Improving node-level efficiency

Single-core performance analysis

- **Are the hotspot routines compute-bound or memory-bound?**
  - If computational intensity > 1.0 the routine is compute-bound, otherwise memory-bound
- **Signature: low L1 and/or L2 cache hit ratios**
  - <96% for L1, <99% for L1+L2
  - Issue: Bad cache alignment
- **Signature: low vector instruction usage**
  - Issue: Non-vectorizable (hotspot) loops
- **Signature: traced ”math” group featuring a significant portion in the profile**
  - Issue: Expensive operations

Doesn't the compiler do everything?

- **Not yet...**
  - Standard answer, unchanged for the last 50 years or so
- **You can make a big difference to code performance**
  - Helping the compiler spot optimisation opportunities
  - Using the insight of your application
  - Removing obscure (and obsolescent) “optimisations” in older code
    - Simple code is the best, until otherwise proven
- **No fixed rules: optimize on case-by-case basis**
  - But first, check what the compiler is already doing

Compiler feedback/output

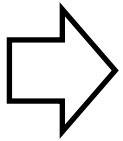
- **Cray compiler:** `ftn -rm …` **or** `cc/CC -hlist=m …`
  - Compiler generates an <source file name>.lst file that contains annotated listing of your source code
- **Intel compiler:** `ftn/cc -opt-report 3 -vec-report 6`
  - If you want this into a file: add `-opt-report-file=filename`
  - See `ifort --help reports`
- **GNU compiler:** `ftn/cc -ftree-vectorizer-verbose=6`

Issue: Bad cache alignment

- **If multi-dimensional arrays are addressed in a wrong order, it causes a lot of cache misses = bad performance**
  - C is row-major, Fortran column-major
  - A compiler may re-order loops automatically (see output)

```
real a(N,M)
real sum = 0;

do i=1,N
  do j=1,M
    sum = sum + a(i,j)
  end do
end do
```



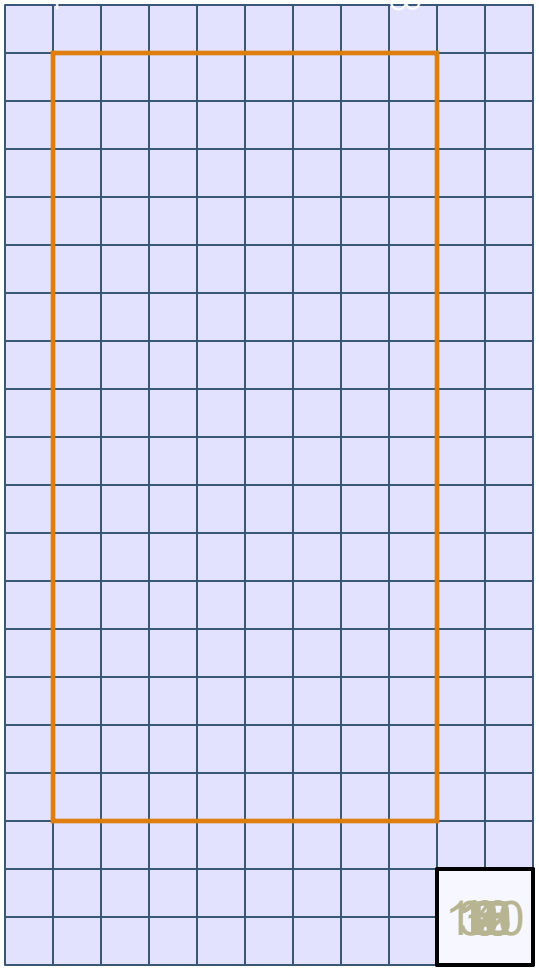
```
real a(N,M)
real sum = 0

do j=1,M
  do i=1,N
    sum = sum + a(i,j)
  end do
end do
```

Issue: Bad cache alignment

- **Loop blocking = Large loops are partitioned by hand such that the data in inner loops stays in caches**
  - A prime example is matrix-matrix multiply coding
- **Complicated optimization: optimal block size is a machine dependent factor as there is a strong connection to L1 and L2 cache sizes**
- **Some compilers do loop blocking automatically**
  - See the compiler output
  - You can assist it using compiler pragmas/directives

