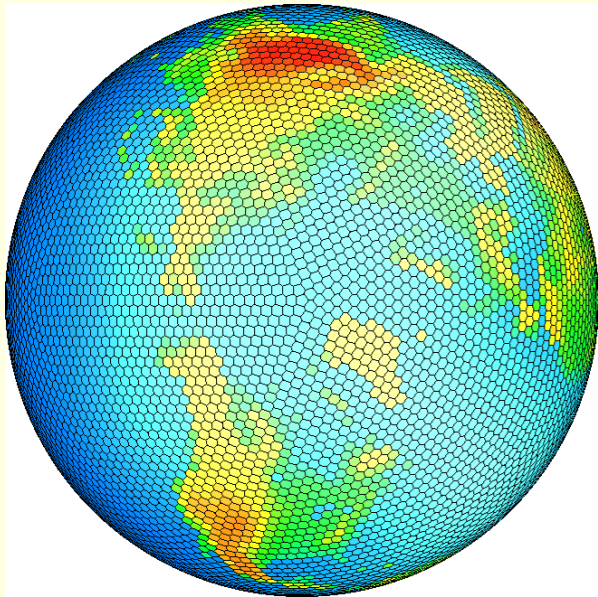
Earth System Research Laboratory
Global Systems Division

Progress on GPU Parallelization of the NIM Prototype Numerical Weather Prediction Dynamical Core

