Using OpenACC Compilers to Run FIM and NIM on GPUs

Mark Govett

NOAA Earth System Research Laboratory
Background

• Developing NIM model to be highly scalable a single source code and performance portable
  – Intel SB, Xeon Phi, Kepler GPU, etc

• Plan to run NIM at 3.5KM resolution in 2014
  – Require a minimum of 2000 GPUs or Xeon-Phi
  – Further work to optimize communications and compute

• First of several models projected to run on Fine-Grain computers in 2014 & 2015
  – FIM, HRRR (variant of WRF-ARW)
Goals of this Talk

• Report on our recent experience with the openACC compilers from PGI, Cray, CAPS
  – Determine ability to handle FIM, NIM codes without code change
  – Maintain performance portability to CPU, GPU, MIC

• Compare performance to F2C-ACC
MIC & GPU Performance for NIM

- 10242 horizontal points, 96 vertical levels
- Kepler results used the F2C-ACC compiler
- Xeon Phi are from the 7110 version on TACC
- The source code for these runs is the identical
  - New diag code has been optimized for MIC, but not run on the Kepler

<table>
<thead>
<tr>
<th>Main Routines</th>
<th>Kepler K20x</th>
<th>Xeon Phi</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOTAL: Main loop</td>
<td>[19.61]</td>
<td>[20.73]</td>
</tr>
<tr>
<td>vdmints</td>
<td>6.29</td>
<td>7.02</td>
</tr>
<tr>
<td>vdmintv</td>
<td>2.43</td>
<td>3.44</td>
</tr>
<tr>
<td>flux</td>
<td>1.11</td>
<td>1.79</td>
</tr>
<tr>
<td>trisol</td>
<td>0.76</td>
<td>0.52</td>
</tr>
<tr>
<td>force</td>
<td>0.64</td>
<td>0.93</td>
</tr>
<tr>
<td>Vdn</td>
<td>0.50</td>
<td>0.93</td>
</tr>
<tr>
<td>Diag</td>
<td>[2.70]</td>
<td>[1.11*]</td>
</tr>
</tbody>
</table>
F2C-ACC Compiler

- Developed in 2008 before commercial compilers were available
- Developed for FIM and NIM
  - New capabilities added as needed
  - Used primarily for model dynamics
- Limited Capabilities, Scope, Support
  - Partial support for Fortran 90
  - Shared with a few outside groups
  - No attempt to conform to openACC standard
  - No new development since the last workshop
- Evaluate commercial compilers
  - Share results, code with vendors
  - Henderson, 2010: pre-OpenACC CAPS, PGI compilers
  - Govett, 2013
Using the OpenACC Compilers

• Simple, easy to use
• Feedback by compilers was useful
• Placement of directives was trivial
  – Same as F2C-ACC, similar to openMP
• OpenACC directives required less information than F2C-ACC to prescribe parallelism
• OpenACC compilers have more capabilities than F2C
Directives to Identify Parallel Regions

- **F2C-ACC**
  - !ACC$REGION (<threads>, <blocks>
    [, <data movement>] )
  - !ACC$REGION END

- **OpenACC**
  - !$acc kernels
  - !$acc end kernels
  - !$acc parallel [num_gangs][num_workers][vector_length]
  [ data movement ]
  - !$acc end parallel
Directives for Loops

- F2C-ACC
  - `!ACC$DO [PARALLEL] [VECTOR]`
- OpenACC
  - `!$acc loop [gang] [worker] [vector]`

Directives for Data Movement

- F2C-ACC
  - `!ACC$REGION, !ACC$DATA`
- OpenACC
  - `!$acc parallel, !$acc data !$acc update`
Additional Directives

• F2C-ACC
  – !ACC$ROUTINE
  – !ACC$THREAD

• OpenACC
  – !$acc routine
  – !$acc async
  – !$acc cache
  – !$acc declare
  – !$acc wait
Standalone Test Cases

- 5 tests from FIM, NIM, WRF
  - Cnuity: FIM Dynamics
  - Trcadv: FIM Dynamics
  - WRF-PBL: WRF Physics
  - Momtum: FIM Dynamics
  - Vdmintv: NIM Dynamics

- Tests run on Titan using F2C-ACC, PGI & Cray

- OpenACC directives used
  - !$acc kernels
  - !$acc parallel [num_gangs] [num_workers] [vector_length]
  - !$acc loop [gang] [vector]

