Lecture 9
Loop Transformations
Part II

John Cavazos

Dept of Computer & Information Sciences

University of Delaware

www.cis.udel.edu/~cavazos/cisc879

Loop Unswitching

- Hoist **invariant** control-flow out of loop nest
  - Invariant means does not change in loop
- Replicate the loop & specialize it
- No tests (branches) in loop body
- Longer segments of straight-line code

Loop Unswitching

```
loop
    statements
    if test then
        then part
    else
        else part
    endif
    more statements
endloop
```

becomes
(unswitch)

```
If test then
    loop
        statements
        then part
        more statements
    endloop
else
    loop
        statements
        else part
        more statements
    endloop
endif
```

Loop Unswitching

```
loop
    statements
    if test then
        then part
    else
        else part
    endif
    more statements
endloop
```

becomes

```
If test then
    loop
      statements
      then part
      more statements
    endloop
else
    loop
      statements
      else part
      more statements
    endloop
endif
```

Loop Unswitching

```
loop
   statements
   if test then
      then part
   else
      else part
   endif
   more statements
endloop
```

becomes

```
If test then
   loop
     statements
     then part
     more statements
    endloop
else
    loop
      statements
       else part
      more statements
   endloop
endif
```

Loop Unswitching

```
loop
    statements
    if test then
        then part
    else
        else part
    endif
    more statements
endloop
```

becomes

```
If test then
   loop
      statements
      then part
      more statements
   endloop
else
   loop
      statements
      else part
      more statements
   endloop
endif
```

Loop Unswitching

becomes

(unswitch)

```
do i = 1 to 100
   a(i) = a(i) + b(i)
   if (expression) then
      d(i) = 0
end
```

→

```
if (expression) then
   do i = 1 to 100
      a(i) = a(i) + b(i)
      d(i) = 0
   end
else
   do i = 1 to 100
      a(i) = a(i) + b(i)
   end
```

Loop Fusion

- Two loops over same iteration space ⇒ one loop
- Safe if does not change values used or defined by any statement in either loop (i.e., does not violate deps)

```
do i = 1 to n
    c(i) = a(i) + b(i)
end

do j = 1 to n
    d(j) = a(j) * e(j)
end
```

becomes (fuse) →

```
do i = 1 to n
    c(i) = a(i) + b(i)
    d(i) = a(i) * e(i)
end
```

For big arrays, a(i) may not be in the cache

a(i) will be found in the cache

Loop Fusion Advantages

- Enhance temporal locality
- Reduce control overhead
- Longer blocks for local optimization & scheduling
- Can convert inter-loop reuse to intra-loop reuse

Loop Fusion of Parallel Loops

- Parallel loop fusion legal if dependences loop independent
  - Source and target of flow dependence map to same loop iteration

Loop distribution (fission)

- Single loop with independent statements ⇒ multiple loops
- Starts by constructing statement level dependence graph
- Safe to perform distribution if:
  - No cycles in the dependence graph
  - Statements forming cycle in dependence graph put in same loop

Loop distribution (fission)

Reads b, c, e, f, h, & k
Writes a, d, & g

```
do i = 1 to n
    a(i) = b(i) + c(i)
    d(i) = e(i) * f(i)
    g(i) = h(i) - k(i)
end
```

***becomes* (fission)**

```
do i = 1 to n
    a(i) = b(i) + c(i)
end
```

Reads b & c
Writes a

```
do i = 1 to n
    d(i) = e(i) * f(i)
end
```

Reads e & f
Writes d

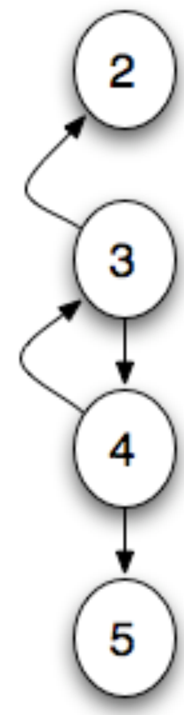
```
do i = 1 to n
    g(i) = h(i) - k(i)
end
```