Cache Use in Stencil Computations

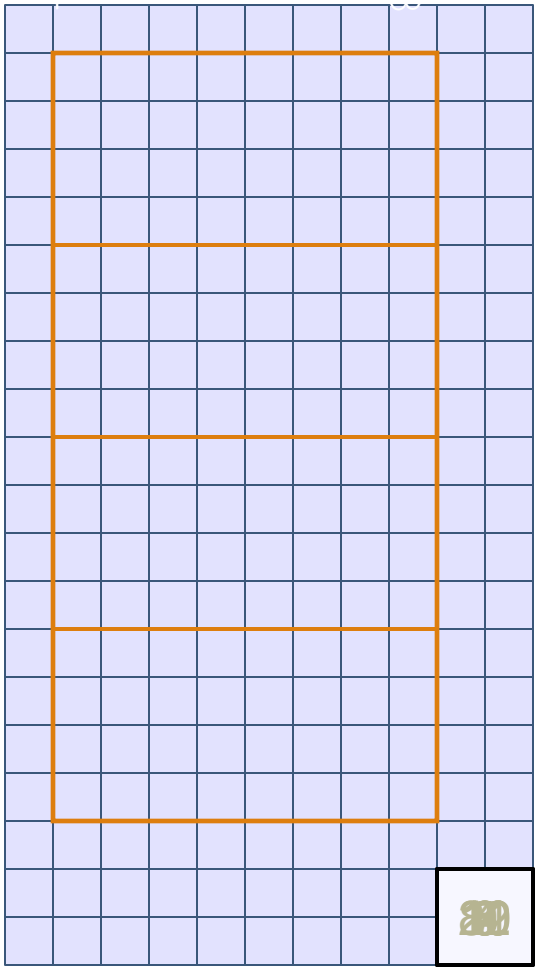


- 2D Laplacian

```
do j = 1, 8
   do i = 1, 16
      a(i,j) = u(i-1,j) + u(i+1,j)   &
           - 4*u(i,j)                &
           + u(i,j-1) + u(i,j+1)
   end do
end do
```

- Cache structure for this example:
  - Each line holds 4 array elements
  - Cache can hold 12 lines of `u` data
- No cache reuse between outer loop iterations

Blocking to Increase Reuse

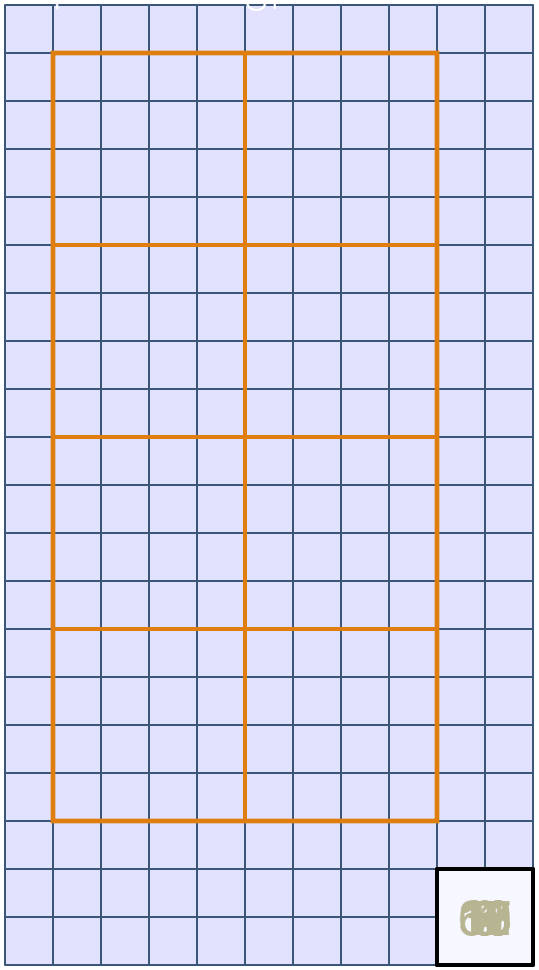


- Unblocked loop: 120 cache misses
- Block the inner loop

```
do IBLOCK = 1, 16, 4
   do j = 1, 8
      do i = IBLOCK, IBLOCK + 3
         a(i,j) = u(i-1,j) + u(i+1,j)   &
                  - 4*u(i,j)            &
                  + u(i,j-1) + u(i,j+1)
      end do
   end do
end do
```

