Introduction to Optimization

John Cavazos

University of Delaware

Lecture Overview

- Motivation
- Loop Transformations

Why study compiler optimizations?

Moore's Law

- Chip density doubles every 18 months
- Reflected in CPU performance doubling every 18 months

Proebsting's Law

- Compilers double CPU performance every 18 years
- 4% improvement per year because of optimizations!

Why study compiler optimizations?

Corollary

- 1 year of code optimization research = 1 month of hardware improvements
- No need for compiler research… Just wait a few months!

Free Lunch is over

Moore's Law

- Chip density doubles every 18 months
- **PAST :** ~~Reflected CPU performance doubling every 18 months~~
- **CURRENT: Density doubling reflected in more cores on chip!**

Corollary

- Cores will become simpler
- Just wait a few months… Your code might get slower!
- Many optimizations now being done by hand! (*autotuning*)

Optimizations: The Big Picture

What are our goals?

- Simple Goal: Make execution time as small as possible

Which leads to:

- Achieve execution of many (all, in the best case) instructions in parallel
- Find **independent** instructions

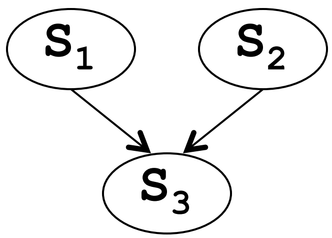
Dependences

- We will concentrate on data dependences
- Simple example of data dependence:

S_1 `PI = 3.14`

S_2 `R = 5.0`

S_3 `AREA = PI * R ** 2`



- Statement S_3 cannot be moved before either S_1 or S_2 without compromising correct results

Dependences

- Formally:

  Data dependence from S_1 to S_2 (S_2 depends on S_1) if:

  1. Both statements access same memory location and one of them stores onto it, and
  2. There is a feasible execution path from S_1 to S_2

Load Store Classification

- Dependences classified in terms of load-store order:
  1. True dependence (RAW hazard)
  2. Antidependence (WAR hazard)
  3. Output dependence (WAW hazard)

Dependence in Loops

- Let us look at two different loops:

```
   DO I = 1, N
S1   A(I+1) = A(I)+ B(I)
   ENDDO
```

```
   DO I = 1, N
S1   A(I+2) = A(I)+B(I)
   ENDDO
```

- In both cases, statement S_1 depends on itself

Transformations

- We call a transformation safe if the transformed program has the same "meaning" as the original program
- But, what is the "meaning" of a program?

For our purposes:

- Two programs are equivalent if, on the same inputs:
  - They produce the same outputs in the same order

Loop Transformations

- Compilers have always focused on loops
  - Higher execution counts
  - Repeated, related operations
- Much of real work takes place in loops

Several effects to attack

- Overhead
  - Decrease control-structure cost per iteration
- Locality
  - Spatial locality ⇒ use of co-resident data
  - Temporal locality ⇒ reuse of same data
- Parallelism
  - Execute independent iterations of loop in parallel

Eliminating Overhead

Loop unrolling (the oldest trick in the book)

- To reduce overhead, replicate the loop body

```
do i = 1 to 100 by 1
   a(i) = a(i) + b(i)
end
```

becomes

(unroll by 4)

```
do i = 1 to 100 by 4
   a(i)   = a(i) + b(i)
   a(i+1) = a(i+1) + b(i+1)
   a(i+2) = a(i+2) + b(i+2)
   a(i+3) = a(i+3) + b(i+3)
end
```

Sources of Improvement

- Less overhead per useful operation
- Longer basic blocks for local optimization

