RISC Optimisation Techniques

Ruud van der Pas

SiliconGraphics

European High Performance Computing Team

Optimisation Techniques

- Several techniques are available to enhance performance
- There are basically two different categories:
  - Improve CPU performance
    - Create more opportunities to go superscalar (high level)
    - Better instruction scheduling (low level)
  - Improve memory performance (often most important)
    - Better memory access pattern
    - Optimal usage of cache lines
    - Re–usage of cached data

Optimisation Techniques

- **The following techniques will be now be discussed:**
  - **Loop Unrolling**
  - **Loop Fusion**
  - **Loop Fission/Splitting**
  - **Loop interchange**
  - **Loop Blocking**
  - **Procedure Inlining**
- **The SGI compilers perform all of the optimisations listed, *so very often this is taken care of automatically***

Loop Unrolling

Basic idea: perform multiple loop iterations at the same time

```
DO I=1, N, 1
     ...(i)...
END DO
```

```
DO I=1, N, unroll
     ...(i)...
     ...(i+1)...
     ...(i+2)...
     ...(i+unroll–1)...
END DO
```

Advantages of loop unrolling:

- ***Better opportunities for superscalar code***
- ***More data re–usage***
- ***Reduction in loop overhead*** (usually a minor advantage)

Disadvantages:

- ***More registers needed***
- ***Clean–up code required to ensure same number of iterations***
  (do i=1,N–mod(N,unroll)),unroll & do i=N–mod(N,unroll)+1,N)

Loop Unrolling – Example

```
DO k=1,N
     DO j=1,N
          DO i=1,N
               C(i,j)=C(i,j)+A(i,k)* B(k,j)
          END DO
     END DO
END DO
```

```
DO k=1,N
     DO j=1,N
          DO i=1,N, 2
               C(i,j)   = C(i,j)   +A(i,k)   * B(k,j)
               C(i+1,j)= C(i+1,j)+A(i+1,k) * B(k,j)
          END DO
     END DO
END DO
```

Advantage:

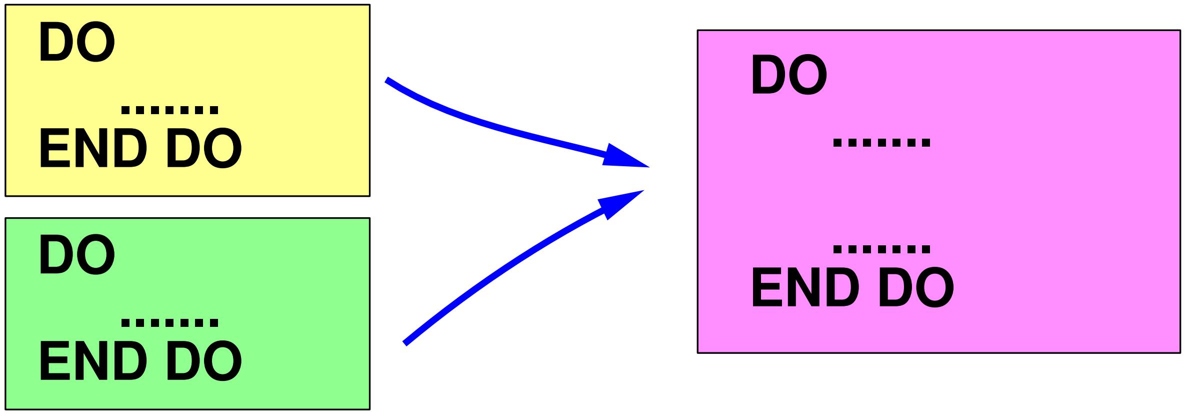
- ***Data values are kept in registers and are re-used***

Disadvantages:

- ***Clean-up code required***
- ***More registers needed***

Loop Fusion

Basic idea: Merge multiple loops into one



Advantages of loop fusion:

- ***More data re–usage***
- ***Better opportunities for superscalar code***
- ***Reduction in loop overhead*** (usually a minor advantage)

Disadvantages:

- ***May give more total data in the fused loop***
- ***More registers needed***

Loop Fusion – Example

```
DO i = 1, N
   B(i) = 2 * A(i)
END DO
```

```
DO k = 1, N
   C(k) = B(k) + D(k)
END DO
```

```
DO ii = 1, N
   B(ii) = 2 * A(ii)
   C(ii) = B(ii) + D(ii)
END DO
```

Advantage:

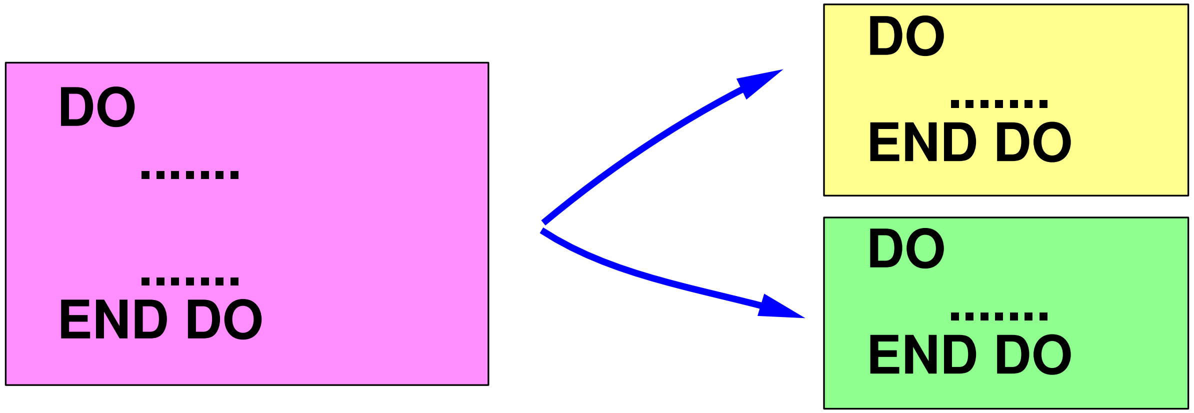
- *Re–usage of array B()*

Disadvantages:

- *In total 4 arrays now contend for cache space*
- *More registers needed*

Loop Fission/Splitting

Basic idea: Split one loop into multiple loops



Advantages of loop fission:

- ***Better data re–usage*** (independent arrays no longer step on each other)
- ***Possible to isolate data dependencies*** (and so create more SWP and parallelisation opportunities)

Disadvantage:

- ***Increase in loop overhead*** (usually a minor disadvantage)

Loop Fission/Splitting – Example

```
DO i = 1, N
    B(i) = 2 * A(i)
    D(i) = D(i–1) + C(i)
END DO
```

```
DO ii = 1, N
    B(ii) = 2 * A(ii)
END DO
```

```
DO ii = 1, N
    D(ii) = D(ii–1) + C(ii)
END DO
```

Advantage:

– ***First loop can be scheduled more efficiently and be parallelised as well***

Disadvantage:

– ***Additional loop created*** (a minor disadvantage)

Loop Interchange

Basic idea: In a nested loop, examine and possibly change the order of the loops

Advantages:

- ***Better memory access patterns*** (leading to improved cache and memory usage)
- ***Elimination of data dependencies*** (to increase the opportunities for CPU optimisations and parallelisation)

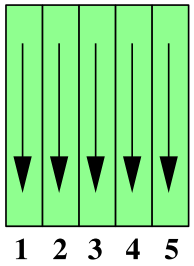
Disadvantage:

- ***May make a short loop innermost*** (which is not good for optimal SWP performance)

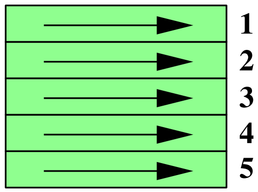
Loop Interchange

The storage order is language dependent:

