Loop Optimizations

Nikhil Hegde

Compiler Optimizations course @ QUALCOMM India Pvt. Ltd.

Loop optimization Techniques

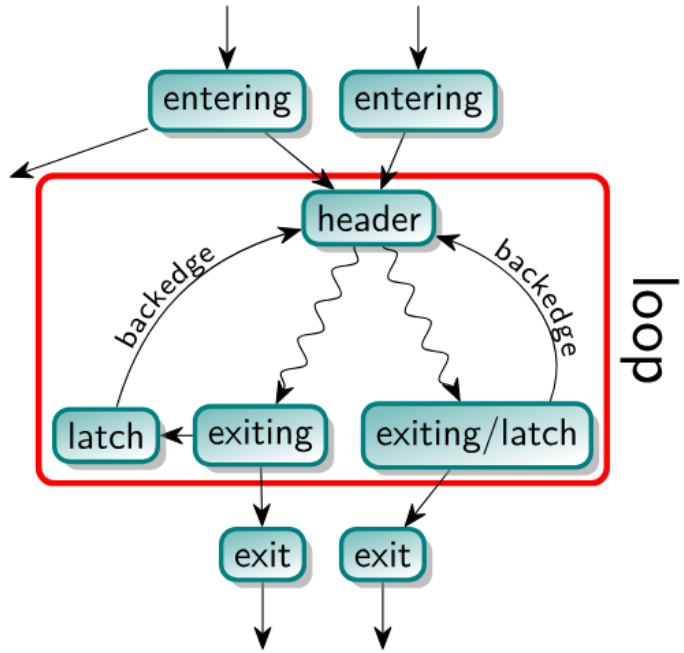
- Compiler based
  - Tiling, Fusion, Distribution, LICM
  - Polyhedral compilation
  - Source-to-source transformation: PLUTO, ROSE
- Library based
  - MKL, GotoBLAS
- DSL
  - Halide, SQL, etc.

Recap: Natural loops

- There is a cycle in the CFG (necessary but not sufficient)
- There is only one vertex with in-edges from outside the set. That one vertex is called the loop-entry/loop-header block. (also the set needs to be strongly connected i.e. there is a path from every vertex to every other vertex in the set).
- For a *backedge A->B,* the *smallest* set of vertices L including A and B such that $\forall\ v \in L,\ predecessors(v) \subseteq L\ or\ v = B$
  - Backedge – an edge from A to B, where B dominates A
  - How to find backedges?

  Do a breadth-first traversal of CFG. Edges that you encounter when discovering new vertices are forward edges. Everything else are backedges.

Natural loops – LLVM Terminology

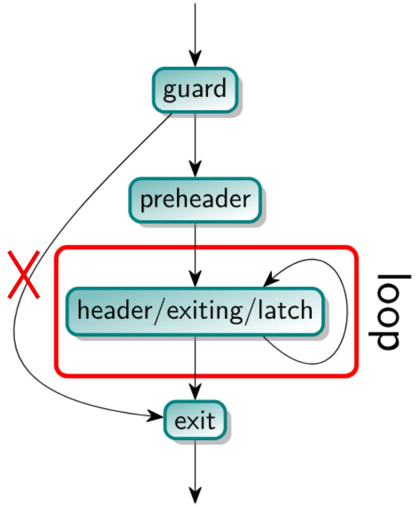


source: https://llvm.org/docs/LoopTerminology.html

- Entering block – BB, that has an edge to a loop node (header). Also called preheader if there is only one such block.
- Latch – loop node that has the edge to header (/ source of a backedge)
- Backedge – edge from latch to header
- Exiting block – BB that takes you to a node, which is outside the loop
- Exit block – BB that

Canonical form of loops

- Adheres to certain restrictions that make it simpler to analyse loops
- E.g. loop simplify form (`passes=loop-simplify`)
  - Has one preheader
  - Has one latch
  - All loop exits (exit blocks) are dominated by loop header



Canonical form of loops

- E.g. loop rotation
  - Converts to a do-while style loop

```
void test(int n) {
        for (int i = 0; i < n; i += 1)
                // Loop body
}
```

```
void test(int n) {
        int i = 0;
        do {
                // Loop body
                i += 1;
        } while (i < n);
}
```

Try it yourself

- For a sample program:
  1. Use the indvars pass to simplify induction variables
  2. Use the loop-simplify pass to view loops in canonical form

Loop Optimizations

- Loop Invariant Code Motion (`passes=licm`)
  - Saw this earlier while looking at an application of computing dominator info

```
for I = 1 to 100
    for J = 1 to 100
        for K = 1 to 100
            A[I][J][K] = (I*J)*K
```

```
for I = 1 to 100
    temp3=A[I]
    for J = 1 to 100
        temp1=temp3[J]
        temp2=I*J
        for K = 1 to 100
            temp1[K] = temp2*K
```

Certain conditions must hold before factoring out invariant expressions (refer to session_2 slides)

Loop Optimizations

- Induction Variable Simplification (`passes=indvars`)

```
for(i=0;i<100;i++)
   foo(a[i])
```

```
a_end=a+stride*100
for(a_i=a;a_i<a_end;a_i+=stride)
   foo(a_i)
```

  - `stride` is the size of array element.
  - `a[i] = a + i*stride //involves an addition and multiplication`
  - Transformation eliminates costly multiplication
  - Demo: `make indvarssimplify` in codeexamples
- Usually followed by Dead Code Elimination and Copy Propagation

Loop Optimizations

- Low level optimization
  - Moving code around in a single loop
  - Examples: loop invariant code motion, strength reduction, loop unrolling
- High level optimization
  - Restructuring loops, often affects multiple loops
  - Examples: loop fusion, loop interchange, loop tiling

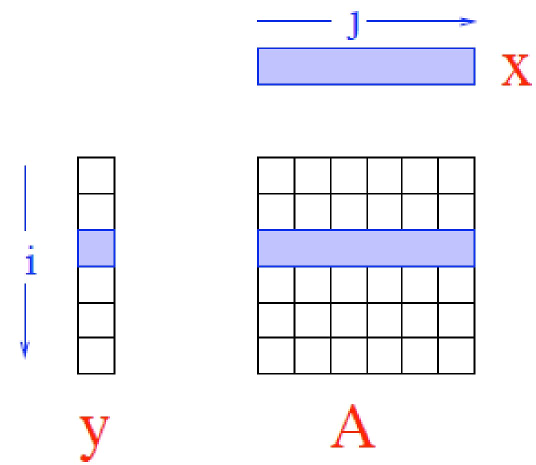
Slide Courtesy: Milind Kulkarni

High level loop optimizations

- Many useful compiler optimizations require *restructuring* loops or sets of loops
  - Combining two loops together (*loop fusion*)
  - Switching the order of a nested loop (*loop interchange*)
  - Completely changing the traversal order of a loop (*loop tiling*)

Cache behavior

- Most loop transformations target cache performance
  - Attempt to increase *spatial* or *temporal* locality
  - Locality can be exploited when there is *reuse* of data (for temporal locality) or recent access of nearby data (for spatial locality)
- Loops are a good opportunity for this: many loops iterate through matrices or arrays
- Consider matrix-vector multiply example
  - Multiple traversals of vector: opportunity for spatial and temporal locality
  - Regular access to array: opportunity for spatial locality



