Scientific and Technical Computing

Hardware and Code Optimization

Lars Koesterke

UT Austin, 11/03/20

Optimizations in Code

Optimizations in Code

Moving invariant code outside the loop

```
do i=1, n
  a(i) = b(i) + x*y
enddo
```

```
z = x*y
do i=1, n
  a(i) = b(i) + z
enddo
```

```
do i=1, n
  a(i) = b(i) + x/y
enddo
```

```
z = x/y
do i=1, n
  a(i) = b(i) + z
enddo
```

Optimizations in Code

Cost of a division: maybe 60 cycles (depends highly on precision and accuracy)

```
do i=1, n
  a(i) = b(i) + c(i)/x
enddo
```

→

```
xi = 1. / x
do i=1, n
  a(i) = b(i) + c(i) * xi
enddo
```

Note that the following is not exactly (bit-wise) the same

```
z1 = y / x

z2 = y * (1./x)
```

Do no assume that z1 is exactly/bit-wise z2

If you change the code, or if you allow the compiler to change the code, you will get a slightly different result (last bits will vary)

Compiler options allow you to control (to some degree)

- Whether operations in code are replaced by cheaper operations
- Whether operations are 'executed' to the fullest accuracy (which is expensive)
  - The default is some 'reasonable' compromise between accuracy and speed

Many optimization techniques will change the rounding bits.

Hence, bit-wise reproducibility is not the goal

Optimizations in Code

Changing result by reordering operations

Assume that the data representation has **4 significant digits**

```
x = 1.e-5
s = 0.
do i=1, 10000
  s = s + x
enddo
s = s + 1.

print s
```

```
x = 1.e-5
s = 1.
do i=1, 10000
  s = s + x
enddo

print s
```

Do we agree that the 2 code versions are intended to calculate the same sum?

Optimizations in Code

Changing result by reordering operations

Assume that the data representation has 4 significant digits

```
x = 1.e-5
s = 0.
do i=1, 10000
  s = s + x
enddo
s = s + 1.
print s
```

```
x = 1.e-5
s = 1.
do i=1, 10000
  s = s + x
enddo

print s
```

What is being printed?

What is 1. + 1.e-5?

TACC

Optimizations in Code

Changing result by reordering operations

Assume that the data representation has 4 significant digits

```
x = 1.e-5
s = 0.
do i=1, 10000
   s = s + x
enddo
s = s + 1.
print s                1.1
```

```
x = 1.e-5
s = 1.
do i=1, 10000
   s = s + x
enddo

print s                      1.
```

Do not hope for bit-wise reproducible results

Too many things out of your control will change actual numeric

Aiming for bit-wise reproducibility will prevent you from

Achieving high performance

Reproducible results are a key ingredient of the ‘scientific method’

But bit-wise reproducibility is not required

Optimizations in Code

What is the problem here (in terms of code performance)?
Independent of what actual language is being used (C or Fortran)

```
do j=1,n
  do i=1,n
    a(i,j)=a(i,j)+s*b(j,i)
  end do
end do
```

Optimizations in Code

What is the problem here (in terms of code performance)?

```
do j=1,n
  do i=1,n
    a(i,j)=a(i,j)+s*b(j,i)
  end do
end do
```

One array is accessed stride-1

One array is accessed with a high stride

Changing the loop order does not help

Optimizations in Code

Strip mining :

Transforms a single loop into two loops to insure that each element of a cache line is used, once it is brought into the cache. In the example below, the loop over j is split into two:

```
do j=1,n
  do i=1,n
    a(i,j)=a(i,j)+s*b(j,i)
  end do
end do
```

→

```
do jouter=1,n,8
  do j=jouter,min(jouter+7,n)
    do i=1,n
      a(i,j)=a(i,j)+s*b(j,i)
    end do
  end do
end do
```

↓

```
do jouter=1,n,8
  do i=1,n
    do j=jouter,min(jouter+7,n)
      a(i,j)=a(i,j)+s*b(j,i)
    end do
  end do
end do
```