- **Fortran stores "column–wise"**
  - **The compiler knows this**



- **C stores "row–wise"**
  - **The compiler knows this**

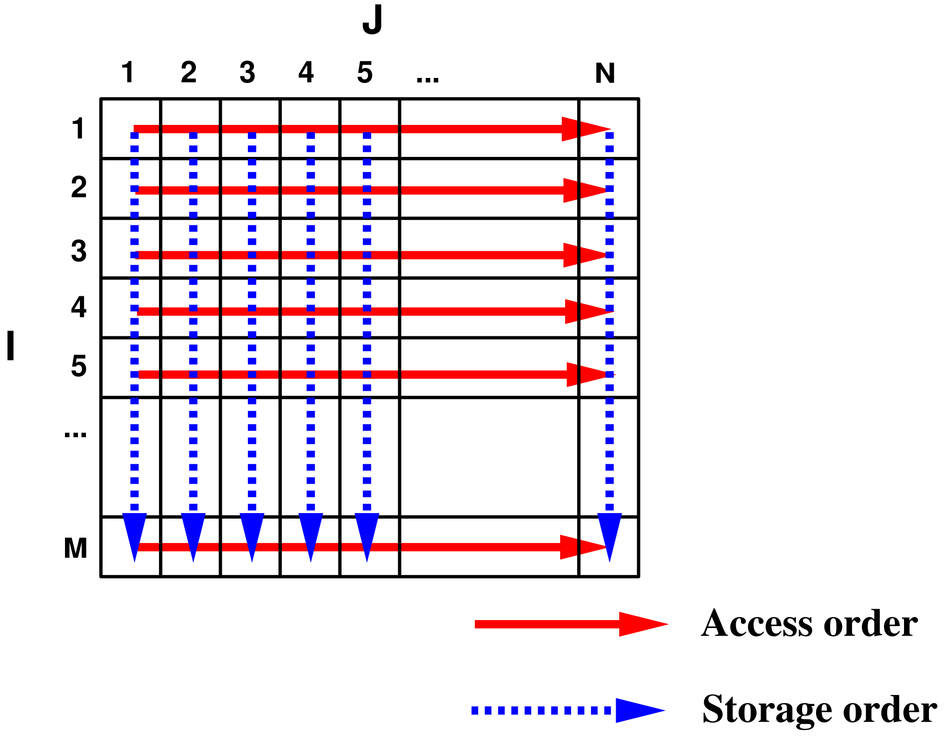


Accessing elements in storage order greatly enhances the performance!

Loop Interchange – Example

Code Fragment:

```
DO i=1,N
     DO j=1,M
          C(i,j) = A(i,j) + B(i,j)
     END DO
 END DO
```

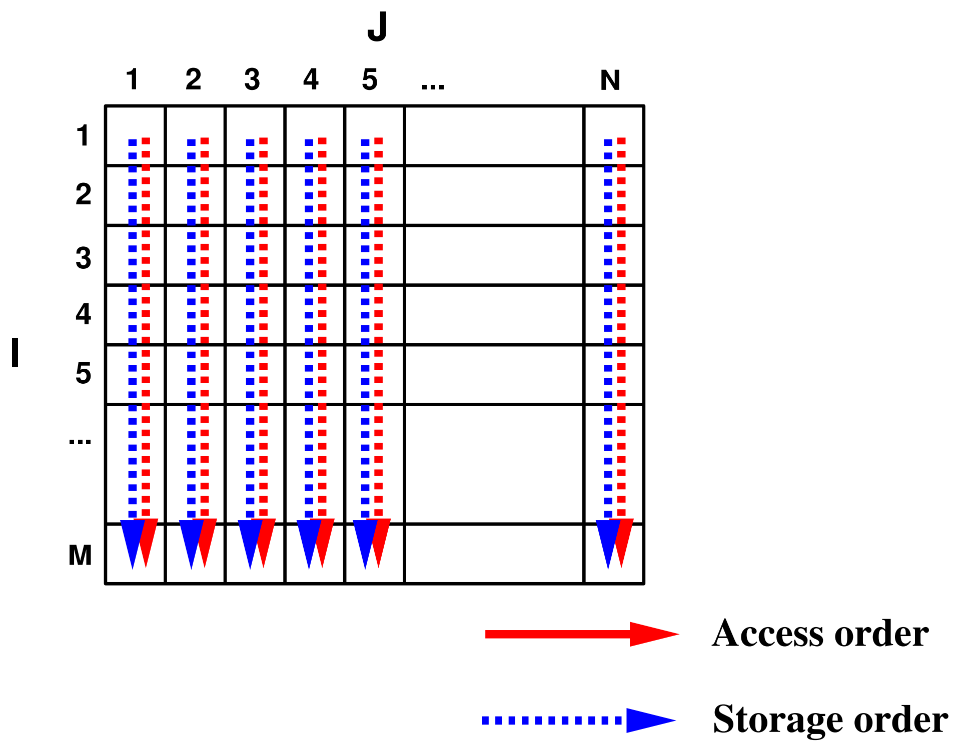


In this nested loop we will severely trash the cache, for M sufficiently large

Loop Interchange – Example

Code Fragment (modified):

```
DO j=1,M
   DO i=1,N
      C(i,j) = A(i,j) + B(i,j)
   END DO
END DO
```



We will now exploit all cache line elements, for any value of M

Loop interchange in C programs

In C, the situation is exactly the opposite

```
for (j=0; j<M; j++)
   for (i=0; i<N; i++)
      C[i][j] = A[i][j] + B[i][j];
```

interchange

index reversal

```
for (i=0; i<N; i++)
   for (j=0; j<M; j++)
      C[i][j] = A[i][j] + B[i][j];
```

```
for (j=0; j<M; j++)
   for (i=0; i<N; i++)
      C[j][i] = A[j][i] + B[j][i];
```

- *The performance benefit is the same in this case*
- *In many practical situations, loop interchange is much easier to achieve than index reversal*

Loop interchange – Another example

```
COMMON /SHARE/B(M,N)

           ......

SUBROUTINE DO_WORK(N,M,A)

REAL A(N,M)
DO i=1,N
   DO j=1,M
      A(i,j) = A(i,j) + B(j,i)
   END DO
 END DO
```

What's the problem here?

Loop interchange – Another example

```
COMMON /SHARE/B(M,N)

          ......

SUBROUTINE DO_WORK(N,M,A)

REAL A(N,M)
DO j=1,M
   DO i=1,N
      A(i,j) = A(i,j) + B(j,i)
   END DO
 END DO
```

Interchange will be good for B, but bad for A

Index reversal on B, with an associated amount of work (!), will do the trick

Loop interchange – Another example

```
COMMON /SHARE/B(N,M)

        ......

SUBROUTINE DO_WORK(N,M,A)

REAL A(N,M)
DO j=1,M
   DO i=1,N
      A(i,j) = A(i,j) + B(i,j)
   END DO
 END DO
```

This optimisation will help on any system ...

