] = B[i];
    B[i+1] = A[i];
}
```

Distance vector is the difference between the target and source iterations.

```
A[0] = B[0];
B[1] = A[0];    i=0
A[1] = B[1];
B[2] = A[1];    i=1
A[2] = B[2];
B[3] = A[2];    i=2
      ⋮
```

$$d = I_t - I_s$$

Exactly the distance of the dependence, i.e.,

$$I_s + d = I_t$$

Example of Distance Vectors

```
for (i=0; i<n; i++)
   for (j=0; j<m; j++){
      A[i,j] =    ;
           = A[i,j];
      B[i,j+1] =    ;
           = B[i,j];
      C[i+1,j] =    ;
           = C[i,j+1] ;
   }
```

j			
	$A_{0,2}=\ \ =A_{0,2}$ $B_{0,3}=\ \ =B_{0,2}$ $C_{1,2}=\ \ =C_{0,3}$	$A_{1,2}=\ \ =A_{1,2}$ $B_{1,3}=\ \ =B_{1,2}$ $C_{2,2}=\ \ =C_{1,3}$	$A_{2,2}=\ \ =A_{2,2}$ $B_{2,3}=\ \ =B_{2,2}$ $C_{3,2}=\ \ =C_{2,3}$
	$A_{0,1}=\ \ =A_{0,1}$ $B_{0,2}=\ \ =B_{0,1}$ $C_{1,1}=\ \ =C_{0,2}$	$A_{1,1}=\ \ =A_{1,1}$ $B_{1,2}=\ \ =B_{1,1}$ $C_{2,1}=\ \ =C_{1,2}$	$A_{2,1}=\ \ =A_{2,1}$ $B_{2,2}=\ \ =B_{2,1}$ $C_{3,1}=\ \ =C_{2,2}$
	$A_{0,0}=\ \ =A_{0,0}$ $B_{0,1}=\ \ =B_{0,0}$ $C_{1,0}=\ \ =C_{0,1}$	$A_{1,0}=\ \ =A_{1,0}$ $B_{1,1}=\ \ =B_{1,0}$ $C_{2,0}=\ \ =C_{1,1}$	$A_{2,0}=\ \ =A_{2,0}$ $B_{2,1}=\ \ =B_{2,0}$ $C_{3,0}=\ \ =C_{2,1}$
		i	

Example of Distance Vectors

```
for (i=0; i<n; i++)
   for (j=0; j<m; j++){
      A[i,j] =    ;
            = A[i,j];
      B[i,j+1] =    ;
            = B[i,j];
      C[i+1,j] =    ;
            = C[i,j+1] ;
   }
```

j \ i			
	$A_{0,2}=$ $=A_{0,2}$ $B_{0,3}=$ $=B_{0,2}$ $C_{1,2}=$ $=C_{0,3}$	$A_{1,2}=$ $=A_{1,2}$ $B_{1,3}=$ $=B_{1,2}$ $C_{2,2}=$ $=C_{1,3}$	$A_{2,2}=$ $=A_{2,2}$ $B_{2,3}=$ $=B_{2,2}$ $C_{3,2}=$ $=C_{2,3}$
j	$A_{0,1}=$ $=A_{0,1}$ $B_{0,2}=$ $=B_{0,1}$ $C_{1,1}=$ $=C_{0,2}$	$A_{1,1}=$ $=A_{1,1}$ $B_{1,2}=$ $=B_{1,1}$ $C_{2,1}=$ $=C_{1,2}$	$A_{2,1}=$ $=A_{2,1}$ $B_{2,2}=$ $=B_{2,1}$ $C_{3,1}=$ $=C_{2,2}$
	$A_{0,0}=$ $=A_{0,0}$ $B_{0,1}=$ $=B_{0,0}$ $C_{1,0}=$ $=C_{0,1}$	$A_{1,0}=$ $=A_{1,0}$ $B_{1,1}=$ $=B_{1,0}$ $C_{2,0}=$ $=C_{1,1}$	$A_{2,0}=$ $=A_{2,0}$ $B_{2,1}=$ $=B_{2,0}$ $C_{3,0}=$ $=C_{2,1}$
		i	

A yields: $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$ B yields: $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ C yields: $\begin{bmatrix} 1 \\ -1 \end{bmatrix}$

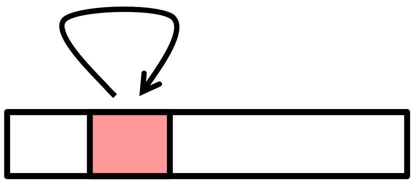
Our Path

- Finding Loops
- Loop Invariant Code Motion (LICM)
- Partial Redundancy Elimination aka Lazy Code Motion (subsumes LICM)
- Understanding Dependencies
- **Understanding Locality**
- SRP
- Finding Dependencies
- Scheduling

Recall: Locality

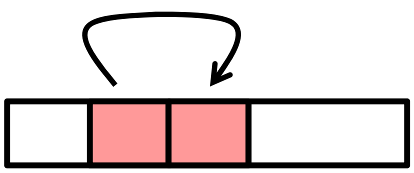
- Principle of Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently

- Temporal locality:



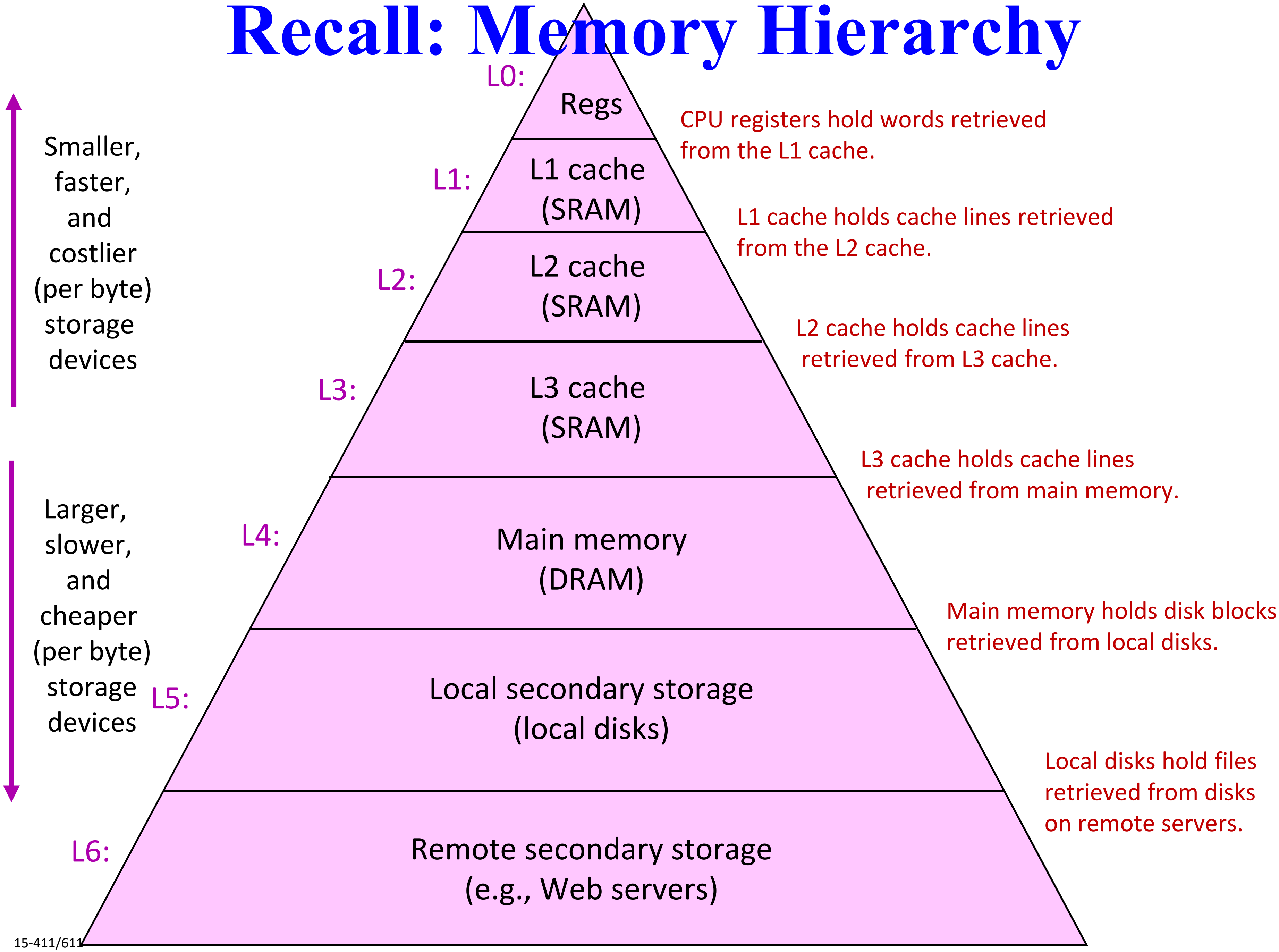
  - Recently referenced items are likely to be referenced again in the near future

- Spatial locality:



  - Items with nearby addresses tend to be referenced close together in time

Recall: Memory Hierarchy



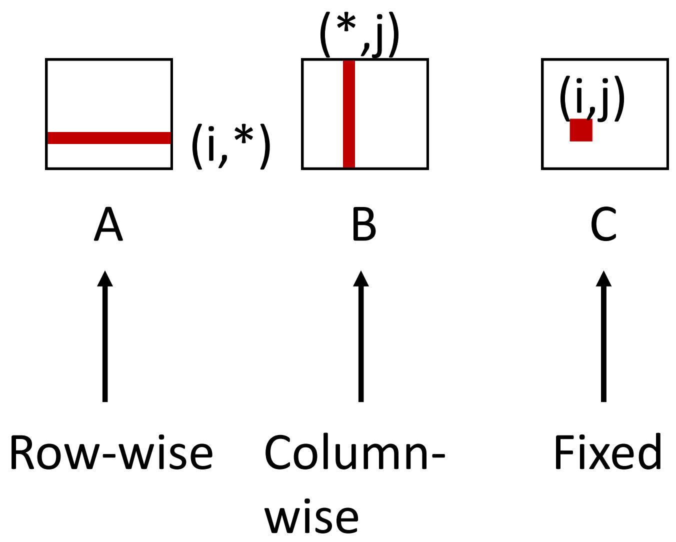
Layout of C Arrays in Memory

- C arrays allocated in row-major order
  - each row in contiguous memory locations
- Stepping through columns in one row:
  - `for (i = 0; i < N; i++)`
    `sum += a[0][i];`
  - accesses successive elements
  - if cache block size (B) > sizeof(a_{ij}) bytes, exploits spatial locality
    - miss rate = sizeof(a_{ij}) / B
- Stepping through rows in one column:
  - `for (i = 0; i < n; i++)`
    `sum += a[i][0];`
  - accesses distant elements
  - no spatial locality!
    - miss rate = 1 (i.e. 100%)

Matrix Multiplication (`ijk`)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Inner loop:



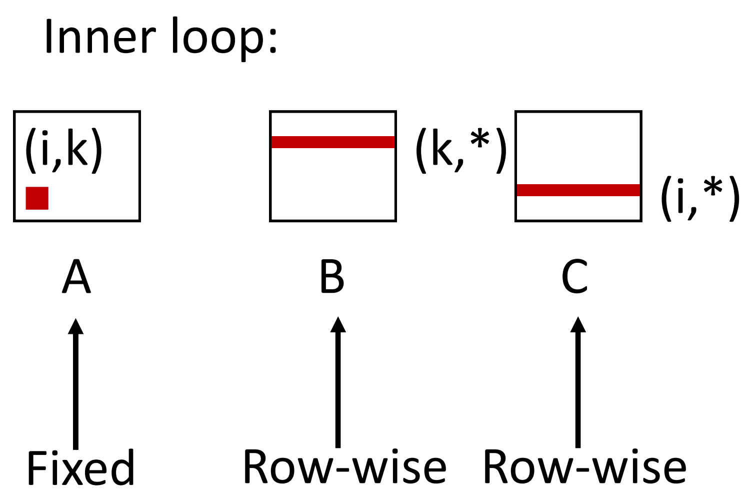
<u>Miss rate for inner loop iterations:</u>

<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

Block size = 32B (four doubles)

Matrix Multiplication (`kij`)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```



Miss rate for inner loop iterations:

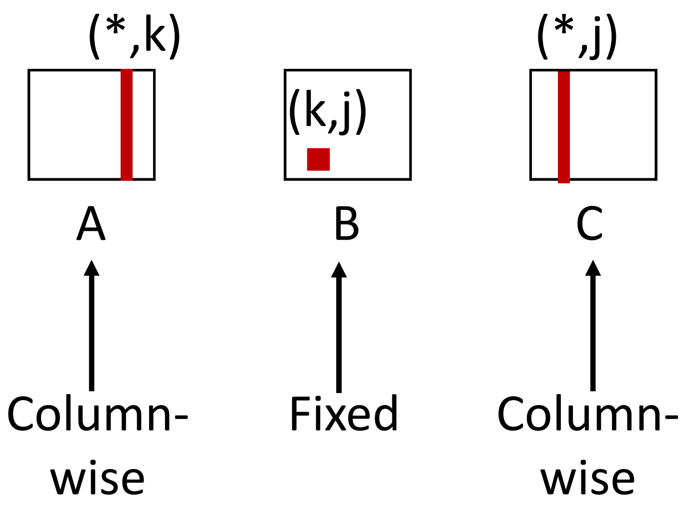
A	B	C
0.0	0.25	0.25

Block size = 32B (four doubles)

Matrix Multiplication (`jki`)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

Inner loop:



Miss rate for inner loop iterations:

A	B	C
1.0	0.0	1.0

Block size = 32B (four doubles)

Summary of Matrix Multiplication

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
   sum = 0.0;
   for (k=0; k<n; k++)
     sum += a[i][k] * b[k][j];
   c[i][j] = sum;
 }
}
```