Tom Henderson
NOAA/OAR/ESRL/GSD/ACE
Thomas.B.Henderson@noaa.gov

Mark Govett, Jacques Middlecoff
Paul Madden, James Rosinski



11/1/2010

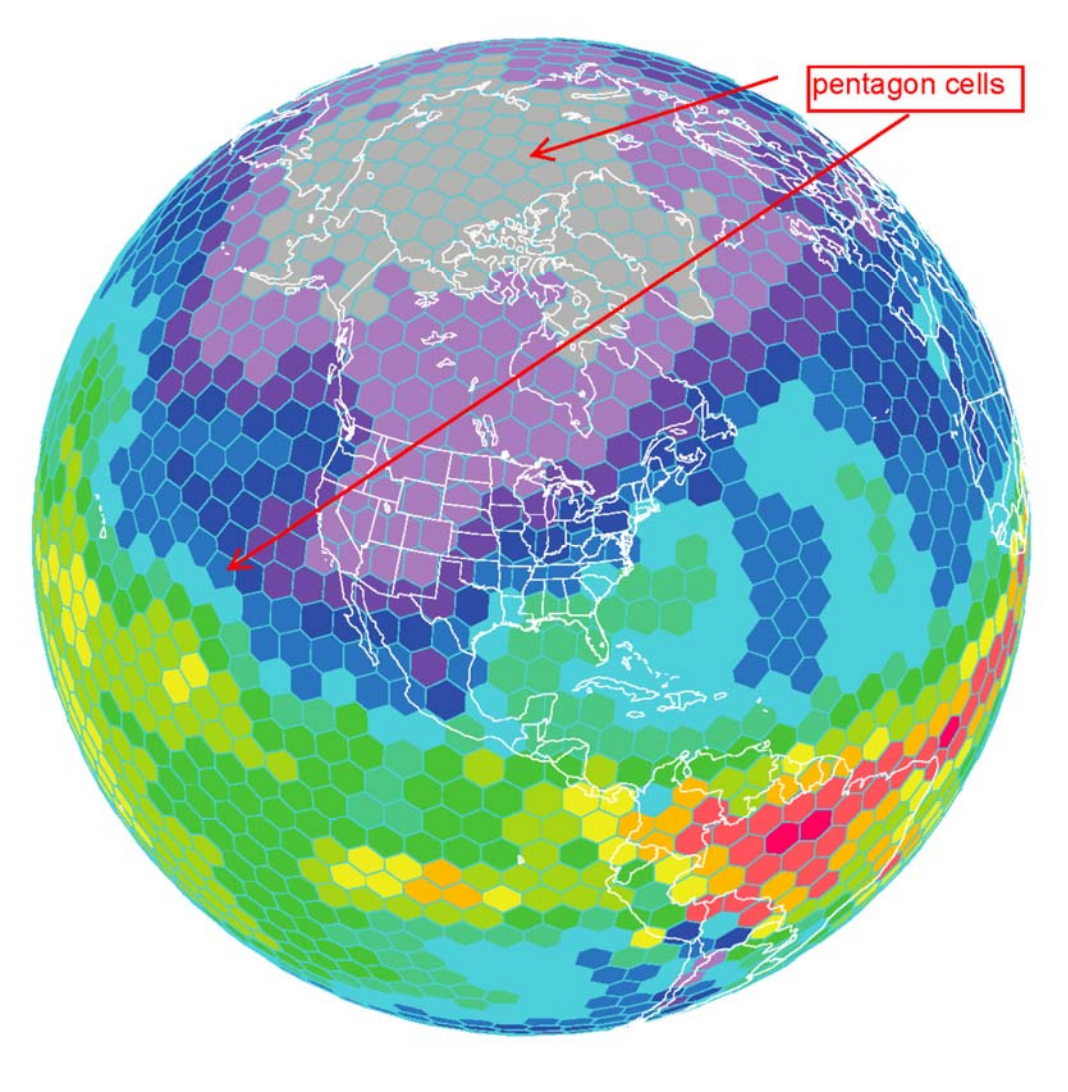
Outline

- NIM and its parent, FIM
- Implementation details unique to NIM/FIM
- Commercial directive-based Fortran GPU compilers
- Step-wise approach
- Initial performance comparisons of CPU and GPU
- Conclusions and future directions

NIM and FIM

- NIM = “Non-Hydrostatic Icosahedral Model”
  - NWP dynamical core prototype
  - Target: global “cloud-permitting” resolutions < 3km (42 million columns)
- FIM = “Flow-following Finite-Volume Icosahedral Model” (http://fim.noaa.gov/)
  - Hydrostatic ancestor of NIM
  - Target resolution ~28km (650K columns)
    - Operational global multi-model ensemble
- Both use 32-bit floating-point in dynamics

Icosahedral Grid



450km
2562 columns

Always 12 pentagons

Direct Addressing Schemes

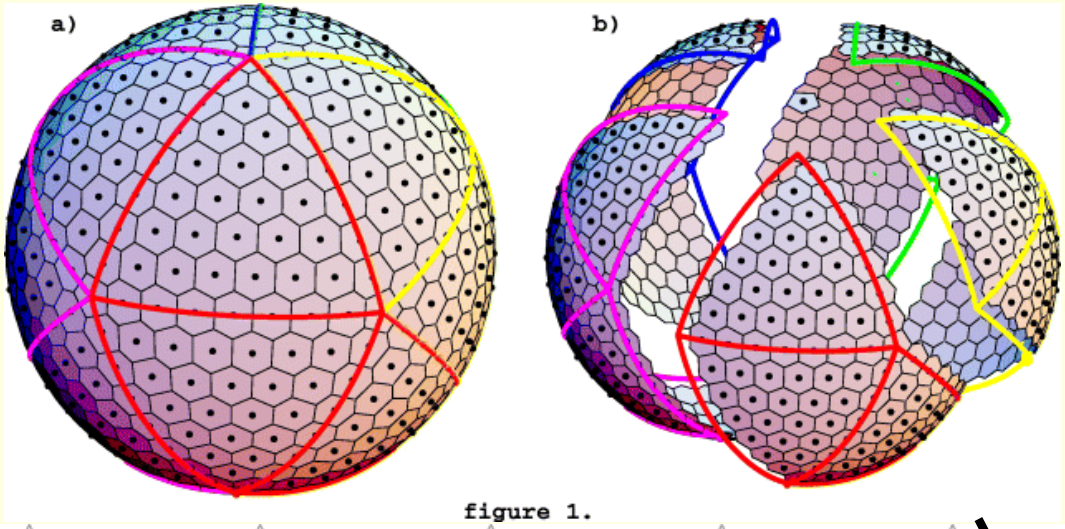


figure 1.