- Now we have reuse of the "j+1" data

Blocking to Increase Reuse



- One-dimensional blocking reduced misses from 120 to 80
- Iterate over 4×4 blocks

```
do JBLOCK = 1, 8, 4
   do IBLOCK = 1, 16, 4
      do j = JBLOCK, JBLOCK + 3
         do i = IBLOCK, IBLOCK + 3
            a(i,j) = u(i-1,j) + u(i+1,j)   &
                     - 4*u(i,j)            &
                     + u(i,j-1) + u(i,j+1)
         end do
      end do
   end do
end do
```

- Better use of spatial locality (cache lines)

Issue: Bad cache alignment

Original loopnest	Blocking with compiler directives	Equivalent explicit code

do k = 6, nz-5
  do j = 6, ny-5
  do i = 6, nx-5
  ! *stencil*
  enddo
  enddo
enddo
	!dir$ blockable(j,k)
	!dir$ blockingsize(16)
	do k = 6, nz-5
	  do j = 6, ny-5
	  do i = 6, nx-5
	  ! *stencil*
	  enddo
	  enddo
	enddo
	
	C:
	#pragma blockable(2)
	#pragma blockingsize(16)
		do kb = 6,nz-5,16
		  do jb = 6,ny-5,16
		  do k = kb,MIN(kb+16-1,nz-5)
		  do j = jb,MIN(jb+16-1,ny-5)
		  do i = 6, nx-5
		  ! *stencil*
		  enddo
		  enddo
		  enddo
		  enddo
		enddo

Loop depth

Issue: Bad cache alignment

- Loop fusion: Useful when the same data is used e.g. in two separate loops: cache-line re-use

Original code	Complete fusion	Partial fusing

do j = 1, Nj
  do i = 1, Ni
  a(i,j)=b(i,j)*2
  enddo
enddo
 	
do j = 1, Nj
  do i = 1, Ni
  a(i,j)=a(i,j)+1
  enddo
enddo
	do j = 1, Nj
	  do i = 1, Ni
	  a(i,j)=b(i,j)*2
	  a(i,j)=a(i,j)+1
	  enddo
	enddo
		do j = 1, Nj
		  do i = 1, Ni
		  a(i,j)=b(i,j)*2
		  enddo
		  do i = 1, Ni
		  a(i,j)=a(i,j)+1
		  enddo
		enddo

What can be vectorized by the compiler?

- **Only loops can be vectorized**
- **Constant (unit) strides are best**
- **Indirect addressing will not vectorize (efficiently)**
- **Only inlined functions/procedures will be vectorized**
- **Needs to know loop tripcount (but only at runtime)**
  - i.e. DO WHILE style loops will not vectorize
- **No recursion allowed**
- **See compiler feedback on why some loops were not vectorized**
  - CCE: `-hlist=a`
  - Intel: `-vec-report[0..5]`
  - GNU: `-ftree-vectorizer-verbose=5`

Improving Vectorization

- **Does the non-vectorized loop have true dependencies?**
  - No: add the pragma/directive `ivdep` on top of the loop
  - Yes: Rewrite the loop
    - Convert loop scalars to vectors
    - Move if-statements out of the loop
- **Some compiler flags can encourage compiler to vectorize more aggressively**
  - CCE: -h restrict=a (C/C++ only)
  - Intel: -fno-alias
- **If you cannot vectorize the entire loop, consider splitting it - so as much of the loop is vectorized as possible**