Lab: Interchange

Lab: Loop interchange

```
SUBROUTINE DO_WORK (N,A,B,C)

DIMENSION A(N,N,N), B(N,N,N), C(N,N,N)

DO i=1,N
   DO j=1,N
      DO k=1,N
         A(i,j,k) = A(i,j,k) + B(i,j,k)*C(i,j,k)
      END DO
   END DO
 END DO
```

- *There are 6 different ways to order these loops*
- *Measure the performance for all variants (N=300, 300MB of memory needed)*

Lab: Loop interchange – Solution

```
DO i=1,300
   DO j=1,300
      DO k=1,300
         A(i,j,k) = A(i,j,k)+B(i,j,k)*C(i,j,k)
      END DO
   END DO
END DO
```

Loop order	R8K@90MHz (4MB cache)	R10K@195MHz (2MB cache)
i j k	176.3	346.1
i k j	136.2	155.8
j i k	51.1	63.9
j k i	11.0	10.7
k i j	10.2	10.3
k j i	9.6	9.2

Timings are in seconds and have been obtained on a PowerChallenge

Loop interchange – Indexing

There are several ways to index arrays:

```
DO j = 1, M
   DO i = 1, N
      .. A(i,j) ..
   END DO
END DO
```

++

Direct

```
DO j = 1, M
   DO i = 1, N
      .. A(i+(j-1)*N) ..
   END DO
END DO
```

+

Explicit

```
DO j = 1, M
   DO i = 1, N
      k = k + 1
      .. A(k) ..
   END DO
END DO
```

–

Loop carried

```
DO j = 1, M
   DO i = 1, N
     .. A(index(i,j)) ..
   END DO
END DO
```

– –

Indirect

The addressing scheme can (and will) have an impact on the performance

Case Study

Case Study: Loop interchange

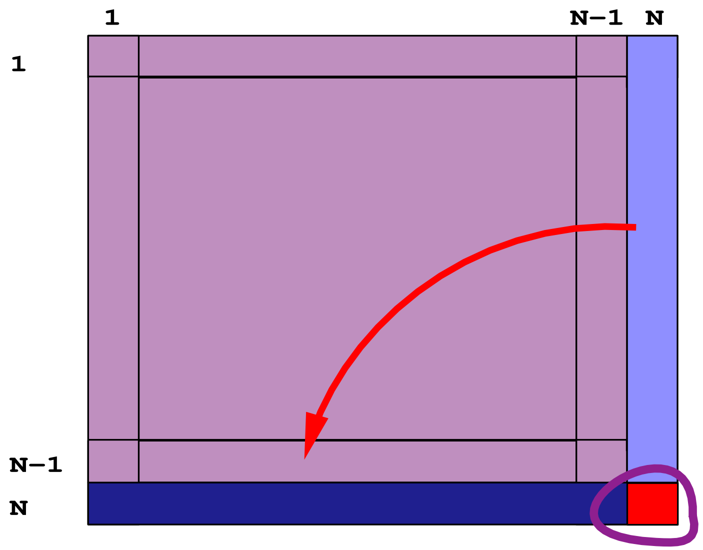
```
DIMENSION  A(N,N,N),B(N,N,N),C(N,N)

DO i=1,N
   DO j=1,N
      DO k=1,N
         A(i,j,k) = A(i,j,k) + B(i,j,k) * C(i,j)
      END DO
   END DO
   C(N,i) = C(i,N) + Cst1
 END DO
```

Diagnosis:

- ***Arrays A, B and C are accessed over the wrong dimension***
- ***So, we would like to interchange the loops***
- ***However, a delicate coupling between A and C exists***

Case Study: Loop interchange

The last row of C is modified:




```
DIMENSION  A(N,N,N),B(N,N,N),C(N,N)

DO i=1,N
   DO j=1,N
      DO k=1,N
         A(i,j,k) = A(i,j,k) + B(i,j,k) * C(i,j)
      END DO
   END DO
   C(N,i) = C(i,N) + Cst1
 END DO
```

All we need to do now is to split those operations that use C(N,1...N−1) and C(N,N)

Needs special care

Case Study: interchange – step #1

```
DO i=1,N-1
   DO j=1,N
      DO k=1,N
         A(i,j,k) = A(i,j,k) + B(i,j,k) * C(i,j)
      END DO
   END DO
END DO
```

The OLD value is needed here

```
DO k=1,N
   A(N,N,k) = A(N,N,k) + B(N,N,k) * C(N,N)
END DO

DO i=1,N
   C(N,i) = C(i,N) + Cst1
END DO

DO j=1,N-1
   DO k=1,N
      A(N,j,k) = A(N,j,k) + B(N,j,k) * C(N,j)
   END DO
END DO
```

Can be fused again

interchange!

Case Study: interchange – step #2

```
DO i=1,N-1
   DO j=1,N
      DO k=1,N
         A(i,j,k) = A(i,j,k) + B(i,j,k) * C(i,j)
      END DO
   END DO
END DO
```

```
DO k=1,N
   A(N,N,k) = A(N,N,k) + B(N,N,k) * C(N,N)
   C(N,k) = C(k,N) + Cst1
END DO

DO k=1,N
   DO j=1,N-1
      A(N,j,k) = A(N,j,k) + B(N,j,k) * C(N,j)
   END DO
 END DO
```

Case Study: interchange – results

Timings in seconds on an R8K@90MHz:

Value of N	Data (MB)	Version	Elapsed time (seconds)	
50	1.0	original	0.0045	
		modified	0.0033	(1.4x)
200	61.0	original	13.9	
		modified	1.5	(9.3x)

Blocking for Cache

Blocking for cache is:

- *An optimization that applies for datasets that do not entirely fit in the data cache*
- *A way to increase temporal locality of reference i.e. improves data re–usage*
- *A way to increase spatial locality of reference i.e. exploit full cache lines*

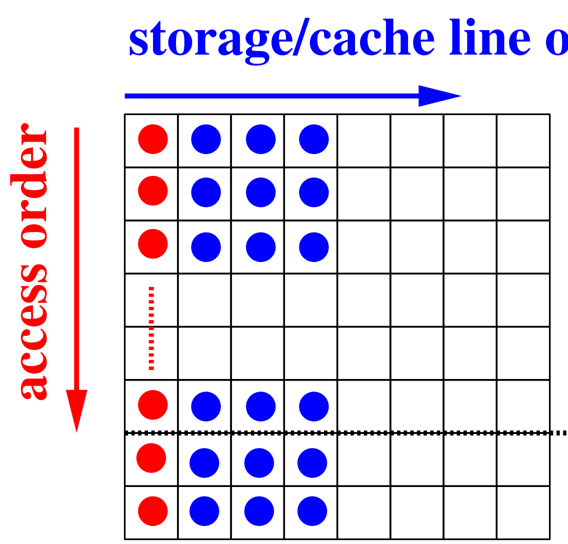
By way of example, we'll examine the following nested loop:

```
for (j=0; j<N; j++)
   for (i=0; i<N; i++)
        A[i][j] += B[i][j];
```

Blocking for Cache – Example

Suppose that N is such that: N*sizeof(datatype) > cachesize

```
for (j=0; j<N; j++)
     for (i=0; i<N; i++)
          A[i][j] += B[i][j];
```



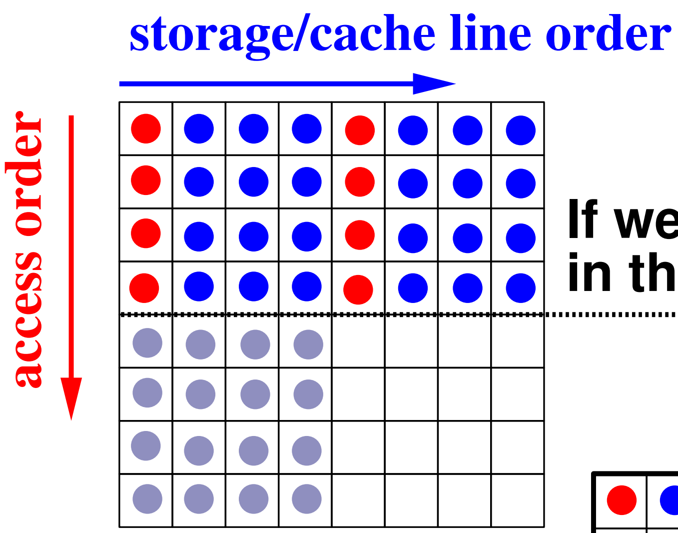
By this time, the cache is full, and so

... this will wipe out ● elements that are needed later on

Side-note:
If N is small enough there won't be a problem!

