New Developments in the F2C-ACC Compiler

Mark Govett
F2C-ACC Basics

• Fortran-to-CUDA (or C) Accelerator
• Directive-based Compiler  \texttt{!ACC$<directive>}
• Developed in 2009 to speed code conversion of NIM
• Goal is to have a single source code that runs on CPU, GPU and MIC
  – Important for code developers (scientists)
  – Reduces development time
  – Allows for direct performance comparisons between CPU, GPU, MIC
• Being used to parallelize
  – NIM, FIM dynamics and WRF / YSU physics
• Working with the GPU compiler vendors
  – CAPS, PGI, CRAY
NIM Parallelization

- 2010: Running all of NIM dynamics on the GPU
  - 25x speedup (1 core of Harpertown vs. Tesla GPU)
- 2011: Multi-GPU runs of NIM dynamics, no physics
  - Small problem size limits scaling
  - waiting for science to progress
  - 4.5x speedup (6 core Westmere vs. single Fermi GPU)
- 2012: New NIM version
  - Runs at 30 KM resolution with YSU or GFS physics
  - GPU parallelization of dynamics took two weeks
    - Optimization to begin next week using F2C-ACC
  - Multi-GPU runs expected by October 1.
F2C-ACC Improvements

• Supports FIM and NIM GPU parallelization
• Ease of Use
  – 10 parallelization directives
  – Automatic generation of data movement
  – Assumes data is resident on the CPU
• Correctness
  – Improvements to mimic Fortran
  – NVCC compiler upgrades
  – Variable promotion
• Performance
  – Variable demotion
  – Control of global, local, shared and register memory
  – Options for blocking and chunking
Standalone Performance Tests

• Share with vendors for performance comparisons & correctness
  – Intel, PGI, CAPS, Cray

<table>
<thead>
<tr>
<th>routine</th>
<th>Model</th>
<th>Num Subroutines</th>
<th>Num GPU Kernels</th>
</tr>
</thead>
<tbody>
<tr>
<td>vdmintv</td>
<td>NIM dynamics</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>trcadv</td>
<td>FIM dynamics</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>cnuity</td>
<td>FIM dynamics</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>momtum</td>
<td>FIM dynamics</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>ysu_pbl</td>
<td>WRF physics</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>
Automatic Generation of Data Movement

**F2C V3**: - user lists thread and block size
- user list the intent and GPU memory for each variable

```fortran
!ACC$REGION(<nvl>,<ime-ims+1>,
!ACC$> <ims,ime,nvl,nedge,permedge,sidevec_e,
!ACC$> <nedge,permedge,sidevec_e,u_edg,v_edg:in,global> &
!ACC$> <vnorm:out,global> ) BEGIN
!ACC$DO PARALLEL(1)
do ipn=ims,ime
   do edgc=1,nedge(ipn) ! loop through edges
      edg = permedge(edgc,ipn)
   !ACC$DO VECTOR(1)
do k=1,nvl
      vnrm(k,edg,ipn) = sidevec_e(2,edg,ipn)* &
         u_edg(k,edg,ipn) - sidevec_e(1,edg,ipn)* &
         v_edg(k,edg,ipn)
   end do
end do
!ACC$REGION END
```
Automatic Generation of Data Movement

**F2C V4:**
- user lists thread and block size
- determines kernel intent
- assumes data is resident on the CPU

```
!ACC$REGION(<nvl>,<ime-ims+1>) BEGIN
!ACC$DO PARALLEL(1)
do ipn=ims,ime
    do edgcount=1,nedge(ipn) ! loop through edges
        edg = permedge(edgcount,ipn)
!ACC$DO VECTOR(1)
do k=1,nvl
    vnorm(k,edg,ipn) = sidevec_e(2,edg,ipn)* &
        u_edg(k,edg,ipn) - sidevec_e(1,edg,ipn)* &
        v_edg(k,edg,ipn)
end do
end do
!ACC$REGION END
```
Generation of Data Movement

