Lecture 15:

Memory Hierarchy Optimizations

I. Caches: A Quick Review
II. Iteration Space & Loop Transformations
III. Types of Reuse

ALSU 7.4.2-7.4.3, 11.2-11.5.1

I. Caches: A Quick Review

- How do they work?
- Why do we care about them?
- What are typical configurations today?
- What are some important cache parameters that will affect performance?

Optimizing Cache Performance

- Things to enhance:
  - temporal locality
  - spatial locality

- Things to minimize:
  - conflicts (i.e. bad replacement decisions)

What can the *compiler* do to help?

Two Things We Can Manipulate

- Time:
  - When is an object accessed?
- Space:
  - Where does an object exist in the address space?

How do we exploit these two levers?

Time: Reordering Computation

- What makes it difficult to know *when* an object is accessed?
- How can we predict a better time to access it?
  - What information is needed?
- How do we know that this would be safe?

Space: Changing Data Layout

- What do we know about an object's location?
  - scalars, structures, pointer-based data structures, arrays, code, etc.
- How can we tell what a better layout would be?
  - how many can we create?
- To what extent can we safely alter the layout?

Types of Objects to Consider

- Scalars
- Structures & Pointers
- Arrays

Scalars

- Locals
- Globals
- Procedure arguments

```
int x;
double y;
foo(int a){
  int i;
  …
  x = a*i;
  …
}
```

- Is cache performance a concern here?
- If so, what can be done?

Structures and Pointers

- What can we do here?
  - within a node
  - across nodes

```
struct {
    int count;
    double velocity;
    double inertia;
    struct node *neighbors[N];
} node;
```

Example: Can rearrange field order to improve cache performance

- What limits the compiler's ability to optimize here?

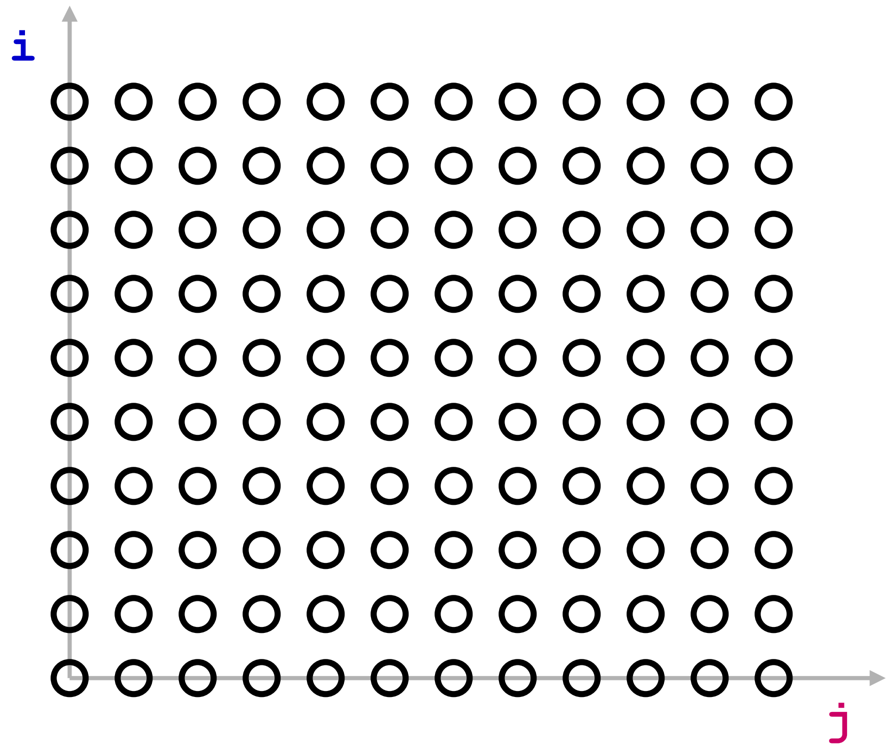
Arrays / Matrices

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

- usually accessed within loops nests
  - makes it easy to understand “time”
- what we know about array element addresses:
  - start of array?
  - relative position within array