`ijk` (& `jik`):

- 2 loads, 0 stores
- avg misses/iter = 1.25

```
for (k=0; k<n; k++) {
 for (i=0; i<n; i++) {
  r = a[i][k];
  for (j=0; j<n; j++)
   c[i][j] += r * b[k][j];
 }
}
```

`kij` (& `ikj`):

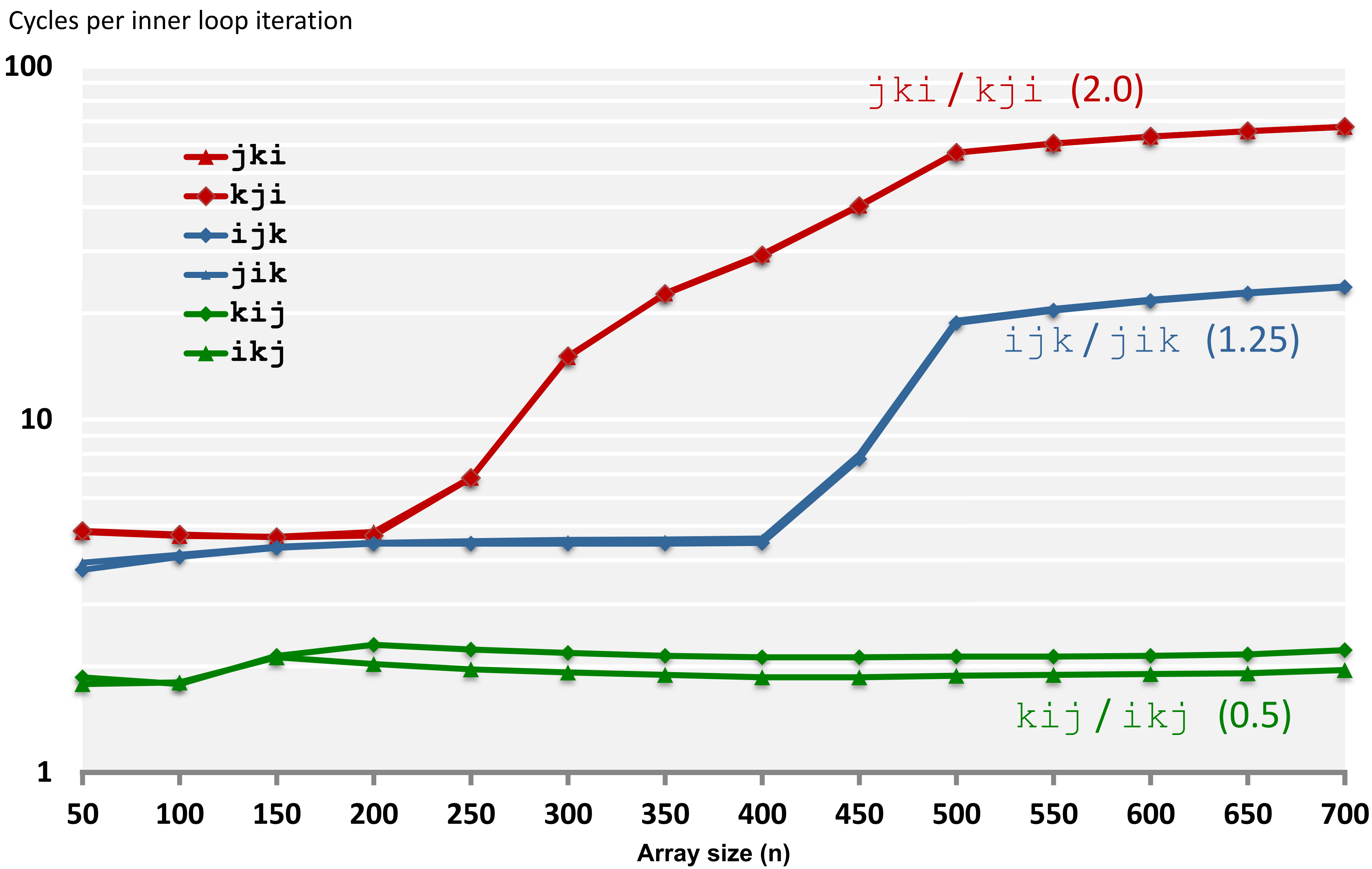
- 2 loads, 1 store
- avg misses/iter = 0.5

```
for (j=0; j<n; j++) {
 for (k=0; k<n; k++) {
   r = b[k][j];
   for (i=0; i<n; i++)
    c[i][j] += a[i][k] * r;
 }
}
```

