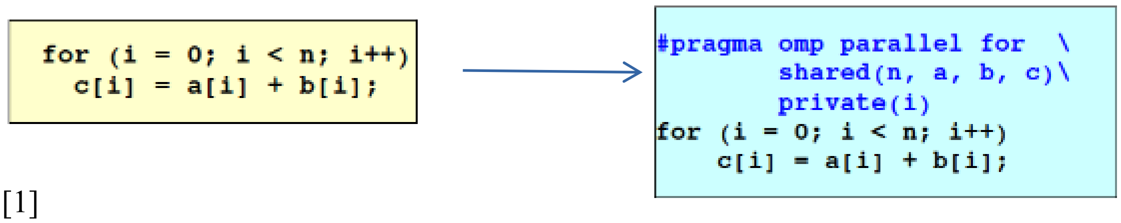
Eric Paulz
CPSC 3300-002
Project 2

Loop Optimization Techniques

Implicit Threading of Loops using OpenMP

Implicit threading is the use of libraries to hide the management of threads. In C, this is commonly done using the OpenMP library. OpenMP uses the *#pragma* compiler directive to detect and insert additional library code at compile time. The goal of implicit threading is to allow the programmer to focus on his/her algorithm and not having to worry about multithreading aspects. The OpenMP implementation injects code from the library to perform thread creation and then join them back at the end.



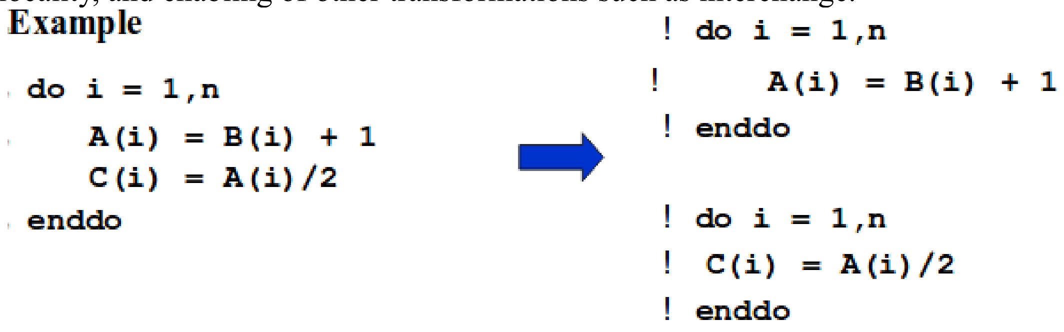
```
 for (i = 0; i < n; i++)
   c[i] = a[i] + b[i];
```

```
#pragma omp parallel for   \
        shared(n, a, b, c)\
        private(i)
for (i = 0; i < n; i++)
    c[i] = a[i] + b[i];
```

[1]

Loop Fission

The idea of loop fission is to split a loop nest into multiple loop nests (the inverse of fusion) Some motivations for using loop fission include: production of multiple less constrained loops, improved locality, and enabling of other transformations such as interchange.



Example

```
do i = 1,n
    A(i) = B(i) + 1
    C(i) = A(i)/2
enddo
```

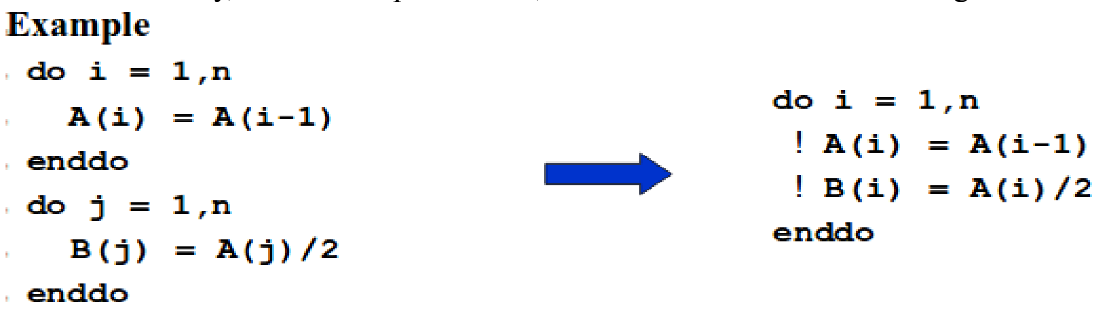
```
! do i = 1,n
!       A(i) = B(i) + 1
! enddo

! do i = 1,n
!  C(i) = A(i)/2
! enddo
```

[2]

Loop Fusion

The idea of loop fusion is to combine multiple loop nests into one single loop. Some advantages of this are: improved data locality, reduced loop overhead, and better instruction scheduling.