- F2C-ACC converts do loops into a thread or block index reference and the loops simply go away
  - Instructions are executed in parallel (effectively simultaneously)
  - This is different than Intel – MIC (see Rosinski talk)

```fortran
!!ACC$REGION(<nvl>,<ime-ims+1>) BEGIN
!!ACC$DO PARALLEL(1)

ipn = blockIdx.x+1;
do edgcount=1,nedge(ipn) ! loop through edges
edg = perm(egcount,ipn)
!!ACC$DO VECTOR(1)

k = threadIdx.x + 1;
vnorm(k,edg,ipn) = sidevec_e(2,edg,ipn)*
   u_edg(k,edg,ipn) - sidevec_e(1,edg,ipn)*
   v_edg(k,edg,ipn)

end do

!!ACC$REGION END
```
Correctness

• Prior to CUDA v4.2, the number of digits of accuracy was used to compare FIM / NIM results

<table>
<thead>
<tr>
<th>Variable</th>
<th>Ndifs</th>
<th>RMS (1)</th>
<th>RMSE</th>
<th>max</th>
<th>DIGITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>rublten</td>
<td>2228</td>
<td>0.1320490309E-03</td>
<td>0.2634E-09</td>
<td>0.3922E-05</td>
<td>5</td>
</tr>
<tr>
<td>rvblten</td>
<td>2204</td>
<td>0.2001348128E-03</td>
<td>0.6318E-09</td>
<td>0.2077E-04</td>
<td>4</td>
</tr>
<tr>
<td>exch_h</td>
<td>3316</td>
<td>0.1670498588E+02</td>
<td>0.8979E-05</td>
<td>0.8379E-05</td>
<td>5</td>
</tr>
<tr>
<td>hpbl</td>
<td>9</td>
<td>0.4522379124E+03</td>
<td>0.2688E-03</td>
<td>0.1532E-04</td>
<td>4</td>
</tr>
<tr>
<td>rqiblten</td>
<td>1082</td>
<td>0.2236843110E-09</td>
<td>0.7502E-17</td>
<td>0.6209E-07</td>
<td>7</td>
</tr>
</tbody>
</table>

– Small differences for 1 timestep can become significant when running a model over many timesteps

• NVCC V4.2 option: –fmad=false
  – No truncation of operation to 32 bits
  – FIM, NIM runs are bitwise exact compared to the CPU
    • Speeds parallelization
  – Exceptions:
    • Where ordering of operations changes the result
    • Use of power function, and possibly other intrinsics
Variable Promotion for Correctness

Example: NIM vdmintv subroutine (nz=32)

**F2C V4:** - promote variables using GPU global memory

```fortran
real :: rhsu(nz,nob), r hsv(nz,nob), tgtu(nz,npp), tgtv(nz,npp)

!ACC$REGION (<nz>,<(ipe-ips+1)>, &
!ACC$> <rhsu,r hsv,tgtu,tgtv:none,global,promote(1:block)> ) BEGIN
!ACC$DO PARALLEL(1)
do ipn=ips,ipe
!ACC$DO VECTOR(1,1:nz-1)
do k=1,nz-1
   rhsu(k,1) = cs(1,ipn)*u(k ,ippl)+sn(1,ipn)*v(k ,ippl) - u(k,ipn)
   rhsu(k,2) = ...
   < Similar calculations on r hsv, tgtu, tgtv >
endo
call solver( ..., rhsu, r hsv, ...)
endo
!ACC$REGION END
```

**Performance:** run-time w/ global memory: 12.51 ms
nvcc will use cache by default
Optimization: Shared Memory

Example: NIM vdmintv subroutine (nz=32)