`jki` (& `kji`):

- 2 loads, 1 store
- avg misses/iter = 2.0

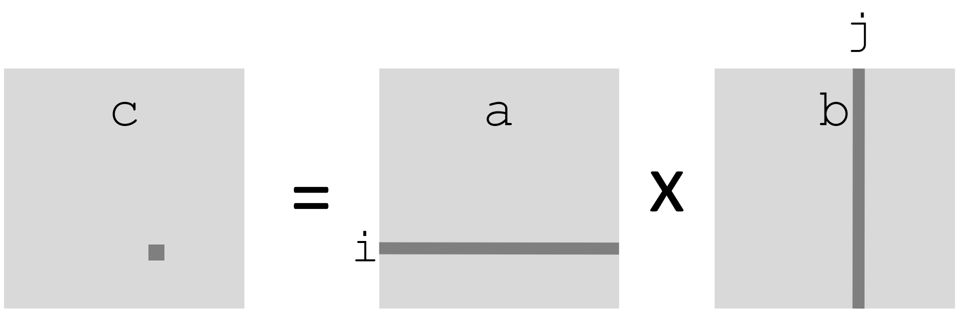
Core i7 Matrix Multiply Performance



Blocking: Matrix Multiplication

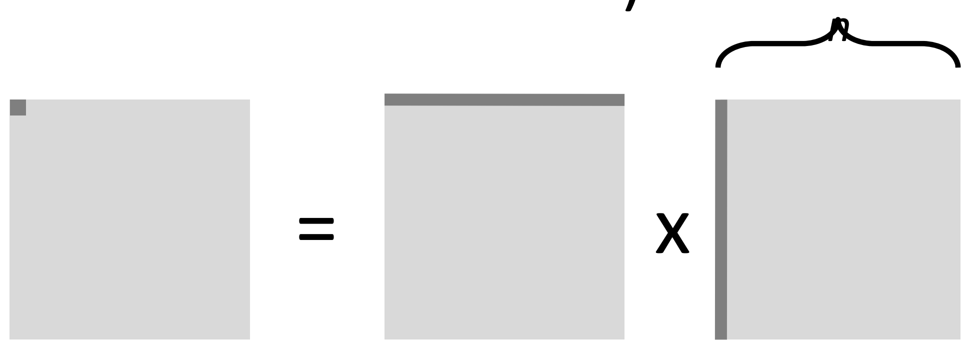
```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n + j] += a[i*n + k] * b[k*n + j];
}
```



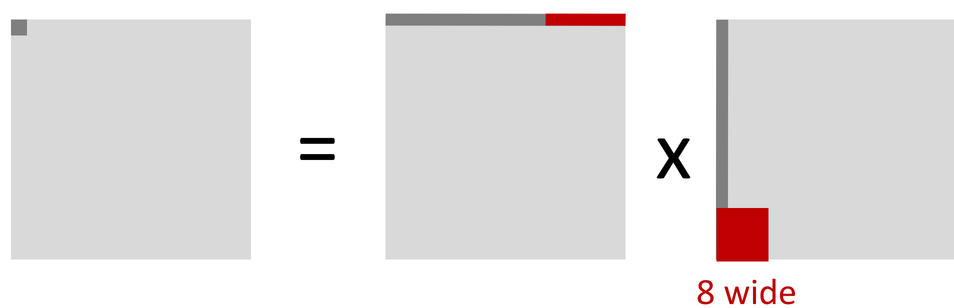
Cache Miss Analysis

- Assume:
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << *n* (much smaller than *n*)



- First iteration:
  - *n*/8 + n = 9*n*/8 misses

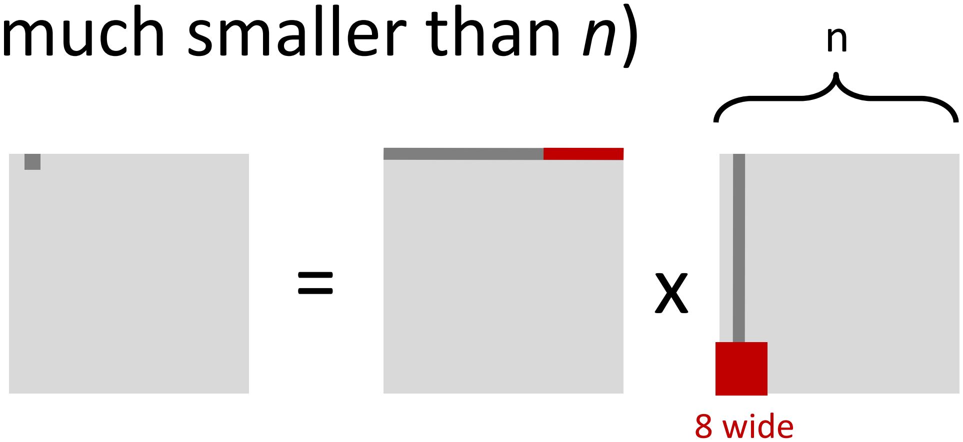
  - Afterwards in cache:
    (schematic)



Cache Miss Analysis

- Assume:
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << *n* (much smaller than *n*)

