NESAP CESM MG2 Update

Helen He (and MG2 team)

Cray Quarterly Meeting
July 22, 2015
CESM NESAP MG2 Team Members

• **NCAR:** John Dennis, **Chris Kerr**, Sean Santos
• **Cray:** Marcus Wagner
• **Intel:** Nadezhda Plotnikova, Martyn Corden
• **NERSC Liaison:** Helen He
MG2 Kernel

- MG2 is a kernel for CESM that represents its radiative transfer workload. Typically consumes about 10% of CESM run time.
  - Brought to Dungeon Session in March

- **Kernel is core bound**
  - Not bandwidth limited at all
  - Shows very little vectorization
    - Some loop bounds are short (e.g. 10)
  - Heavy use of math intrinsicsics that do not vectorize
    - `pow()`, `gamma()`, `log10()`.
    - Intel intrinsic `gamma()` is 2.6x slower than MG2 internal function

- **Kernel has long complex loops with interleaved conditionals and elemental function calls.**
  - Mixed conditionals and non-inlined functions inhibit vectorization
  - Some send array sections to elemental functions
MG2 Vectorization Prototype

• Use compiler report to check and make sure key functions are vectorized (and all functions on the call stack are vectorized too)
  – Elemental functions need to be inlined
  – “-qopt-report=5” reports highest level of details.
  – “-ipo” is needed if functions are in different source codes.
• Add !$OMP DECLARE SIMD and !$DIR$ ATTRIBUTE FORCEINLINE when needed.
Recommendations from Dungeon Session

• Divide major loops when possible and localize vectorization: work to be done by MG2 developers.

• Implement inlining as close to hotspot as possible; or use vector functions on the low level

• Follow up with MKL team on Gamma function vectorization.

• Work with compiler team for a flag to replace FORCEINLINE, and portable options for other compilers.
Changes Made to Improve Performance (1)

• Remove ‘elemental’ attribute and move the ‘mgncol’ loop inside routine

Before change:

```fortran
elemental function wv_sat_svp_to_qsat(es, p)
result(qs)
  real(r8), intent(in) :: es ! SVP
  real(r8), intent(in) :: p
  real(r8) :: qs

  ! If pressure is less than SVP, set qs to maximum of 1.
  if ( (p - es) <= 0._r8 ) then
      qs = 1.0_r8
  else
      qs = epsilo*es / (p - omeps*es)
  end if

end function wv_sat_svp_to_qsat
```

After change:

```fortran
function wv_sat_svp_to_qsat(es, p, mgncol) result(qs)
  integer, intent(in) :: mgncol
  real(r8), dimension(mgncol), intent(in) :: es ! SVP
  real(r8), dimension(mgncol), intent(in) :: p
  real(r8), dimension(mgncol) :: qs
  integer :: i

  do i=1,mgncol
    if ( (p(i) - es(i)) <= 0._r8 ) then
      qs(i) = 1.0_r8
    else
      qs(i) = epsilo*es(i) / (p(i) - omeps*es(i))
    end if
  enddo

end function wv_sat_svp_to_qsat
```
Impact of Code Changes for Elemental Functions

• No changes to algorithm
• Algorithm gives same answers
• Code readability not effected
• Revised code looks almost identical to original
• Provide scalar and vector version of functions
• Overload function names to maintain single naming convention
Changes Made to Improve Performance (2)

• Structure routine: don’t use assumed-shaped arrays:

**Before change:**
subroutine size_dist_param_liq(qcic, ...,)
  real, intent(in) :: qcic(:)
  do i=1,SIZE(qcic)

**After change:**
subroutine size_dist_param_liq(qcic, ..., mgncol)
  real, dimension(mgncol), intent(in) :: qcic
  do i=1,mgncol
Changes Made to Improve Performance (3)

• Divide loop blocks into manageable sizes. Allows compiler to vectorize loops. Can fuse loops during optimization step.
• Remove array syntax: plev(:, :) and replace with loops
• Replace divides: 1.0/plev(i, k) with *plev_inv(i, k)
• Remove initialization of variables that are over written
Changes Made to Improve Performance (4)

• Rearrange loop order to allow for data alignment

Before change:
do i=1,mgncol
do k=1,nlev
plev(i,k) = ...

After change:
Do k=1,nlev
do i=1,mgncol
plev(i,k) = ...

