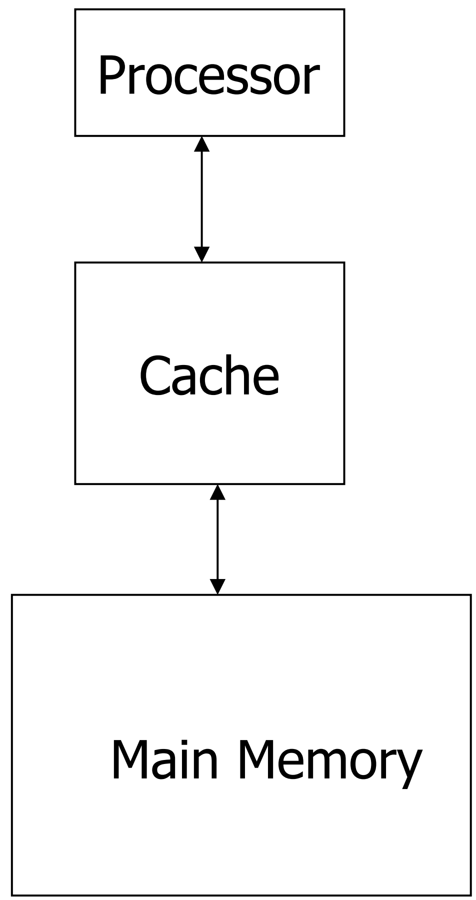
Memory Reuse Analysis (Objectives)

- To be able to define the types of memory reuse
- To be able to compute the reuse properties of array references and loops
- To be able to calculate loop cost
- To be able to use loop cost to determine best loop permutation