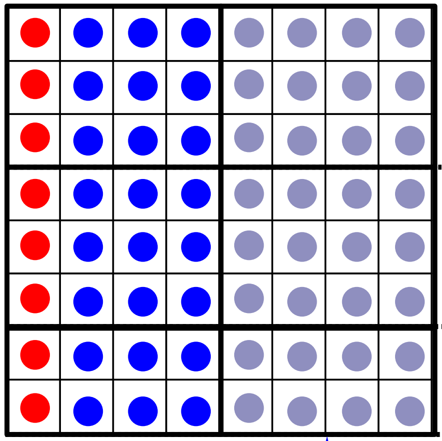
Blocking for Cache – Example

```
for (j=0; j<N; j++)
    for (i=0; i<N; i++)
        A[i][j] += B[i][j];
```



If we stop here, the ● elements are still in the cache !

Blocking:



We do the work on this block first

then we move on to the second block

and a leftover part

Repeat this process for the remaining part

Blocking for Cache

The example just shown can of course be solved in other, more simple ways

BUT .....

- **It can be very tedious/impossible to:**
  - **Just revert dimensions of arrays**
  - **Interchange loops**
- **There can also be a fundamental problem, like this:**

a[i][j] = b[j][i];

Blocking for Cache

Blocking is very powerful and simple to do (local impact only)

BUT .....

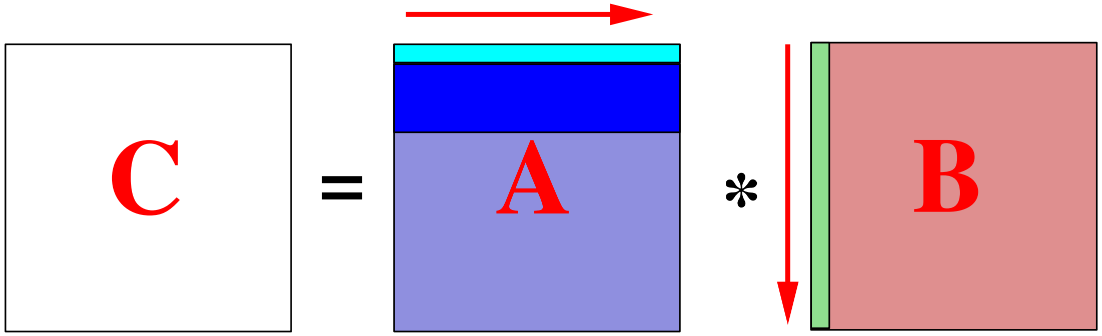
The resulting code does not get any prettier ....

```
DO i = 1, N
   ...i...
```

```
DO i1 = 1, N, Nblocking
   DO i2 = 0, min(N-i1+1, Nblocking)-1
      ...i = i1 + i2...
```

Blocking for Cache – Another example

*Matrix Multiply: C = A * B*



For reasonably large matrices A, B, and C, cache contention will occur

Blocking for Cache – Another example

*Matrix Multiply: C = A * B*

$$\begin{bmatrix} C11 & C12 \\ C21 & C22 \end{bmatrix} = \begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix} * \begin{bmatrix} B11 & B12 \\ B21 & B22 \end{bmatrix}$$

(C11 = A11*B11 + A12*B21)

With the appropriate blocking, A11 and A12 can be kept in the cache and are re used to compute C12

Blocking for Cache

Rule of thumb

Number of loops is greater than dimensions:

potential for blocking

Procedure Inlining

Inlining

Replacement of a function/subroutine call by the source code

Advantages

- Reduce the calling overhead (usually minor effect)
- Increase the opportunities for CPU optimisations (SWP)
- Increase auto–parallelisation opportunities

Procedure Inlining

Candidates for inlining are modules that:

- **Are "small"** and
- **Are called very often** and
- **Do not take much time per call**

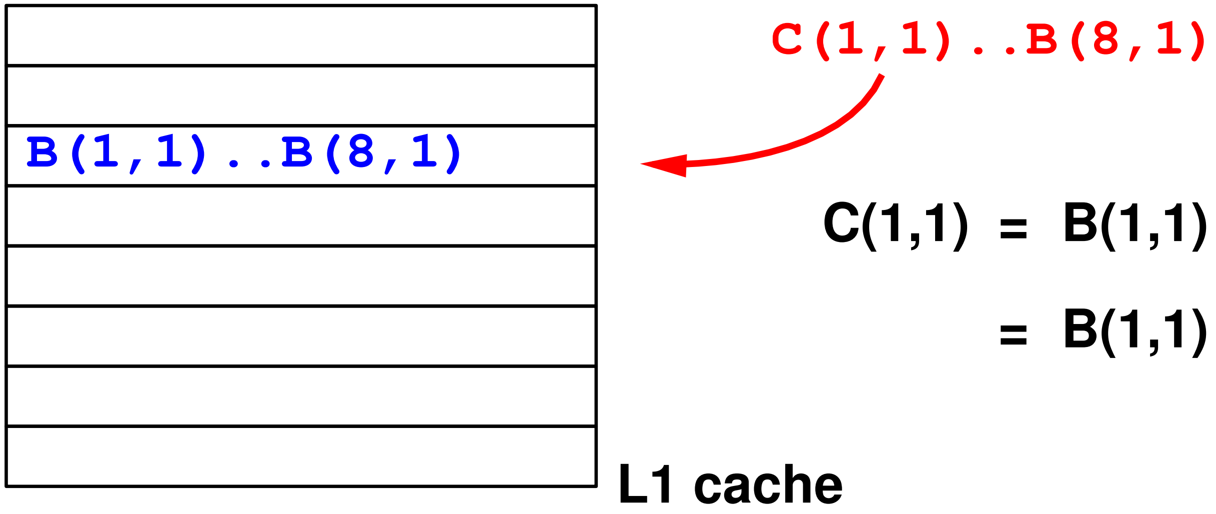
A typical example:

```
Do  i = 1, N
     A(i) = B(i) * 2.0
     Call Do_Work(A(i),C(i))
End Do

Subroutine Do_Work(X,Y)
Y = 1 + X * (1.0 + X*0.5)
```

Array Padding – Example

Out of the 3 references A(i,j), B(i,j) and C(i,j), two will map on the same cache line. This will cause cache trashing:

```
COMMON /SHARED/A(1024,1024),
               B(1024,1024),
               C(1024,1024)
DO j = 1, 1024
   DO i = 1, 1024
      A(i,j) = a(i,j) + b(i,j)*c(i,j)
   END DO
END DO
```



C(1,1) = B(1,1) + 1024*1024*4

= B(1,1) mod 32KB

Array Padding – Example

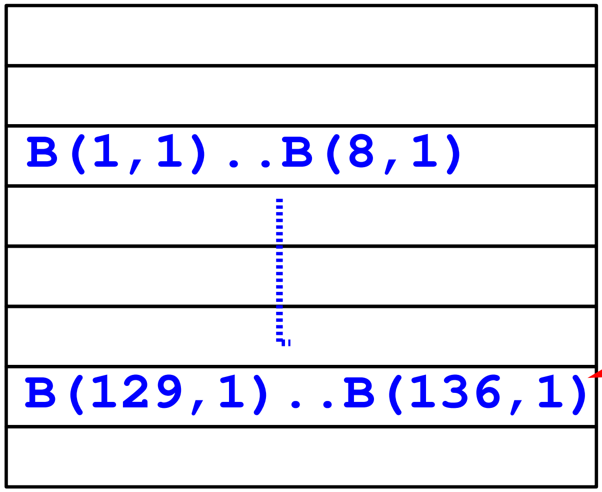
```
COMMON /SHARED/A(1024,1024),PAD1(129),
               B(1024,1024),PAD2(129),
               C(1024,1024)
DO j = 1, 1024
   DO i = 1, 1024
      A(i,j) = a(i,j) + b(i,j)*c(i,j)
   END DO
END DO
```