**F2C V4:** - use GPU shared memory for rhsu,rhsv,tgtu,tgtv

```fortran
real :: rhsu(nz,nob), rhsv(nz,nob), tgtu(nz,npp), tgtv(nz,npp)

!ACC$DATA(<rhsu,rhsv,tgtu,tgtv:none,shared>) !declaration required
!ACC$REGION (<nz>,<(ipe-ips+1)>,
!ACC$> <rhsu,rhsv,tgtu,tgtv:none,shared> ) BEGIN
!ACC$DO PARALLEL(1)
do ipn=ips,ipe
!ACC$DO VECTOR(1,1:nz-1)
do k=1,nz-1
  rhsu(k,1) = cs(1,ipn)*u(k ,ipp1)+sn(1,ipn)*v(k ,ipp1) - u(k,ipn)
  rhsu(k,2) = ...
  < Similar calculations on rhsv, tgtu, tgtv >
endo
call solver( ..., rhsu, rhsv, ...)
endo
!ACC$REGION END
```

Performance: run-time w/ shared memory: 7.30 ms
1.7x speedup over global memory w/ cache
Optimization: Increase threads / block

Example: NIM vdmintv subroutine (nz=32)

**F2C V4:** - increase thread count to 96 (3 blocks of 32)

```fortran
real :: rhsu(nz,nob), rhsv(nz,nob), tgtu(nz,npp), tgtv(nz,npp)
 !ACC$DATA(<rhsu,rhsv,tgtu,tgtv:none,shared(96,6)>)! declaration

!ACC$REGION (<nz:block=3>,<(ipe-ips+1)>,
 &
 !ACC$> <rhsu,rhsv,tgtu,tgtv:none,shared> ) BEGIN
 !ACC$DO PARALLEL(1)
do ipn=ips,ipe
 !ACC$DO VECTOR(1,1:nz-1)
do k=1,nz-1
   rhsu(k,1) = cs(1,ipn)*u(k ,ipp1)+sn(1,ipn)*v(k ,ipp1) - u(k,ipn)
   rhsu(k,2) = ...
   < Similar calculations on rhsv, tgtu, tgtv >
endo
call solver( ..., rhsu, rhsv, ...)
endo
!ACC$REGION END
```

**Performance:** run-time w/ shared memory: 5.49 ms
2.3x speedup over global memory w/ cache
Optimization: Local Memory

**Example:** NIM vdmintv subroutine (nz=32)
Credit to NVIDIA’s Paulius Micikevicius for this optimization

**F2C V4:** - Use local memory – private to each thread

```fortran
real :: rhsu(nz,nob), rhsv(nz,nob), tgtu(nz,npp), tgtv(nz,npp)
!ACC$DATA(<tgtu,tgtv:none,shared(64,6)>) !declaration

!ACC$REGION (<nz:block=2>,<(ipe-ips+1)>, &
!ACC$> <rhsu,rhsv:none,local>,<tgtu,tgtv:none,shared>) BEGIN
!ACC$DO PARALLEL(1)
do ipn=ips,ipe
!ACC$DO VECTOR(1,1:nz-1)
do k=1,nz-1
    rhsu(k,1) = cs(1,ipn)*u(k ,ipp1)+sn(1,ipn)*v(k ,ipp1) - u(k,ipn)
    rhsu(k,2) = ...
    < Similar calculations on rhsv, tgtu, tgtv >
endo
call solver( ..., rhsu, rhsv, ...)
endo
!ACC$REGION END
```

**Performance:** run-time w/ shared memory: 3.69 ms
3.4x speedup over global memory w/ cache
Optimization: Variable Demotion
Example: FIM trcadv subroutine

**F2C V4:** - Demote variables + shared, local or register memory

```
!ACC$REGION(<nv1:block=2>,<ipe-ips+1>,
!ACC$>  <s_plus,s_mnus:none,local,demote(1)> ) BEGIN
!ACC$DO PARALLEL(1)
  do ipn=ips,ipe
!ACC$DO VECTOR(1)
   do k=1,nvl
     s_plus(k) = 0.
     s_mnus(k) = 0.
   end do
   do edg=1,nprox(ipn)
!ACC$DO VECTOR(1)
   do k=1,nvl
     s_plus(k) = s_plus(k) - min(0., antiflx(k,edg,ipn))
     s_mnus(k) = s_mnus(k) + max(0., antiflx(k,edg,ipn))
   end do
end do
```