- Share results with vendors
  - Collaborate on tuning
Bitwise Exact Results

- Important to validate the parallelization
  - Speeds parallelization, leaves no doubt with scientists
- Correctness may be needed for long simulations
- Compiler options needed to generate correct results
  - Can be turned off for speed, but essential for correctness

F2C-ACC w/ Intel, CUDA compiler

<table>
<thead>
<tr>
<th>Tracer</th>
<th>Diffs Found</th>
<th>Possible Diffs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>655488</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>655488</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>655488</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>655488</td>
</tr>
</tbody>
</table>

Cray, PGI compilers

<table>
<thead>
<tr>
<th>Tracer</th>
<th>Diffs Found</th>
<th>Possible Diffs</th>
<th>Max Diff</th>
<th>K, IPN</th>
<th>Number of Digits</th>
<th>Max Relative Diff</th>
<th>K, IPN</th>
<th>Number of Digits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1551</td>
<td>655488</td>
<td>2.44E-04</td>
<td>64</td>
<td>3285</td>
<td>2.12E-07</td>
<td>15</td>
<td>1654</td>
</tr>
<tr>
<td>2</td>
<td>10807</td>
<td>655488</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FIM Dynamics: CNUITY

!ACC$REGION(<64>,<10242>,<flxhi:none,local>) BEGIN
!$acc parallel num_gangs(ihe-ips+1) vector_length(64) private(flxhi)
!ACC$DO PARALLEL(1)
!$acc loop gang
  do ipn=ips,ihe
!ACC$DO VECTOR(1)
!$acc loop vector
  do k=1,nvl
    flxhi(k) = vnorm(k,edg,ipn)*dp_edg(k,edg,ipn)
    massfx(k,edg,ipn,nf) = 0.5*((vnorm(k,edg,ipn) + abs(vnorm(k,edg,ipn)))*delp(k,ipn) - &
      (vnorm(k,edx,ipx) + abs(vnorm(k,edx,ipx)))*delp(k,ipx))
  enddo
enddo
!$acc end parallel
!ACC$REGION END

- FIM has 64 vertical levels, 10242 horizontal points / GPU
- One horizontal dimension – ipn
- One vertical dimension - k
FIM Dynamics: CNUITY

### Compiler versions & settings used
- **Cray V8.1.9:** `-O3 -h noomp,acc -em -ef -eZ -ra`
  - `-fast_addr` option uses 32 bit addressing for array references
- **PGI V13.7.0:** `-O3 -acc -Minfo=accel`
  - `-ta=nvidia:cuda5.0` sometimes yielded 10-20% faster runtimes

### Kernel execution times for Kepler K20x
- Does not include data movement

### Code modified for PGI due to a bug with handling “private” (GPU local) variables resulted in a significant performance penalty

### Both !$acc parallel and !$acc kernels were tried
- !$acc parallel *almost* always faster than !$acc kernels

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Cnuity1</th>
<th>Cnuity2</th>
<th>Cnuity3</th>
<th>Cnuity4</th>
<th>Cnuity5</th>
<th>Total (% slower)</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2C</td>
<td>325,</td>
<td>295</td>
<td>421</td>
<td>13,</td>
<td>1024</td>
<td>3552</td>
</tr>
<tr>
<td>PGI: parallel</td>
<td>616,</td>
<td>463</td>
<td>798</td>
<td>159,</td>
<td>1286</td>
<td>8941</td>
</tr>
<tr>
<td>Cray: parallel</td>
<td>682,</td>
<td>745</td>
<td>964</td>
<td>43,</td>
<td>1893</td>
<td>4031</td>
</tr>
<tr>
<td>Cray: fast32</td>
<td>555,</td>
<td>432</td>
<td>748</td>
<td>37,</td>
<td>1322</td>
<td>3903</td>
</tr>
</tbody>
</table>
FIM Dynamics: TRCADV

- 64 vertical levels, 10242 horizontal points
- Computation time only – does not include data movement

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Trcadv1</th>
<th>Trcadv2</th>
<th>Trcadv3</th>
<th>Total(% slower)</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2C-ACC</td>
<td>1031</td>
<td>463</td>
<td>679</td>
<td>2173</td>
</tr>
<tr>
<td>PGI</td>
<td>1615</td>
<td>885</td>
<td>871</td>
<td>3371 (55%)</td>
</tr>
<tr>
<td>Cray</td>
<td>2895</td>
<td>757</td>
<td>1233</td>
<td>4885 (124%)</td>
</tr>
<tr>
<td>Cray-fast32</td>
<td>1205</td>
<td>607</td>
<td>1076</td>
<td>2888 (32%)</td>
</tr>
</tbody>
</table>
Code Example: WRF-PBL

```fortran
!$acc parallel num_gangs((ite-its+1)/64+1) vector_length(64)
do k = kts,kte            !vertical dimension loop
!$acc loop gang vector
  do i = its,ite           !horizontal dimension loop
    zq(i,k+1) = dz8w2d(i,k)+zq(i,k)
  enddo
enddo
!
do k = kts,kte
!$acc loop gang vector
  do i = its,ite
    za(i,k) = 0.5*(zq(i,k)+zq(i,k+1))
    dzq(i,k) = zq(i,k+1)-zq(i,k)
    del(i,k) = p2di(i,k)-p2di(i,k+1)
  enddo
enddo
```