Reads h & k
Writes g

Loop distribution (fission)

```
(1) for I = 1 to N do
(2)    A[I] = A[i] + B[i-1]
(3)    B[I] = C[I-1]*X+C
(4)    C[I] = 1/B[I]
(5)    D[I] = sqrt(C[I])
(6) endfor
```

Has the following dependence graph



Loop distribution (fission)

```
(1) for I = 1 to N do
(2)    A[I] = A[i] + B[i-1]
(3)    B[I] = C[I-1]*X+C
(4)    C[I] = 1/B[I]
(5)    D[I] = sqrt(C[I])
(6) endfor
```

becomes (fission)

```
(2)   A[I] = A[i] + B[i-1]
(3) endfor
(4) for
(5)   B[I] = C[I-1]*X+C
(6)   C[I] = 1/B[I]
(7)endfor
(8)for
(9)   D[I] = sqrt(C[I])
(10)endfor
```

Loop Fission Advantages

- Enables other transformations
  - E.g., Vectorization
- Resulting loops have smaller cache footprints
  - More reuse hits in the cache

Loop Interchange

```
do i = 1 to 50
   do j = 1 to 100
      a(i,j) = b(i,j) * c(i,j)
    end
 end
```

becomes

(interchange)

```
do j = 1 to 100
   do i = 1 to 50
      a(i,j) = b(i,j) * c(i,j)
   end
 end
```

- Swap inner & outer loops to rearrange iteration space

Effect

- Improves reuse by using more elements per cache line
- Goal is to get as much reuse into inner loop as possible

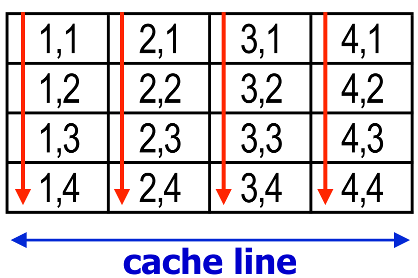
Loop Interchange Effect

- If one loop carries all dependence relations
  - Swap to outermost loop and all inner loops executed in parallel
- If outer loops iterates many times and inner only a few
  - Swap outer and inner loops to reduce startup overhead
- Improves reuse by using more elements per cache line
- Goal is to get as much reuse into inner loop as possible

Reordering Loops for Locality

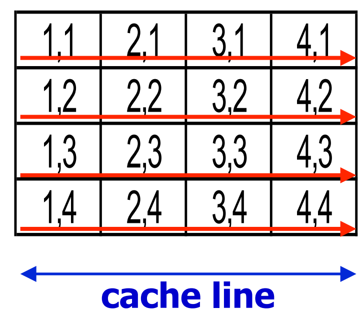
In row-major order, the opposite loop ordering causes the same effects

In Fortran's column-major order, a(4,4) would lay out as



As little as 1 used element per line

After interchange, direction of Iteration is changed



Runs down cache line

Loop permutation

- Interchange is degenerate case
  - Two perfectly nested loops
- More general problem is called permutation

Safety

- Permutation is safe iff no data dependences are reversed
  - The flow of data from definitions to uses is preserved

Loop Permutation Effects

- Change order of access & order of computation
- Move accesses closer in time ⇒ increase temporal locality
- Move computations farther apart ⇒ cover pipeline latencies

Strip Mining

- Splits a loop into two loops

```
do j = 1 to 100
  do i = 1 to 50
    a(i,j) = b(i,j) * c(i,j)
  endend
```

becomes

(strip mine)

```
do j = 1 to 100
  do ii = 1 to 50 by 8
    do i = ii to min(ii+7,50)
      a(i,j) = b(i,j) * c(i,j)
    end
  end
end
```