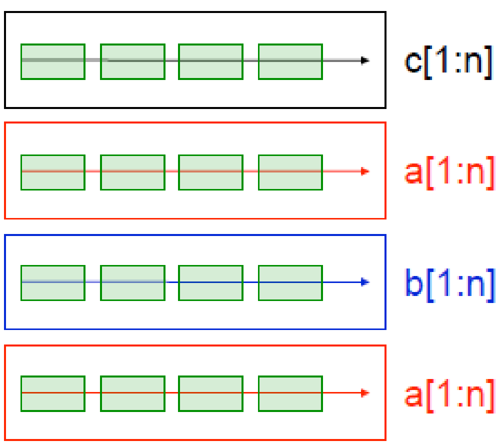
$$y = Ax$$

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


Slide Courtesy: Milind Kulkarni

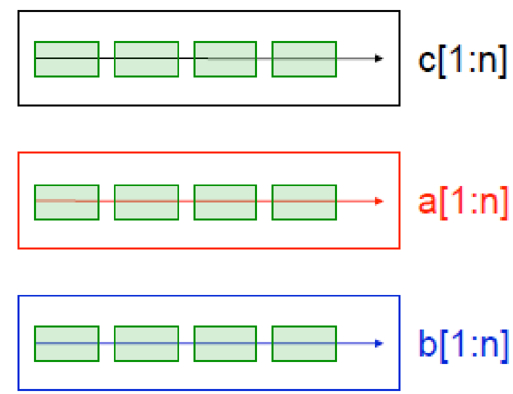
Loop fusion

```
do I = 1, n
  c[i] = a[i]
end do
do I = 1, n
  b[i] = a[i]
end do
```



- Combine two loops together into a single loop
- Why is this useful?
- Is this always legal?

```
do I = 1, n
  c[i] = a[i]
  b[i] = a[i]
end do
```



Slide Courtesy: Milind Kulkarni

Loop Fusion – Another Example

```
for(i=0;i<100;i++)
    b[i]=foo(a[i])

for(i=0;i<100;i++)
    c[i]=bar(b[i])
```

⇨

```
for(i=0;i<100;i++)
    b[i]=foo(a[i])
    c[i]=bar(b[i])
```

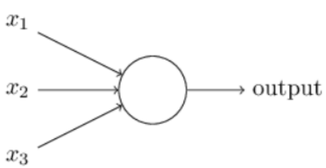
If b is dead outside the loop, then:

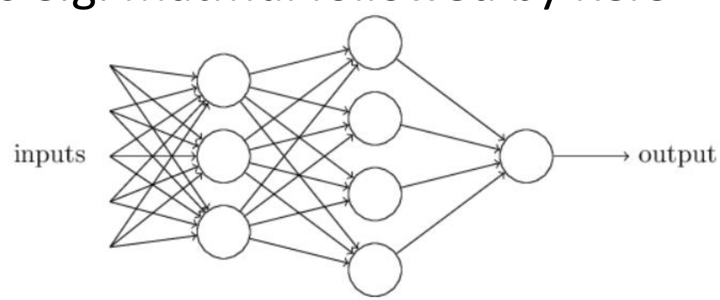
⇨

```
for(i=0;i<100;i++)
    c[i]=bar(foo(a[i]))
```

This is a big saving – eliminate memory store, reduce memory footprint etc.

Common in DNN applications e.g. MatMul followed by RelU

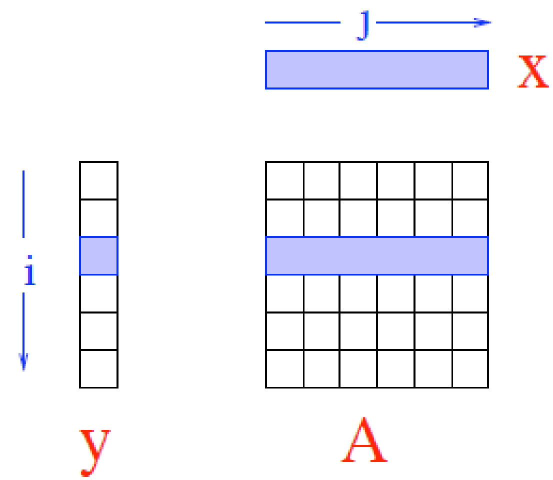




output = weighted sum of inputs, Multiple inputs => Matrix of inputs * Matrix of weights

Loop interchange

- Change the order of a nested loop
- This is not always legal – it changes the order that elements are accessed!
- Why is this useful?
  - Consider matrix-matrix multiply when A is stored in column-major order (i.e., each column is stored in contiguous memory)

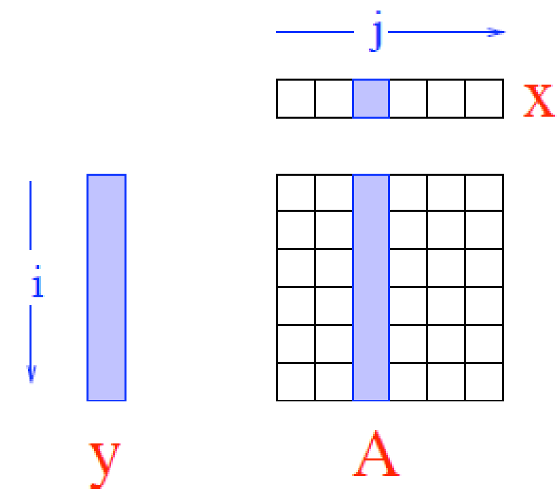



```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    y[i] += A[i][j] * x[j]
```


Slide Courtesy: Milind Kulkarni

Loop interchange

- Change the order of a nested loop
- This is not always legal – it changes the order that elements are accessed!
- Why is this useful?
    - Consider matrix-matrix multiply when A is stored in column-major order (i.e., each column is stored in contiguous memory)




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

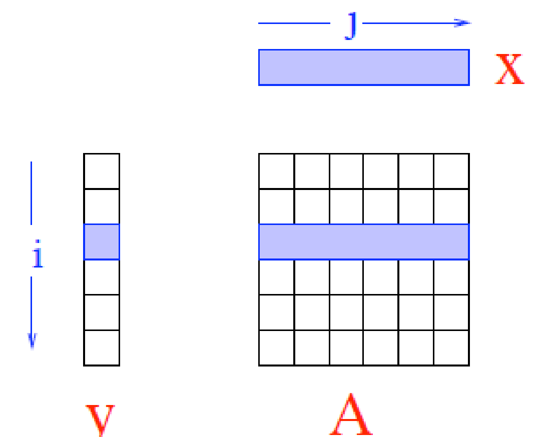

Slide Courtesy: Milind Kulkarni

Loop tiling

- Also called "loop blocking"
- One of the more complex loop transformations
- Goal: break loop up into smaller pieces to get spatial and temporal locality
  - Create new inner loops so that data accessed in inner loops fit in cache
- Also changes iteration order, so may not be legal

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

```
for (ii = 0; ii < N; ii += B)
  for (jj = 0; jj < N; jj += B)
    for (i = ii; i < ii+B; i++)
      for (j = jj; j < jj+B; j++)
        y[i] += A[i][j] * x[j]
```



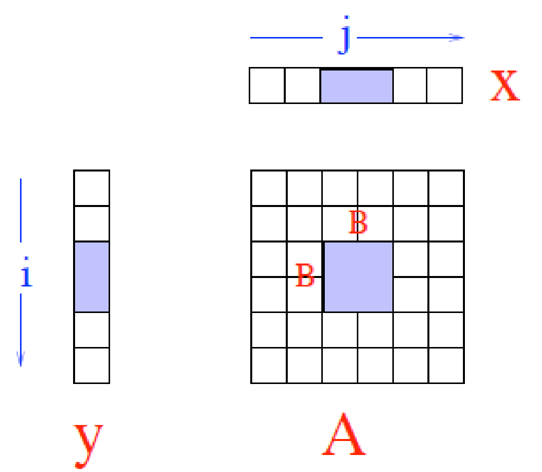
Slide Courtesy: Milind Kulkarni

Loop tiling

- Also called "loop blocking"
- One of the more complex loop transformations
- Goal: break loop up into smaller pieces to get spatial and temporal locality
  - Create new inner loops so that data accessed in inner loops fit in cache
- Also changes iteration order, so may not be legal