II. Iteration Space and Loop Transformations

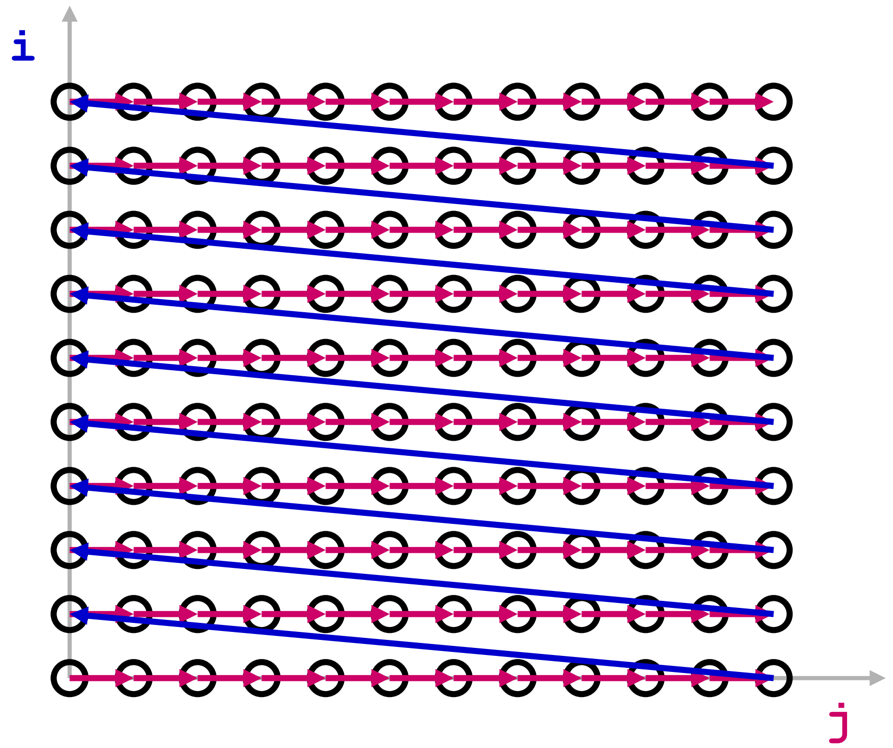
```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```



- each position represents an iteration (not an array element)

Visitation Order in Iteration Space

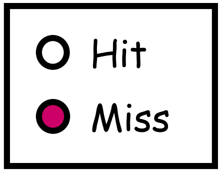
```
for i = 0 to N-1
   for j = 0 to N-1
      A[i][j] = B[j][i];
```

