Dangerously Clever X1
Application Tricks

CUG 2004
James B. White III (Trey)
trey@ornl.gov
Acknowledgement

Research sponsored by the Mathematical, Information, and Computational Sciences Division, Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.
Outline

• Outlandish optimizations that are universally applicable and guaranteed to enhance performance*
• Proof by anecdote
  – With slide after slide of source code
• * Fine print
Optimizations

• Avoid using cache
• Replace BLAS calls with do loops
• Minimize vector length
• Move if statements inside loops
• Use more pointers
• Add infinite loops
Avoid using cache

• Why?
  – No spatial locality of memory references
  – Avoiding cache can provide much higher bandwidth

• How?
  – !dir$ no_cache_alloc variable

• Example
  – Strided triad benchmark
Strided triad benchmark

real(8), allocatable :: a(:,), b(:,), c(:)
!dir$ no_cache_alloc a,b,c

...  
do stride = 1, 500  
   ...  
   do iter = 1, iters  
      ...  
      !dir$ unroll(8)  
a(:,:stride) = b(:,:stride) + s*c(:,:stride)  
   ...

...
BLAS calls → do loops

• Why?
  – Compiler can optimize special cases
  – BLAS calls aren’t yet inlined
  – BLAS calls aren’t highly optimized
    • True for fewer and fewer calls
• CUG 2003 example: CGEMM
  – BLAS improved, no longer an advantage
• Example: Benchmark loop over DGER
Benchmark loop over DGER

do iter = 1, niters
   call dger(n,n,alpha,x,1,y,1,a,n)
   do j = 1, n
      do i = 1, n
         a(i,j) = a(i,j) + alpha*x(i)*y(j)
      end do
   end do
end do
Benchmark loop over DGER

- N = 4480, nites = 100, one MSP
- DGER performance
  - 2.4 GF
  - 20% efficiency
- Do-loop performance
  - 138 GF
  - 1078% efficiency
Loopmarks?

\[ \text{Di--------< do iter} = 1, \text{niters} \]
\[ \text{Di Mr------< do j} = 1, n \]
\[ \text{Di Mr Vm--< do i} = 1, n \]
\[ \text{Di Mr Vm} \quad a(i,j) = \ldots \]
\[ \text{Di Mr Vm--> end do} \]
\[ \text{Di Mr------> end do} \]
\[ \text{Di--------> end do} \]
Loopmarks!

\[
\begin{align*}
&\text{Di} \quad \text{do iter} = 1, \text{niters} \\
&\text{Di Mr} < \quad \text{do } j = 1, n \\
&\text{Di Mr Vm} < \\
&\text{Di Mr Vm} \quad \text{do } i = 1, n \\
&\text{Di Mr Vm} \quad \text{a}(i,j) = \ldots \\
&\text{Di Mr Vm} \quad \text{end do} \\
&\text{Di Mr} \quad \text{end do} \\
&\text{Di} \quad \text{end do}
\end{align*}
\]
Loop over DGER?

do iter = 1, niters
  do j = 1, n
    do i = 1, n
      a(i,j) = a(i,j) + alpha*x(i)*y(j)
    end do
  end do
end do
No loop over DGER!

\[
\text{nalpha} = \text{niters} \times \alpha \\
\text{do } j = 1, n \\
\hspace{1cm} \text{do } i = 1, n \\
\hspace{2cm} a(i,j) = a(i,j) + \text{nalpha} \times x(i) \times y(j) \\
\hspace{1cm} \text{end do} \\
\text{end do}
\]
Minimize vector length*
* within reason

• Why?
  – shortloop - Loops disappear in favor of pure vector instructions
  – Cache and vector-register locality

• How?
  – Tile vector loops in blocks of 64
    • Multistream an outer loop
  – Look for “Vs” in loopmarks
  – Add “!dir$ shortloop” if compiler can’t tell

