Experience Applying Fortran GPU Compilers to Numerical Weather Prediction

Tom Henderson
NOAA Global Systems Division
Thomas.B.Henderson@noaa.gov

Mark Govett, Jacques Middlecoff
Paul Madden, James Rosinski, Craig Tierney
Outline

- Motivation for GPU investigation
- The Non-hydrostatic Icosahedral Model (NIM)
- GPU software design issues
- Commercial directive-based Fortran GPU compilers
- Step-wise approach
- Initial performance comparisons
- Conclusions and future directions
Motivation for NOAA GPU Investigation

- Run global NWP and climate models at higher resolution with more sophisticated physical parameterizations to improve forecast skill
  - Research: shop for FLOPs
  - Operations: technology must be mature
- Michalakes early GPU work
- Continue to maintain single source code for all desired execution modes
  - Single and multiple CPU
  - Single and multiple GPU
  - Prefer a directive-based Fortran approach
NIM NWP Dynamical Core

- NIM = “Non-Hydrostatic Icosahedral Model”
  - NWP dynamical core prototype
  - Global “cloud-permitting” resolutions < 3km (42 million columns)
- 32-bit floating-point computations
- Computations structured as simple vector ops with indirect addressing and inner vertical loop
  - “GPU-friendly”, also good for CPU
Icosahedral Grid

- 450km
- 2562 columns
- Always 12 pentagons
- Good geometric properties compared to traditional latitude/longitude grid
NIM/FIM Indirect Addressing (MacDonald, Middlecoff)

- Single horizontal index
- Store number of sides (5 or 6) in “nprox” array
  - nprox(34) = 6
- Store neighbor indices in “prox” array
  - prox(1,34) = 515
  - prox(2,19) = 3
- Place directly-addressed vertical dimension fastest-varying for speed
- Very compact code
- Indirect addressing costs <1%
Simple Loop With Indirect Addressing

- Compute sum of all horizontal neighbors
  - nip = number of columns
  - nvl = number of vertical levels

\[ \text{xnsum} = 0.0 \]
\[
\text{do } \text{ipn}=1,\text{nip} \quad \text{! Horizontal loop}
\]
\[
\quad \text{do } \text{isn}=1,\text{nprox(ipn)} \quad \text{! Loop over edges (sides, 5 or 6)}
\]
\[
\quad \quad \text{ipp} = \text{prox(isn,ipn)} \quad \text{! Index of neighbor across side “isn”}
\]
\[
\quad \text{do } k=1,\text{nvl} \quad \text{! Vertical loop}
\]
\[
\quad \quad \text{xnsum}(k,\text{ipn}) = \text{xnsum}(k,\text{ipn}) + x(k,\text{ipp})
\]
\[
\quad \text{enddo}
\]
\[
\text{enddo}
\]
\[
\text{enddo}
\]
GPU Software Design Issues

- CPU controls high level program flow
  - I/O, message passing, coarse-grained parallelism
    - MPI parallelism via the Scalable Modeling System (SMS)
      - Directive-based

- GPU performs all computations
  - Fine-grained parallelism
  - Implemented by a GPU compiler

- Model data is resident on the GPU
  - Invert traditional “GPU-as-accelerator” model
    - Initial data read by the CPU and passed to the GPU
    - Data passed back to the CPU only for output & message-passing
    - Minimizes overhead of data movement between CPU & GPU
Massive fine-grained parallelism
- GPU “global memory” is slow
  - Need lots of threads to hide memory latency
  - Key trade-off: number of threads vs. thread resources

Coalesced loads
- Multiple memory accesses in one load
- Adjacent threads must access adjacent data
  - Use unit stride access in loops that will be threaded
GPU Software Design Issues

- Reduce branching
  - Code should vectorize well using CPU compilers
- Redesign algorithms if needed
- Optimize inter-CPU (MPI) communication
  - Computation time is a much smaller fraction of total run time when multiple GPUs are used
  - Overlap communication with computation
  - Use redundant computation to eliminate communication
GPU Fortran Compilers

- Commercial directive-based compilers
  - CAPS HMPP 2.3.5
    - Generates CUDA-C and OpenCL
    - Supports NVIDIA and AMD GPUs
  - Portland Group PGI Accelerator 11.7
    - Supports NVIDIA GPUs
    - Previously used to accelerate WRF physics packages
- F2C-ACC (Govett, 2008) directive-based compiler
  - “Application-specific” Fortran->CUDA-C compiler for performance evaluation
- Other directive-based compilers
  - Cray (beta)
Current GPU Compiler Limitations

- Limited support for Fortran language features such as modules, derived types
- Support not yet strong for automatic inlining, __device__ routines
- Both PGI and HMPP prefer “tightly nested outer loops” (not a limitation for F2C-ACC)

```fortran
! This is OK
do ipn=1,nip
  do k=1,nvl
    <statements>
  enddo
enddo

! This is NOT OK
do ipn=1,nip
  <statements>
    do k=1,nvl
      <statements>
    enddo
  enddo
enddo
```
Directive Comparison: Loops