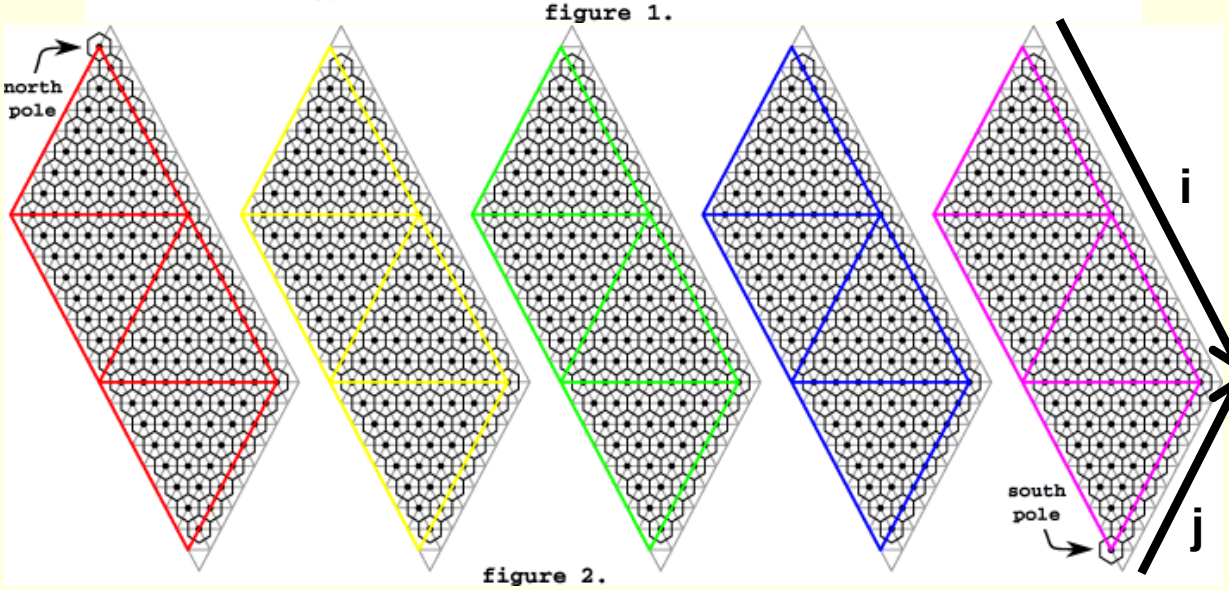
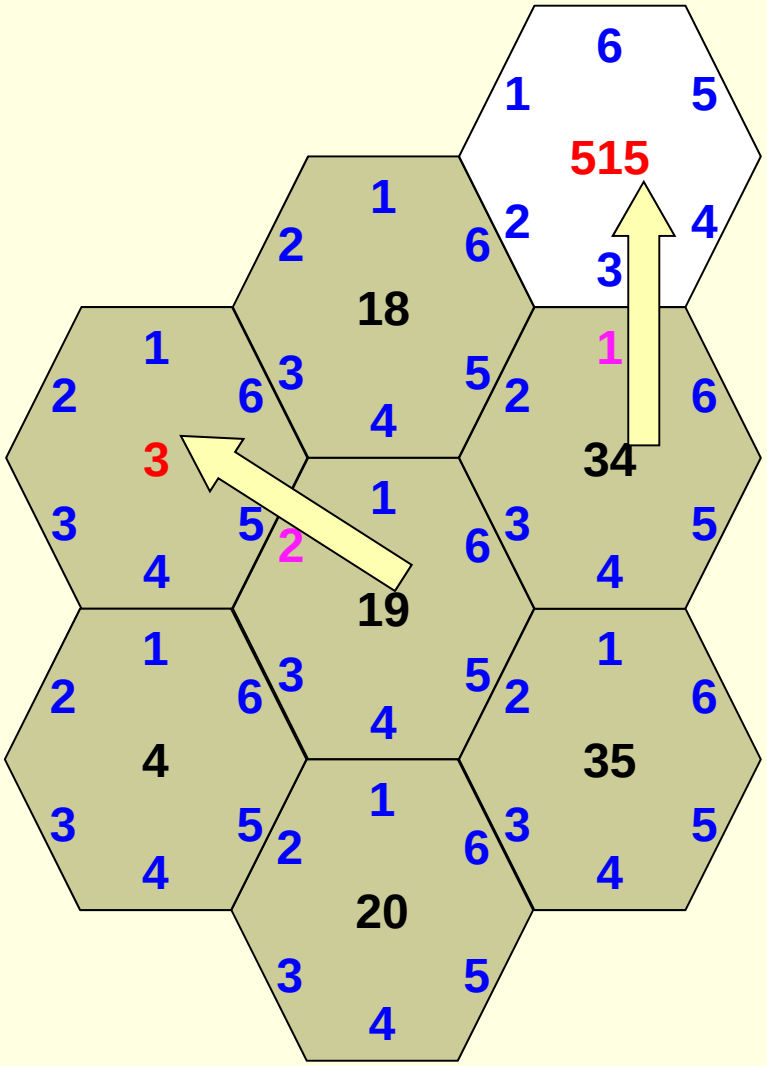


