ECE 1754

Loop Transformations

by: Eric LaForest

Motivation

- Improving loop behaviour/performance
  - usually parallelism
  - sometimes memory, register usage

Motivation

- Improving loop behaviour/performance
  - usually parallelism
  - sometimes memory, register usage
- Why focus on loops?
  - they contain most of the busy code

Motivation

- Improving loop behaviour/performance
  - usually parallelism
  - sometimes memory, register usage
- Why focus on loops?
  - they contain most of the busy code
  - informal proof:
    - else, program size would be proportional to data size

Data-Flow-Based Transformations

Strength Reduction (Loop-Based)

- Replaces an expression in a loop with an equivalent one that uses a less expensive operator
  - Before

```
do i = 1, n
   a[i] = a[i] + c*i
end do
```

  - After

```
T = c
do i = 1, n
   a[i] = a[i] + T
   T = T + c
end do
```

- Similar operations for exponentiation, sign reversal, and division, where economical.

Induction Variable Elimination

- Frees the register used by the variable, reduces the number of operations in the loop framework.

- Before

```
for(i = 0; i < n; i++){
   a[i] = a[i] + c;
}
```

- After

```
A = &a;
T = &a + n;
while(A < T){
   *A = *A + c;
   A++;
}
```

Loop-Invariant Code Motion

- A specific case of code hoisting
- Needs a register to hold the invariant value
  - Ex: multi-dim. indices, pointers, structures

- Before

```
do i = 1, n
   a[i] = a[i] + sqrt(x)
end do
```

- After

```
if (n > 0) C = sqrt(x)
do i = 1, n
   a[i] = a[i] + C
end do
```

Loop Unswitching

- What:
  - Moving loop-invariant conditionals outside of a loop.
- How:
  - replicate loop inside each branch
- Benefits:
  - no conditional testing each iteration
  - smaller loops
  - expose parallelism

Loop Unswitching

- Before

```
do i = 2, n
    a[i] = a[i] + c
    if (x < 7) then
        b[i] = a[i] * c[i]
    else
        b[i] = a[i-1] * b[i-1]
    end if
end do
```

- After

```
if (n > 2) then
    if (x < 7) then
        do all i = 2, n
            a[i] = a[i] + c
            b[i] = a[i] * c[i]
        end do
    else
        do i = 2, n
            a[i] = a[i] + c
            b[i] = a[i-1] * b[i-1]
        end do
    end if
end if
```

Loop Reordering Transformations

(changing the relative iteration order of nested loops)

Loop Interchange

- Exchange loops in a perfect nest.
- Benefits:
  - enable vectorization
  - improve parallelism by increasing granularity
  - reduce stride (and thus improve cache behaviour)
  - move loop-invariant expressions to inner loop
- Legal when:
  - new dependencies and loop bounds are legal

Loop Interchange

- Before

```
do i = 1, n
   do j = 1, n
      b[i] = b[i] + a[i,j]
   end do
end do
```

- After

```
do j = 1, n
   do i = 1, n
      b[i] = b[i] + a[i,j]
   end do
end do
```

Loop Interchange

```
do i = 2, n
  do j = 1, n-1
    a[i,j] = a[i-1,j+1]
  end do
end do
```

- Cannot be interchanged due to (1,-1) dependence.
  - would end up using a prior uncomputed value

Loop Skewing

- Used in wavefront computations
- How:
  - By adding the outer loop index multiplied by a skew factor, f, to the bounds of the inner iteration variable, and then subtracting the same quantity from every use of the inner variable.
- Always legal because of subtraction.
- Benefit:
  - allows inner loop (once exchanged) to execute in parallel.