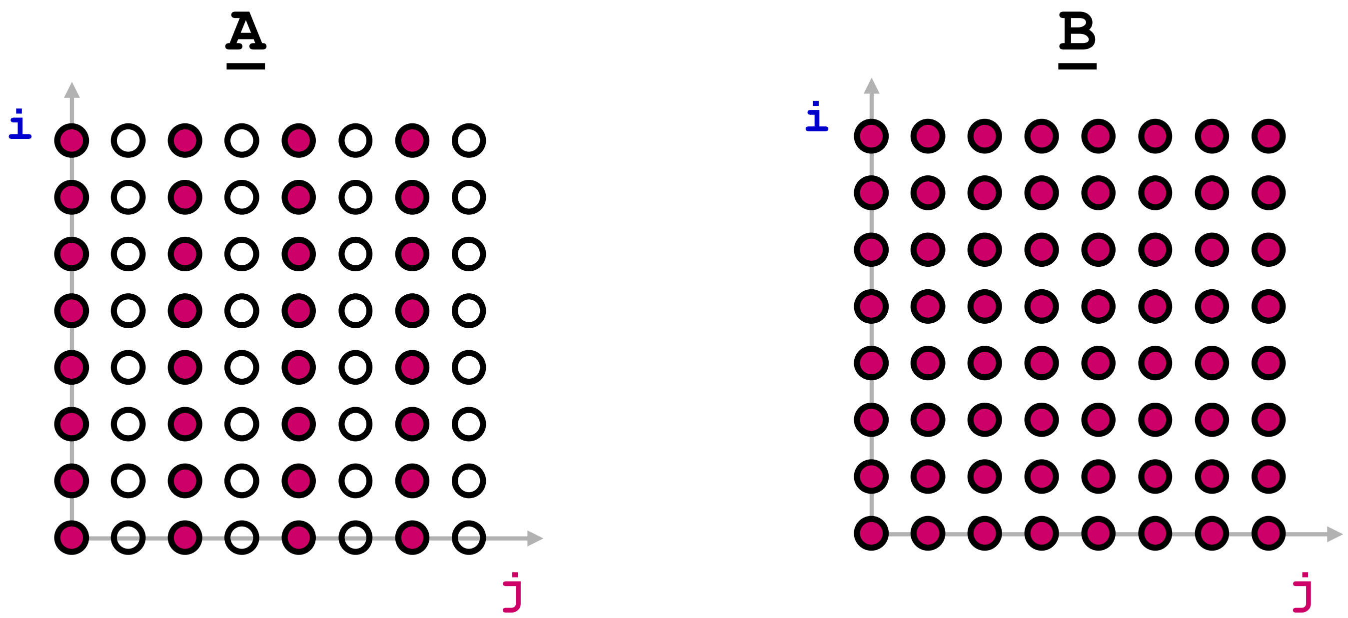


- Note: iteration space ≠ data space

When Do Cache Misses Occur?

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





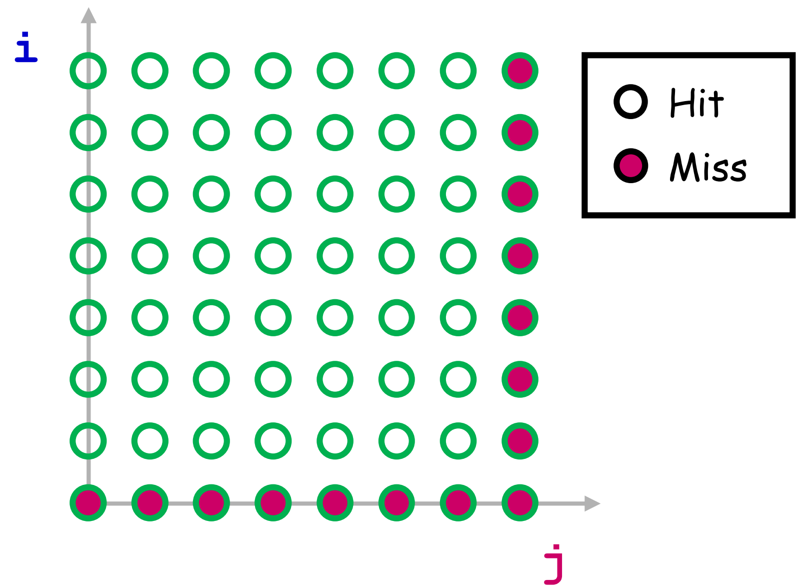
Assume row major order, N large, 2 elements per cache line

Row major layout: A[0][0] A[0][1]…A[0][N-1] A[1][0] A[1][1]…A[1][N-1] A[2][0]…

When Do Cache Misses Occur?