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

```
for (ii = 0; ii < N; ii += B)
  for (jj = 0; jj < N; jj += B)
    for (i = ii; i < ii+B; i++)
      for (j = jj; j < jj+B; j++)
        y[i] += A[i][j] * x[j]
```



Slide Courtesy: Milind Kulkarni

Legality of High Level Loop Optimizations

- Dependence Analysis (slide courtesy: Milind Kukarni)

Dependence Analysis

Motivating question

- Can the loops on the right be run in parallel?
  - *i.e.*, can different processors run different iterations in parallel?
- What needs to be true for a loop to be parallelizable?
  - Iterations cannot interfere with each other
  - No *dependence* between iterations

```
for (i = 1; i < N; i++) {
  a[i] = b[i];
  c[i] = a[i - 1];
}
```

```
for (i = 1; i < N; i++) {
  a[i] = b[i];
  c[i] = a[i] + b[i - 1];
}
```

Dependences

- A *flow dependence* occurs when one iteration writes a location that a *later* iteration reads

```
for (i = 1; i < N; i++) {
  a[i] = b[i];
  c[i] = a[i - 1];
}
```

i = 1	i = 2	i = 3	i = 4	i = 5
W(a[1])	W(a[2])	W(a[3])	W(a[4])	W(a[5])
R(b[1])	R(b[2])	R(b[3])	R(b[4])	R(b[5])
W(c[1])	W(c[2])	W(c[3])	W(c[4])	W(c[5])
R(a[0])	R(a[1])	R(a[2])	R(a[3])	R(a[4])

Running a loop in parallel

- If there is a dependence in a loop, we cannot guarantee that the loop will run correctly in parallel
  - What if the iterations run out of order?
    - Might read from a location before the correct value was written to it
  - What if the iterations do not run in lock-step?
    - Same problem!

Other kinds of dependence

- *Anti dependence* – When an iteration *reads* a location that a later iteration *writes* (why is this a problem?)

```
for (i = 1; i < N; i++) {
  a[i - 1] = b[i];
  c[i] = a[i];
}
```

- *Output dependence* – When an iteration *writes* a location that a later iteration *writes* (why is this a problem?)

```
for (i = 1; i < N; i++) {
  a[i] = b[i];
  a[i + 1] = c[i];
}
```

Data dependence concepts

- Dependence *source* is the earlier statement (the statement at the tail of the dependence arrow)
- Dependence *sink* is the later statement (the statement at the head of the dependence arrow)

i = 1	i = 2	i = 3	i = 4	i = 5
W(a[1])	W(a[2])	W(a[3])	W(a[4])	W(a[5])
R(b[1])	R(b[2])	R(b[3])	R(b[4])	R(b[5])
W(c[1])	W(c[2])	W(c[3])	W(c[4])	W(c[5])
R(a[0])	R(a[1])	R(a[2])	R(a[3])	R(a[4])

- Dependences can only go forward in time: always from an earlier iteration to a later iteration.

Using dependences

- If there are no dependences, we can parallelize a loop
  - None of the iterations interfere with each other
- Can also use dependence information to drive other optimizations
  - Loop interchange
  - Loop fusion
  - (We will discuss these later)
- Two questions:
  - How do we represent dependences in loops?
  - How do we determine if there are dependences?

Representing dependences

- Focus on flow dependences for now
- Dependences in straight line code are easy to represent:
  - One statement writes a location (variable, array location, etc.) and another reads that same location
  - Can figure this out using reaching definitions
- What do we do about loops?
  - We often care about dependences between the same statement in different iterations of the loop!

```
for (i = 1; i < N; i++) {
  a[i + 1] = a[i] + 2
}
```

Iteration space graphs

- Represent each *dynamic* instance of a loop as a point in a graph
- Draw arrows from one point to another to represent dependences

```
for (i = 0; i < N; i++) {
  a[i + 2] = a[i]
}
```

Iteration space graphs

- Represent each *dynamic* instance of a loop as a point in a graph
- Draw arrows from one point to another to represent dependences

```
for (i = 0; i < N; i++) {
  a[i + 2] = a[i]
}
```

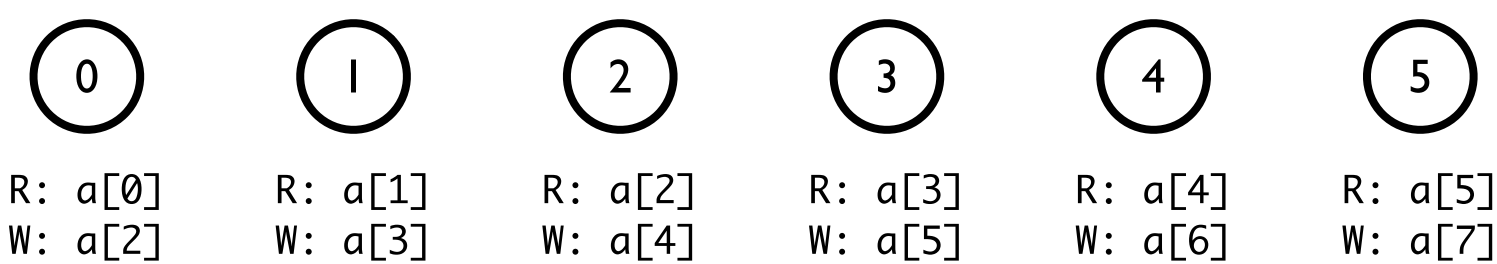
- Step 1: Create nodes, 1 for each iteration
  - Note: not 1 for each array location!

Iteration space graphs

- Represent each *dynamic* instance of a loop as a point in a graph
- Draw arrows from one point to another to represent dependences

```
for (i = 0; i < N; i++) {
  a[i + 2] = a[i]
}
```

- Step 2: Determine which array elements are read and written in each iteration

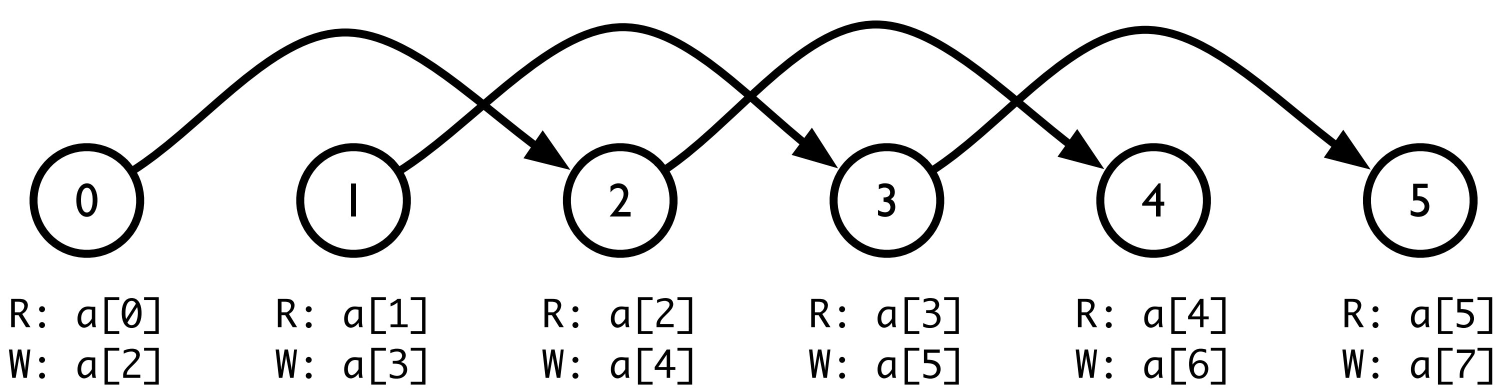


Iteration space graphs

- Represent each *dynamic* instance of a loop as a point in a graph
- Draw arrows from one point to another to represent dependences

```
for (i = 0; i < N; i++) {
  a[i + 2] = a[i]
}
```

