

New Developments in the F2C-ACC Compiler

Mark Govett

F2C-ACC Basics

- Fortran-to-CUDA (or C) Accelerator
- Directive-based Compiler !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

routine	Model	Num Subroutines	Num GPU Kernels
vdmintv	NIM dynamics	1	1
trcadv	FIM dynamics	3	3
cnuity	FIM dynamics	5	8
momtum	FIM dynamics	3	5
ysu_pbl	WRF physics	1	7

Automatic Generation of Data Movement

F2C V3: - user lists thread and block size

- user list the intent and GPU memory for each variable

```
!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$DO PARALLEL(1)
do ipn=ims,ime
    do edgcount=1,nedge(ipn)           ! loop through edges
        edg = permedge(edgcount,ipn)
!ACC$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
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$DO VECTOR(1)
  do k=1,nvl
    vnorm(k,edg,ipn) = sidevec_e(2,edg,ipn) *
                         u_edg(k,edg,ipn) - sidevec_e(1,edg,ipn) *
                         v_edg(k,edg,ipn)
  end do
end do
end do
!ACC$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)

```
!!ACC$REGION(<nvl>,<ime-ims+1>) BEGIN
!!ACC$DO PARALLEL(1)
ipn = blockIdx.x+1;
do edgcount=1,nedge(ipn)           ! loop through edges
   edg = permedge(edgcount,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

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

- 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

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

!ACC$REGION (<nz>,<(ipe-ips+1)>,
!ACC$> <rhsu,rhsv,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 ,ipp1)+sn(1,ipn)*v(k ,ipp1) - u(k,ipn)
    rhsu(k,2) = ...
      < Similar calculations on rhsv, tgtu, tgtv >
  enddo
  call solver( ..., rhsu, rhsv, ... )
enddo
!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

```
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$DO VECTOR(1,1:nz-1)
    do k=1,nz-1
        rhsu(k,1) = cs(1,ipn)*u(k ,ipp1)+sn(1,ipn)*v(k ,ipp1) - u(k,ipn)
        rhsu(k,2) = ...
            < Similar calculations on rhsv, tgtu, tgtv >
    enddo
    call solver( ..., rhsu, rhsv, ... )
enddo
!ACC$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)

```
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 >
  enddo
  call solver( ..., rhsu, rhsv, ... )
enddo
!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

```
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 >
  enddo
  call solver( ..., rhsu, rhsv, ... )
enddo
!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$REGION(<nvl:block=2>,<ipe-ips+1>,
&
!ACC$> <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
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

```
!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$DO PARALLEL(1)
do ipn=ips, ipe
  do edgcount=1, nedge(ipn) ! loop through edges
    edg = permedge(edgcount, ipn)
    do k=nvl, 1, -1
      dpdx(k) = dpdx(k) + lp_edg(k, edg, ipn)*sidevec_c(2, edg, ipn)
      dpdy(k) = dpdy(k) - lp_edg(k, edg, ipn)*sidevec_c(1, edg, ipn)
    end do
  end do
  < other calculations
!ACC$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

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

- 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