• Example: NAS FT C (5 directives)
NAS FT C Performance

Performance (MF) vs. MSPs

Max VL

VL = 64
Performance Improvement of VL=64

Relative performance vs. MSPs
Move if statements inside loops

• Do what?
  – Move if statements inside loops even though they are loop-independent

• Why?
  – Fuse loops together
  – Demote temporary arrays to scalars
  – Compiler promotes scalars to vector registers
  – Register blocking - reduce memory load/store

• Check loopmarks afterward for “V”

• Example: POP “state” subroutine
POP “state”, before

... if (present(RHOOUT)) then
    RHOOUT = merge(((unt0 + RHO0)*BULK_MOD*DENOMK)*p001, $
                    c0, KMT >= k)
 endif

if (present(RHOFULL)) then
    RHOFULL = merge(((unt0 + RHO0)*BULK_MOD*DENOMK)*p001, $
                     c0, KMT >= k)
 endif
...

POP “state”, after

\begin{verbatim}
do j = 1, jmt ; do i = 1, imt
  ...
  if (present(RHOOUT)) then
    RHOOUT(i,j) = merge($
                         ((unt0 + rho0)*bulk_mod*denomk)*p001,
                         $ c0, kmt_mask)
  endif
  if (present(RHOFULL)) then
    RHOFULL(i,j) = merge($
                         ((unt0 + rho0)*bulk_mod*denomk)*p001,
                         $ c0, kmt_mask)
  endif
  ...
end do; end do
\end{verbatim}
“state” performance

• POP benchmark run on one MSP
  – 320x384x40 grid points worldwide
• Results from “samp_cs_time”
• Before
  | 16.3% | 33.7% | 19545 | state@state_mod
• After
  | 13.5% | 31.7% | 15663 | state@state_mod
• 25% performance improvement in “state”
Use more pointers: Why?

- Communication optimization
- Replace MPI with direct load/store
- Co-Array Fortran requires non-local changes (arguments must be Co-Arrays)
- Use pointers to cheat
Use more pointers: How?

• Declare a co-array of INTEGER(8)
• Declare Cray pointer on receiver
• Sender stores array addresses in receiver co-array location (address from LOC)
• Receiver associates pointer with local value of integer co-array (assigned by sender)
• Receiver uses pointer to access sender data
• Example: Scatter from POP/CICE
POP/CICE scatter (sender)

```fortran
integer(8) :: remote_address(NPROC_X*NPROC_Y)[*]
real(dbl_kind) :: workg(imt_global,jmt_global)

integer(8) :: address

if (my_image == master_image) then
    address = loc(workg)
    do i = 1, num_images()
        remote_address(master_image)[i] = address
    end do
end if

... ! Synchronize
```
**POP/CICE scatter (receivers)**

```fortran
integer(8) :: remote_address(NPROC_X*NPROC_Y)[*]
real(dbl_kind) :: work(ilo:ihi,jlo:jhi)

real(dbl_kind) :: $ workg_remote(imt_global,jmt_global)
pointer(workg_address, workg_remote)

... ! Synchronize

workg_address = remote_address(master_image)
work(ilo:ihi,jlo:jhi) = $ workg_remote(ilog:ihig,jlog:jhig)
```
While you’re at it, add some infinite loops!

• Why?! … Synchronization!
• How?
  – Initialize integer co-array to zero
  – Sender puts address (guaranteed nonzero)
  – Receivers spin-wait for nonzero value
• Declare the co-array VOLATILE!
  – Otherwise you might spin-wait on a register
  – Wait for comic ray to change register value?
Scatter synchronization

integer(8), volatile :: remote_address(NPROC_X*NPROC_Y)[*]
logical, volatile :: remote_flag(NPROC_X*NPROC_Y)[*]
...
do while (remote_address(master_image) == 0)
end do
... ! Copy data
remote_address(master_image) = 0
remote_flag(my_image)[master_image] = .true.

if (my_image == master_image) then
  do i = 1, num_images()
    do while (.not. remote_flag(i))
      end do
    remote_flag(i) = .false.
  end do
end if
Scatter performance