Padding

Padding will cause cache lines to be placed onto different cache locations

The SGI compiler will try to do padding for you automatically

C(1,1)..B(8,1)



L1 cache

C(1,1) = B(1,1) + 1024*1024*4 + 129*4

= B(129,1) mod 32KB

Serial Optimisations

Tuning Techniques Summary

- **Optimise memory access ("stride 1"):**
  - Interchange
- **Maximize data usage:**
  - Unrolling, fusion, fission, blocking
- **Improved instruction scheduling:**
  - Unrolling, fission, inlining

In realistic situations, a combination of several of these techniques may have to be applied for optimal performance

Compilers for RISC systems

Technique	Instruction	Memory
Software Pipelining	++	--
Loop Unrolling	++	+
Loop Fusion	+	++
Loop Fission	+	++
Loop Interchange	-	++
Loop Blocking	-	++
Procedure Inlining	++	++

Case Study

Case Study: Matrix Multiply

```
DO i=1,m
    DO j=1,n
       DO k=1,p
          C(i,j)=C(i,j)+A(i,k)* B(k,j)
       END DO
    END DO
 END DO
```

Diagnosis:

- *C(i,j) is loop invariant and will be hoisted/sunk out of the loop*
- *Memory access on A is bad, but on B it is good*

Case Study: Matrix Multiply–Opt #1

```
DO i=1,m
     DO j=1,n
          DO k=1,p
               C(i,j)=C(i,j)+A(i,k)* B(k,j)
          END DO
     END DO
END DO
```

Interchange
+
Scalar move

```
DO j=1,n
     DO k=1,p
          t = B(k,j)
          DO i=1,m
               C(i,j)=C(i,j)+A(i,k)* t
          END DO
     END DO
END DO
```

Case Study: Matrix Multiply–Opt #2

```
DO j=1,n
    DO k=1,p
       t = B(k,j)
       DO i=1,m
          C(i,j)=C(i,j)+A(i,k)* t
       END DO
    END DO
END DO
```

major advantage:
re-use of t

inner loop unrolling

```
DO j=1,n                                   nb=m/4
    DO k=1,p
       t=B(k,j)
       DO i=1,4*nb, 4
          C(i  ,j)=C(i  ,j)+A(i  ,k)*t
          C(i+1,j)=C(i+1,j)+A(i+1,k)*t
          C(i+2,j)=C(i+2,j)+A(i+2,k)*t
          C(i+3,j)=C(i+3,j)+A(i+3,k)*t
       END DO
       DO i=4*nb+1,m
          C(i,j)=C(i,j)+A(i,k)* t
       END DO
    END DO
END DO
```

cleanup loop

Case Study: Matrix Multiply–Opt #3

```
DO j=1,n
    DO k=1,p
       t = B(k,j)
       DO i=1,m
          C(i,j)=C(i,j)+A(i,k)* t
       END DO
    END DO
 END DO
```

major advantage:
less load/stores on C(i,j)

middle loop unrolling

nb=np/4

```
DO j=1,n
   DO k=1,4*nb, 4
       t0=B(k  ,j); t1=B(k+1,j)
       t2=B(k+2,j); t3=B(k+3,j)
       DO i=1,m
          C(i,j)=C(i,j)+ A(i,k)   * t0
                       + A(i,k+1) * t1
                       + A(i,k+2) * t2
                       + A(i,k+3) * t3
       END DO
    END DO
    DO k=4*nb+1,p
       t=B(k,j)
       DO i = 1,m
          C(i,j)=C(i,j)+A(i,k)* t
       END DO
    END DO
 END DO
```

cleanup loop

Case Study: Matrix Multiply–Opt #4

```
DO j=1,n
    DO k=1,p
       t = B(k,j)
       DO i=1,m
          C(i,j)=C(i,j)+A(i,k)* t
       END DO
    END DO
 END DO
```

major advantage:
re-use of A(i,k)
(plus t0, t1, t2 and t3)

outer loop unrolling

nb=n/4

```
DO j=1,4*nb, 4
   DO k=1,p
       t0=B(k,j);t1=B(k,j+1)
       t2=B(k,j+2);t3=B(k,j+3)
       DO i=1,m
          C(i,j  )=C(i,j  )+ A(i,k) * t0
          C(i,j+1)=C(i,j+1)+ A(i,k) * t1
          C(i,j+2)=C(i,j+2)+ A(i,k) * t2
          C(i,j+3)=C(i,j+3)+ A(i,k) * t3
       END DO
    END DO
END DO
DO j=4*nb+1,n
   DO k=1,p
      t=B(k,j)
      DO i = 1,m
         C(i,j)=C(i,j)+A(i,k)* t
      END DO
   END DO
 END DO
```

cleanup loop

Case Study: Matrix Multiply–Opt #5

```
DO j=1,n
    DO k=1,p
       t = B(k,j)
       DO i=1,m
          C(i,j)=C(i,j)+A(i,k)* t
       END DO
    END DO
 END DO
```

major advantage:
re-use of A(i,k) for every value of j

blocking

```
DO j1 = 1,n,Nbj
   DO k1 = 1,p, Nbk
      DO i1 = 1, m, Nbi

         DO k2 = 0, min(p-k1+1,Nbk)-1
            DO j2 = 0, min(n-j1+1,Nbj)-1
               k = k1+k2
               j = j1+j2
               t = B(k,j)
               DO i2 = 0, min(m-i1+1,Nbi)-1
                  i = i1 + i2
                  C(i,j)=C(i,j)+ A(i,k)*t
               END DO
            END DO
         END DO

      END DO
   END DO
END DO
```

Original loop structure again

Case Study: Matrix Multiply

The optimal blocking factor is a function of:

- Loop length
- Number of arrays involved and their datatypes

In our example:

- 3 NBxNB matrices A, B and C
- Each matrix element is double precision and so takes 8 bytes
- Condition for NB: 3*NB*NB*8 < Cache_Size_In_Bytes
- Therefore: **NB < SQRT(Cache_Size_In_Bytes/24)**
- Round down to ~10% less (to store the scalar variables)

Case Study: Matrix Multiply

We timed the following versions of the matrix multiply code:

1. original code
2. loop reversal and loop constant
3. loop reversal and inner loop unrolling
4. loop reversal and middle loop unrolling
5. loop reversal and outer loop unrolling
6. loop reverval and middle and outer loop unrolling
7. loop reversal and loop blocking
8. loop reversal and middle and outer unrolling with blocking

Case Study: Matrix Multiply

Compiler: 7.0 Options –O2 –64 –mips4

R10000@195MHz 2MB cache	–SWP:=ON	R8000@90MHz 4MB cache	–SWP:=ON	
156.0	88.3	108.1	108.0	Original code
84.4	87.0	85.2	52.2	Loop reversal and loop constant
82.7	87.3	63.5	70.8	LR and innerloop unrolling
56.5	60.0	62.1	39.5	LR and middle loop unrolling
37.7	37.4	119.1	30.5	LR and outer loop unrolling
35.2	35.1	92.9	24.5	LR and middle+outer loop unrolling
32.6	31.9	68.2	31.8	LR and loop blocking
15.3	14.0	87.5	15.0	LR and middle+outer loop unrolling + blocking
15.1				–O3 option on original code

The Scientific Libraries

SGI provides a tuned library for the most popular intrinsic functions (sin, cos, exp, log, ...)