- Second iteration:
  - Again: $n/8 + n = 9n/8$ misses

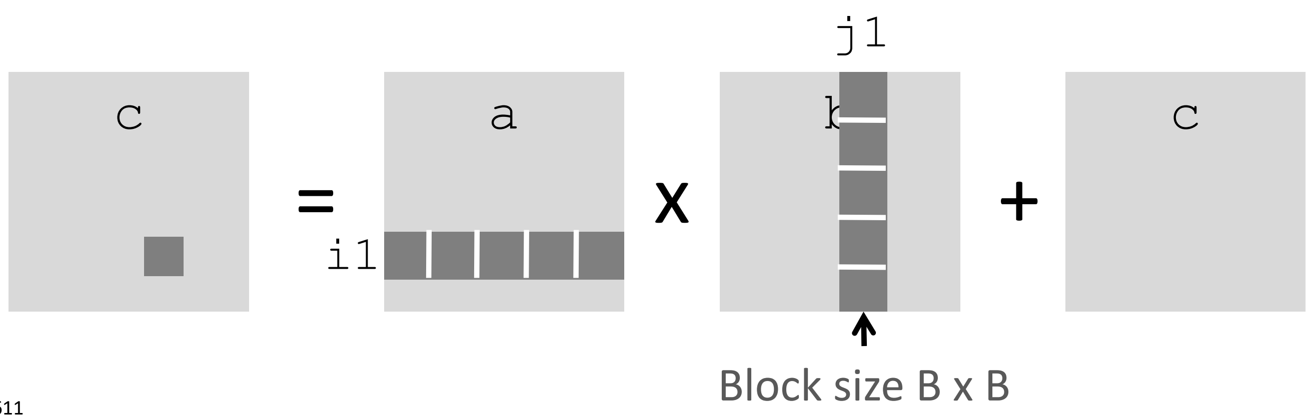


- Total misses:
  - $9n/8 \; n^2 = (9/8) \; n^3$

Blocked Matrix Multiplication

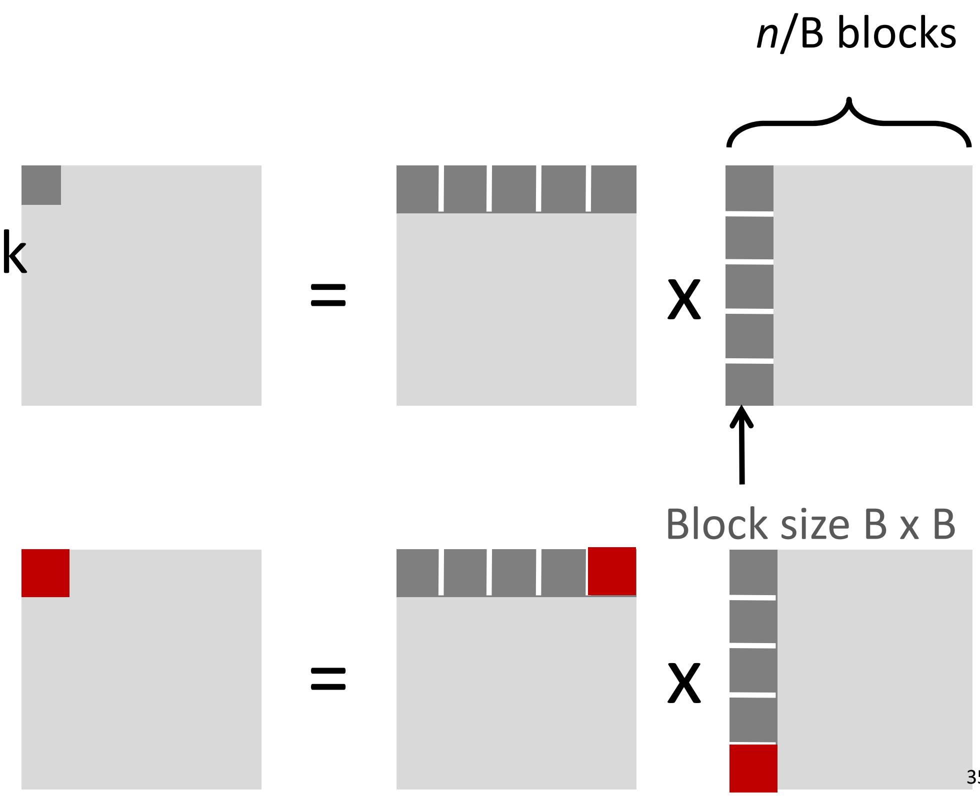
```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i1++)
                    for (j1 = j; j1 < j+B; j1++)
                        for (k1 = k; k1 < k+B; k1++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```



Cache Miss Analysis

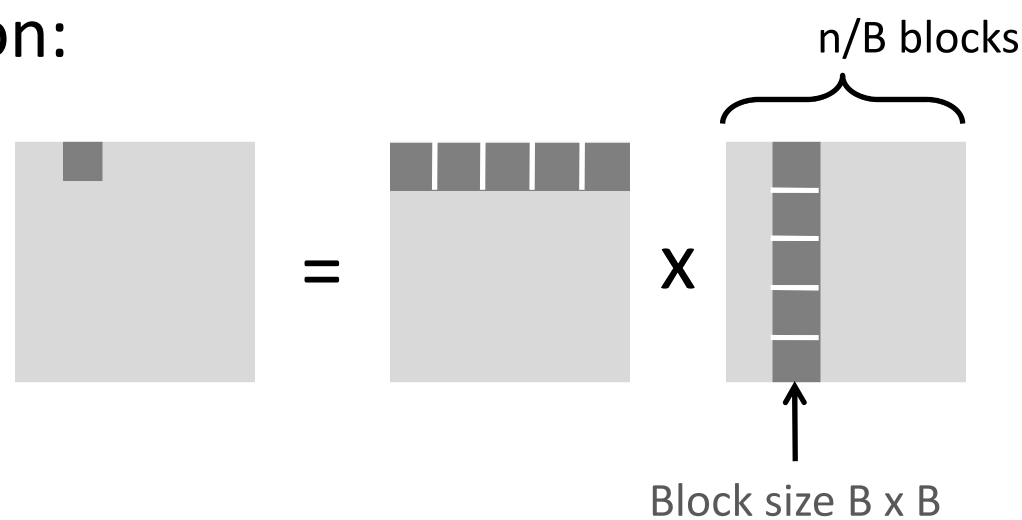
- Assume:
  - Cache block = 8 doubles
  - Cache size C << *n* (much smaller than n)
  - Three blocks fit into cache: $3B^2 < C$
- First (block) iteration:
  - $B^2/8$ misses for each block
  - $2n/B \times B^2/8 = nB/4$ (omitting matrix c)
  - Afterwards in cache (schematic)



Cache Miss Analysis

- Assume:
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks fit into cache: $3B^2 < C$

- Second (block) iteration:
  - Same as first iteration
  - $2n/B \times B^2/8 = nB/4$



- Total misses:
  - $nB/4 * (n/B)^2 = n^3/(4B)$

Blocking Summary

- No blocking: (9/8) n^3 misses
- Blocking: (1/(4B)) n^3 misses

- Use largest block B, such that B satisfies $3B^2 < C$
  - Fit three blocks in cache! Two input, one output.
- Reason for dramatic difference:
  - Matrix multiplication has inherent temporal locality:
    - Input data: $3n^2$, computation $2n^3$
    - Every array elements used $O(n)$ times!
  - But program has to be written properly

Outline

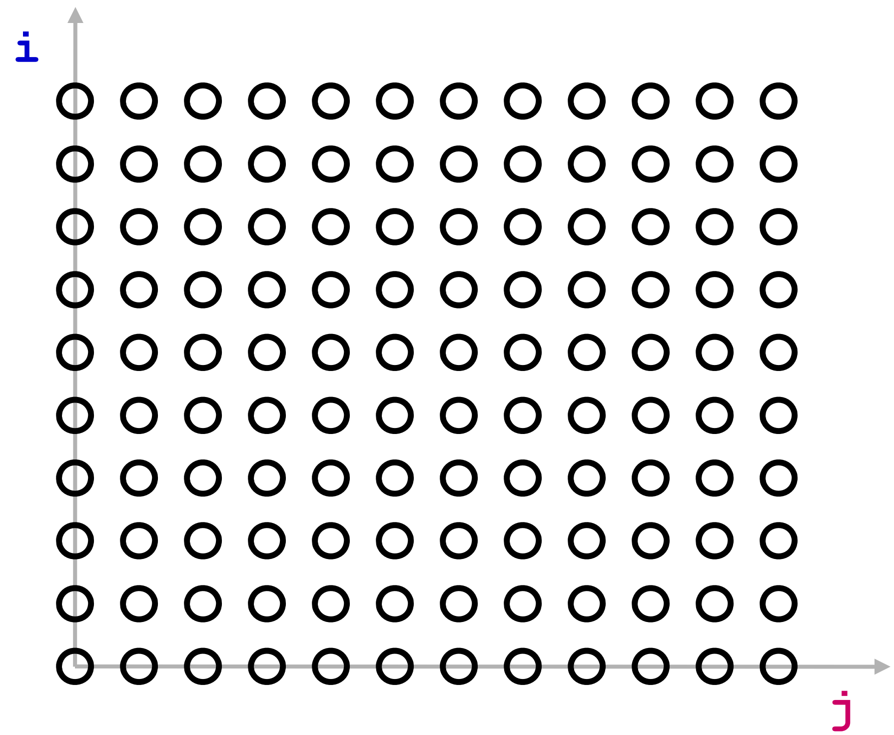
- The Problem
- Loop Transformations
  - dependence vectors
  - Transformations
  - Unimodular transformations
- Locality Analysis
- SRP

The Problem

- How to increase locality by transforming loop nest
- Matrix Mult is simple as it is both
  - legal to tile
  - advantageous to tile
- Can we determine the benefit?
  (reuse vector space and locality vector space)
- Is it legal (and if so, how) to transform loop?
  (unimodular transformations)

Handy Representation: "Iteration Space"

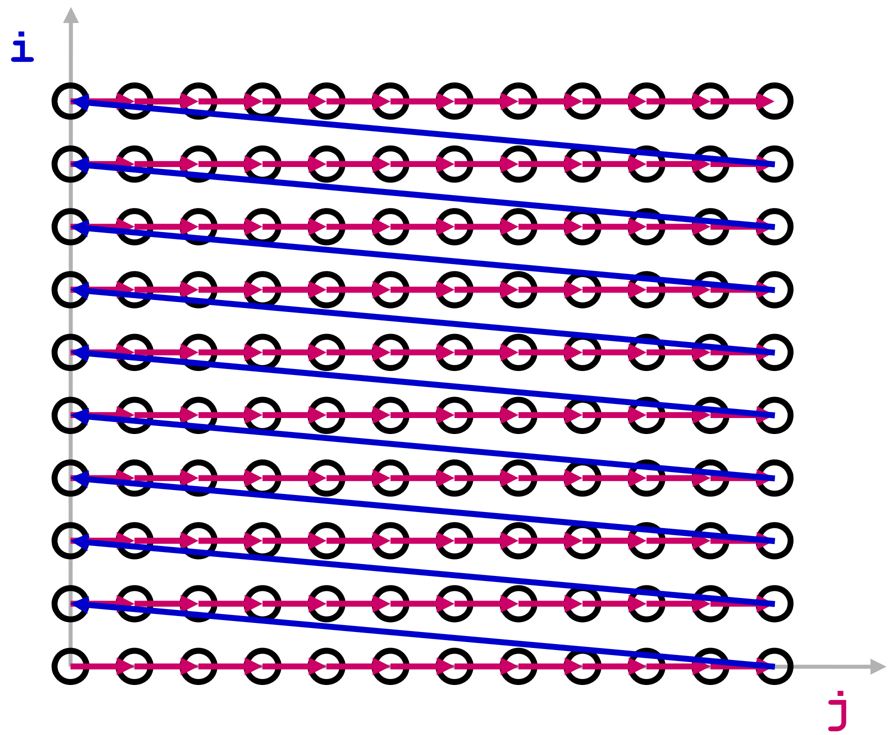
```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```