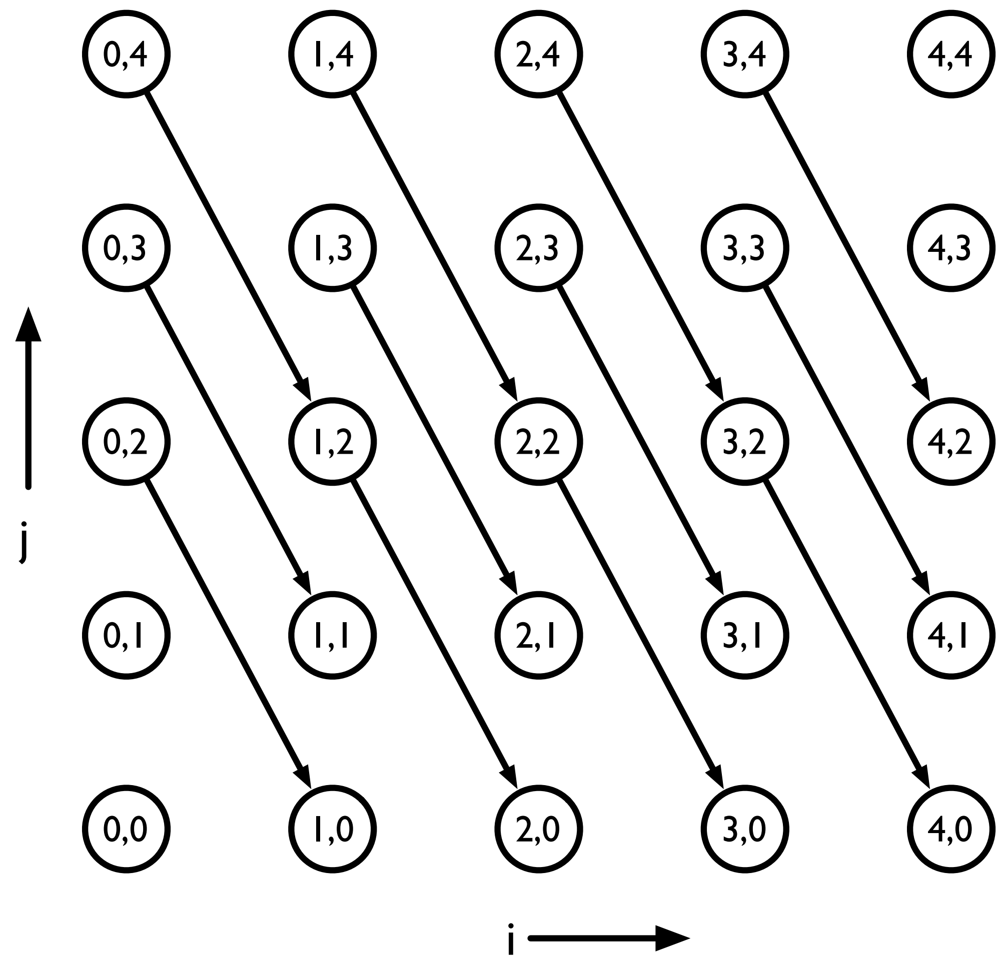
- Step 3: Draw arrows to represent dependences



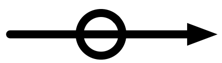
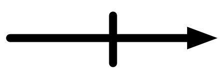
2-D iteration space graphs

- Can do the same thing for doubly-nested loops
  - 2 loop counters

```
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
  a[i+1][j-2] = a[i][j] + 1
```



Iteration space graphs

- Can also represent output and anti dependences
  - Use different kinds of arrows for clarity. *E.g.*
  -  for output
  -  for anti

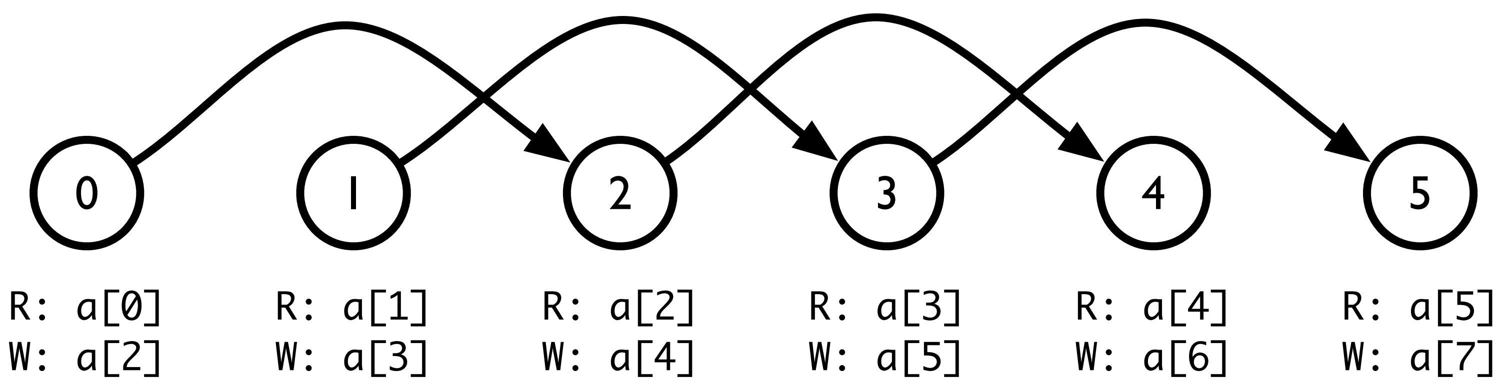
- Crucial problem: Iteration space graphs are potentially infinite representations!
  - Can we represent dependences in a more compact way?

Distance and direction vectors

- Compiler researchers have devised *compressed* representations of dependences
  - Capture the same dependences as an iteration space graph
  - May lose *precision* (show more dependences than the loop actually has)
- Two types
  - Distance vectors: captures the "shape" of dependences, but not the particular source and sink
  - Direction vectors: captures the "direction" of dependences, but not the particular shape