figure 2.

*figures from Ringler, Colorado State University

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
- Put vertical on inside for speed
- **Very compact code**

Simple Loop With Indirect Addressing

- Compute sum of all horizontal neighbors
  - nip = number of grid points in horizontal
  - nvl = number of vertical levels

```
xnsum = 0.0
do ipn=1,nip                ! Horizontal loop
  do isn=1,nprox(ipn)       ! Loop over edges (sides)
    ipp = prox(isn,ipn)     ! Index of neighbor across side “isn”
    do k=1,nvl              ! Vertical loop
      xnsum(k,ipn) = xnsum(k,ipn) + x(k,ipp)
    enddo
  enddo
enddo
```

Does Indirect Addressing Degrade CPU Performance?

- Compared FIM approach with several directly addressed schemes
  - Extensive performance testing shows that cost of FIM indirect addressing is <1%
    - IBM, Cray, Linux clusters
    - Directly-addressed inner “k” dimension is key
      - IJHPCA paper accepted
- This approach has now been adopted by NCAR for MPAS (Skamarock et. al.)
- … but what about GPUs?

FIM CPU Scaling: 10km 1-day on ORNL Cray (P. Johnsen)

MPI Tasks	Total Time (seconds)	Main Loop TIme	Main Loop Efficiency	Total Efficiency	Total 7-day Projected Efficiency
1821	5818	5508	1.00	1.00	1.00
3621	3211	2823	0.98	0.91	0.97
7221	1738	1445	0.96	0.84	0.94
14421	1245	776	0.90	0.59	0.83
21621	910	536	0.87	0.54	0.79
28821	720	366	0.95	0.51	0.84
36021	777	320	0.87	0.38	0.73

- 21 asynchronous write tasks included
  - “Main Loop” includes write times

Commercial GPU Fortran Compilers

- Directive-based compilers
  - CAPS HMPP 2.3.5
    - Generates CUDA-C and OpenCL
    - Supports NVIDIA and AMD GPUs
  - Portland Group PGI Accelerator 10.8
    - Supports NVIDIA GPUs
    - Previously used to accelerate WRF physics packages
- Other compilers
  - Portland Group “CUDA Fortran”
    - Not evaluated

Step-Wise Approach to GPU Parallelization

- GPU tools and debuggers are still relatively primitive
  - Bugs can be difficult to diagnose
- Create test case and establish tolerances
  - 50 time steps, 5-6 digits accuracy (single-precision)
  - Matches tolerance observed from a CPU compiler upgrade
- Make small changes and test after each
  - Much easier to find and fix errors

Step-Wise Approach to GPU Parallelization

- Add directives to move entire model state to GPU “global” memory
  - HMPP only so far
  - CPU becomes a “communication co-processor for the GPU”
    - Transfer data back to CPU only for I/O and MPI communication
- Then optimize computational performance
  - Fuse loops
  - Add explicit scalar temporary variables
  - And other things modern CPU compilers already do well…

Example: Loop Fusion

- Simplify code so compilers can recognize nested loops and parallelize both
  - We expect future compilers to do this

```
do ipn=1,nip
  do k=1,nvl
    u(k,ipn) = u(k,ipn) + …
  enddo
  do k=1,nvl
    v(k,ipn) = v(k,ipn) + …
  enddo
enddo
```

```
do ipn=1,nip
  do k=1,nvl
    u(k,ipn) = u(k,ipn) + …
    v(k,ipn) = v(k,ipn) + …
  enddo
enddo
```