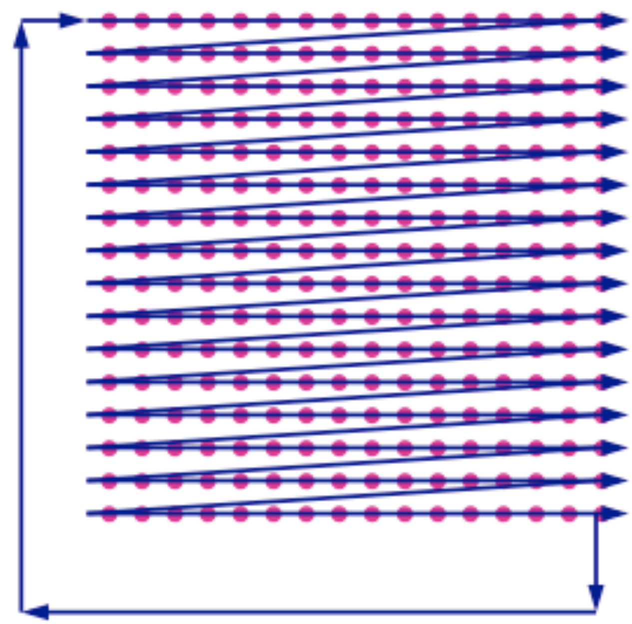
Note: This is always safe, but used by itself not profitable!

Strip Mining Effects

- May slow down the code (extra loop)
- Enables vectorization

Loop Tiling (blocking)

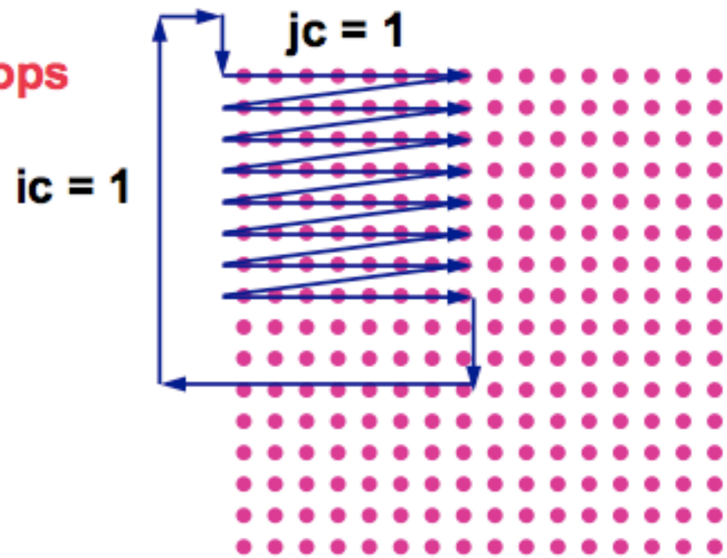
```
do t = 1,T
  do i = 1,n
    do j = 1,n
       … a(i,j) …
    end do
  end do
end do
```



Loop Tiling (blocking)

```
do ic = 1, n, B
  do jc = 1, n, B
    do t = 1,T
      do i = ic, min(n,ic+B-1), 1
        do j = jc, min(n, jc+B-1), 1
          … a(i,j) …
        end do
      end do
    end do
  end do
end do
```

control loops

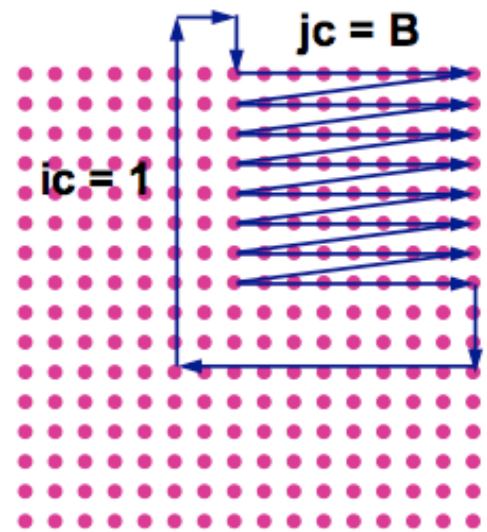


B: Block Size

Loop Tiling (blocking)

```
do ic = 1, n, B
  do jc = 1, n, B
    do t = 1,T
      do i = ic, min(n,ic+B-1), 1
        do j = jc, min(n, jc+B-1), 1
          … a(i,j) …
        end do
      end do
    end do
  end do
end do
```