Distance vector

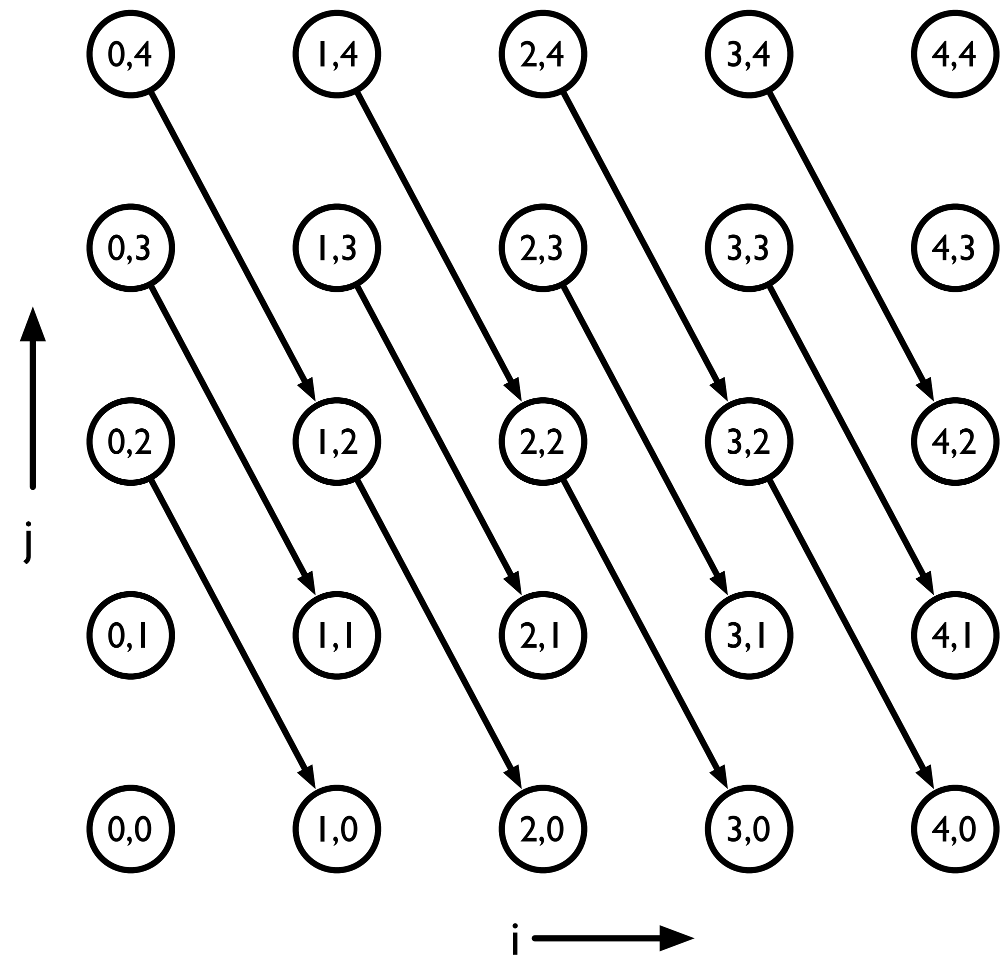
- Represent each dependence arrow in an iteration space graph as a vector
  - Captures the “shape” of the dependence, but loses where the dependence originates



- Distance vector for this iteration space: (2)
  - Each dependence is 2 iterations forward

2-D distance vectors

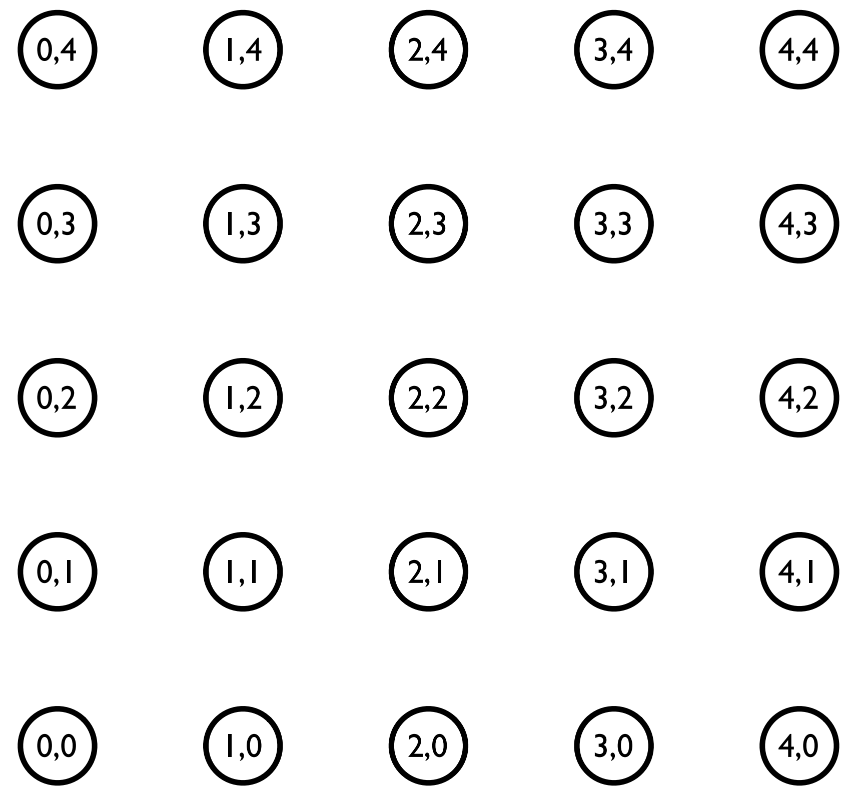
- Distance vector for this graph:
  - (1, -2)
  - +1 in the i direction, -2 in the j direction
- Crucial point about distance vectors: they are always "positive"
  - First non-zero entry has to be positive
  - Dependences can't go backwards in time



More complex example

- Can have multiple distance vectors

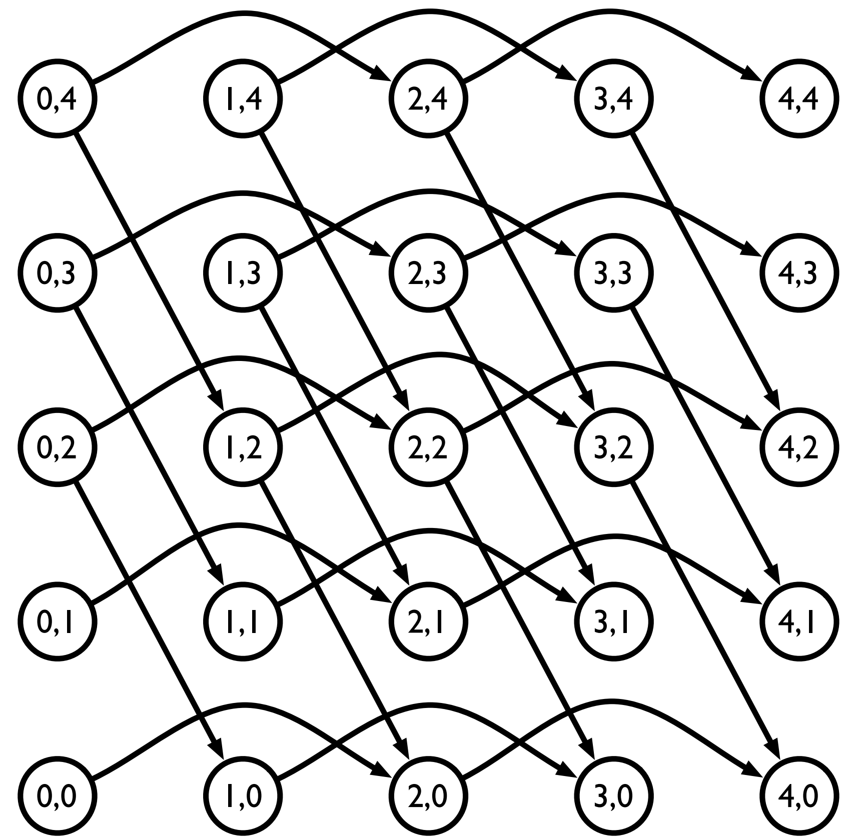
```
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
  a[i+1][j-2] = a[i][j] +
                a[i-1][j-2]
```



More complex example

- Can have multiple distance vectors

```
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
  a[i+1][j-2] = a[i][j] +
                a[i-1][j-2]
```

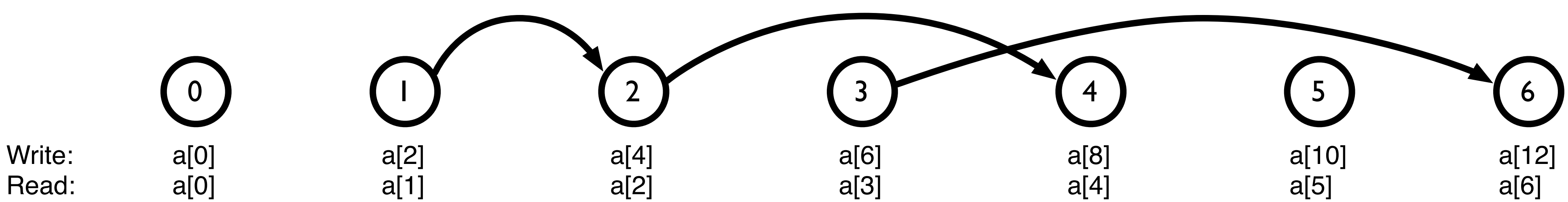


- Distance vectors
  - (1, -2)
  - (2, 0)
- Important point: order of vectors depends on order of loops, not use in arrays