Example: Vectorization

In the compiler listing File:

```
 1519.  + 1 2-----< loop2:  do ji = j0, j1
 1520.    1 2                 j = 1 + mod( ji - 1, N1 )
 1521.    1 2
                              …
 1526.    1 2                 RXij = RXi - RX2(j)

                                       … more lines like this …

 1532.    1 2                 PXij = PXij - anint( RXij )

                                       … more lines like this …

 1538.    1 2                 RijSquared = RXij**2 + RYij**2 + RZij**2
 1539.    1 2                 if( RijSquared >= RCutoffSquared ) cycle loop2
 1540.  + 1 2                 RijSquaredInv = SigmaSquared / RijSquared
                               …
```

caused by

```
ftn-6339 ftn: VECTOR File = ms2_potential.F90, Line = 1519
  A loop starting at line 1519 was not vectorized because of a potential hazard
  in conditional code on line 1555.
```

Example: Vectorization

```
Use explain to get the meaning of ftn-6339

explain ftn-6339

VECTOR:  A loop starting at line %s was not vectorized because of a potential
hazard in conditional code on line num.

Vectorization of conditional code on architectures that lack predicated
execution requires careful manipulation to avoid triggering floating point
exceptions or memory segmentation faults.  The line identified by this
message has one such hazard, and at present it inhibits vectorization of the
entire loop.

This restriction may be removed in a future compiler release.
```

This is better:

```
1529.  + M m 3-----< loop2:  do ji = j0, j1
1530.    M m 3                j = 1 + mod( ji - 1, N1 )
1537.    M m 3                RXij = RXi - RX2(j)
                                    …
1540.    M m 3                RXij = RXij - anint( RXij )
                                    …
1543.    M m 3                RijSquared = RXij**2 + RYij**2 + RZij**2
1544.    M m 3                if( RijSquared >= RCutoffSquared ) cycle loop2
1545.    M m 3                counter=counter+1
1546.    M m 3                map(counter)=j
1547.    M m 3------>         end do loop2
1548.    M m Vp----< loop2a: do k=1,counter
1549.    M m Vp               j=map(k)
1550.    M m Vp               RXij = RXi - RX2(j)

                              … more lines like this …

1556.    M m Vp               PXij = PXij - anint( RXij )

                              … more lines like this …

1562.    M m Vp               RijSquared = RXij**2 + RYij**2 + RZij**2
1563.    M m Vp      !mn         if( RijSquared >= RCutoffSquared ) cycle loop2
1564.  + M m Vp               RijSquaredInv = SigmaSquared / RijSquared
1565.    M m Vp               Rij6Inv = RijSquaredInv**3
                                …
```

Issue: Expensive operations

- Cost of different scalar FP operations is roughly as follows:

  ~1 cycle: +, *

  ~20 cycles: /, sqrt()

  ~100-300 cycles: sin, cos, exp, log, ...

- Note that there is also instruction latency and issues related to the pipelining

Issue: Expensive operations

- **Loop hoisting: try to get the expensive operations out of innermost loops**
- **Minimize the use of sin, cos, exp, log, pow, ...**
  - Consider precomputing values to lookup table
  - Use identities, e.g.
    - pow(x,2.5) = x*x*sqrt(x)
    - sin(x)*cos(x) = 0.5*sin(2*x)

Huge Pages

- **The Aries performs better with HUGE pages than with 4K pages. The Aries can map more pages using fewer resources meaning communications may be faster.**

- **Huge Pages will also affect TLB performance:**
  - Your code may run with fewer TLB misses (hence faster)
  - However, your code may load extra data and so run slower
  - Only way to know is by experimentation.

- **Use modules to change default page sizes (`man intro_hugepages`):**
  - `e.g. module load craype-hugepages#`
    - `craype-hugepages128K (not on XC30)`
    - `craype-hugepages512K (not on XC30)`
    - **`craype-hugepages2M`**
    - **`craype-hugepages8M`**
    - `craype-hugepages16M`
    - `craype-hugepages64M`

Most commonly successfully on Cray XC

Summary

- **Do not forget the algorithms**
- **Do the performance analysis!**
- **Determine the limiting resource**
  - Compute or memory bound (or something else like IO)
- **Vectorize, check compiler diagnostics**
- **Pay attention to the memory access**
  - Try to maximize the cache usage