Two libraries are available:

1. Link with –lfastm instead of using –lm
speed–up without any modifications

2. Use the vectorized intrinsics ("man math") in libm
higher speed–up, but possibly needs code modifications

Syntax of vector intrinsics:

```
v"intrinsic"[f] ("type" *x,
                 "type' *y,
                 long length,
                 long incx,
                 long incy)
```

Vector Intrinsics

The supported vector intrinsics are:

```
vacos(  double *x, double *y, long count, long stridex, long stridey )
vasin(  double *x, double *y, long count, long stridex, long stridey )
vatan(  double *x, double *y, long count, long stridex, long stridey )
vcos (  double *x, double *y, long count, long stridex, long stridey )
vexp (  double *x, double *y, long count, long stridex, long stridey )
vlog (  double *x, double *y, long count, long stridex, long stridey )
vsin (  double *x, double *y, long count, long stridex, long stridey )
vsqrt(  double *x, double *y, long count, long stridex, long stridey )
vtan (  double *x, double *y, long count, long stridex, long stridey )
```

Comments:

- **The single precision versions have an "f" added to their name e.g. vacos –> vacosf**
- **Length and strides need to be 64 bit integers**
- **The compiler tries to generate calls to these functions (at –O3)**

Vector Intrinsics – Example

```
subroutine vintr(n, x, y)
implicit none

integer i, n
real x(n), y(n)

do i = 1, n
   x(i) = i
   y(i) = sin(x(i))
end do

return
end
```

f77 –c –64 –O2 vintr.f

```
SUBROUTINE vintr(n, x, y)
IMPLICIT NONE
INTEGER*4 n
REAL*4 x(n)
REAL*4 y(n)

**** Variables and functions ****

INTEGER*4 i

**** Temporary variables ****

INTEGER*4 preg

**** statements ****

preg = n
DO i = 1, preg, 1
  x(i) = REAL(i)
  y(i) = SIN(x(i))
END DO
RETURN
END ! vintr
```

Vector Intrinsics – Example

```
subroutine vintr(n, x, y)
implicit none

integer i, n
real x(n), y(n)

do i = 1, n
   x(i) = i
   y(i) = sin(x(i))
end do

return
end
```

f77 –c –64 –O3 vintr.f

```
SUBROUTINE vintr(n, x, y)
IMPLICIT NONE
INTEGER*4 n
REAL*4 x(n)
REAL*4 y(n)

**** Temporary variables ****

INTEGER*4 i0

**** statements ****

DO i0 = 1, n, 1
  x(i0) = REAL(i0)
END DO
CALL vsinf$(x(1),y(1),%val(n),%val(1_8),%val(1_8))
RETURN
END ! vintr
```

compiler splitted the loop first and then generated the call

The Cray/SGI Scientific Library

SGI also provides a tuned Scientific Library:

1. ***Basic Linear Algebra Operations and Algorithms***
   BLAS1, BLAS2, BLAS3, LAPACK, LINPACK (man blas, man lapack)

2. ***FFTs and Convolutions***
   A variety in functionality is available (1D,2D,3D,..)
   (man fft, man conv)

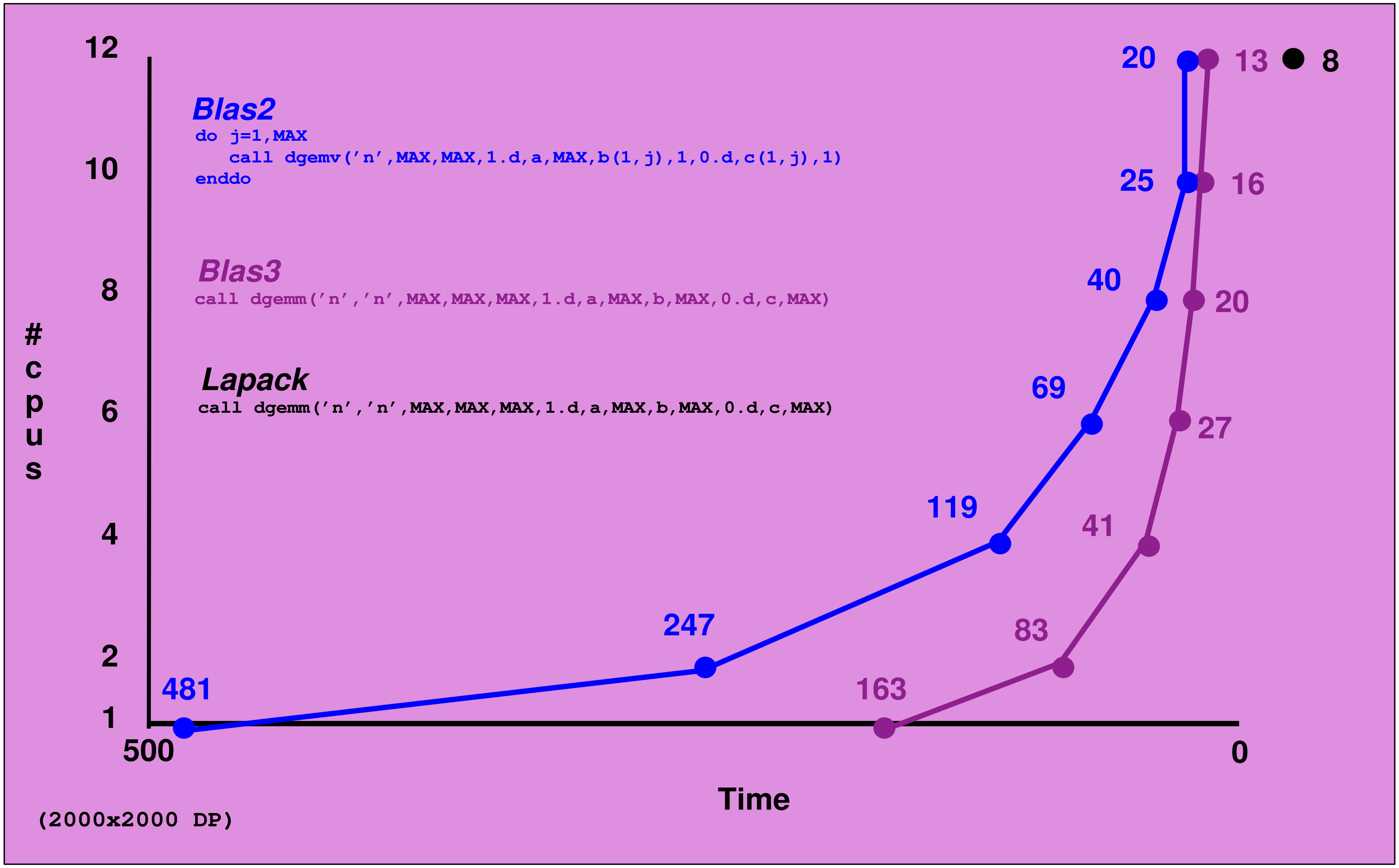
3. ***Sparse Solvers***
   Two specific direct solvers are available (man solvers)

Notes:

- Link with -lcomplib.sgimath to get the sequential version
- Where relevant, routines have also been parallelised (link with -lcomplib.sgimath_mp)
- The new name will be SCSL (=SGI Cray Scientific Library)

The Cray/SGI Scientific Library

Matrix Multiplication – BLAS2/BLAS3/LAPACK



I/O Optimisation

Implicit I/O

- Add "mmap"
- "read"/"write" Disappear
- Reduces Overhead
- Still Operates On File