Problems with distance vectors

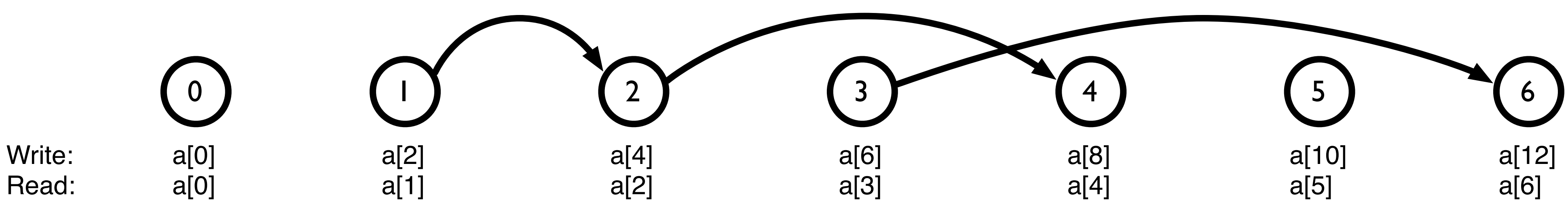
- The preceding examples show how distance vectors can summarize all the dependences in a loop nest using just a small number of distance vectors
- Can't always summarize as easily
- Running example:

```
for (i = 0; i < N; i++)
  a[2*i] = a[i];
```



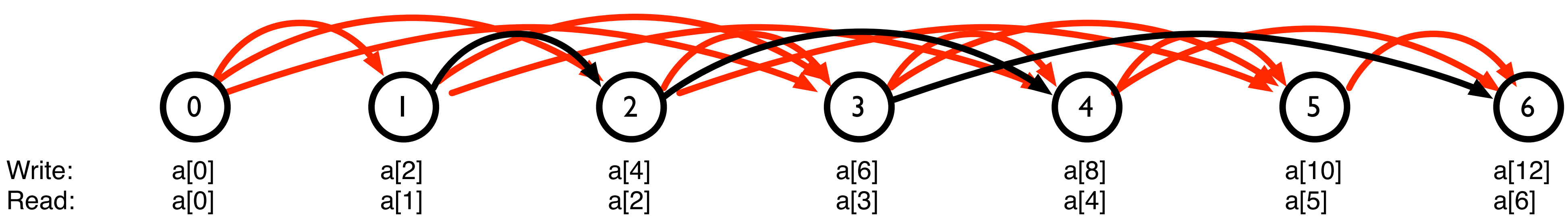
Loss of precision

- What are the distance vectors for this code?
  - (1), (2), (3), (4) ...
- Note: we have information about the length of each vector, but not about the source of each vector
  - What happens if we try to reconstruct the iteration space graph?



Loss of precision

- What are the distance vectors for this code?
  - (1), (2), (3), (4) ...
- Note: we have information about the length of each vector, but not about the source of each vector
  - What happens if we try to reconstruct the iteration space graph?



Direction vectors

- The whole point of distance vectors is that we want to be able to succinctly capture the dependences in a loop nest
    - But in the previous example, not only did we add a lot of extra information, we still had an infinite number of distance vectors
- Idea: summarize distance vectors, and save only the *direction* the dependence was in
    - (2, -1) → (+, –)
    - (0, 1) → (0, +)
    - (0, -2) → (0, –)
        - (can’t happen; dependences have to be positive)
    - Notation: sometimes use ‘<‘ and ‘>’ instead of ‘+’ and ‘–’

Why use direction vectors?

- Direction vectors lose a lot of information, but do capture some useful information
  - Whether there is a dependence (anything other than a ‘0’ means there is a dependence)
  - Which dimension and direction the dependence is in
- Many times, the only information we need to determine if an optimization is legal is captured by direction vectors
  - Loop parallelization
  - Loop interchange

Loop parallelization

Loop-carried dependence

- The key concept for parallelization is the *loop carried dependence*
  - A dependence that crosses loop iterations
- If there is a loop carried dependence, then that loop *cannot* be parallelized
  - Some iterations of the loop depend on other iterations of the same loop

Examples

```
for (i = 0; i < N; i++)
  a[2*i] = a[i];
```

Later iterations of i loop depend on earlier iterations

```
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
  a[i+1][j-2] = a[i][j] + 1
```

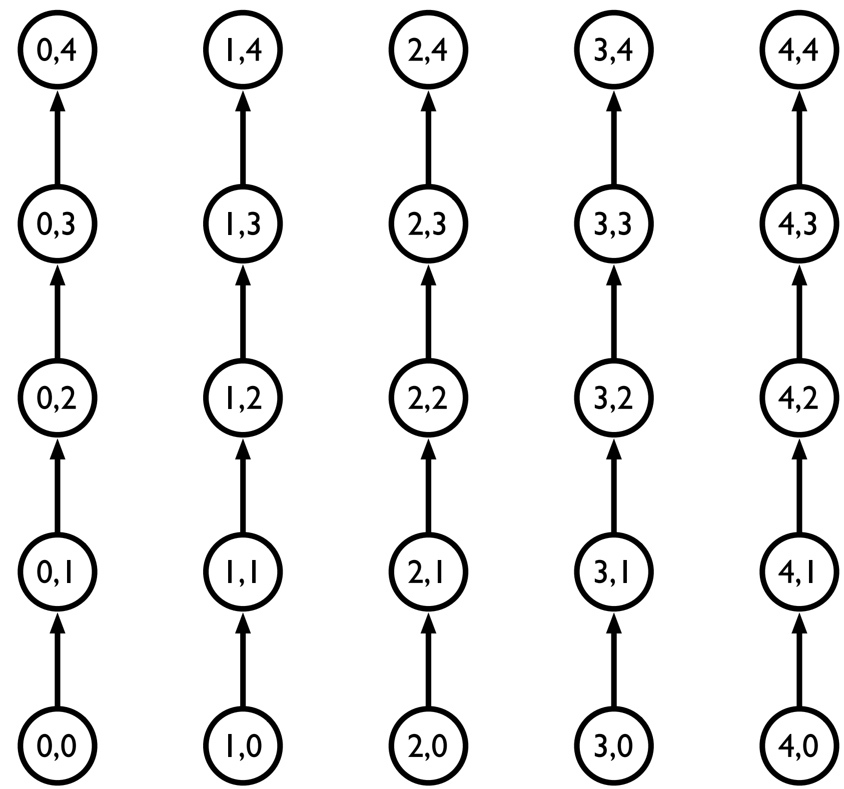
Later iterations of both i and j loops depend on earlier iterations

Some subtleties

- Dependences might only be carried over one loop!

```
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
  a[i][j+1] = a[i][j] + 1
```