• 8-MSP POP/CICE production-like run
  – Ten simulation days
• “global_scatter” MPI
  – mpi_isend, mpi_irecv, mpi_wait
  – 31,635 samples from “samp_cs_time”
• “global_scatter” CAF
  – 1,767 samples
  – 18x faster for this subroutine
  – 4.2% → 0.3% of runtime
Scatter call-tree profiling

- **pat_report -b functions,callers**
- **MPI version**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4%</td>
<td>3.4%</td>
<td>26048</td>
<td>global_scatter@ice_mpi_internal_</td>
</tr>
<tr>
<td>0.5%</td>
<td>14.0%</td>
<td>3502</td>
<td>global_scatter@ice_mpi_internal_</td>
</tr>
<tr>
<td>0.0%</td>
<td>14.9%</td>
<td>1</td>
<td>global_scatter@ice_mpi_internal_</td>
</tr>
<tr>
<td>0.3%</td>
<td>47.9%</td>
<td>2015</td>
<td>global_scatter@ice_mpi_internal_</td>
</tr>
<tr>
<td>0.0%</td>
<td>48.1%</td>
<td>1</td>
<td>global_scatter@ice_mpi_internal_</td>
</tr>
<tr>
<td>0.0%</td>
<td>87.7%</td>
<td>3</td>
<td>global_scatter@ice_mpi_internal_</td>
</tr>
<tr>
<td>0.0%</td>
<td>93.7%</td>
<td>15</td>
<td>global_scatter@ice_mpi_internal_</td>
</tr>
<tr>
<td>0.0%</td>
<td>94.1%</td>
<td>6</td>
<td>global_scatter@ice_mpi_internal_</td>
</tr>
<tr>
<td>0.0%</td>
<td>97.2%</td>
<td>2</td>
<td>global_scatter@ice_mpi_internal_</td>
</tr>
<tr>
<td>0.0%</td>
<td>97.6%</td>
<td>3</td>
<td>global_scatter@ice_mpi_internal_</td>
</tr>
<tr>
<td>0.0%</td>
<td>98.7%</td>
<td>1</td>
<td>global_scatter@ice_mpi_internal_</td>
</tr>
<tr>
<td>0.0%</td>
<td>99.5%</td>
<td>1</td>
<td>global_scatter@ice_mpi_internal_</td>
</tr>
<tr>
<td>0.0%</td>
<td>99.9%</td>
<td>37</td>
<td>global_scatter@ice_mpi_internal_</td>
</tr>
</tbody>
</table>

- **Co-array version**

| 0.3% | 93.2% | 1767 | global_scatter@ice_mpi_internal_ |
Scatter line profiling

- `pat_report -b functions,lines`
- MPI version - not useful, it's all in calls
- Co-array version (first sync loop)

| 0.3% | 93.2% | 1767 | global_scatter@ice_mpi_internal_
|--------------------------------------|
| 0.2% | 93.2% | 1552 | line.253
| 0.0% | 93.2% | 100  | line.256
| 0.0% | 93.2% | 67   | line.255
| 0.0% | 93.2% | 34   | line.264
| 0.0% | 93.2% | 10   | line.243
| 0.0% | 93.2% | 3    | line.216
| 0.0% | 93.2% | 1    | line.295

---
Is any of this useful?

• Avoid using cache
  – Rarely, more of a tweak
  – In real apps, not often obvious when to apply

• Replace BLAS calls with do loops
  – Only if outer loops over BLAS are independent
  – Ask Cray when BLAS will automatically inline

• Minimize vector length
  – Yes, register re-use is good
  – May be more important with X1E
Is any of this useful?

• Move if statements inside loops
  – Maybe, again more of a tweak
  – More important for X1E?

• Use more pointers
  Add infinite loops
  – Yes! Minimize latency! Eliminate copies!
  – Needed less often when optimized MPI collectives are available
  – Deadlock prone, not portable
Questions?