• Use more aggressive compiler options
  – -O3 -xAVX -fp-model fast=2 -no-prec-div -no-prec-sqrt
    -ip -fimf-precision=low -override-limits -qopt-report=5
    -no-inline-max-total-size -inline-factor=200

• Use Profile-guided Optimization (PGO) to improve code performance

• Compare performance of code with different vendors compilers
Changes Made to Improve Performance (5)

- Align data on specific byte boundaries; directive based approach with OMP directive:
  - Portable solution:
    
    ```
    !$OMP SIMD ALIGNED
    (qc,qcn,nc,ncn,qi,qin,ni,nin,qr,qrn,nr,nrn,qs,qsn,ns,nsn)
    ```
    - Tells the compiler that the arrays are aligned
    - Asserts that there are no dependencies
    - Requires to use PRIVATE or REDUCTION clauses to ensure correctness
    - Forces the compiler to vectorize, whether or not it thinks it is a good idea or not
  
  - As compared to:
    
    ```
    !$OMP SIMD ALIGNED
    ```
    - Intel compiler specific, not portable

- !$OMP SIMD ALIGNED is independent of vendor, however it can be overly intrusive in code
- 8% improvement in overall performance
The “ALIGNED” attribute is important for performance
However, providing the list of variables for the aligned list is tedious and error-prone, and often times impossible in large real applications.

- !$OMP SIMD ALIGNED added in 48 loops in MG2 kernel, many with list of 10+ variables

Working with Intel compiler team to find a more manageable solution: How can compilers know better which arrays are aligned?

Desired for other compilers too.

<table>
<thead>
<tr>
<th>!$OMP SIMD ALIGNED</th>
<th>!$OMP SIMD</th>
<th>!dir$ VECTOR ALIGNED</th>
<th>-align array64byte</th>
<th>-openmp</th>
<th>Time per iteration (usec) on Edison</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>444</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>446</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td>484</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td>482</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>452</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>456</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>473</td>
</tr>
</tbody>
</table>
Srinath Vadlamani’s testSIMD Suite

- Python test script to see which of the SIMD options are able to get close to AVX performance.
- “aligned” is essential
- Tests ran on Edison. Use “ifort” native compiler (15.0.1.133), default “-O2” optimization: not completely –no-vec

<table>
<thead>
<tr>
<th>Compiler and language options</th>
<th>Run Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>4.0509</td>
</tr>
<tr>
<td>-xavx</td>
<td>3.2940</td>
</tr>
<tr>
<td>!$omp declare simd(init)</td>
<td>40.0168</td>
</tr>
<tr>
<td>!$omp declare simd(init) uniform(n)</td>
<td>40.0029</td>
</tr>
<tr>
<td>!$omp declare simd(init) simdlen(4) uniform(n)</td>
<td>37.8277</td>
</tr>
<tr>
<td>!$omp declare simd(init) simdlen(4)</td>
<td>37.7145</td>
</tr>
<tr>
<td>!$omp declare simd(init) aligned(a:32)</td>
<td>4.2609</td>
</tr>
<tr>
<td>!$omp declare simd(init) aligned(a:32) uniform(n)</td>
<td>4.2955</td>
</tr>
<tr>
<td>!$omp declare simd(init) simdlen(4) aligned(a:32)</td>
<td>4.2598</td>
</tr>
<tr>
<td>!$omp declare simd(init) simdlen(4) aligned(a:32) uniform(n)</td>
<td>4.2779</td>
</tr>
</tbody>
</table>
### Performance Comparisons on Different Compilers and Hardware

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Compiler</th>
<th>Performance [usec per iteration]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sandy-Bridge</td>
<td>Intel-15.0.2</td>
<td>541</td>
</tr>
<tr>
<td>Sandy-Bridge</td>
<td>PGI-15.5</td>
<td>600</td>
</tr>
<tr>
<td>Ivy-Bridge</td>
<td>Intel-15.0.1</td>
<td>407</td>
</tr>
<tr>
<td>Ivy-Bridge</td>
<td>Cray-8.3.11</td>
<td>347</td>
</tr>
</tbody>
</table>

- Fastest run on Edison: 407 sec (not easily reproducible when run again with same executable)
- Original performance on Sandy-Bridge with Intel/15.0.2 is 1035 usec
- Cray compiler is fastest
Summary

• Directives and flags can be helpful, however not a replacement for programmers work on code modifications.

• Break up loops and push loops into functions where vectorization can be dealt with directly and can expose logic to compiler.

• Incremental improvements not necessary a BIG win from any one thing. Accumulative results matter.

• Performance and portability is a major goal: use !$OMP SIMD proves to be beneficial but very hard to use regarding the need of providing the aligned list.
Thank you.