- Can parallelize i loop, but not j loop

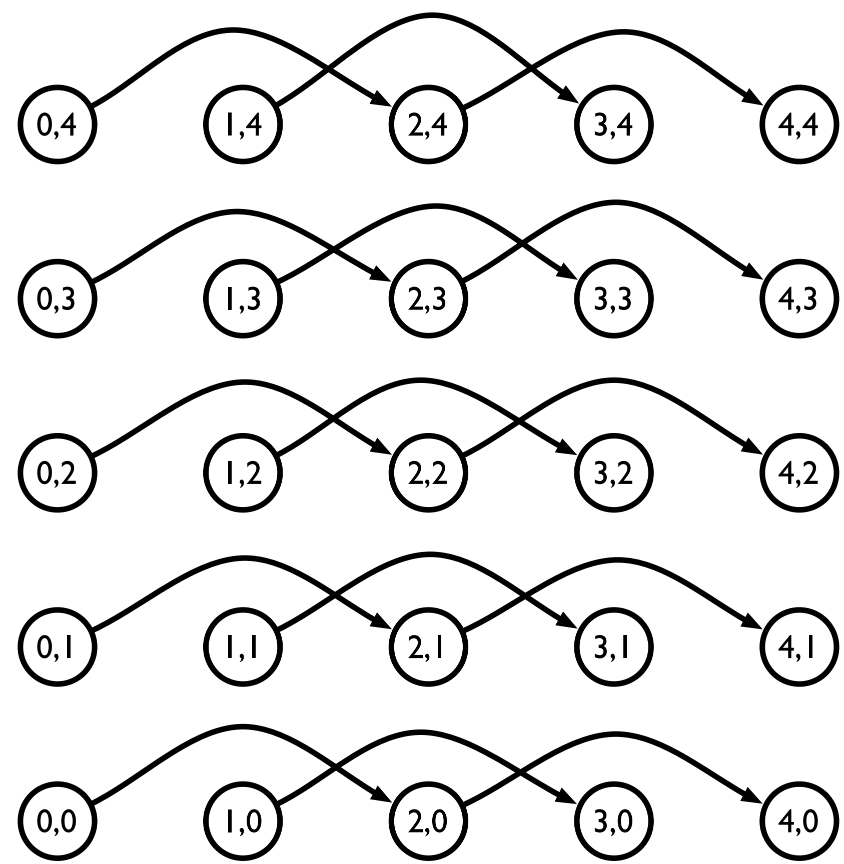


Some subtleties

- Dependences might only be carried over one loop!

```
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
  a[i+1][j] = a[i-1][j] + 1
```

- Can parallelize j loop, but not i loop



Direction vectors

- So how do direction vectors help?
  - If there is a non-zero entry for a loop dimension, that means that there is a loop carried dependence over that dimension
  - If an entry is zero, then that loop can be parallelized!
- May be able to parallelize inner loop even if entry is not zero, but you have to carefully structure parallel execution

Other loop optimizations

Loop interchange

- We've seen this one before
- Interchange doubly-nested loop to
  - Improve locality
  - Improve parallelism
    - Move parallel loop to outer loop (coarse grained parallelism)

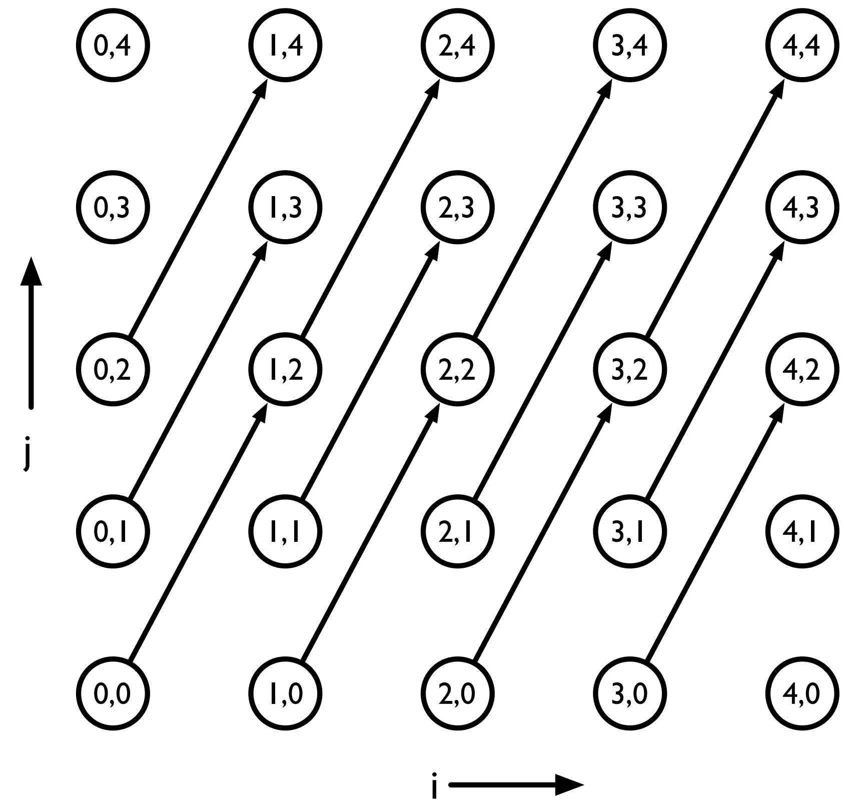
Loop interchange legality

- We noted that loop interchange is not always legal, because it reorders a computation
- Can we use dependences to determine legality?

Loop interchange dependences

- Consider interchanging the following loop, with the dependence graph to the right:

```
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
  a[i+1][j+2] = a[i][j] + 1
```

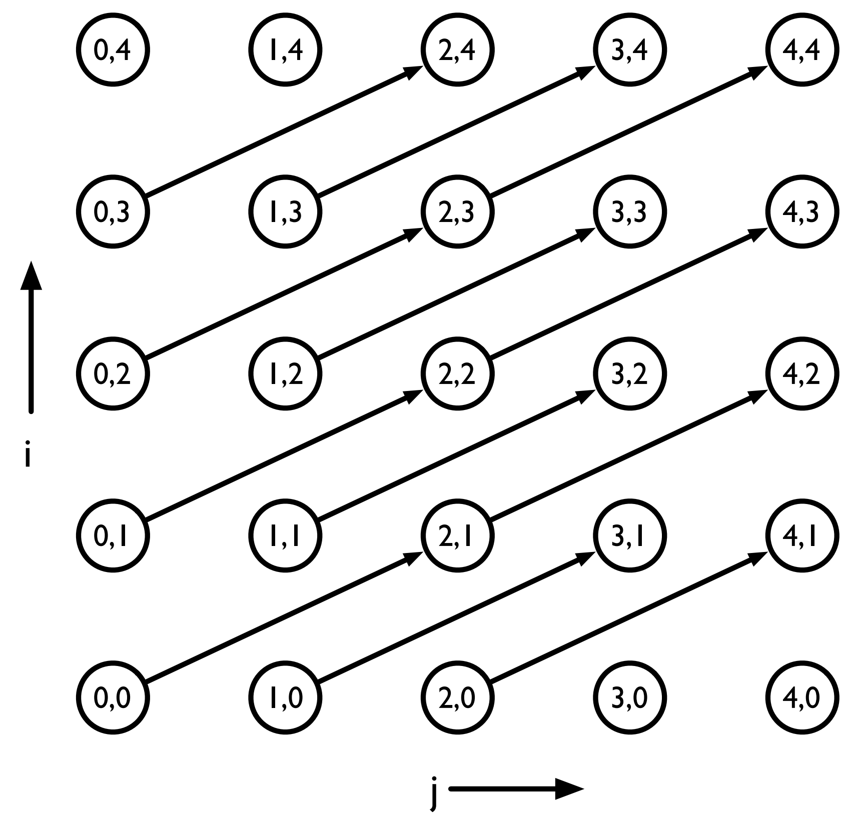


- Distance vector (1, 2)
- Direction vector (+, +)

Loop interchange dependences

- Consider interchanging the following loop, with the dependence graph to the right:

```
for (j = 0; j < N; j++)
 for (i = 0; i < N; i++)
  a[i+1][j+2] = a[i][j] + 1
```



- Distance vector (2, 1)
- Direction vector (+, +)
- Distance vector gets swapped!