- each position represents an iteration

Visitation Order in Iteration Space

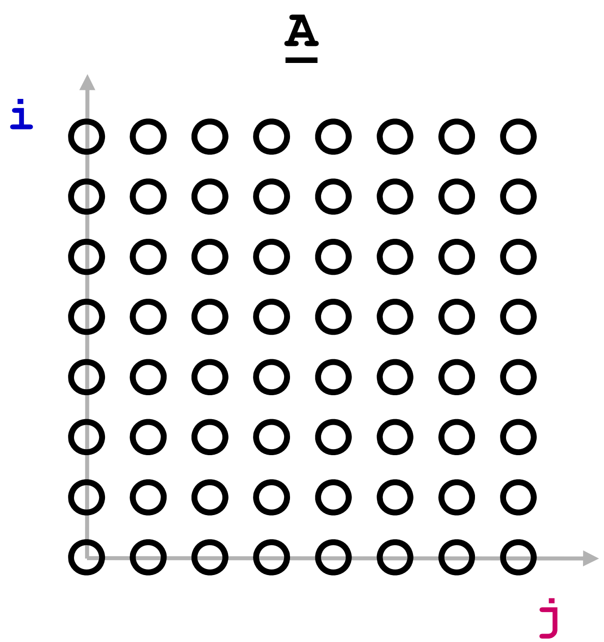
```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

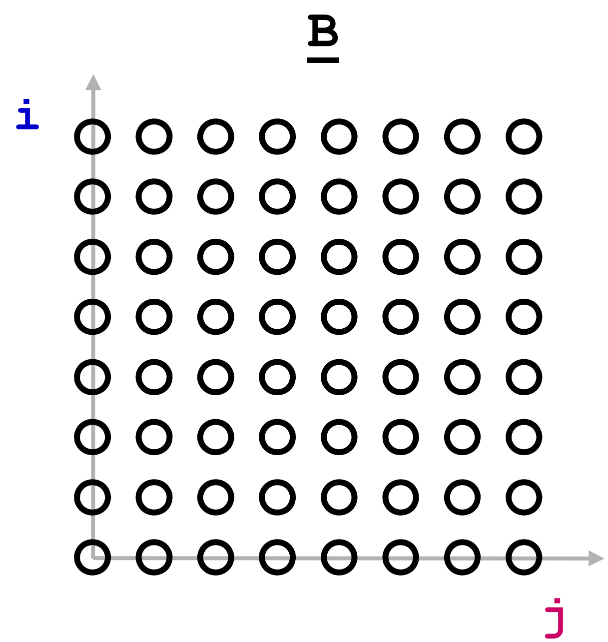


- Note: iteration space is not data space

When Do Cache Misses Occur?

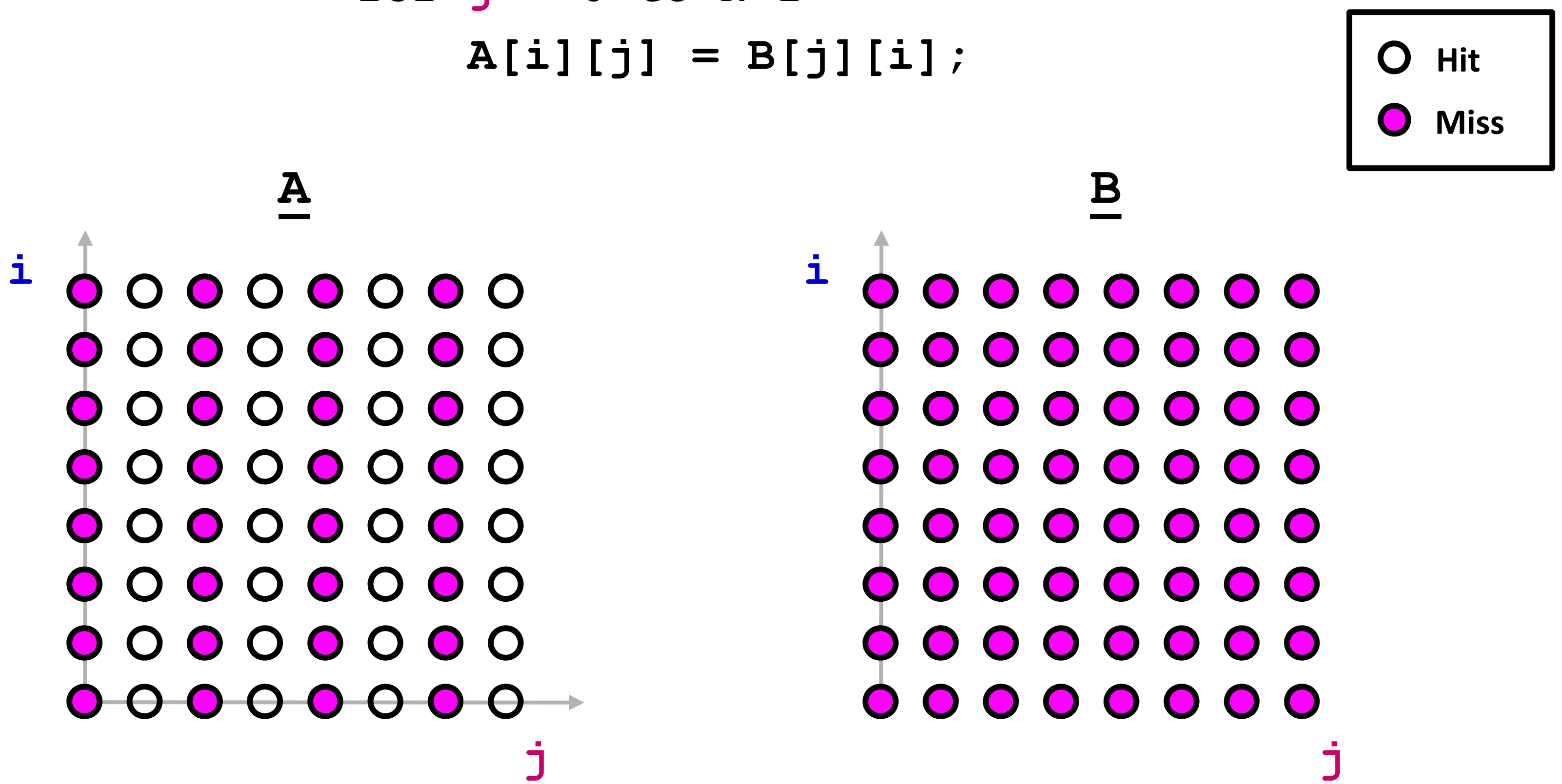
```
for i = 0 to N-1
    for j = 0 to N-1
        A[i][j] = B[j][i];
```





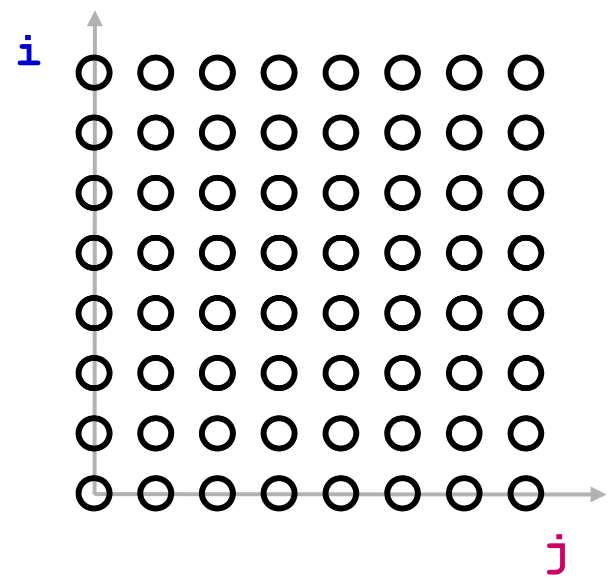
When Do Cache Misses Occur?

```
for i = 0 to N-1
    for j = 0 to N-1
        A[i][j] = B[j][i];
```



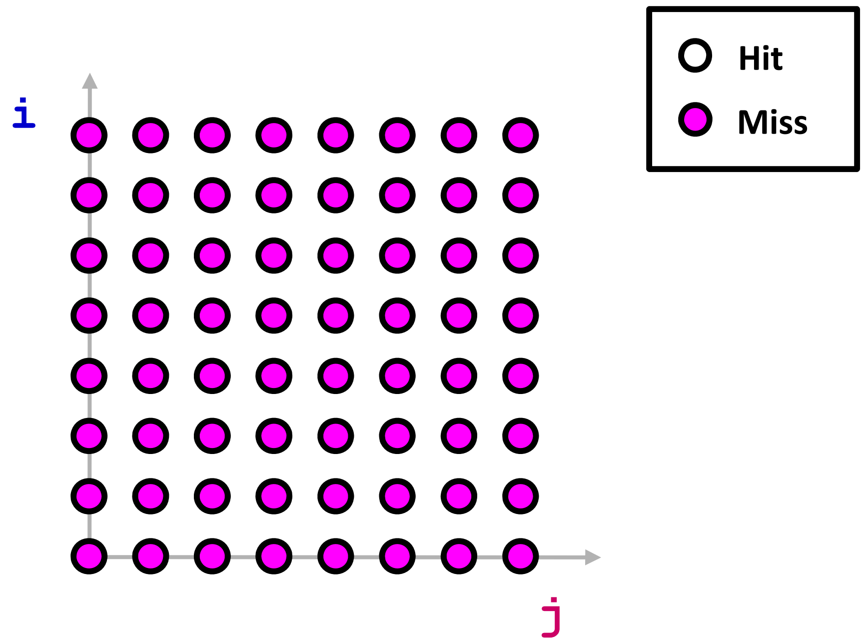
When Do Cache Misses Occur?