Memory Tuning

Array Blocking

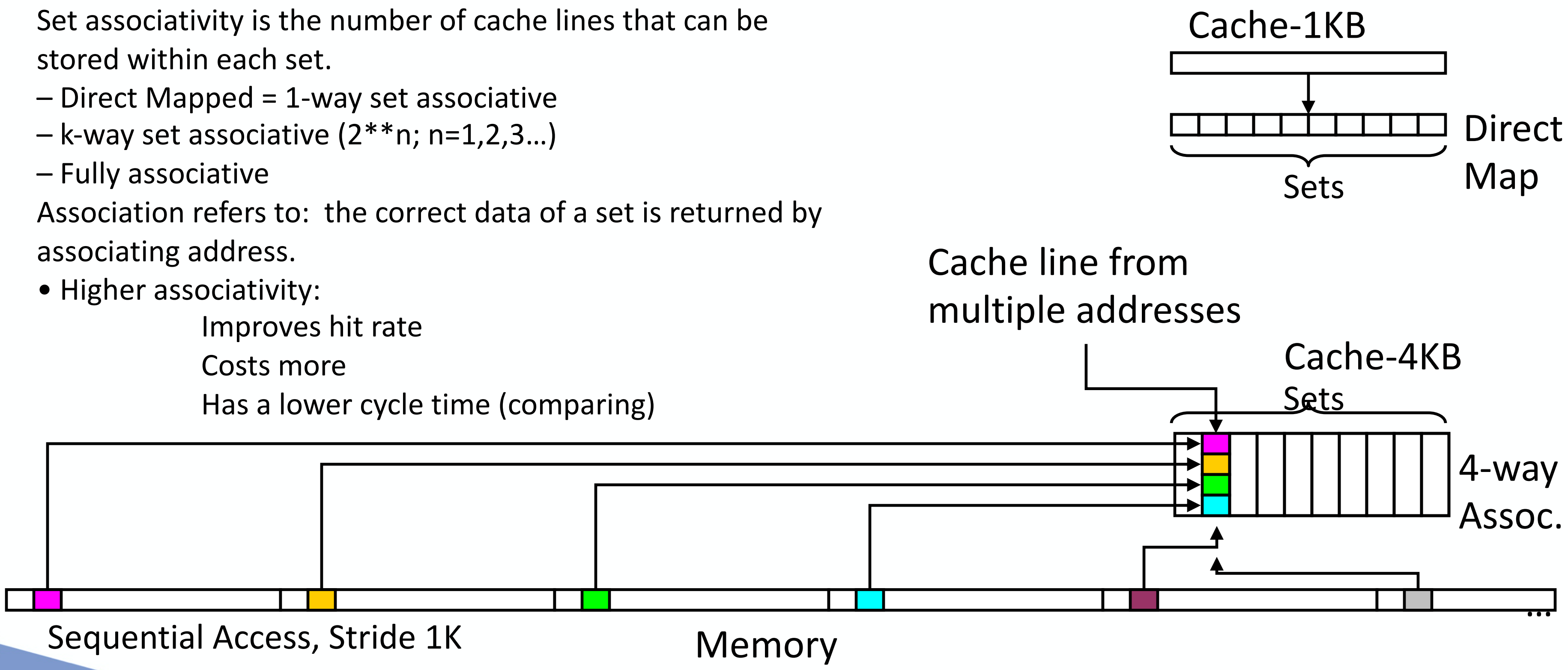
The objective of array blocking is to work with small array blocks when expressions contain mixed-stride operations. It uses complete cache lines when they are brought in from memory, and hence avoid possible eviction that would otherwise ensue without blocking.

```
do i=1,n
do j=1,n
 A(j,i)=B(i,j)
end do
end do
```

```
do i=1,n,2
do j=1,n,2
 A(j  ,i  )=B(i  ,j  )
 A(j+1,i  )=B(i+1,j  )
 A(j  ,i+1)=B(i  ,j+1)
 A(j+1,i+1)=B(i+1,j+1)
end do
end do
```

Cache Set Associativity

Caches are divided into “sets”.
Set associativity is the number of cache lines that can be stored within each set.
– Direct Mapped = 1-way set associative
– k-way set associative (2**n; n=1,2,3…)
– Fully associative
Association refers to: the correct data of a set is returned by associating address.
• Higher associativity:
  Improves hit rate
  Costs more
  Has a lower cycle time (comparing)



TACC

Cache Tuning

Memory access with a stride of a high power of two usually leads to this form of cache thrashing, because data cache sizes are a (high) power of two in bytes. The following **cache trashing** code references array elements with a stride of 8KB:

```
program MAIN
integer, parameter :: n=1000
real*8 :: A(1024,n), B(1024,n)
real*8 :: C(1024,n), D(1024,n)
common /Arrays/ A,B,C,D
...
do i=1,n
   D(64,i)= c1*A(64,i)+c2*B(64,i)+c3*C(64,i)
end do
...
end program
```

```
Integer, Parameter :: n=1000, pad=24
Real*8 :: A(1024+pad,n)
Real*8 :: B(1024+pad,n)
Real*8 :: C(1024+pad,n)
Real*8 :: D(1024+pad,n)
```

TACC

Memory Tuning

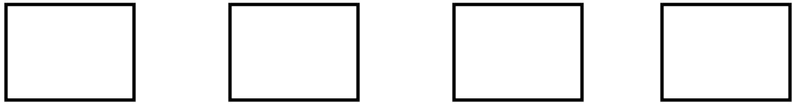
Array Blocking for matrix x matrix multiplication

```
real*8 a(n,n), b(n,n), c(n,n)
do ii=1,n,nb
   do jj=1,n,nb
      do kk=1,n,nb
         do i=ii,min(n,ii+nb-1)
            do j=jj,min(n,jj+nb-1)
               do k=kk,min(n,kk+nb-1)

            c(i,j)=c(i,j)+a(j,k)*b(k,i)
```

Simple implementation has 3 loops

Let’s not get into the details here



nb x nb   nb x nb   nb x nb   nb x nb

```
end do; end do; end do; end do; end do; end do
```

TACC

More is better!

But then there are also power concerns …