Memory Hierarchy

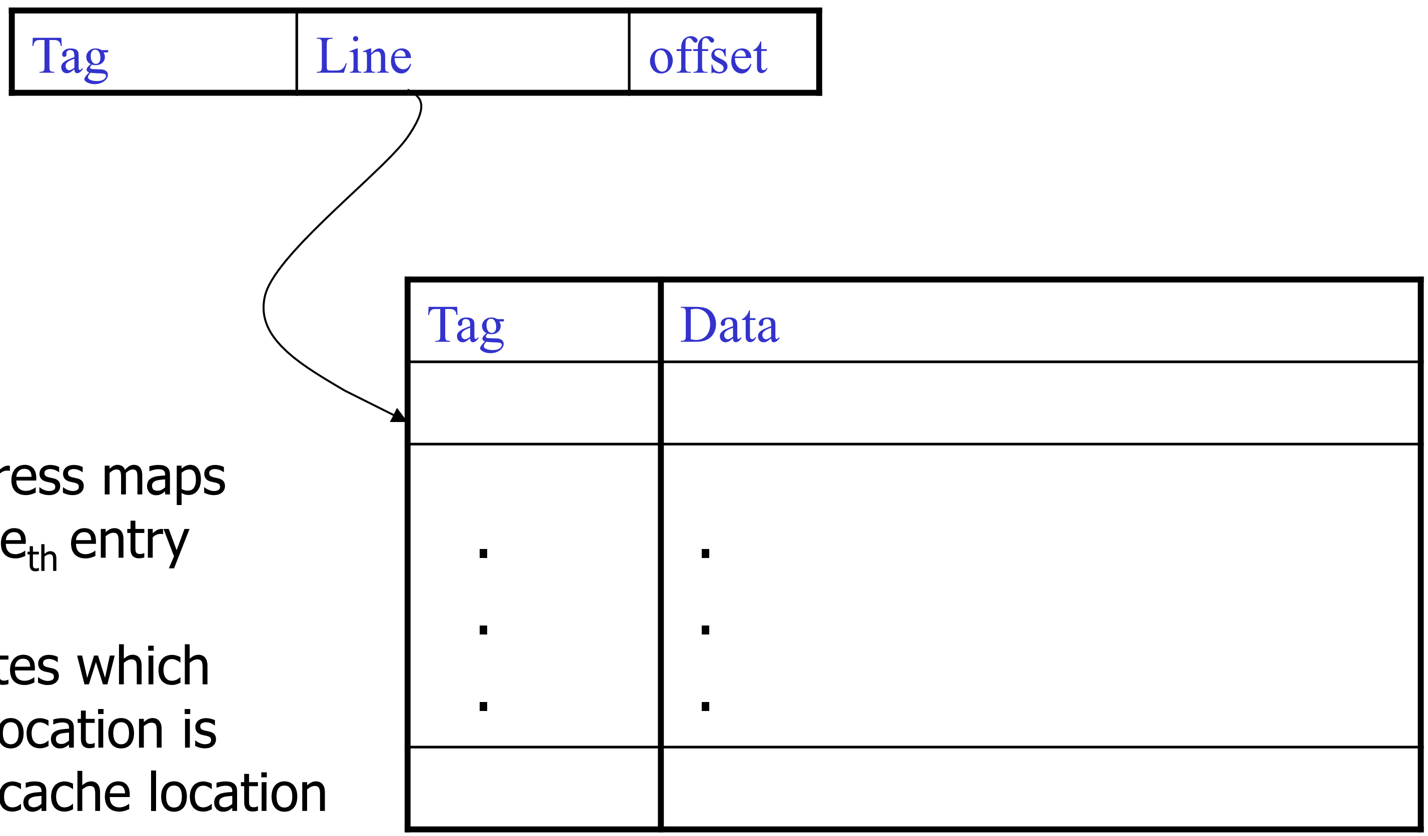
Keys:
- Inclusion
- Speed



May have multiple levels
(on- or off- chip)

Basic Cache Structure (direct-mapped)

Address



N entries

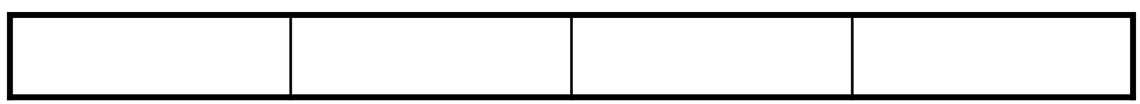
Each address maps to the Line$_{th}$ entry

Tag denotes which memory location is stored in cache location

Cache Line (Block)

- The unit of memory in a cache is a line (block)
- A line may have 1 or more words (a word is typically 4 or 8 bytes)
- Typical Cache Line