```
double A[2N-1][N];

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



Assume row major order, 2 elements per cache line

Row major layout of A:
A[0][0] A[0][1]…A[0][N-1] A[1][0]…A[1][N-1]…A[2N-2][0]…A[2N-2][N-1]

If N large then all misses. What if N is small? see above

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

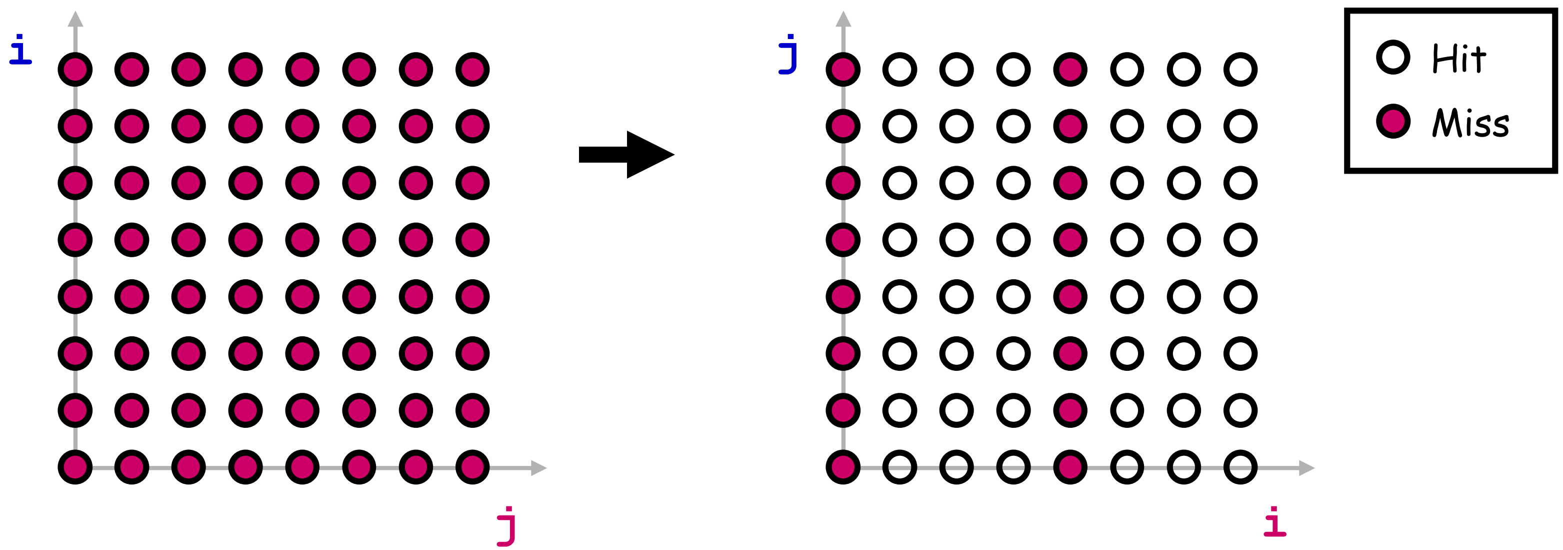
- Loop Interchange
- Cache Blocking
- Skewing: iterate through iteration space in the loops at an angle
- Loop Reversal: execute iterations in a loop in reverse order
- …

(we will briefly discuss the first two;
see ALSU 11.7.8 for others)