Example

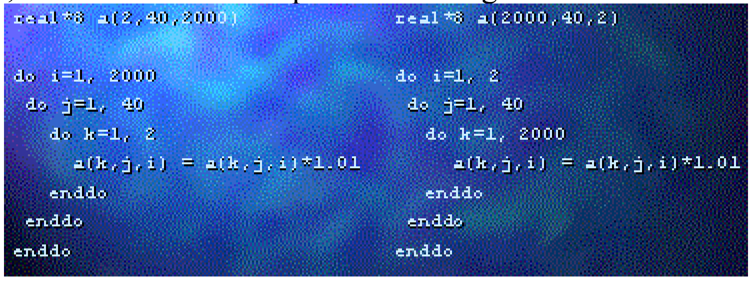
```
do i = 1,n
   A(i) = A(i-1)
enddo
do j = 1,n
   B(j) = A(j)/2
enddo
```

```
do i = 1,n
 ! A(i) = A(i-1)
 ! B(i) = A(i)/2
enddo
```

[2]

Eric Paulz
CPSC 3300-002
Project 2

Loop Interchange
By swapping the nesting order of loops we can: minimize stride, reduce loop overhead where inner loop counts are smaller, and allow better compiler scheduling.

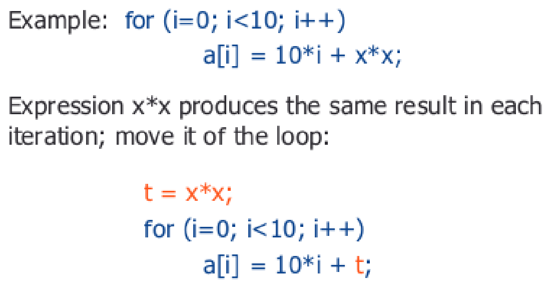


```
real*8 a(2,40,2000)                real*8 a(2000,40,2)

do i=1, 2000                       do i=1, 2
 do j=1, 40                         do j=1, 40
   do k=1, 2                          do k=1, 2000
     a(k,j,i) = a(k,j,i)*1.01            a(k,j,i) = a(k,j,i)*1.01
   enddo                              enddo
 enddo                              enddo
enddo                              enddo
```

[3]

Loop Invariant Code Motion
The idea of this concept is that if a computation produces the same result in all loop iterations, move it out of the loop.



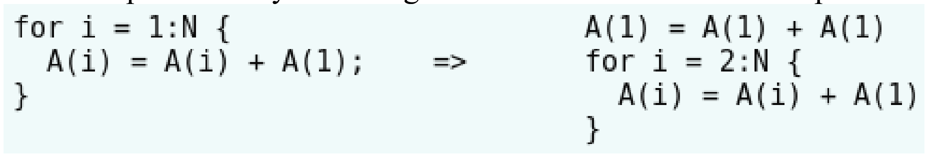
Example:
```
for (i=0; i<10; i++)
    a[i] = 10*i + x*x;
```

Expression x*x produces the same result in each iteration; move it of the loop:

```
t = x*x;
for (i=0; i<10; i++)
    a[i] = 10*i + t;
```

[4]

Loop Peeling
Eliminate loop-carried dependence by unfolding the first few iterations of a loop.



```
for i = 1:N {                              A(1) = A(1) + A(1)
  A(i) = A(i) + A(1);     =>               for i = 2:N {
}                                            A(i) = A(i) + A(1)
                                           }
```

[5]

Loop Strip Mining
By breaking a large loop up into smaller segments (or strips), this technique transforms the loop structure in two ways:

- it increases locality in the data cache
- it reduces the number of iterations of the loop by a factor of the length of each vector

```
i = 1
do while (i<=n)
a(i) = b(i) + c(i) ! Original loop code
i = i + 1
end do
```

$\longrightarrow$

```
i = 1
do while (i < (n - mod(n,4)))
! Vector strip-mined loop.
a(i:i+3) = b(i:i+3) + c(i:i+3)
i = i + 4
end do
do while (i <= n)
a(i) = b(i) + c(i)      !Scalar clean-up loop
i = i + 1
end do
```

[6]