32 bytes

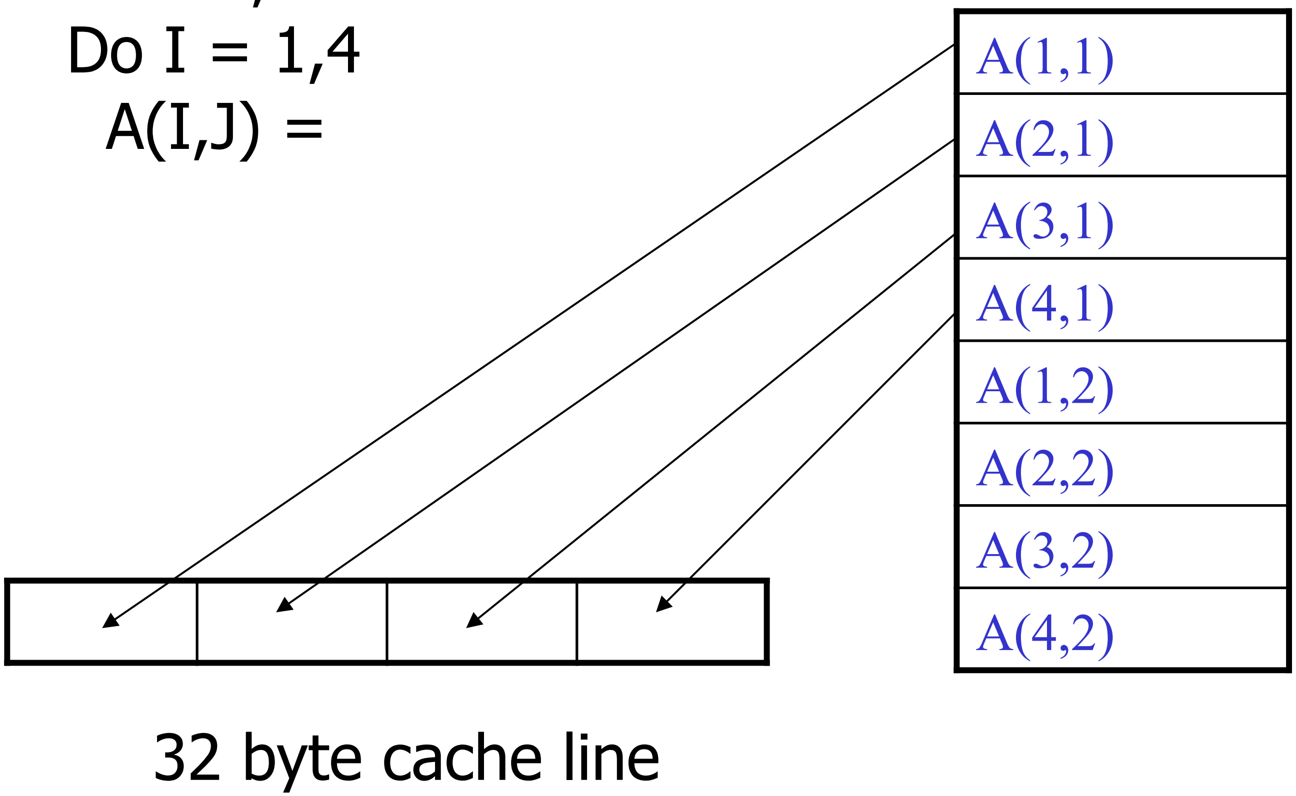


- Accessing any member of the line brings the entire line into cache (replacing whatever was there previously)
- Interference - multiple lines that map to the same cache location and need to be in the cache at the same time

Stride One Access

- If cache size is 1 block then we get a 75% hit rate

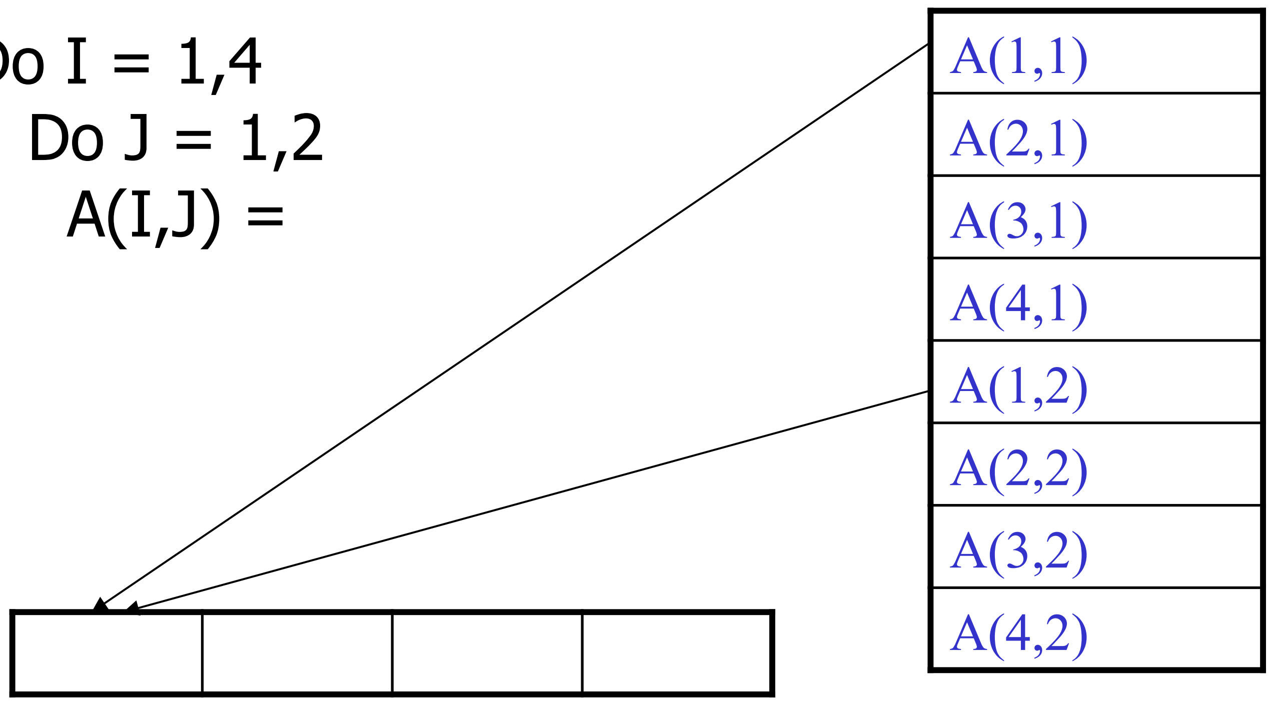
```
Do J = 1,2
  Do I = 1,4
    A(I,J) =
```