Loop Skewing

- Before

```
do i = 2, n-1
   do j = 2, m-1
      a[i,j] = (
      a[a-1,j] +
      a[i,j-1] +
      a[i+1,j] +
      a[i,j+1]
      )/4
   end do
end do
```

{(1,0),(0,1)}

- After Skewing (f = 1)

```
do i = 2, n-1
   do j = i+2, i+m-1
      a[i,j-i] = (
      a[a-1,j-i] +
      a[i,j-1-i] +
      a[i+1,j-i] +
      a[i,j+1-i]
      )/4
   end do
end do
```

{(1,1),(0,1)}

Loop Skewing

- After Skewing (f = 1)

```
do i = 2, n-1
   do j = i+2, i+m-1
      a[i,j-i] = (
      a[a-1,j-i] +
      a[i,j-1-i] +
      a[i+1,j-i] +
      a[i,j+1-i]
      )/4
   end do
end do
```

{(1,1),(0,1)}

- After Interchange

```
do j = 4, m+n-2
   do i = max(2, j-m+1),
     min(n-1, j-2)
      a[i,j-i] = (
      a[a-1,j-i] +
      a[i,j-1-i] +
      a[i+1,j-i] +
      a[i,j+1-i]
      )/4
   end do
end do
```

{(1,0),(1,1)}

Loop Reversal

- Changes the iteration direction
- By making the iteration variable run down to zero, the loop condition reduces to a BNEZ.
- May eliminate temporary arrays (see later)
- Legal when resulting dependence vector remains lexicographically positive
  - This also helps loop interchange.

Loop Reversal

- Before

```
do i = 1, n
   do j = 1, n
      a[i,j] =
      a[i-1,j+1] + 1
   end do
end do
```

(1,-1)

- After

```
do i = 1, n
   do j = 1, n, -1
      a[i,j] =
      a[i-1,j+1] + 1
   end do
end do
```

(1,1)

Strip Mining

- Adjusts the granularity of an operation
  - usually for vectorization
  - also controlling array size, grouping operations
- Often requires other transforms first

Strip Mining

- Before

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

- After

```
TN = (n/64)*64
do TI = 1, TN, 64
   a[TI:TI+63] =
   a[TI:TI+63] + c
end do
do i= TN+1, n
   a[i] = a[i] + c
end do
```

Cycle Shrinking

- Specialization of strip mining:
  - parallelize when dependence distance > 1
- Legal when:
  - distance must be constant and positive

Cycle Shrinking

- Before

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

- After

```
do TI = 1, n, k
   do all i = TI,
     TI+k-1
      a[i+k] = b[i]
      b[i+k] =
      a[i] + c[i]
   end do all
end do
```

Loop Tiling

- Multidimensional specialization of strip mining
- Goal: to improve cache reuse
- Adjacent loops can be tiled if they can be interchanged.

Loop Tiling

- Before

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

- After

```
do TI = 1, n, 64
   do TJ = 1, n, 64
      do i = TI, min(TI+63, n)
         do j = TJ, min(TJ+63, n)
            a[i,j] = b[j,i]
         end do
      end do
   end do
end do
```

Loop Fission

- a.k.a. Loop Distribution
- Divide loop statements into separate similar loops
- Benefits:
  - create perfect loops nests
  - reduce dependences, memory use
  - improve locality, register reuse
- Legal when sub-loops are placed in same dependency order as original statements.

Loop Fission

- Before

```
do i = 1, n
   a[i] = a[i] + c
   x[i+1] = x[i]*7 +
     x[i+1] + a[i]
end do
```

- After

```
do all i = 1, n
   a[i] = a[i] + c
end do all
do i = 1, n
   x[i+1] = x[i]*7 +
     x[i+1] + a[i]
end do
```

Loop Fusion

- a.k.a. loop jamming
- Legal when bounds are identical and when not inducing dependencies (S2 < S1).
- Benefits:
  - reduced loop overhead
  - improved parallelism, locality
  - fix load balance

Restructuring Transformations

(Alters the structure, but not the computations or iteration order)

Loop Unrolling

- Replicates the loop body
- Benefits:
  - reduces loop overhead
  - increased ILP (esp. VLIW)
  - improved locality (consecutive elements)
- Always legal.