\$hmppcg parallel
\ldo ipn=1,nip
\$hmppcg parallel
\ldo k=1,nvl
\hspace{1cm}HMPP \ldo isn=1,nprox(ipn)
\hspace{2cm}xnsum(k,ipn) = xnsum(k,ipn) + x(k,ipp)
\hspace{1cm}\ldo enddo
\ldo enddo
\ldo enddo

\$acc do parallel
\ldo ipn=1,nip
\$acc do vector
\ldo k=1,nvl
\hspace{1cm}PGI \ldo isn=1,nprox(ipn)
\hspace{2cm}xnsum(k,ipn) = xnsum(k,ipn) + x(k,ipp)
\hspace{1cm}\ldo enddo
\ldo enddo
\ldo enddo
Directive Comparison: Array Declarations

```
real :: u(nvl,nip)
...
call diag(u, ...)  
call vd(u, ...)  
call diag(u, ...)  
...
subroutine vd(fin, ...)  
...
subroutine diag(u, ...)  
...
```

Original Code
Directive Comparison: Array Declarations

real :: u(nvl,nip)
!$hmpp map, args[vd::fin;diag1::u;diag2::u]
...
!$hmpp diag1 callsite
call diag(u, ...)
!$hmpp vd callsite
call vd(u, ...)
!$hmpp diag2 callsite
call diag(u, ...)
...
!$hmpp vd codelet
subroutine vd(fin, ...)
...
!$hmpp diag1 codelet
!$hmpp diag2 codelet
subroutine diag(u, ...)
...
Directive Comparison:
Array Declarations

 !$acc mirror (u)
real :: u(nvl,nip)

! Must make interfaces explicit via interface block or use association
include “interfaces.h”

... call diag(u, ...)
call vd(u, ...)
call diag(u, ...)
...

subroutine vd(fin, ...)
 !$acc reflected (fin, ...)
...

subroutine diag(u, ...)
 !$acc reflected (u, ...)
...
Directive Comparison: Explicit CPU-GPU Data Transfers

HMPP

!$hmpp diag1 advancedLoad, args[u]
...
!$hmpp diag2 delegatedStore, args[u]

PGI

!$acc update device(u)
...
!$acc update host(u)
Step-Wise Approach to GPU Parallelization

- GPU tools and debuggers are still relatively primitive
  - Bugs can be difficult to diagnose
- Create test cases and establish tolerances
  - Match tolerances observed from CPU compiler optimization changes
- Make small changes and test after each
  - Much easier to find and fix errors
Step-Wise Approach to GPU Parallelization

- Compare HMPP and PGI output and performance with F2C-ACC compiler
  - Use F2C-ACC to prove existence of bugs in commercial compilers
  - Use F2C-ACC to prove that performance of commercial compilers can be improved
- Both HMPP and F2C-ACC generate “readable” CUDA code
  - Re-use function and variable names from original Fortran code
  - Allows straightforward use of CUDA profiler
  - Eases detection and analysis of compiler correctness and performance bugs
Initial Performance Results

- Optimize for both CPU and GPU
  - Some code divergence
  - Always use fastest code
- CPU = Intel Nehalem (2.8GHz) or Intel Westmere (2.66GHz)
- GPU = NVIDIA GTX280 “Tesla” or C2050 “Fermi”
- Work in-progress…
Initial Performance Results

- Small “G4-L96” test case
  - 2562 columns, 96 levels, 50 time steps
    - Fraction of time spent in init/input is unrealistically large
- Large “G5-L96” test case
  - 10242 columns, 96 levels, 1000 time steps
    - Newer version of NIM code
- Update to newer NIM code turned out to be non-trivial!
- Many GPU optimizations remain untried
### Run Times for Single GPU vs. Single Nehalem Core, “G4-L96”

<table>
<thead>
<tr>
<th>NIM routine</th>
<th>Nehalem CPU Time (sec)</th>
<th>F2C-ACC Tesla GPU Time (sec)</th>
<th>HMPP Tesla GPU Time (sec)</th>
<th>PGI Tesla GPU Time (sec)*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>106.6</td>
<td>10.8</td>
<td>10.3</td>
<td>--</td>
</tr>
<tr>
<td>vdmints</td>
<td>50.6</td>
<td>2.5</td>
<td>2.3</td>
<td>4.6</td>
</tr>
<tr>
<td>vdmintv</td>
<td>23.3</td>
<td>0.93</td>
<td>0.99</td>
<td>0.93</td>
</tr>
<tr>
<td>flux</td>
<td>10.4</td>
<td>1.15</td>
<td>1.05</td>
<td>0.43</td>
</tr>
<tr>
<td>vdn</td>
<td>4.6</td>
<td>0.58</td>
<td>0.73</td>
<td>--</td>
</tr>
<tr>
<td>diag</td>
<td>4.0</td>
<td>0.093</td>
<td>0.085</td>
<td>0.12</td>
</tr>
<tr>
<td>force</td>
<td>3.4</td>
<td>0.11</td>
<td>0.19</td>
<td>0.09</td>
</tr>
<tr>
<td>trisol</td>
<td>2.0</td>
<td>1.9</td>
<td>1.4</td>
<td>--</td>
</tr>
</tbody>
</table>