Long Stride Access

- If cache size is 1 block them we get a 0% hit rate
- If cache size is large enough, we get a 75% hit rate

```
Do I = 1,4
   Do J = 1,2
     A(I,J) =
```



32 byte cache line

Temporal Reuse Definition

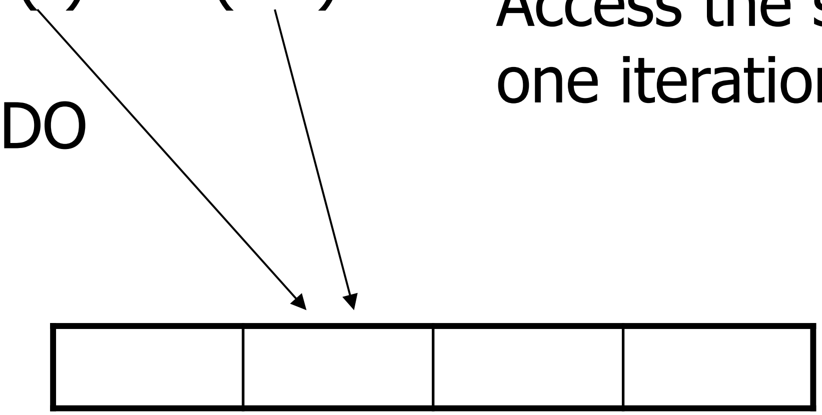
- Temporal reuse - reuse of the same memory location

```
Do I = 2, N

    A(I) = A(I-1)

ENDDO
```

Access the same element one iteration apart



Note that the true dependence from A(I) to A(I-1) tells that the temporal reuse exists

Spatial Reuse Definition

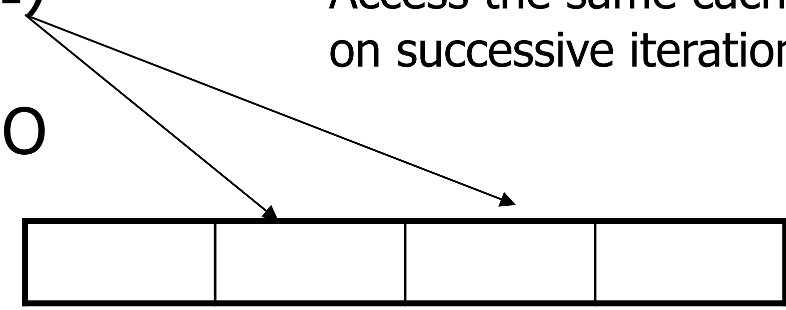
- Spatial reuse - reuse of a cache block with a nearby memory reference

```
Do I = 1, N

    A(I) =

ENDDO
```