Example: Scalar Temporary Variables

- Simplify code so NVIDIA nvcc compiler can identify values to be stored in register
  - Useful when a value is re-used several times
    - Some directive support in HMPP

```
do ipn=1,nip
  do k=1,nvl
    u(k,ipn) = u(k,ipn) + …
    uf(k,ipn) = z*u(k,ipn)
    …
  enddo
enddo
```

```
do ipn=1,nip
  do k=1,nvl
    utmp = u(k,ipn)
    utmp = utmp + …
    uf(k,ipn) = z*utmp
    …
    u(k,ipn) = utmp
  enddo
enddo
```

Initial Performance Results

- Optimize for both CPU and GPU
  - Some code divergence
  - Always use fastest code
- CPU = Intel Nehalem (2.8GHz)
- GPU = NVIDIA GTX280 “Tesla” or C2050 “Fermi”
- Compare HMPP with hand-tuned CUDA-C via “F2C-ACC” approach
  - Also, PGI results hot off the press…
- Many optimizations remain untried

Run Times for Single GPUs vs. Single Nehalem CPU Core

	Nehalem CPU Time	F2C-ACC CUDA-C Tesla GPU Time	F2C-ACC CUDA-C Fermi GPU Time	HMPP Tesla GPU Time	PGI Tesla GPU Time
Total	106.6	--	--	10.32	--
vdmints	50.6	2.56	2.11	2.35	4.78
vdmintv	23.3	0.94	0.76	0.99	0.97
flux	10.4	1.11	0.42	1.05	--
vdn	4.6	0.56	0.55	0.73	--
diag	4.0	0.094	0.096	0.085	0.077
force	3.4	0.11	0.10	0.19	
trisol	2.0	--	--	1.38	--
input/init	1.0	--	--	2.0	--
output	0.2	--	--	0.2	

Speedups for Single GPU vs. Single Nehalem CPU Core

	Nehalem CPU Time	F2C-ACC CUDA-C Tesla GPU Speedup	F2C-ACC CUDA-C Fermi GPU Speedup	HMPP Tesla GPU Speedup	PGI Tesla GPU Speedup
Total	106.6	--	--	10	--
vdmints	50.6	20	24	22	11
vdmintv	23.3	25	31	24	24
flux	10.4	9	25	10	--
vdn	4.6	8	8	6	--
diag	4.0	43	42	47	52
force	3.4	31	34	18	--
trisol	2.0	--	--	1.4	--

GFLOPS for Single GPU and Single Nehalem CPU Core

	Nehalem CPU GFLOPS	F2C-ACC CUDA-C Tesla GPU GFLOPS	F2C-ACC CUDA-C Fermi GPU GFLOPS	HMPP Tesla GPU GFLOPS	Computational Intensity
Total	2.00	--	--	20	1.05
vdmints	2.29	45	55	50	1.18
vdmintv	2.32	58	72	56	1.10
flux	1.54	14	39	15	0.74
vdn	0.35	3	3	2	0.31
diag	1.2	52	53	56	1.03
force	1.5	47	43	27	1.11
trisol	1.7	--	--	2.4	0.81

- Used PAPI performance counters on CPU
- ~10-15% of theoretical peak on CPU

Conclusions and Future Directions

- Encouraging initial results
  - Target: 30-40x may be achievable
  - Level of effort similar to OpenMP for HMPP and PGI directives
- Continue to improve GPU performance
  - Hand-tuned CUDA-C via F2C-ACC
  - Tuning options via commercial compilers
  - Test AMD “Firestream” GPUs

Conclusions and Future Directions

- Create more realistic test cases
  - Write output every 1000 time steps (not 50)
  - Fill GPU memory with more columns
  - Run much longer
    - Fraction of time spent on input/init is unrealistic
- Test NWP physics packages on GPU (Michalakes)
  - Double-precision
  - More complex “k” dependencies

Questions?