- **Typical loop structure for WRF physics**
- **Apply block (gang) and thread (vector) level parallelization to a single dimension for NIM**
  - Dependence on “k” prevents parallelization
### WRF Physics: PBL

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Kernel1</th>
<th>Kernel2</th>
<th>Kernel3</th>
<th>Kernel4</th>
<th>Kernel5</th>
<th>Kernel6</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2C</td>
<td>178</td>
<td>197</td>
<td>665</td>
<td>703</td>
<td>613</td>
<td>91</td>
<td>2447</td>
</tr>
<tr>
<td>Cray</td>
<td>250</td>
<td>281</td>
<td>[740]</td>
<td>[1090]</td>
<td>[699]</td>
<td>87</td>
<td>3147</td>
</tr>
<tr>
<td>PGI</td>
<td>351</td>
<td>[301]</td>
<td>[829]</td>
<td>652, BUG</td>
<td>[731]</td>
<td>84</td>
<td>NA</td>
</tr>
<tr>
<td>Cray: fast</td>
<td>193</td>
<td>202</td>
<td>[613]</td>
<td>[780]</td>
<td>[566]</td>
<td>75</td>
<td>2494</td>
</tr>
</tbody>
</table>

- WRF Version 3.3 code with minor changes
  - F2C-ACC has limited Fortran support
  - To get bitwise exact results compared to the CPU
    - calculations with **2 modified to multiply factors directly
- Square brackets indicate more than one kernel was used
Why are the openACC Compilers Slower?

• Use of Memory: Local, Shared, Global, Registers?
  – Significant benefit using private (thread local) memory observed by all compilers
  – Minimal benefit using shared memory (F2C), did not test with Cray, PGI
  – 10-30% more registers used in Cray, PGI than F2C-ACC

• Parallelism: Increasing the number of threads / block from 64 – 128
  – 32% performance improvements for F2C-ACC routines observed
  – Degraded performance for Cray, PGI compilers observed
    • No combination of gangs, workers, vectors yielded benefit

<table>
<thead>
<tr>
<th>Compiler - threads</th>
<th>Cnuity1</th>
<th>Cnuity2</th>
<th>Cnuity3</th>
<th>Cnuity4</th>
<th>cnuity5</th>
<th>% benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2C – 64</td>
<td>, 701</td>
<td>400</td>
<td>506</td>
<td>, 1384</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2C – 128</td>
<td>325, 514</td>
<td>295</td>
<td>421</td>
<td>13, 1024</td>
<td>3552</td>
<td>32% faster</td>
</tr>
<tr>
<td>Cray – 64</td>
<td>555, 980</td>
<td>431</td>
<td>750</td>
<td>13, 1322</td>
<td>3895</td>
<td></td>
</tr>
<tr>
<td>Cray – 128</td>
<td>,1272</td>
<td>457</td>
<td>974</td>
<td>, 1419</td>
<td></td>
<td>18% slower</td>
</tr>
<tr>
<td>PGI – 64</td>
<td>616, 993</td>
<td>463</td>
<td>798</td>
<td>159, 1286</td>
<td>8941</td>
<td></td>
</tr>
<tr>
<td>PGI – 128</td>
<td>,1094</td>
<td>480</td>
<td>977</td>
<td>, 1403</td>
<td></td>
<td>11% slower</td>
</tr>
</tbody>
</table>
Summary

• Goal is to have performance benchmarks of the FIM and NIM in the next few months
  – Both physics and dynamics if possible

• OpenACC compilers are the future for GPU programming
  – Standard is sufficient for our applications
  – Easy to use, parallelization is straightforward
  – No code changes required for parallelization
    • Except for the PGI handling of “private” variables

• We will work with vendors to improve performance
  – Modify our use of the openACC directives
    • are we missing something?
  – Provide stand-alone tests to vendors on our website for profiling and analysis