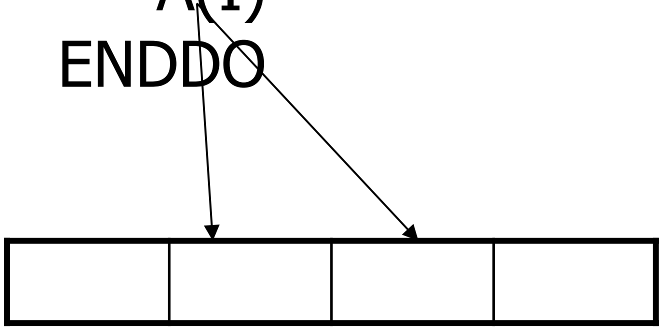
Access the same cache line on successive iterations



Self Reuse Definition

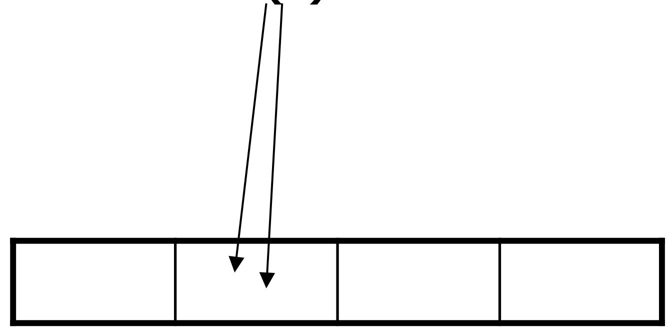
- self reuse – reuse that arises due to a single static reference
  - Self temporal
  - Self spatial

```
DO I = 1, N
   A(I) =
ENDDO
```



Access the same cache line
on successive iterations

```
DO I = 1, N
   DO J=1,N
      A(I) =
```

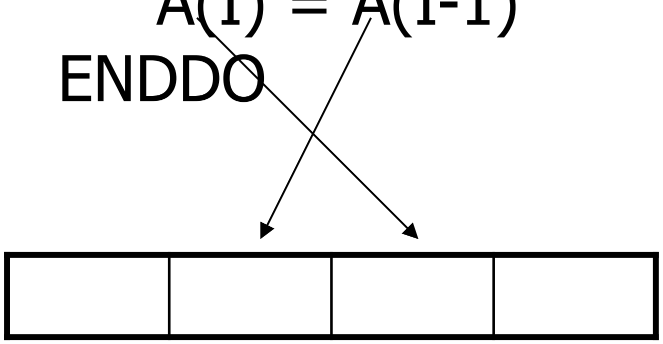


Access the same cache location
on successive iterations

Group Reuse Definition

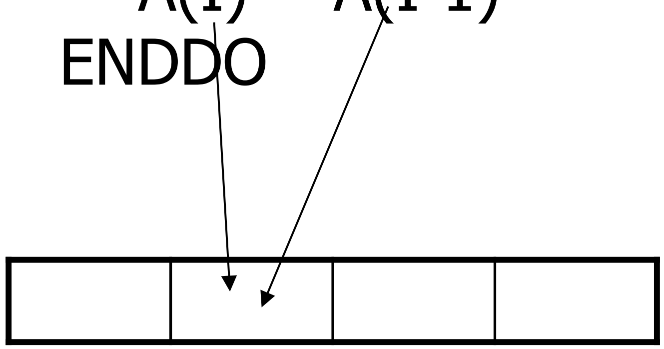
- group reuse - reuse that arises due to multiple static references
  - Group temporal
  - Group spatial

```
DO I = 1, N
    A(I) = A(I-1)
ENDDO
```



Access the same cache line
on same iteration (group spatial)

```
DO I = 1, N
    A(I) = A(I-1)
ENDDO
```



Access the same element one
iteration apart (group temporal)

Computing Reuse

- Optimization-oriented
- One example (McKinley, Carr & Tseng 96)
  - Compute reuse across the innermost loop only
  - Self reuse
    - Examine the subscript
    - Temporal - missing the innermost induction variable
    - Spatial - innermost induction variable in the 1st subscript position only
  - Group reuse
    - look at reference connected by a dependence carried by innermost loop or loop independent

