Simple measures, large impact

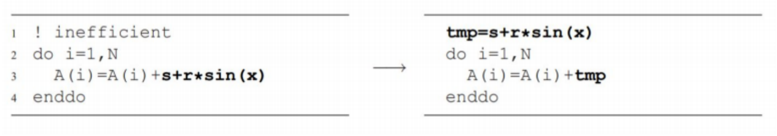
Vipin Vasu

Outline

Elimination of Common Subexpressions

- Common subexpression elimination is an optimization that is often considered a task for compilers
- Basically one tries to save time by precalculating parts of complex expressions and assigning them to temporary variables before a code construct starts that uses those parts multiple times.

Elimination of Common Subexpressions



```
! inefficient
do i=1,N
  A(i)=A(i)+s+r*sin(x)
enddo
```

$\longrightarrow$

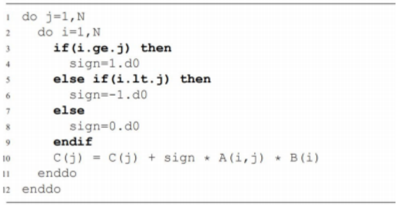
```
tmp=s+r*sin(x)
do i=1,N
  A(i)=A(i)+tmp
enddo
```

This optimization is also called loop invariant code motion

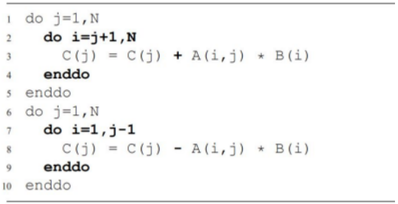
Avoiding Branches

- “Tight” loops, i.e., loops that have few operations in them, are typical candidates for software pipelining , loop unrolling, and other optimization techniques .
- Compiler optimization fails or is inefficient, performance will suffer if the loop body contains conditional branches.

Avoiding Branches



```
do j=1,N
  do i=1,N
    if(i.ge.j) then
      sign=1.d0
    else if(i.lt.j) then
      sign=-1.d0
    else
      sign=0.d0
    endif
    C(j) = C(j) + sign * A(i,j) * B(i)
  enddo
enddo
```

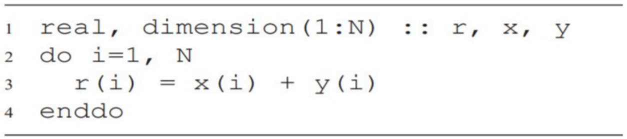


```
do j=1,N
  do i=j+1,N
    C(j) = C(j) + A(i,j) * B(i)
  enddo
enddo
do j=1,N
  do i=1,j-1
    C(j) = C(j) - A(i,j) * B(i)
  enddo
enddo
```

Using SIMD instruction sets

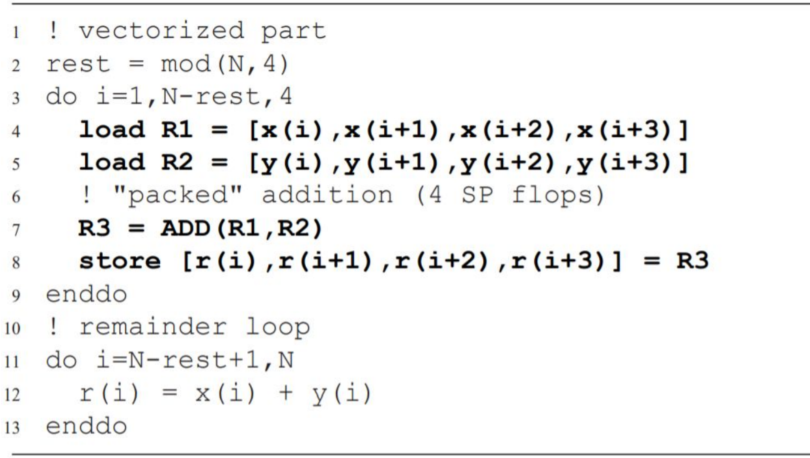
- The use of SIMD in microprocessors is often termed “vectorization,”
- Generally speaking, a “vectorizable” loop in this context will run faster if more operations can be performed with a single instruction.
- Preferring SIMD instructions over scalar ones is no guarantee for a performance improvement.
- If the code is strongly limited by memory bandwidth, no SIMD technique can bridge this gap.

Using SIMD instruction sets



```
real, dimension(1:N) :: r, x, y
do i=1, N
  r(i) = x(i) + y(i)
enddo
```

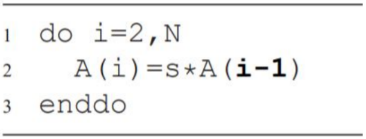
Using SIMD instruction sets



```
! vectorized part
rest = mod(N,4)
do i=1,N-rest,4
  load R1 = [x(i),x(i+1),x(i+2),x(i+3)]
  load R2 = [y(i),y(i+1),y(i+2),y(i+3)]
  ! "packed" addition (4 SP flops)
  R3 = ADD(R1,R2)
  store [r(i),r(i+1),r(i+2),r(i+3)] = R3
enddo
! remainder loop
do i=N-rest+1,N
  r(i) = x(i) + y(i)
enddo
```

Using SIMD instruction sets

- Some SIMD instruction sets distinguish between aligned and unaligned data.
- In cases where the compiler knows nothing about the alignment of arrays used in a vectorized loop and cannot otherwise influence it, unaligned loads and stores must be used, incurring some performance penalty.
- A loop with a true dependency cannot be SIMD vectorized.



```
do i=2,N
   A(i)=s*A(i-1)
enddo
```

Using SIMD instruction Sets

- There are no fixed guidelines for when a loop qualifies as vectorized.
- Load and store instructions could still be scalar; compilers tend to report such loops as "vectorized".
- On x86 processors with SSE(Streaming SIMD Extensions) support, the lower and higher 64 bits of a register can be moved independently.

Using SIMD instruction Sets

```
rest = mod(N,2)
do i=1,N-rest,2
  ! scalar loads
  load R1.low = x(i)
  load R1.high = x(i+1)
  load R2.low = y(i)
  load R2.high = y(i+1)
  ! "packed" addition (2 DP flops)
  R3 = ADD(R1,R2)
  ! scalar stores
  store r(i) = R3.low
  store r(i+1) = R3.high
enddo
! remainder "loop"
if(rest.eq.1) r(N) = x(N) + y(N)
```

Using SIMD instruction Sets

- If the compiler cannot be convinced to properly vectorize a loop even with additional command line options or source code directives, before using assembly language altogether is to employ compiler intrinsics.
- Intrinsics are constructs that resemble assembly instructions so closely that they can usually be translated 1:1 by the compiler.

The End