Loop Tiling/Blocking

Main purpose is to eliminate as many cache misses as possible. This technique transforms the memory domain into smaller chunks rather than sequentially traversing through the entire memory domain. Each chunk should be small enough to fit all the data for a given computation into the cache, thereby maximizing data reuse.

```
REAL A(MAX,MAX), B(MAX,MAX)
DO I =1, MAX
DO J = 1, MAX
  A(I,J) = A(I,J) + B(J,I)
ENDDO
ENDDO
```

```
REAL A(MAX,MAX), B(MAX,MAX)
DO I =1, MAX, BS
DO J = 1, MAX, BS
  DO II = I, I+MAX, BS-1
    DO J = J, J+MAX, BS-1
      A(II,JJ) = A(II,JJ) + B(JJ,II)
    ENDDO
  ENDDO
ENDDO
ENDDO
```

[6]

Loop Unrolling

Loop unrolling reduces loop overhead and improves effectiveness of other transformations such as code scheduling and CSE. It does this by first making n copies of the loop (n is the 'unrolling factor'), and then adjusting the loop bounds accordingly.

Example

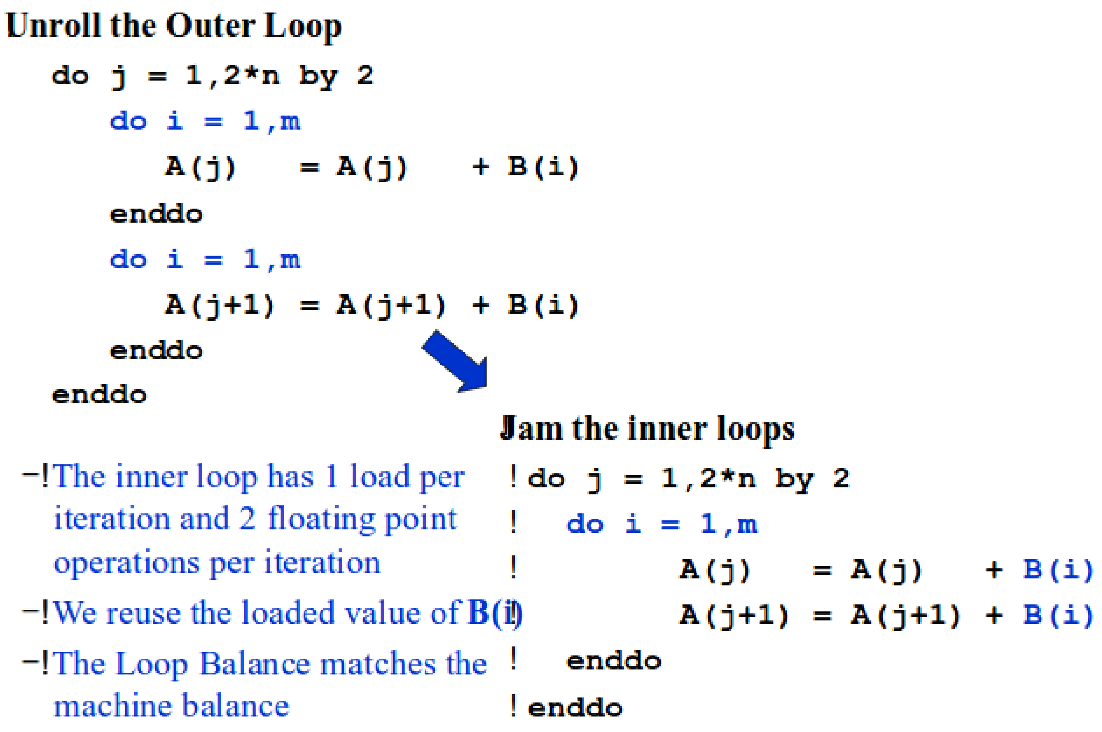
```
do i=1,n
   A(i) = B(i) + C(i)
enddo
```

```
do i=1,n by 2
   A(i)   = B(i) + C(i)
   A(i+1) = B(i+1) + C(i+1)
enddo
```

[2]

Loop Unroll and Jam

The basic idea of Unroll and Jam is to restructure loops so that loaded values are used many times per iteration. First, you must unroll the outer loop some number of times, and then fuse (jam) the resulting inner loops.