control loops

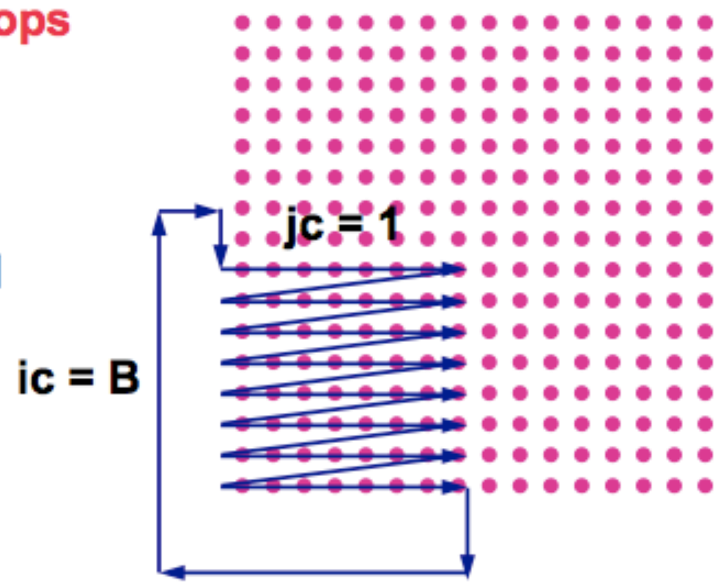


B: Block Size

Loop Tiling (blocking)

```
do ic = 1, n, B
  do jc = 1, n, B
    do t = 1,T
      do i = ic, min(n,ic+B-1), 1
        do j = jc, min(n, jc+B-1), 1
          … a(i,j) …
        end do
      end do
    end do
  end do
end do
```

control loops

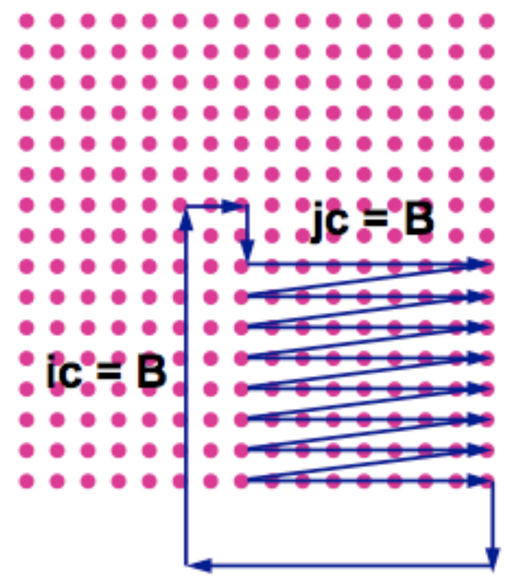


B: Block Size

Loop Tiling (blocking)

```
do ic = 1, n, B
  do jc = 1, n, B
    do t = 1,T
      do i = ic, min(n,ic+B-1), 1
        do j = jc, min(n, jc+B-1), 1
          … a(i,j) …
        end do
      end do
    end do
  end do
end do
```

control loops



B: Block Size

When is this legal?

Loop Tiling Effects

- Reduces volume of data between reuses
  - Works on one "tile" at a time (*tile size is* B *by* B)
- Choice of tile size is crucial

Scalar Replacement

- Allocators never keep c(i) in a register
- We can trick the allocator by rewriting the references

The plan

- Locate patterns of consistent reuse
- Make loads and stores use temporary scalar variable
- Replace references with temporary's name

Scalar Replacement

```
do i = 1 to n
   do j = 1 to n
     a(i) = a(i) + b(j)
   end
end
```

becomes

(scalar replacement)

```
do i = 1 to n
   t = a(i)
   do j = 1 to n
      t =  t + b(j)
   end
   a(i) = t
end
```

Almost any register allocator can get t into a register

Scalar Replacement Effects

- Decreases number of loads and stores
- Keeps reused values in names that can be allocated to registers
- In essence, this exposes the reuse of a(i) to subsequent passes