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;
```



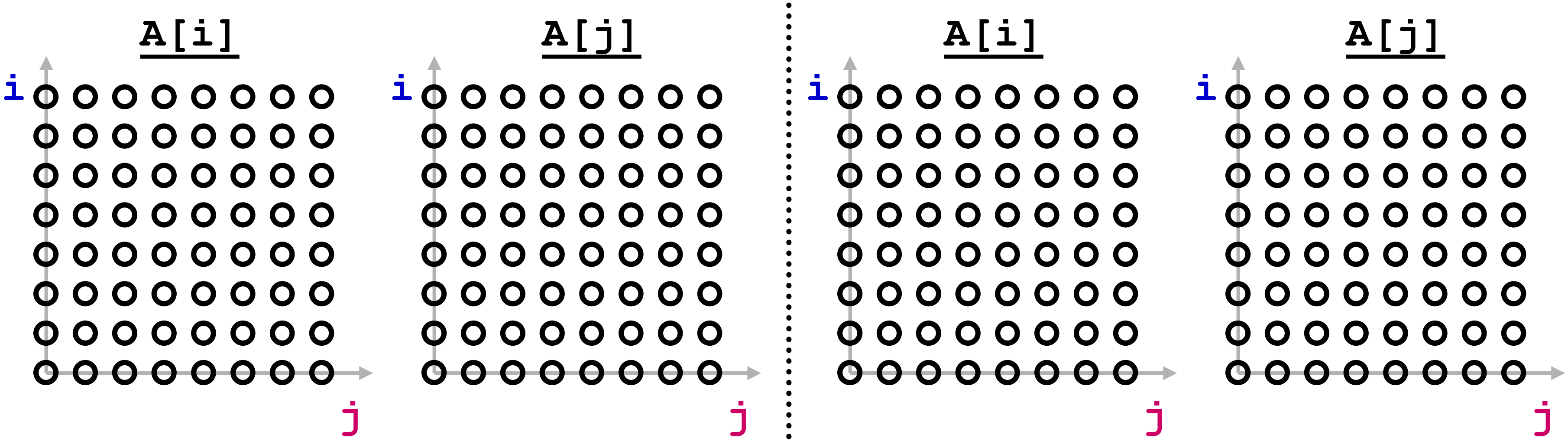
Assume row major order, N large, 4 elements per cache line

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

L elements per cache line

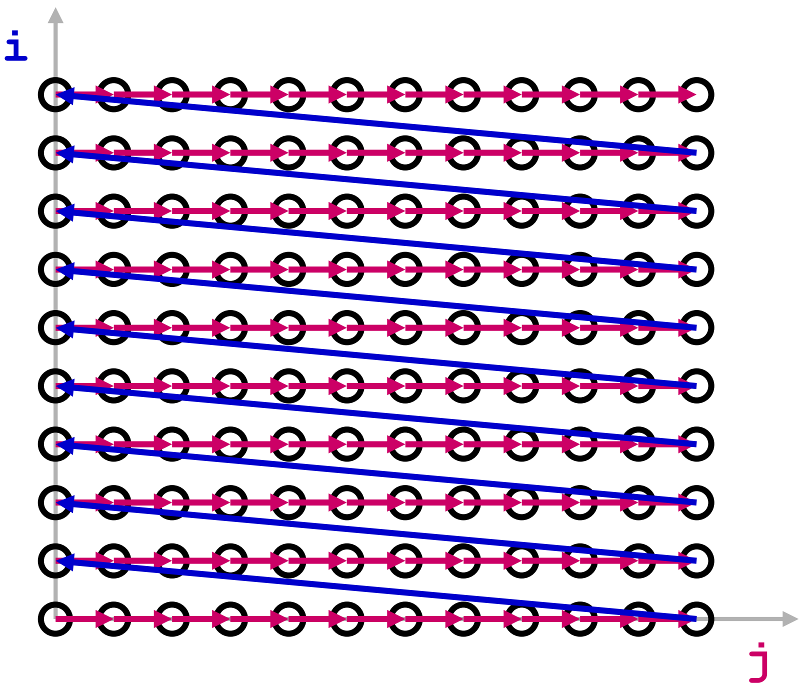


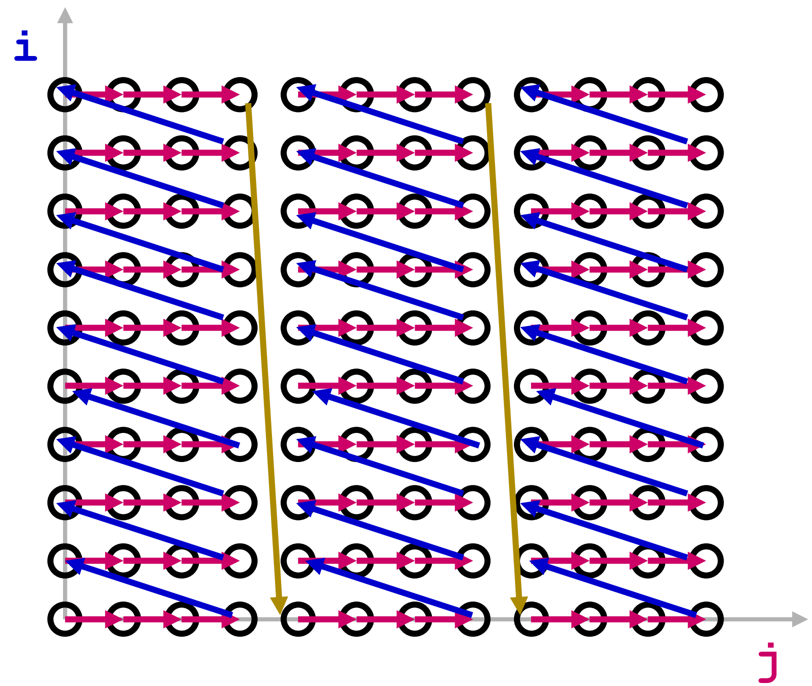
now we can exploit temporal locality

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





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];
```