Unroll the Outer Loop

```
do j = 1,2*n by 2
   do i = 1,m
      A(j)   = A(j)   + B(i)
   enddo
   do i = 1,m
      A(j+1) = A(j+1) + B(i)
   enddo
enddo
```

Jam the inner loops

```
! do j = 1,2*n by 2
!    do i = 1,m
!         A(j)   = A(j)   + B(i)
          A(j+1) = A(j+1) + B(i)
!    enddo
! enddo
```

−!The inner loop has 1 load per iteration and 2 floating point operations per iteration

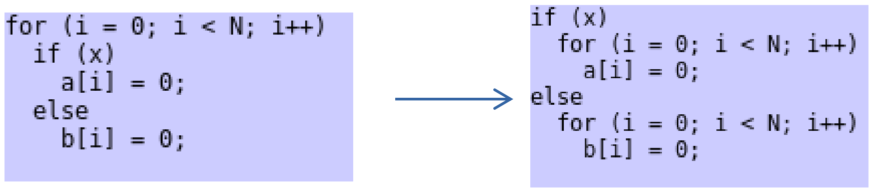
−!We reuse the loaded value of **B(i)**

−!The Loop Balance matches the machine balance

[2]

Loop Unswitching

A loop containing a loop-invariant IF statement can be transformed into an IF/ELSE statement containing two loops.



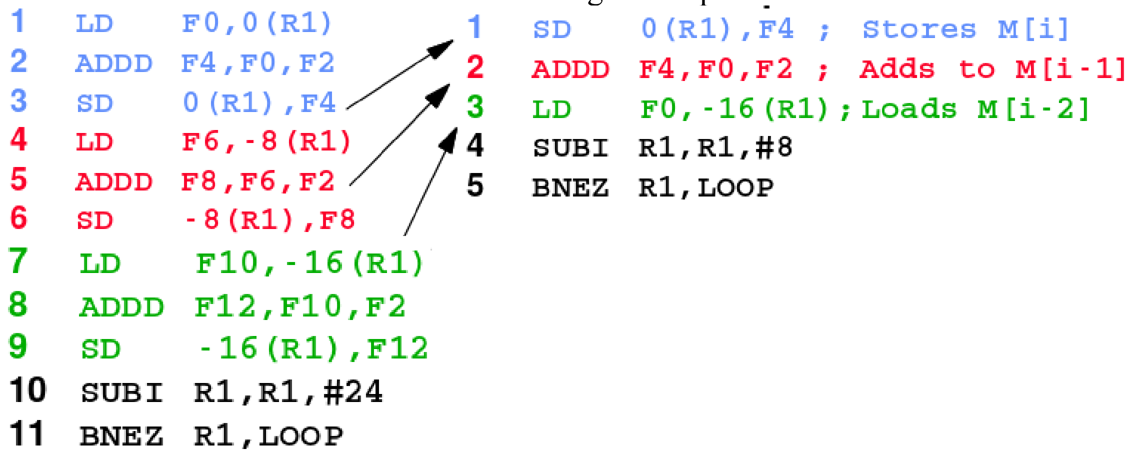
```
for (i = 0; i < N; i++)
  if (x)
    a[i] = 0;
  else
    b[i] = 0;
```

```
if (x)
  for (i = 0; i < N; i++)
    a[i] = 0;
else
  for (i = 0; i < N; i++)
    b[i] = 0;
```

[7]

Software Pipelining

The key observation needed to understand the function of Software Pipelining is that if iterations from loops are independent, they can get Instruction Level Parallelism (ILP) by taking instructions from different iterations. Software Pipelining reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop.



```
LD    F0,0(R1)
ADDD  F4,F0,F2
SD    0(R1),F4
LD    F6,-8(R1)
ADDD  F8,F6,F2
SD    -8(R1),F8
LD    F10,-16(R1)
ADDD  F12,F10,F2
SD    -16(R1),F12
SUBI  R1,R1,#24
BNEZ  R1,LOOP
```

```
SD    0(R1),F4 ; Stores M[i]
ADDD  F4,F0,F2 ; Adds to M[i-1]
LD    F0,-16(R1);Loads M[i-2]
SUBI  R1,R1,#8
BNEZ  R1,LOOP
```

[8]