RefGroup Definition

- Represent group reuse
- All references in one reference group will have some kind of group reuse
- Two references R_1 and R_2 are in the same reference group with respect to Loop L if either of the following holds
  - $R_1 \vec{\delta} R_2$ (group-temporal reuse)
    - $\vec{\delta}$ is a loop-independent dependence, or
    - δ_L is a small constant d and all other entries are zero
  - R_1 and R_2 differ in the in the first subscript dimension by a small constant d and all other subscripts are identical. (group-spatial reuse) (d is decided by cache line size and array element size)

RefGroup Example

```
DO K = 1, N
   DO J=1,N
      DO I=1,N
```

(0,1,0) (0,0,1)

A(I,J,K) = B(I,J+1,K)+B(I,J,K)+ B(I+1,J,K)

(0,1,-1)

- What are the RefGroups with respect to K, J and I?

RefGroup Leader

- The reference that brings in the cache lines accessed by other members
- Leader is
  - the reference without an incoming dependence
  - or all outgoing edges are loop carried and references are invariant at source and sink
  - or any incoming edge is from a reference that is not in the RefGroup

Leader Example

```
DO K = 1, N
   DO J=1,N
      DO I=1,N
```

(0,1,0)   (0,0,1)

```
         A(I,J,K) = B(I,J+1,K)+B(I,J,K)+ B(I+1,J,K)
```

(0,1,-1)

```
         C(J) = C(J) + A(I,J,K)
```

- What are the leaders?

Loop Cost

- Loop Cost gives the number of cache lines that are brought into the cache if a given loop is innermost. This relates to the locality of the loop.
- Given the RefGroups we can compute the cost of a loop, *LC(l)* by summing the cost of each RefGroup, *RC(R, l)*.
  - Let R be the leader of a RefGroup

$$LC(l) = \sum_{1 \le k \le n} RC(R_k, l) \times \prod_{h \ne l} trip_h$$

where $trip_h$ is the number of loop iterations for loop h

RefGroup Cost

$$RC(R_k, l) = \begin{cases} 1 & if \quad R_k \text{ is self-temporal} \\ \dfrac{trip_l}{cls / stride} & if \quad R_k \text{ is self-spatial} \\ trip_l & if \quad R_k \text{ has no self-reuse} \end{cases}$$

Loop Cost Example

```
DO K = 1, N
   DO J=1,N
      DO I=1,N
```

(0,1,0) (0,0,1)

A(I,J,K) = B(I,J+1,K)+B(I,J,K)+ B(I+1,J,K)

(0,1,-1)

- What is the loop cost of K, J, and I, assuming cls=4 words?

Loop Permutation (Objectives)

- To be able to apply loop-cost analysis to program transformation
- To be able to correctly adhere to the constraints imposed on program transformation by dependences

How To Use Loop Cost

- The loop cost gives the number of cache lines accessed by a loop as if it were innermost
- The number of cache lines is a measure of locality
- Lower loop cost implies better locality
- Order loops based on decreasing cost from outer to inner

Loop Permutation Safety

- **Is it always safe?**
  - No, permuting a nest is legal only if no true, anti or output dependence changes direction
- Example

```
Do I = 1,N
   Do J = 1,N
      A(I,J) = A(I-1, J+1)
```

(1, -1), (<, >)

If we permute the loop ordering to be J, I the dependence becomes (>,<) which is an anti-dependence in the opposite direction.

Essentially, we cannot permute the loop such that a ">" would be in the outermost non-equal/non-zero entry of a direction vector of a true, anti or output dependence (input is irrelevant).

Nearby Permutation

- P will contain the new loop order, L is the order sorted by decreasing loop cost

```
P =∅ ;  k = 0; m = |L|
While L <> ∅ do
    for j=1 to m do
        if  {P_1, …, P_k, L_j} is a legal ordering
        then
            P = {P1, …, Pk, Lj};
            L -= L_j; m--; k++;
        endif
    endfor
endwhile
```

Nearby Permutation Example

```
DO I = 1, N
   DO J=1,N
        DO K=1,N

           A(I,J,K) = A(I+1,J-1,K)+B(J,I,K)
```