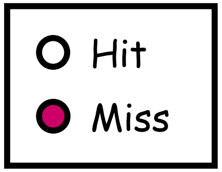
```
for JJ = 0 to N-1 by B
  for KK = 0 to N-1 by B
    for i = 0 to N-1
      for j = JJ to min(N-1,JJ+B-1)
        for k = KK to min(N-1,KK+B-1)
          c[i,k] += a[i,j]*b[j,k];
```

- brings square sub-blocks of matrix "**b**" into the cache
- completely uses them up before moving on
- reduces the number of misses from $\frac{N^3}{L}$ or N^3 to only $\frac{2N^3}{L\,C}$ (C=cache size, L=line size)

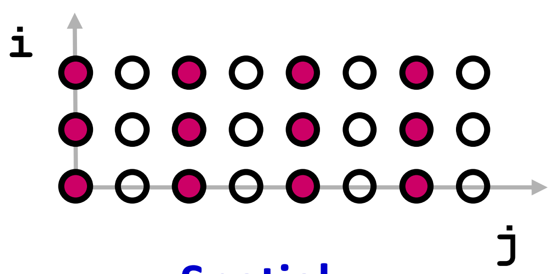
III. Types of Data Reuse/Locality

```
double A[3][N], B[N][3];

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

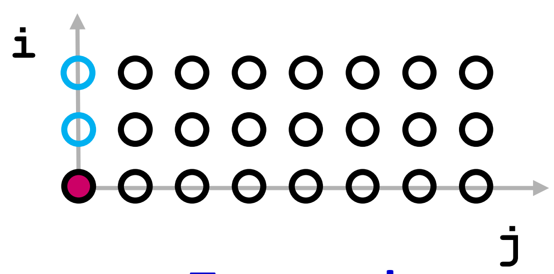


A[i][j]



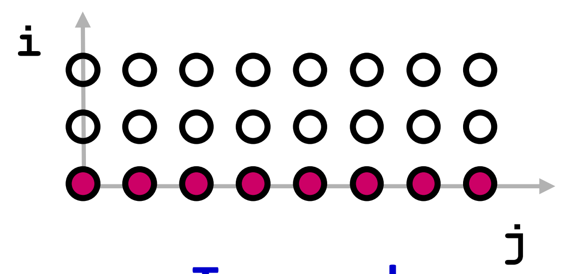
Spatial
(Self)

B[j][0]



Temporal
(Group)
except for ○

B[j+1][0]



Temporal
(Self)

(assume row-major, 2 elements per cache line, N small)

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 $\cap$ localized iteration space $\Rightarrow$ locality

To be covered in a future lecture...

Today's Class: Memory Hierarchy Optimizations

I. Caches: A Quick Review
II. Iteration Space & Loop Transformations
III. Types of Reuse

At 3 pm Today

- Sign up for Discussion Topics & Slots

Monday's Class

- Brian Railing, Guest Lecture on Compiler-based Instrumentation