Hurdles to Loop Optimization

Loop-Carried Dependencies

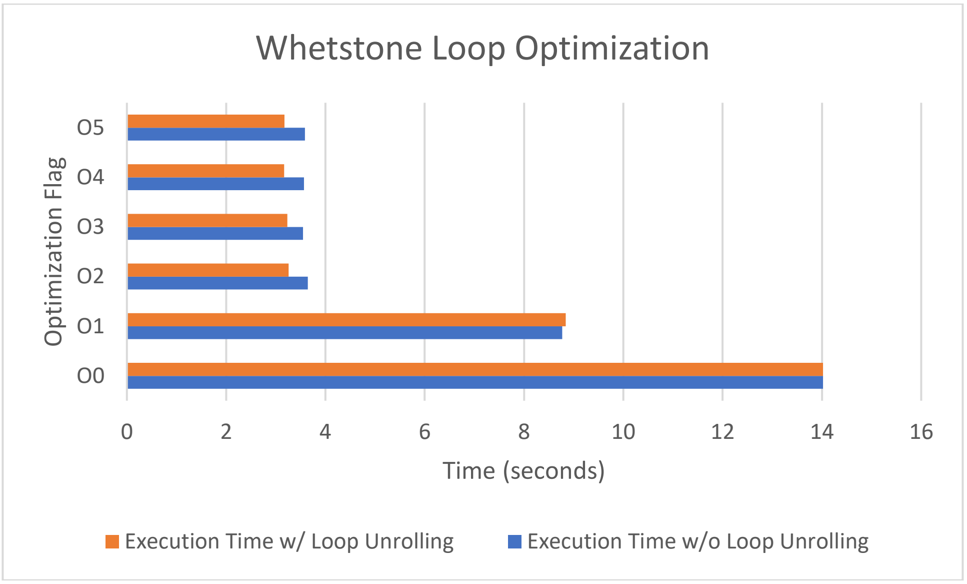
A key concept to observe when looking at parallelization is loop carried dependence, which is a dependence that crosses loop iterations. If a loop carried dependence exists, that means that the loop cannot be parallelized. This is because some number of iterations of the loop depend on other iterations of the same loop.

Array Element Aliasing

Aliasing describes a situation in which a data location in memory can be accessed through symbolic names in the program. By doing this, modifying the data through one name will modify any values associated with all aliased names. The C language does not allow a programmer to create two pointers of different types that point to the same memory location. This is known as the *strict aliasing rule*, which can sometimes allow for significant increases in performance. However, it is considered unsafe in many cases and tends to be very difficult.

Experimentation

For my experimentation I used a ThinkPad running Windows 10 with an Intel® Core™ i5-6300U CPU @ 2.40GHz (2 cores) and 8GB of memory. I ran compiled whetstone.c with a total of 10 different optimization flags: -O0, -O1,…,-O5 and -O0 -funroll-loops, -O1 -funroll -loops,…,-O5 -funroll -loops. The chart below shows the execution times of each, where the blue bars represent execution without loop unrolling and the orange bars represent the same optimization flag but with -funroll -loops added to it.



References

[1] College of Integrated Science and Engineering, "Computer Systems Fundamentals," *James Madison University*, 2017. [Online]. Available: https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/cs361/html/ImpThr.html. [Accessed: April 6, 2018].

[2] M. Strout, "Home Page-Michelle Mills Strout, Ph.D.", October, 2014. [Online]. Available: http://www.cs.colostate.edu/~mstrout/CS553Fall09/Slides/lecture18-fusion- KellyPugh.ppt.pdf. [Accessed: April 6, 2018].

[3] P. Mucci, "Home Page-Phil Mucci", September, 2007. [Online]. Available: http://www.icl.utk.edu/~mucci/MPPopt/hpcug-html/sld047.htm. [Accessed: April 6, 2018].

[4] Cornell University Computer Science Department, "Introduction to Compilers", *Cornell University,* 2008. [Online]. Available: https://courses.cs.cornell.edu/cs412/2004sp/lectures/lec29.pdf. [Accessed: April 7, 2018].

[5] A. Chauhan, "Home Page-Arun Chauhan", Fall 2006. [Online]. Available: https://www.cs.indiana.edu/~achauhan/Teaching/B629/2006-Fall/LectureNotes/27-loop-transformations.html. [Accessed: April 7, 2018].

[6] Z. Moravec, "Home Page-Zdenek Moravec, Ph. D.", February 2018. [Online]. Available: http://physics.ujep.cz/~zmoravec/prga/main_for/mergedProjects/optaps_for/common/optaps_vec_mine.htm. [Accessed: April 6, 2018].

[7] Nullstone Corporation, "Loop Fusion", *Compiler Optimizations,* 2012. [Online]. Available: http://compileroptimizations.com/category/loop_fusion.htm. [Accessed: April 8, 2018].

[8] D. Patterson, "Home Page-David A. Patterson", Fall 1996. [Online]. Available: https://people.eecs.berkeley.edu/~pattrsn/252F96/Lecture05.pdf. [Accessed: April 6, 2018].