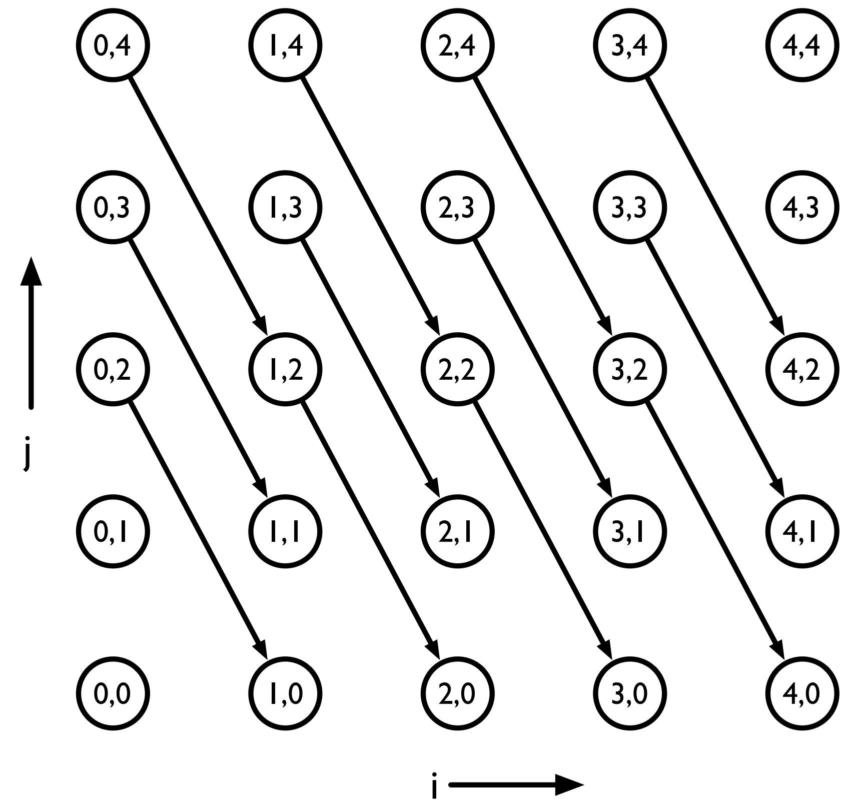
Loop interchange legality

- Interchanging two loops swaps the order of their entries in distance/direction vectors
  - (0, +) → (+, 0)
  - (+, 0) → (0, +)
- But remember, we can't have backwards dependences
  - (+, –) → (–, +)
  - Illegal dependence → Loop interchange not legal!

Loop interchange dependences

- Example of illegal interchange:

```
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
  a[i+1][j-2] = a[i][j] + 1
```

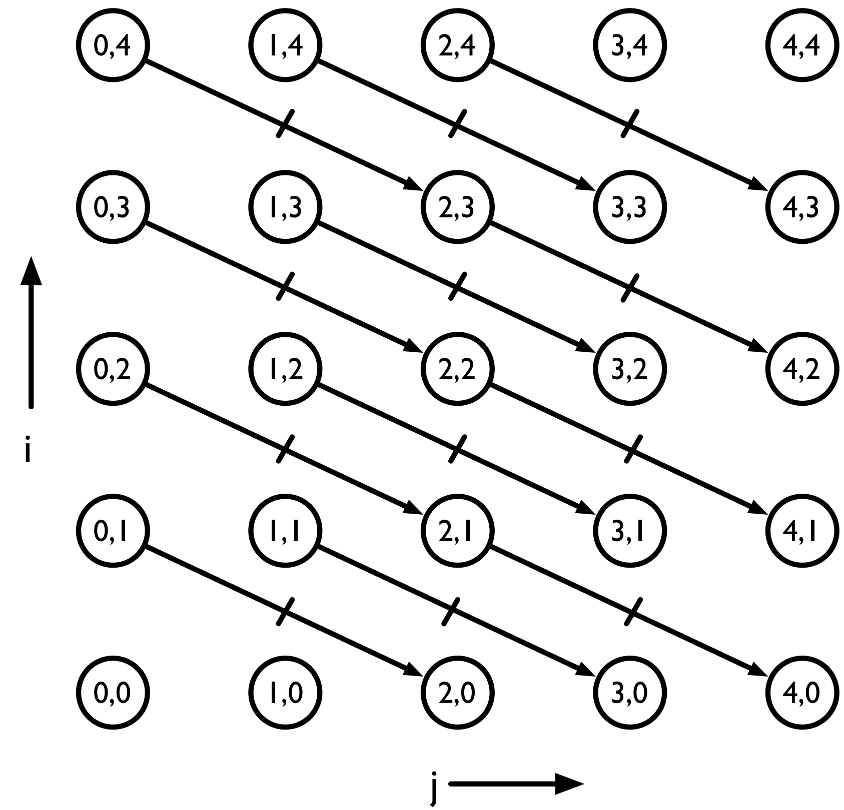


Loop interchange dependences

- Example of illegal interchange:

```
for (j = 0; j < N; j++)
 for (i = 0; i < N; i++)
  a[i+1][j-2] = a[i][j] + 1
```

- Flow dependences turned into anti-dependences
  - Result of computation will change!



Loop fusion/distribution

- Loop fusion: combining two loops into a single loop
  - Improves locality, parallelism
- Loop distribution: splitting a single loop into two loops
  - Can increase parallelism (turn a non-parallelizable loop into a parallelizable loop)
- Legal as long as optimization maintains dependences
  - Every dependence in the original loop should have a dependence in the optimized loop
  - Optimized loop should not introduce new dependences

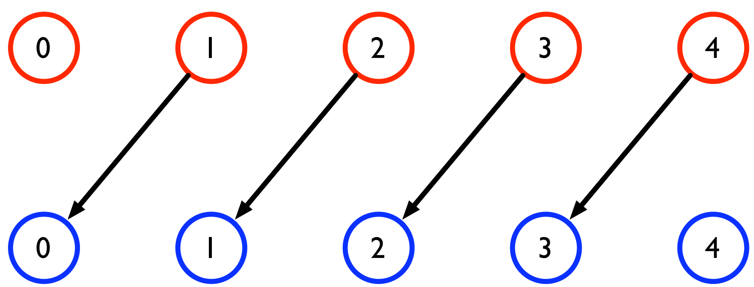
Fusion/distribution example

- Code 1:

```
for (i = 0; i < N; i++)
  a[i - 1] = b[i]

for (j = 0; j < N; j++)
  c[j] = a[j]
```

- Dependence graph

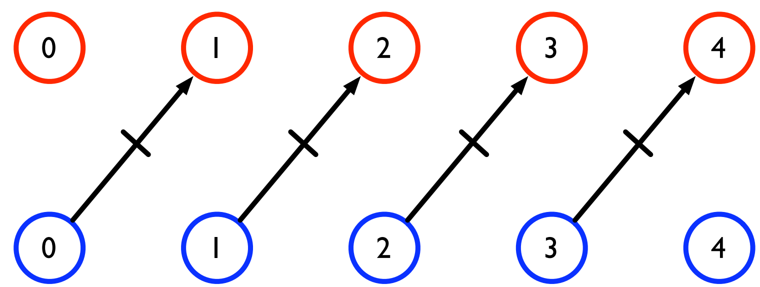


- All red iterations finish before blue iterations → flow dependence

- Code 2:

```
for (i = 0; i < N; i++)
  a[i - 1] = b[i]
  c[i] = a[i]
```

- Dependence graph



- i iterations finish before i+1 iterations → flow dependence now an anti dependence!

Fusion/distribution utility

```
for (i = 0; i < N; i++)
  a[i] = a[i - 1]

for (j = 0; j < N; j++)
  b[j] = a[j]
```

Fusion →

← Distribution

```
for (i = 0; i < N; i++)
  a[i] = a[i - 1]
  b[i] = a[i]
```

- Fusion and distribution both legal
- Right code has better locality, but cannot be parallelized due to loop carried dependences
- Left code has worse locality, but blue loop can be parallelized