Loop Unrolling

- Before

```
do i = 2, n-1
   a[i] = a[i] +
    a[i-1] * a[i+1]
end do
```

- After

```
do i = 1, n-2, 2
   a[i] = a[i] + a[i-1] *
     a[i+1]
   a[i+1] = a[i+1] + a[i]
     * a[i+2]
end do
if (mod(n-2,2) = 1) then
   a[n-1] = a[n-1] +
    a[n-2] * a[n]
end if
```

Software Pipelining

- Before

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

- After (approx.)

```
do i = 1, n, 3
   a[i] = a[i] + c
   a[i+1] = a[i+1] + c
   a[i+2] = a[i+2] + c
end do
```

note: assume a 2-way superscalar CPU

Loop Coalescing

- Combines a loop nest into a single loop
  - results in a single induction variable
- Always legal: doesn't change iteration order
- Improves load balancing on parallel machines

Loop Coalescing

- Before

```
do all i = 1, n
   do all j = 1, m
      a[i,j] = a[i,j] + c
   end do all
end do all
```

- After

```
do all T = 1, n*m
   i = ((T-1) / m) * m + 1
   j = mod(T-1, m) + 1
   a[i,j] = a[i,j] + c
end do all
```

Note: assume n, m slightly larger than P

Loop Collapsing

- Reduce the number of loop dimensions
- Eliminates overhead of nested or multidimensional loops
- Best when stride is constant

Loop Collapsing

- Before

```
do all i = 1, n
   do all j = 1, m
      a[i,j] = a[i,j] + c
   end do all
end do all
```

- After

```
real TA[n*m]
equivalence(TA,a)
do all T = 1, n*m
   TA[T] = TA[T] + c
end do all
```

Loop Peeling

- Extract a number of iterations at start or end
- Reduces dependencies, allows adjusting bounds for later loop fusion
- Always legal

Loop Peeling

- Before

```
do i = 2, n
  b[i] = b[i] + b[2]
end do
do all i = 3, n
  a[i] = a[i] + c
end do all
```

- After

```
if (2 <= n) then
  b[2] = b[2] + b[2]
end if
do all i = 3, n
  b[i] = b[i] + b[2]
  a[i] = a[i] + c
end do all
```

Loop Normalization

- Converts induction variables to be of the form:
  - $i = 1, n, 1$
- Makes analysis easier

Loop Normalization

- Before

```
do i = 1, n
    a[i] = a[i] + c
end do
do i = 2, n+1
    b[i] = a[i-1] * b[i]
end do
```

- After

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

note: new loops can be fused

Loop Spreading

- Move some of the second to the first
- Enables ILP by stepping over dependences
- Delay 2nd loop by max. dep. distance between 2nd and 1st loop statements, plus 1.

Loop Spreading

- Before

```
do i = 1, n/2
    a[i+1] = a[i+1] + a[i]
end do
do i = 1, n-3
    b[i+1] = b[i+1] +b[i] *
      a[i+3]
end do
```

- After

```
do i = 1, n/2
    COBEGIN
        a[i+1] = a[i+1] + a[i]
        if(i > 3) then
            b[i-2] = b[i-2] +
              b[i-3] * a[i]
        end if
    COEND
end do
do i = n/2-3, n-3
    b[i+1] = b[i+1] + b[i] *
      a[i+3]
end do
```

Replacement Transformations

(these change everything)

Reduction Recognition

- Before

```
do i = 1, n
   s = s + a[i]
end do
```

- After

```
real TS[64]
TS[1:64] = 0.0
do TI = 1, n, 64
   TS[1:64] = TS[1:64] +
     a[TI: TI+63]
end do
do TI = 1, 64
   s = s + TS[TI]
end do
```

note: legal if *fully* associative (watch out for FP ops...)

Array Statement Scalarization

- What do you do when you can't vectorize in hardware?