Loop Fusion

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

```
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
- Each iteration can execute in parallel

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)

(1) for I = 1 to N do

(2) A[I] = A[i] + B[i-1]

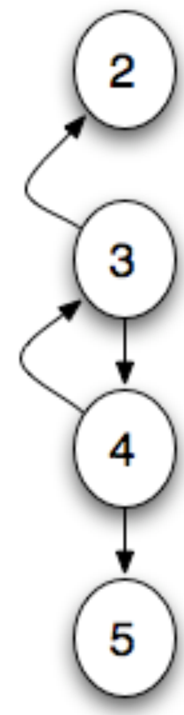
(3) B[I] = C[I-1]*X+C

(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)

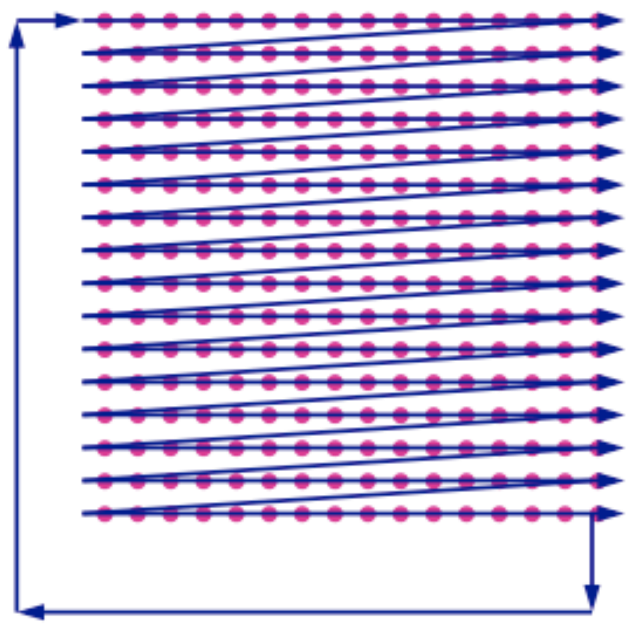
```
(1) for I = 1 to N do
(2)   A[I] = A[i] + B[i-1]
(3) endfor
(4) for
(5)     B[I] = C[I-1]*X+C
(6)     C[I] = 1/B[I]
(7) endfor
(8)  for
(9)      D[I] = sqrt(C[I])
(10) endfor
```

Loop Fission Advantages

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

Loop Tiling (blocking)

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

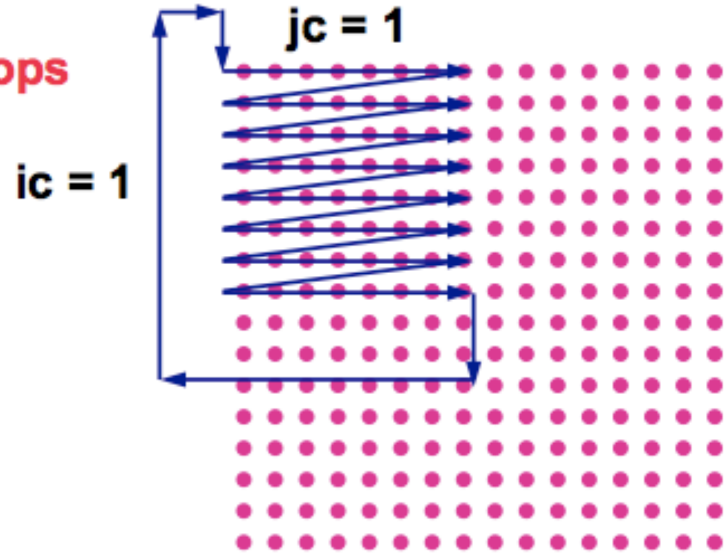


Want to exploit temporal locality
in loop nest.

Loop Tiling (blocking)

control loops

```
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
```

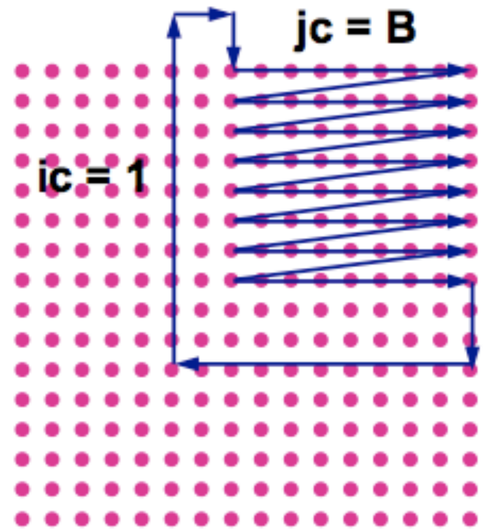


B: Block Size

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

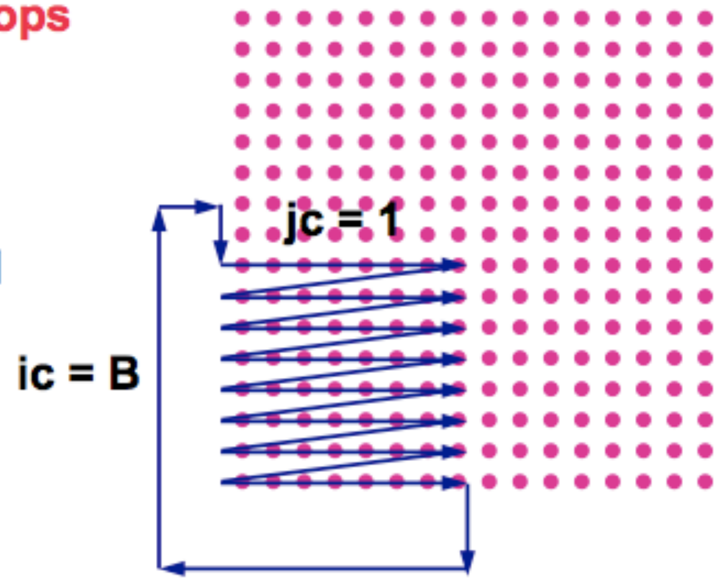


B: Block Size

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

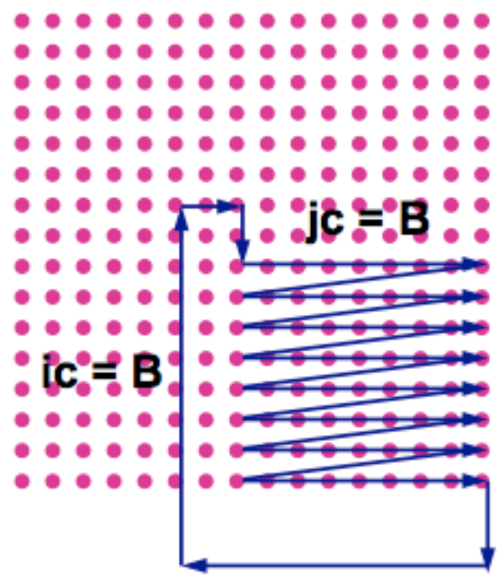


B: Block Size

Loop Tiling (blocking)

control loops

```
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
```



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