Performance: 1.8x faster than global memory / cache
Optimization: Chunking

Example: FIM continuity subroutine

**F2C V4:** - recurrence relation prevents parallelism in the “k” dimension
- in general, vector units handle this type of parallelism much better than GPUs

```fortran
!ACC$DATA(<lyrtend, dpdx, dpdy: none, shared(6, 64)>)

!ACC$REGION(<6: chunk>,<(ipe-ips+1)/6>,
!ACC$> <lyrtend, dpdx, dpdy: none, shared, promote(1:thread)>) BEGIN
!ACC$DO PARALLEL(1)
  do ipn=ips,ipe
     do edgcount=1,nedge(ipn)  ! loop through edges
       edg = permedge(edgcount, ipn)
       do k=nvl,1,-1
         dpdx(k) = dpdx(k) + lp_edg(k, edg, ipn)*sidevec_c(2, edg, ipn)
         dpdy(k) = dpdy(k) - lp_edg(k, edg, ipn)*sidevec_c(1, edg, ipn)
       end do
     end do
   end do
< other calculations
!ACC$REGION END
```

**Performance:** 4x improvement over single thread execution
Performance Results
Westmere versus C2050 Fermi

- Explicit use of GPU memories was always better than GLOBAL memory with cache.
- Different optimizations were effective for different routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>GPU - F2C 1 socket GLOBAL MEMORY</th>
<th>GPU – F2C 1 socket SHARED MEMORY</th>
<th>GPU – F2C 1 socket Shared + Demotion</th>
<th>GPU – F2C 1 socket BLOCK or CHUNKING</th>
<th>GPU – F2C 1 socket BEST</th>
<th>CPU Westmere 2 sockets BEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>trcadv</td>
<td>2.067</td>
<td>1.79</td>
<td>1.72</td>
<td>1.53</td>
<td>1.28</td>
<td>4.22</td>
</tr>
<tr>
<td>cnuity</td>
<td>5.21</td>
<td>3.20</td>
<td></td>
<td>1.19</td>
<td>1.08</td>
<td>1.46</td>
</tr>
<tr>
<td>momtum</td>
<td>0.57</td>
<td>0.52</td>
<td></td>
<td>0.41</td>
<td>1.67</td>
<td></td>
</tr>
<tr>
<td>vdmintv</td>
<td>12.5</td>
<td>7.50</td>
<td></td>
<td>3.68</td>
<td>58.7*</td>
<td></td>
</tr>
<tr>
<td>wrf_pbl</td>
<td>52.3</td>
<td></td>
<td></td>
<td>3.04</td>
<td>39.0*</td>
<td></td>
</tr>
</tbody>
</table>

- Data transfer times between CPU and GPU are not included
- CPU runtimes for vdmintv and wrf_pbl are for a single core, the rest are with 12 cores
Summary

• F2C-ACC has supported our GPU parallelization of FIM & NIM dycores
  – Plans to move to OpenACC once the compilers mature
• F2C-ACC and CUDA upgrades speed parallelization
  – Bitwise exact results takes away the guesswork
  – Automatic generation of data movement
• Single source code for CPU, GPU and MIC allow direct performance comparisons
  – Optimizations for Fine-grain parallel also improve CPU performance
• GPU parallelization of FIM and NIM has been quite easy once the code has been adapted for fine-grain parallel architectures
  – more time is spent adapting the code than the GPU parallelization
  – Parallelization of a FIM routine typically takes a few hours
  – Parallelization of NIM dynamics took 2 weeks
• Demonstrated the value in explicit use of memory & parallelism
  – shared, local, and register memory, promotion, demotion, blocking, chunking