- Before

```
a[2:n-1] = a[2:n-1] +
  a[1:n-2]
```

- After (wrong)

```
do i = 2, n-1
   a[i] = a[i] + a[i-1]
end do
```

Array Statement Scalarization

- Before

```
a[2:n-1] = a[2:n-1] +
  a[1:n-2]
```

- After (wrong)

```
do i = 2, n-1
   a[i] = a[i] + a[a-1]
end do
```

- After (right)

```
do i = 2, n-1
   T[i] = a[i] + a[i-1]
end do
do i = 2, n-1
   a[i] = T[i]
end do
```

Array Statement Scalarization

- After (right)

```
do i = 2, n-1
   T[i] = a[i] + a[a-1]
end do
do i = 2, n-1
   a[i] = T[i]
end do
```

- After (even better)

```
do i = n-1, 2, -1
   a[i] = a[i] + a[a-1]
end do
```

Array Statement Scalarization

- However:

```
a[2:n-1] = a[2:n-1] + a[1:n-2] + a[3:n]
```

- must use a temporary, because:

```
do i = 2, n-1
   a[i] = a[i] + a[i-1] + a[i+1]
end do
```

- has antidependence either way.

Memory Access Transformations

(love your DRAM)

Array Padding

- Before

```
real a[8,512]
do i = 1, 512
   a[1,i] = a[1,i] + c
end do
```

- After

```
real a[9,512]
do i = 1, 512
   a[1,i] = a[1,i] + c
end do
```

note: assumes 8 banks of memory, similar for cache and TLB sets

Scalar Expansion

- Converts scalars to vectors
- Removes antidependences from temporaries
- Must be done when vectorizing

Scalar Expansion

- Before

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

- After

```
real T[n]
do all i = 1, n
   T[i] = b[i]
   a[i] = a[i] + T[i]
end do all
```

Array Contraction

- Before

```
real T[n,n]

do i = 1, n

   do all j = 1, n

      T[i,j] = a[i,j]*3

      b[i,j[ = T[i,j] +
       b[i,j]/T[i,j]

   end do all

end do
```

- After

```
real T[n]

do i = 1, n

   do all j = 1, n

      T[j] = a[i,j]*3

      b[i,j[ = T[j] +
       b[i,j]/T[j]

   end do all

end do
```

Scalar Replacement

- Before

```
do i = 1, n
   do j = 1, n
      total[i] =
       total[i] + a[i,j]
   end do
end do
```

- After

```
do i = 1, n
   T = total[i]
   do j = 1, n
      T = T + a[i,j]
   end do
   total[i] = T
end do
```

Transformations for Parallel Machines

(sharing the load)

Guard Introduction

- Before

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

- After

```
LBA = (n/Pnum)*Pid + 1
UBA = (n/Pnum)*(Pid + 1)
LBB = (n/Pnum)*Pid + 1
UBB = (n/Pnum)*(Pid + 1)
do i = 1, n
   if (LBA <= 1 .and. i <= UBA)
      a[i] = a[i] + c
   if (LBB <= 1 .and. i <= UBB)
      b[i] = b[i] + c
end do
```

Redundant Guard Elimination

- Before

```
LBA = (n/Pnum)*Pid + 1
UBA = (n/Pnum)*(Pid + 1)
LBB = (n/Pnum)*Pid + 1
UBB = (n/Pnum)*(Pid + 1)
do i = 1, n
   if (LBA <= 1 .and. i <= UBA)
      a[i] = a[i] + c
   if (LBB <= 1 .and. i <= UBB)
      b[i] = b[i] + c
end do
```

- After

```
LB = (n/Pnum)*Pid + 1
UB = (n/Pnum)*(Pid + 1)
do i = 1, n
   if (LB <= 1 .and. i <= UB)
      a[i] = a[i] + c
      b[i] = b[i] + c
   end if
end do
```

Bounds Reduction

- After

```
LB = (n/Pnum)*Pid + 1
UB = (n/Pnum)*(Pid + 1)
do i = LB, UB
       a[i] = a[i] + c
       b[i] = b[i] + c
end do
```

- Before

```
LB = (n/Pnum)*Pid + 1
UB = (n/Pnum)*(Pid + 1)
do i = 1, n
   if (LB <= 1 .and. i <= UB)
       a[i] = a[i] + c
       b[i] = b[i] + c
   end if
end do
```

That's all folks!