* Note error in paper, speedups erroneously listed for PGI
## Estimated GFLOPS for GPU and Single Nehalem Core “G4-L96”

<table>
<thead>
<tr>
<th>NIM routine</th>
<th>Nehalem 1-core CPU GFLOPS</th>
<th>F2C-ACC CUDA-C Tesla GPU GFLOPS</th>
<th>HMPP Tesla GPU GFLOPS</th>
<th>Computational Intensity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>3.2</td>
<td>31</td>
<td>32</td>
<td>1.68</td>
</tr>
<tr>
<td>vdmints</td>
<td>3.8</td>
<td>77</td>
<td>85</td>
<td>1.96</td>
</tr>
<tr>
<td>vdmintv</td>
<td>3.9</td>
<td>99</td>
<td>95</td>
<td>1.85</td>
</tr>
<tr>
<td>flux</td>
<td>2.3</td>
<td>21</td>
<td>23</td>
<td>1.11</td>
</tr>
<tr>
<td>vdn</td>
<td>1.0</td>
<td>9</td>
<td>6</td>
<td>0.89</td>
</tr>
<tr>
<td>diag</td>
<td>1.3</td>
<td>57</td>
<td>62</td>
<td>1.12</td>
</tr>
<tr>
<td>force</td>
<td>1.9</td>
<td>61</td>
<td>35</td>
<td>1.41</td>
</tr>
<tr>
<td>trisol</td>
<td>2.3</td>
<td>2.2</td>
<td>3</td>
<td>1.10</td>
</tr>
</tbody>
</table>

- Used PAPI performance counters on CPU (GPTL)

- Estimated ~29% of peak (11.2 GFLOPS) on CPU
Fermi GPU vs. Single/Multiple Westmere CPU cores, “G5-L96”

<table>
<thead>
<tr>
<th>NIM routine</th>
<th>Westmere CPU 1-core Time (sec)</th>
<th>Westmere CPU 6-core Time (sec)</th>
<th>F2C-ACC Fermi GPU Time (sec)</th>
<th>Fermi Speedup vs. 6-core CPU (1 socket ea.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>8654</td>
<td>2068</td>
<td>449</td>
<td>4.6</td>
</tr>
<tr>
<td>vdmints</td>
<td>4559</td>
<td>1062</td>
<td>196</td>
<td>5.4</td>
</tr>
<tr>
<td>vdmintv</td>
<td>2119</td>
<td>446</td>
<td>91</td>
<td>4.9</td>
</tr>
<tr>
<td>flux</td>
<td>964</td>
<td>175</td>
<td>26</td>
<td>6.7</td>
</tr>
<tr>
<td>vdn</td>
<td>131</td>
<td>86</td>
<td>18</td>
<td>4.8</td>
</tr>
<tr>
<td>diag</td>
<td>389</td>
<td>74</td>
<td>42</td>
<td>1.8</td>
</tr>
<tr>
<td>force</td>
<td>80</td>
<td>33</td>
<td>7</td>
<td>4.7</td>
</tr>
<tr>
<td>trisol</td>
<td>119</td>
<td>38</td>
<td>31</td>
<td>1.2</td>
</tr>
</tbody>
</table>
“G5-96” with PGI and HMPP

HMPP:
- Each kernel passes correctness tests in isolation
- Run times of individual kernels very close to F2C-ACC
- Unresolved error in “map”/data transfers

PGI:
- Started with PGI 11.7 six days ago
- Entire model runs but does not pass correctness tests
- Run times of most expensive kernels very close to F2C-ACC
- Data transfers appear to be correct
- Likely error in one (or more) kernel(s)
Ongoing Work With WRF Physics

- Legacy codes not designed with GPU in mind
- Much more difficult than NIM
- Initial candidate: YSU PBL scheme
- WRF physics (i,k,j) ordering good for coalesced loads (Michalakes, others)
  - Must transpose from NIM (k,ipn)
  - Transpose costs appear small
  - Memory may be an issue
Early Work With Multi-GPU Runs

- F2C-ACC + SMS directives
  - Correct results on different numbers of GPUs
  - Poor scaling because compute has sped up but communication has not
  - Working on communication optimizations
- Demonstrates that single source code can be used for single/multiple CPU/GPU runs
- Should be possible to mix HMPP/PGI directives with SMS too
Conclusions

- Some grounds for optimism
  - Fermi is ~4-5x faster than 6-core Westmere
  - Once compilers mature, expect level of effort similar to OpenMP for “GPU-friendly” codes like NIM
- HMPP strengths: more flexible low-level loop transformation directives, user-readable CUDA-C
- PGI strengths: simpler directives for making data persist in GPU memory
- This is still very much a work-in-progress
Future Directions

- Continue to improve GPU performance
  - Tuning options via commercial compilers
  - Test AMD GPU/APUs (HMPP->OpenCL)
- Address GPU scaling issues
- Cray GPU compiler
  - Working with beta releases
- Intel MIC
- OpenMP extensions?
- OpenHMPP?
Thank You