```
for i = 0 to N-1
    for j = 0 to N-1
        A[i+j][0] = i*j;
```



When Do Cache Misses Occur?

```
for i = 0 to N-1
    for j = 0 to N-1
        A[i+j][0] = i*j;
```



Optimizing the Cache Behavior of Array Accesses

- We need to answer the following questions:
  - when do cache misses occur?
    - use "locality analysis"
  - can we change the order of the iterations (or possibly data layout) to produce better behavior?
    - evaluate the cost of various alternatives
  - does the new ordering/layout still produce correct results?
    - use "dependence analysis"

Examples of Loop Transformations

- Loop Interchange
- Cache Blocking
- Skewing
- Loop Reversal
- …

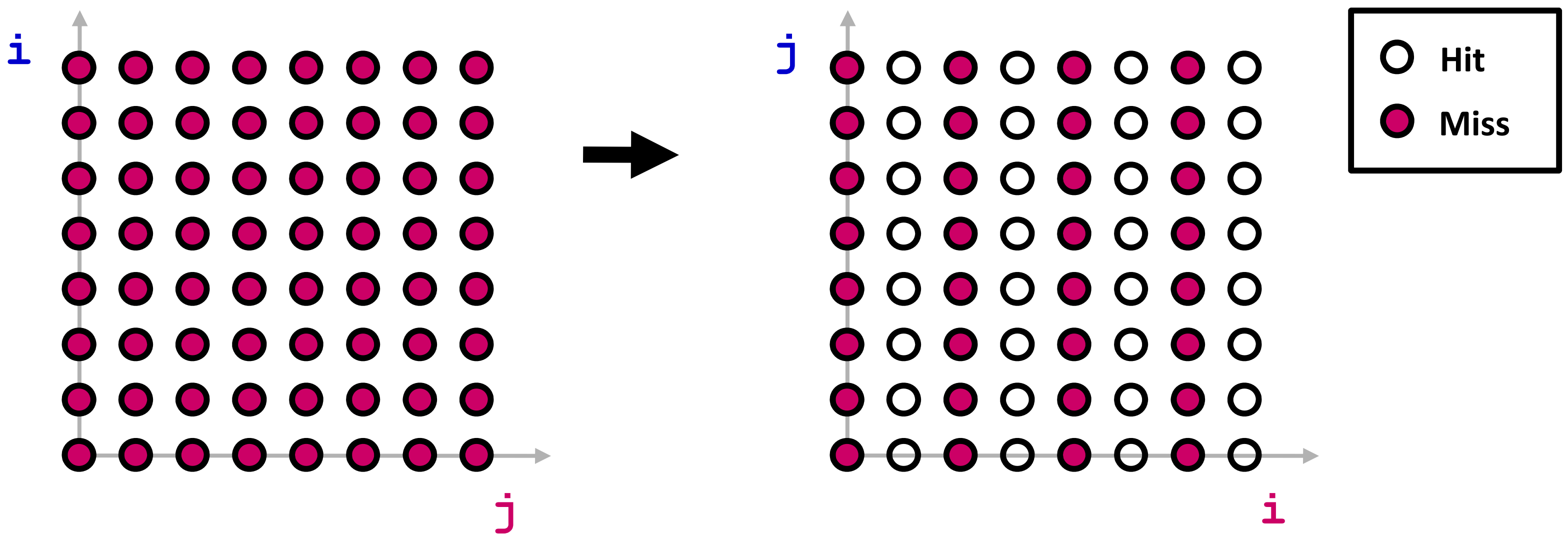
Can improve locality

Can enable above

Loop Interchange

```
for i = 0 to N-1
  for j = 0 to N-1
    A[j][i] = i*j;
```

```
for j = 0 to N-1
  for i = 0 to N-1
    A[j][i] = i*j;
```

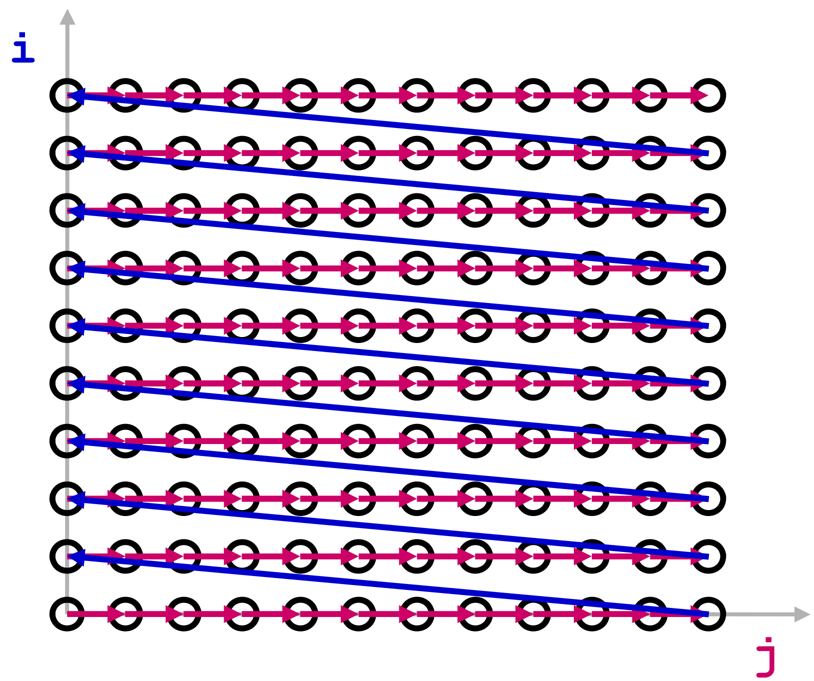


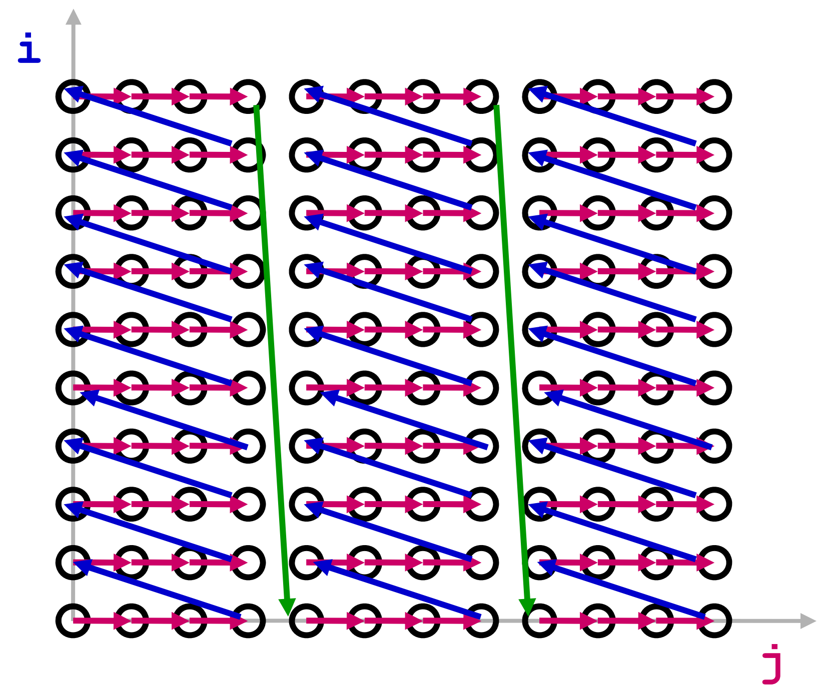
- *(assuming N is large relative to cache size)*

Impact on Visitation Order in Iteration Space