I/O Optimisation

```
float f1;
int i,fd;
fd = open("file",O_RDWR,0644);
for (i=1;i!=N;i++) {
      read(fd,&f1,sizeof(f1));
      f1 +=  2.0;
      lseek(fd,-1*sizeof(f1),
           SEEK_CUR);
      write(fd,&f1,sizeof(f1));
 }
```

Using MMAP

```
float *f1;
int i,fd;
fd = open("file",O_RDWR,0644);

f1 = mmap(0,N*sizeof(*f1),PROT_READ|PROT_WRITE,
      MAP_PRIVATE,fd,0);
for (i=1;i!=N;i++)
      f1[i] += 2.0;
```

Case Study

Case Study: Vector update

```
ll=0
do jj=1,nj
   do ii=1,ni
      ll=ll+1
      res=0
      do n=1,nib
         na=ii+(n-1)*nra+(i-1)*nru+(l-1)*nra*nrub
         nb=n+(jj-1)*nib
         ndb1=nmb1/2
         naa1=nma1+na
         nbb1=ndb1+nb
         res=res+p(naa1)*dp(nbb1)
      end do
      nde1=nme1/2
      lle1=nde1+ll
      dp(lle1)=dp(lle1)+res
   end do
end do
```

Profiling tells us that we spend all of our time in this part

This is the net result of all the computations

Case Study: Vector update – Step 1

```
do n=1,nib
   na=ii+(n-1)*nra+(i-1)*nru+(l-1)*nra*nrub
   nb=n+(jj-1)*nib
   ndb1=nmb1/2
   naa1=nma1+na
   nbb1=ndb1+nb
   res=res+p(naa1)*dp(nbb1)
end do
```

Stride on array P controlled by naa1:

naa1 = nma1 + ii + (n–1)*nra + (i–1)*nru + (l–1)*nra*nrub

stride = nra

Stride on array DP controlled by nbb1:

nbb1 = nbd1 + n + (jj–1)*nib

stride = 1

Case Study: Vector update

Inner loop over:	n	ii	jj
Stride on P:	nra	1	0
Stride on DP:	1	0	nib

```
...
do jj=1,nj
   do ii=1,ni
      ...
      do n=1,nib
         na=ii+(n-1)*nra+(i-1)*nru+(l-1)*nra*nrub
         nb=n+(jj-1)*nib
         ndb1=nmb1/2
         naa1=nma1+na
         nbb1=ndb1+nb
         res=res+p(naa1)*dp(nbb1)
      end do
      .....
   end do
end do
```

Note: nra and nib are both ~5000

Case Study: Vector update

- It is now clear what needs to be done:

  we need to move the loop over ii inward

- The question is:

  how do we do that?

- A useful first step is to *simplify the indexing*
- We do this by eliminating temporary variables
- This will initially make the code to look more complicated (...), but the compiler will fix this for us

Case Study: Vector update – Step 2

```
do jj=1,nj
   do ii=1,ni
      res=0
      do n=1,nib
         ndb1=nmb1/2
         naa1=nma1+ii+(n-1)*nra+(i-1)*nru+(l-1)*nra*nrub
         nbb1=ndb1+n+(jj-1)*nib
         res=res+p(naa1)*dp(nbb1)
      end do
      nde1=nme1/2
      lle1=nde1+ii+(jj-1)*ni
      dp(lle1)=dp(lle1)+res
   end do
end do
```

Expanded NA

Expanded NB

Eliminated LL

- *We are now almost ready to interchange the loops*
- *Statement* `res=0` *is the only obstacle left*
- *This can be fixed easily by substition*

Case Study: Vector update – Step 3

```
do jj=1,nj
   do ii=1,ni
      do n=1,nib
        ndb1=nmb1/2
        naa1=nma1+ii+(n-1)*nra+(i-1)*nru+(l-1)*nra*nrub
        nbb1=ndb1+n+(jj-1)*nib
        nde1=nme1/2
        lle1=nde1+ii+(jj-1)*ni
        dp(lle1)=dp(lle1)+p(naa1)*dp(nbb1)
      end do
   end do
end do
```

Can now be interchanged

Substituted the expressions

Note:

We can now also move some constants out of the loop again (the compiler will do this too ...)

Case Study: Vector update – Step 4

```
ndb1=nmb1/2
nde1=nme1/2
do jj=1,nj
   do n=1,nib
      do ii=1,ni
       naa1=nma1+ii+(n-1)*nra+(i-1)*nru+(l-1)*nra*nrub
       nbb1=ndb1+n+(jj-1)*nib
       lle1=nde1+ii+(jj-1)*ni
       dp(lle1)=dp(lle1)+p(naa1)*dp(nbb1)
    end do
  end do
end do
```

Have been interchanged

stride=1

stride=0!

Note:

Next we simplify the index calculations in the innermost loop

Case Study: Vector update – Step 5

```
ndb1=nmb1/2
nde1=nme1/2
do jj=1,nj
   do n=1,nib
      naa1=nma1+(n-1)*nra+(i-1)*nru+(l-1)*nra*nrub
      dp_temp=dp(ndb1+n+(jj-1)*nib)
      lle1=nde1+(jj-1)*ni
      do ii=1,ni
         dp(lle1+ii)=dp(lle1+ii)+p(naa1+ii)*dp_temp
      end do
   end do
end do
```

Indexing can be simplified more

Note:

We can simplify the index calculations even further

Case Study: Vector update – Step 6

```
ndb1=nmb1/2
nde1=nme1/2
id1 =nma1+(i-1)*nru+(l-1)*nra*nrub
do jj=1,nj
   id2 = ndb1+(jj-1)*nib
   lle1= nde1+(jj-1)*ni
   id3 = id1
   do n=1,nib
      dp_temp=dp(id2+n)

      do ii=1,ni
         dp(lle1+ii)=dp(lle1+ii)+p(id3+ii)*dp_temp
      end do

      id3 = id3 + nra
  end do
end do
```

This is a DAXPY operation !

Case Study: Vector update

```
do ii=1,ni
   dp(lle1+ii)=dp(lle1+ii)+p(id3+ii)*dp_temp
end do
```

is equivalent to:

```
CALL DAXPY(ni, dp_temp, p(id3), 1, dp(lle1),1 )
```

```
%man daxpy

subroutine daxpy( n, a, x, incx, y, incy )
    integer           incx, incy, n
    double precision  a
    double precision  x( 1 ), y( 1 )

                              ......
DESCRIPTION
    DAXPY, SAXPY, ZAXPY and CAXPY compute a constant A times a vector X plus
    a vector Y. The result ovewrites the initial values of vector Y.
        Y <--- Y + a . X
    INCX and INCY specify the increment between two consecutive elements of
    respectively vector X and Y.
```

Case Study: Vector update

The code is now in such a shape that we might consider:

- Outer loop unrolling
- Blocking

This is similar to the matrix multiply case

Well, okay, let's have a look then !