```
for i = 0 to N-1
  for j = 0 to N-1
    f(A[i],A[j]);
```

```
for JJ = 0 to N-1 by B
  for i = 0 to N-1
    for j = JJ to max(N-1,JJ+B-1)
      f(A[i],A[j]);
```

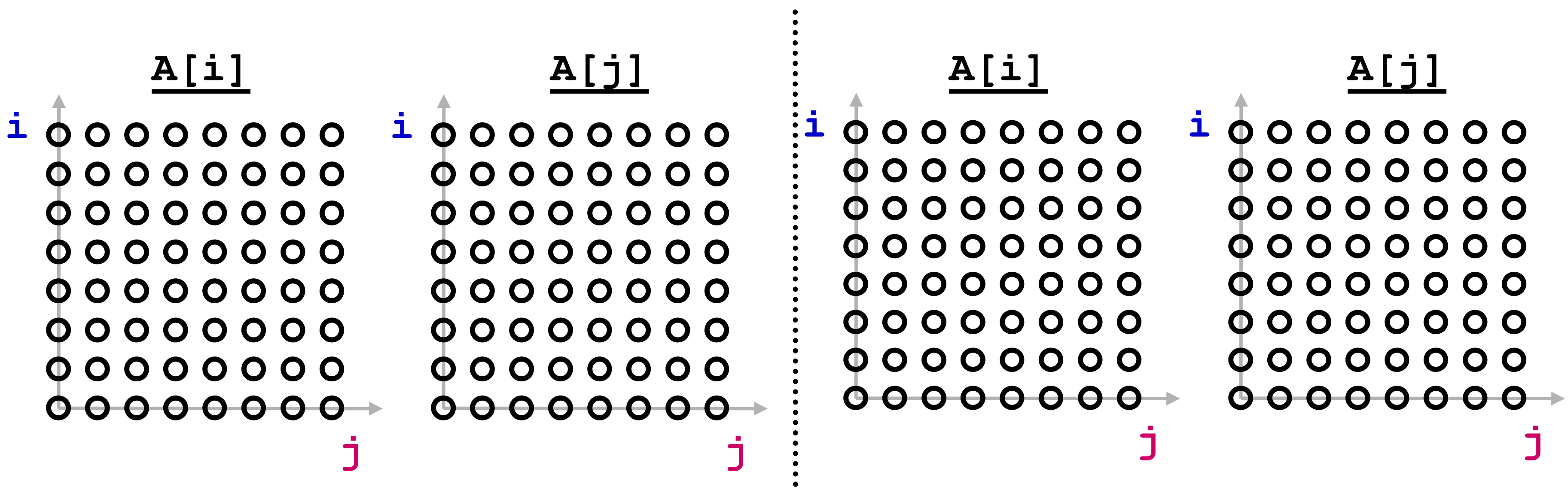




Cache Blocking (aka “Tiling”)

```
for i = 0 to N-1
  for j = 0 to N-1
    f(A[i],A[j]);
```

```
for JJ = 0 to N-1 by B
  for i = 0 to N-1
    for j = JJ to max(N-1,JJ+B-1)
      f(A[i],A[j]);
```

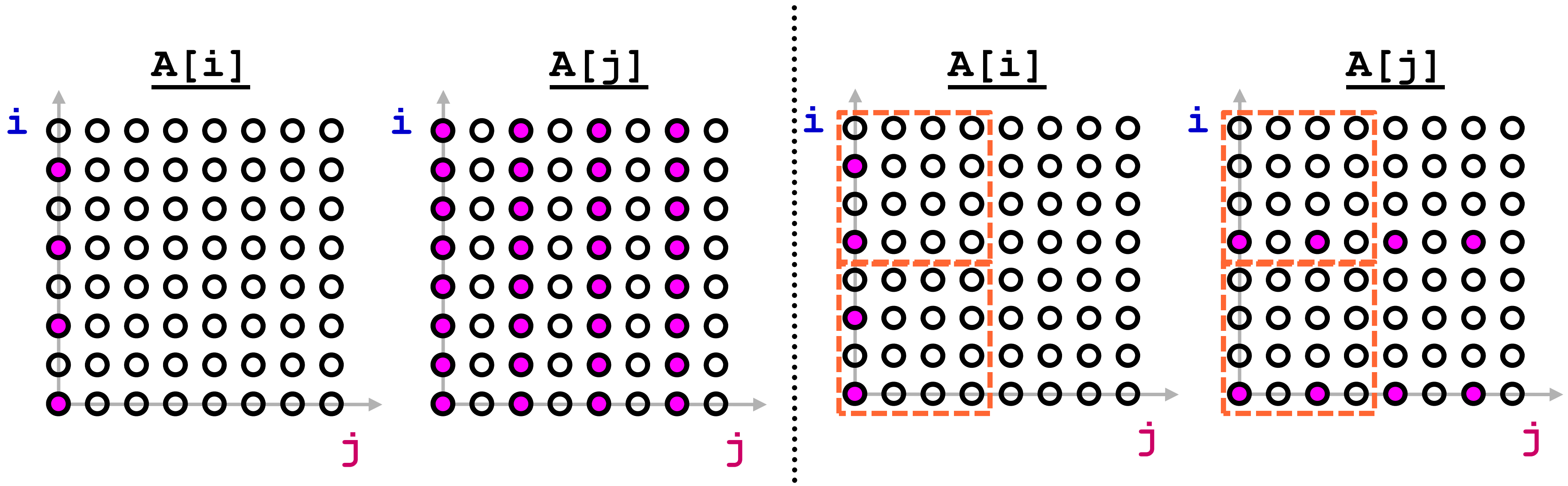


now we can exploit locality

Cache Blocking (aka "Tiling")

```
for i = 0 to N-1
  for j = 0 to N-1
    f(A[i],A[j]);
```

```
for JJ = 0 to N-1 by B
  for i = 0 to N-1
    for j = JJ to max(N-1,JJ+B-1)
      f(A[i],A[j]);
```



now we can exploit temporal locality

Cache Blocking in Two Dimensions

```
for i = 0 to N-1
    for j = 0 to N-1
        for k = 0 to N-1
            c[i,k] += a[i,j]*b[j,k];
```

```
for JJ = 0 to N-1 by B
  for KK = 0 to N-1 by B
    for i = 0 to N-1
      for j = JJ to max(N-1,JJ+B-1)
        for k = KK to max(N-1,KK+B-1)
              c[i,k] += a[i,j]*b[j,k];
```

- brings square sub-blocks of matrix "**`b`**" into the cache
- completely uses them up before moving on

Predicting Cache Behavior through "Locality Analysis"

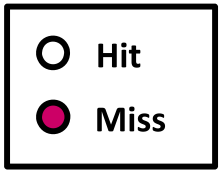
- Definitions:
  - Reuse:
    accessing a location that has been accessed in the past
  - Locality:
    accessing a location that is now found in the cache
- Key Insights
  - Locality only occurs when there is reuse!
  - BUT, reuse does not necessarily result in locality.
  - Why not?

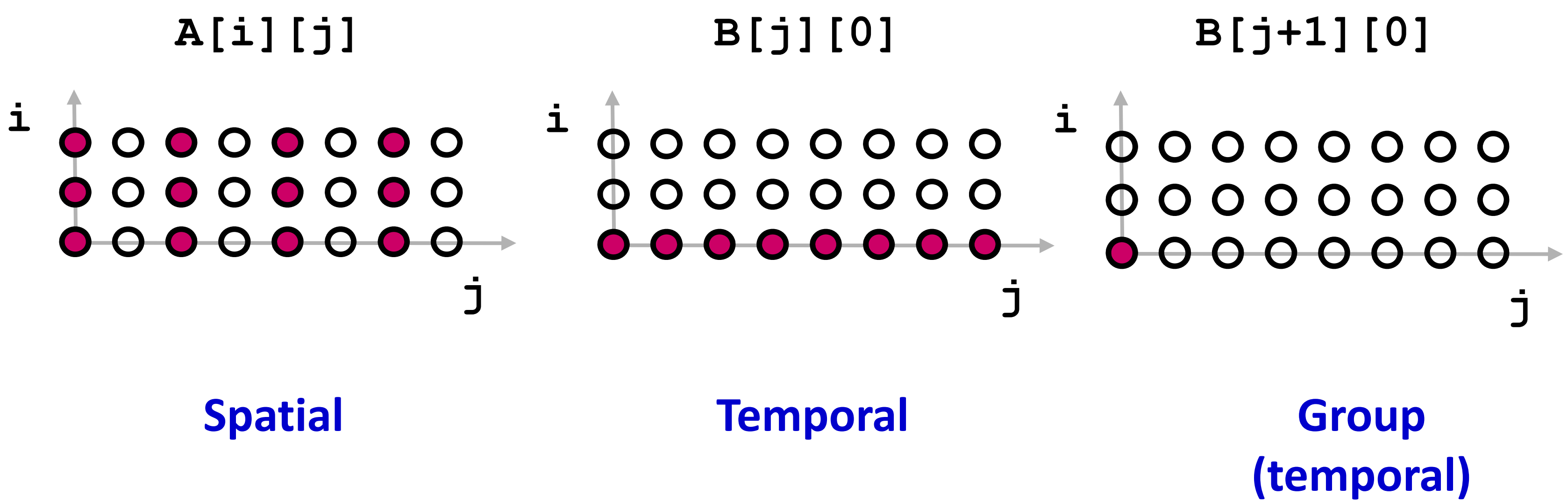
Steps in Locality Analysis

1. Find data reuse
   – if caches were infinitely large, we would be finished
2. Determine "localized iteration space"
   – set of inner loops where the data accessed by an iteration is expected to fit within the cache
3. Find data locality:
   – reuse $\supseteq$ localized iteration space $\supseteq$ locality

Types of Data Reuse/Locality

```
for i = 0 to 2
   for j = 0 to 100
       A[i][j] = B[j][0] + B[j+1][0];
```





Kinds of reuse and the factor

```
for i = 0 to N-1
  for j = 0 to N-1
    f(A[i],A[j]);
```

What kinds of reuse are there?

A[i]?

A[j]?

Kinds of reuse and the factor

for $I_1 := 0$ to 5
  for $I_2 := 0$ to 6
    $A[I_2 + 1] = 1/3 * (A[I_2] + A[I_2 + 1] + A[I_2 + 2])$

Kinds of reuse and the factor

for I_1 := 0 to 5
  for I_2 := 0 to 6
    $A[I_2+1] = 1/3 * (A[I_2] + A[I_2+1] + A[I_2+2])$

self-temporal in 1, self-spatial in 2
Also, group spatial in 2

What is different about this and previous?

```
for i = 0 to N-1
  for j = 0 to N-1
    f(A[i],A[j]);
```

Uniformly Generated references

- f and g are indexing functions: $Z^n \rightarrow Z^d$
  - n is depth of loop nest
  - d is dimensions of array, A
- Two references A[f(**i**)] and A[g(**i**)] are uniformly generated if

$$f(\mathbf{i}) = H\mathbf{i} + c_f \text{ AND } g(\mathbf{i})=H\mathbf{i}+c_g$$

- H is a linear transform
- c_f and c_g are constant vectors

Eg of Uniformly generated sets

These references all belong to the same uniformly generated set: H = [0 1]

for I_1 := 0 to 5
  for I_2 := 0 to 6
    $A[I_2 + 1] = 1/3 * (A[I_2] + A[I_2 + 1] + A[I_2 + 2])$

$A[I_2 + 1]$ $$[\,0\ 1\,]\begin{pmatrix} I_1 \\ I_2 \end{pmatrix} + [\,1\,]$$

$A[I_2]$ $$[\,0\ 1\,]\begin{pmatrix} I_1 \\ I_2 \end{pmatrix} + [\,0\,]$$

$A[I_2 + 2]$ $$[\,0\ 1\,]\begin{pmatrix} I_1 \\ I_2 \end{pmatrix} + [\,2\,]$$

Quantifying Reuse

- Why should we quantify reuse?
- How do we quantify locality?

Quantifying Reuse

- Why should we quantify reuse?
- How do we quantify locality?

- Use vector spaces to identify loops with reuse
- We convert that reuse into locality by making the “best” loop the inner loop
- Metric: memory accesses/iter of innermost loop. No locality → mem access

Self-Temporal

- For a reference, A[H**i**+**c**], there is self-temporal reuse between **m** and **n** when H**m**+**c**=H**n**+**c**, i.e., H(**r**)=**0**, where **r**=**m**-**n**.
- The direction of reuse is **r**.
- The self-temporal reuse vector space is: R_{ST} = Ker H
- There is locality if R_{ST} is in the localized vector space.

Recall that for nxm matrix A,
the ker A = nullspace(A) = $\{x^m \mid Ax = 0\}$

- Reuse is $s^{\dim(Rst)}$
- $R_{ST} \cap L$ = locality
- # of mem refs = $\frac{1}{s^{\dim(R_{ST} \cap L)}}$

Example of self-temporal reuse

```
for I₁ := 1 to n
   for I₂ := 1 to n
      for I₃ := 1 to n
         C[I₁,I₃] += A[I₁,I₂] * B[I₂,I₃]
```

Access	H	ker H	reuse?	Local?
$C[I_1,I_3]$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	span{(0,1,0)}	n in I_2	
$A[I_1,I_2]$	$\begin{pmatrix} \; \\ \; \end{pmatrix}$			
$B[I_2,I_3]$	$\begin{pmatrix} \; \\ \; \end{pmatrix}$			

Example of self-temporal reuse

```
for I1 := 1 to n
   for I2 := 1 to n
      for I3 := 1 to n
         C[I1,I3] += A[I1,I2] * B[I2,I3]
```

Access	H	ker H	reuse?	Local?
$C[I_1,I_3]$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	span{(0,1,0)}	n in I_2	
$A[I_1,I_2]$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$	span{(0,0,1)}		
$B[I_2,I_3]$	$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	span{(1,0,0)}		

Self-Spatial

- Occurs when we access in order
  - A[i,j]: best gain, l
  - A[i,j*k]: best gain, l/k if |k| <= l
- How do we get spatial reuse for uniformly generated H?

Self-Spatial

- Occurs when we access in order
  - A[i,j]: best gain, l
  - A[i,j*k]: best gain, l/k if |k| <= l
- How do we get spatial reuse for UG: H?
- Since all but row must be identical, set last row in H to 0, H_s
  self-spatial reuse vector space = R_{SS}

$$R_{SS} = \ker H_S$$

- Notice, $\ker H \subseteq \ker H_s$
- If, $R_{ss} \cap L = R_{ST} \cap L$, then no additional benefit to SS

Example of self-spatial reuse

```
for I1 := 1 to n
   for I2 := 1 to n
      for I3 := 1 to n
         C[I1,I3] += A[I1,I2] * B[I2,I3]
```

Access	H_s	ker H_s	reuse?	Local?
$C[I_1,I_3]$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$	span{(0,1,0), (0,0,1)}		1/l
$A[I_1,I_2]$	$\begin{pmatrix} & & \\ & & \end{pmatrix}$			
$B[I_2,I_3]$	$\begin{pmatrix} & & \\ & & \end{pmatrix}$			

Example of self-spatial reuse

```
for I₁ := 1 to n
   for I₂ := 1 to n
      for I₃ := 1 to n
         C[I₁,I₃] += A[I₁,I₂] * B[I₂,I₃]
```

Access	H_s	ker H_s	reuse?	Local?
$C[I_1,I_3]$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$	span{(0,1,0), (0,0,1)}		1/l
$A[I_1,I_2]$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$	span{(0,0,1), (0,1,0)}		
$B[I_2,I_3]$	$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$	span{(1,0,0), (0,0,1)}		

Self-spatial reuse/locality

- $\text{Dim}(R_{SS})$ is dimensionality of reuse vector space.
- If $R_{SS}=0 \rightarrow$ no reuse
- If $R_{SS}=R_{ST}$ no extra reuse from spatial
- Reuse of each element is $k/ls^{\dim(R\_SS)}$ where, s is number of iters per dim.
- $R_{SS} \cap L$ is amount of reuse exploited, therefore number of memory references generated is:

$$k/ls^{\dim(R\_ST \cap L)}$$

Group Temporal

- Two refs A[H**i**+**c**] and A[H**i**+**d**] can have group temporal reuse in L iff
  – they are from same uniformly generated set
  – There is an $\mathbf{r} \in L$ s.t. $H\mathbf{r} = \mathbf{c} - \mathbf{d}$
- if $\mathbf{c}-\mathbf{d} = \mathbf{r_p}$, then there is group temporal reuse, $R_{GT} = \ker H + \mathrm{span}\{\mathbf{r_p}\}$
- However, there is no extra benefit if $R_{GT} \cap L = R_{ST} \cap L$

Example:

```
for i = 1 to n
  for j=i to n
    A[i,j] = 0.2*(A[i,j]+A[i+1,j]+
                  A[i-1,j]+A[i,j+1]+A[i,j-1])
```

If L = span{j}, since ker H = $\emptyset$:

A[i,j] and A[i,j-1] → (0,0)-(0,-1) $\in$span{(0,1)} yes

A[i,j-1] and A[i+1,j] → (0,-1)-(1,0) $\notin$span{(0,1)} no

Notice equivalence classes

Evaluating group temporal reuse

- Divide all references from a uniformly generated set into equiv classes that satisfy the R_{GT}
- For a particular L and g references
  - Don't count any group reuse when $R_{GT} \cap L = R_{ST} \cap L$
  - number of equiv classes is g_T.
  - Number of mem references is g_T instead of g

Total memory accesses

- For each uniformly generated set
  localized space, L
  line size, z

$$\frac{g_S + (g_T - g_S)/z}{z^e s^{\dim(R\_SS \cap L)}}$$

$$\text{where } e = \begin{cases} 0 & \text{if } R_{ST} \cap L = R_{SS} \cap L \\ 1 & \text{otherwise} \end{cases}$$

Now what?

- We have a way to characterize
  - Reuse (potential for locality)
  - Local iteration space
- Can we transform loop to take advantage of reuse?
- If so, can we?

Our Path

- Finding Loops
- Loop Invariant Code Motion (LICM)
- Partial Redundancy Elimination aka Lazy Code Motion (subsumes LICM)
- Understanding Dependencies
- Understanding Locality
- **SRP**
- Finding Dependencies
- Scheduling