Case Study: Vector update

```
      subroutine opt2( ni,nj,nib,nra,nru,nrub,
     2                 nma1,nmb1,nme1,i,l,dp,p )
      implicit none
      integer ni,nj,nib,nra,nru,nrub,nma1,nmb1,nme1,i,l
      real    dp(*),p(*)
c
      integer ii,jj,ndb1,nde1,lle1,n
      real    dp_temp
      integer id1,id2,id3
c
      ndb1=nmb1/2
      nde1=nme1/2
      id1 =nma1+(i-1)*nru+(l-1)*nra*nrub
c
      call opt2a(ni,nj,nib,nra,nru,nrub,
    2           nma1,nmb1,nme1,i,l,dp(nde1),
    3           dp(ndb1),p(id1))

      return
      end
```

Case Study: Vector update

```
      subroutine opt2a( ni,nj,nib,nra,nru,nrub,
     2                  nma1,nmb1,nme1,i,l,dp1,dp2,p )
      implicit none
      integer ni,nj,nib,nra,nru,nrub,nma1,nmb1,nme1,i,l
      real    dp1(ni,*),dp2(nib,*),p(nra,*)
c
      integer ii,jj,ndb1,nde1,lle1,n
      real    dp_temp
c
      do jj=1,nj
         do n=1,nib
            dp_temp=dp2(n,jj)
            do ii=1,ni
               dp1(ii,jj)=dp1(ii,jj)+p(ii,n)*dp_temp
            end do
         end do
      end do
      return
      end
```

Arrays are 2D to allow auto blocking by the compiler !

Case Study: Vector update

```
DO tile2jj = 1, nj, 126
  DO tile1ii = 1, ni, 544
    DO n = 1, (nib + -3), 4
      DO jj = tile2jj, MIN((nj + -1), (tile2jj + 124)), 2
        mi0 = dp2(n, jj)
        mi1 = dp2(n + 3, jj + 1)
        mi2 = dp2(n + 2, jj + 1)
        mi3 = dp2(n + 1, jj + 1)
        mi4 = dp2(n, jj + 1)
        mi5 = dp2(n + 1, jj)
        mi6 = dp2(n + 2, jj)
        mi7 = dp2(n + 3, jj)
        DO ii = tile1ii, MIN((tile1ii + 543), ni), 1
          dp1(ii, jj) = (dp1(ii, jj) +(p(ii, n) * mi0))
          dp1(ii, jj) = (dp1(ii, jj) +(p(ii, n + 1) * mi5))
          dp1(ii, jj) = (dp1(ii, jj) +(p(ii, n + 2) * mi6))
          dp1(ii, jj) = (dp1(ii, jj) +(p(ii, n + 3) * mi7))
          dp1(ii, jj + 1) = (dp1(ii, jj + 1) +(p(ii, n) * mi4))
          dp1(ii, jj + 1) = (dp1(ii, jj + 1) +(p(ii, n + 1) * mi3))
          dp1(ii, jj + 1) = (dp1(ii, jj + 1) +(p(ii, n + 2) * mi2))
          dp1(ii, jj + 1) = (dp1(ii, jj + 1) +(p(ii, n + 3) * mi1))
        END DO
      END DO
      DO wd_jj0 = jj, MIN((tile2jj + 125), nj), 1
        mi8 = dp2(n, wd_jj0)
        mi9 = dp2(n + 1, wd_jj0)
        mi10 = dp2(n + 3, wd_jj0)
        mi11 = dp2(n + 2, wd_jj0)
```

Case Study: Vector update

```
          DO ii0 = tile1ii, MIN((tile1ii + 543), ni), 1
            dp1(ii0, wd_jj0) = (dp1(ii0, wd_jj0) +(p(ii0, n) * mi8))
            dp1(ii0, wd_jj0) = (dp1(ii0, wd_jj0) +(p(ii0, n + 1) * mi9))
            dp1(ii0, wd_jj0) = (dp1(ii0, wd_jj0) +(p(ii0, n + 2) * mi11))
            dp1(ii0, wd_jj0) = (dp1(ii0, wd_jj0) +(p(ii0, n + 3) * mi10))
          END DO
        END DO
      END DO
      DO wd_n = n, nib, 1
        DO jj0 = tile2jj, MIN((nj + -1), (tile2jj + 124)), 2
          mi12 = dp2(wd_n, jj0)
          mi13 = dp2(wd_n, jj0 + 1)
          DO ii1 = tile1ii, MIN((tile1ii + 543), ni), 1
            dp1(ii1, jj0) = (dp1(ii1, jj0) +(p(ii1, wd_n) * mi12))
            dp1(ii1, jj0 + 1) = (dp1(ii1, jj0 + 1) +(p(ii1, wd_n) * mi13))
          END DO
        END DO
        DO wd_jj = jj0, MIN((tile2jj + 125), nj), 1
          mi14 = dp2(wd_n, wd_jj)
          DO ii2 = tile1ii, MIN((tile1ii + 543), ni), 1
            dp1(ii2, wd_jj) = (dp1(ii2, wd_jj) +(p(ii2, wd_n) * mi14))
          END DO
        END DO
      END DO
    END DO
  END DO
```

Case Study: Vector update

The program has been run for 3 implementations:

original – as presented initially
opt1 – daxpy formulation, one dimensional version
opt2 – daxpy formulation, two dimensional version

Timings in seconds on an Origin2000@195MHz (nra=nib=5000)

```
% ./a.out
    15.34893     seconds (original)
     1.98022     seconds (opt1)
     0.83715     seconds (opt2)
%
```

Case Study: Vector update

After all these manual optimisations, one might wonder what the compiler does to the un–modified code

Case Study: Vector update

```
      ll = 0
      preg1 = nj
      DO jj = 1, preg1, 1
        preg = ni
        DO ii = 1, preg, 1
          ll = (ll + 1)
          res = 0.0
          preg0 = nib
          DO n = 1, preg0, 1
            na = (((((n + -1) * nra) + ii) +((i + -1)
    &                * nru)) +(((l + -1) * nra) * nrub))
            nb = (((jj + -1) * nib) + n)
            ndb1 = (nmb1 / 2)
            naa1 = (nma1 + na)
            nbb1 = (nb + ndb1)
            res = ((p(naa1) * dp(nbb1)) + res)
          END DO
          nde1 = (nme1 / 2)
          lle1 = (nde1 + ll)
          dp(lle1) = (dp(lle1) + res)
        END DO
      END DO
```

Compiled with option –O2:

Nearly identical to original source

Case Study: Vector update

```
      ll = 0
      DO jj = 1, nj, 1
        preg = ll
        DO ii = 1, ni, 1
          res = 0.0
          DO n = 1, nib, 1
            ndb1 = (nmb1 / 2)
            res = ((p(((((((n + -1) * nra) + ii) +
     &            ((i + -1) * nru)) +(((l + -1) * nra)
     &            * nrub)) + nma1)) *
     &            dp(((((jj + -1) * nib) + n)
     &            + ndb1))) + res)
          END DO
          nde1 = (nme1 / 2)
          dp(((ii + preg) + nde1)) =
     &                 (dp(((ii + preg) + nde1)) + res)
        END DO
        ll = ((ii + preg) + -1)
      END DO
```

Compiled with option –O3:

Still much less than the hand–tuned version

Case Study: Vector update

Software pipelining with –O3:

```
#<swps>
#<swps> Pipelined loop line 17 steady state
#<swps>
#<swps>  25 estimated iterations before pipelining
#<swps>   4 unrollings before pipelining
#<swps>   8 cycles per 4 iterations
#<swps>   8 flops         ( 50% of peak) (madds count as 2)
#<swps>   4 flops         ( 25% of peak) (madds count as 1)
#<swps>   4 madds         ( 50% of peak)
#<swps>   8 mem refs      (100% of peak)
#<swps>   6 integer ops   ( 37% of peak)
#<swps>  18 instructions ( 56% of peak)
#<swps>   1 short trip threshold
#<swps>   8 integer registers used.
#<swps>   9 float registers used.
#<swps>
```

RISC Architectures and Optimisation

Conclusions

- *The memory/cache is a crucial component*
- *For optimal performance pay attention to the coding style (language dependent)*
- *Compilers are good at instruction level optimisations*
- *Memory optimisations are performed, but often benefit from high level insight*
- *Pays–off to look at the datastructure details*