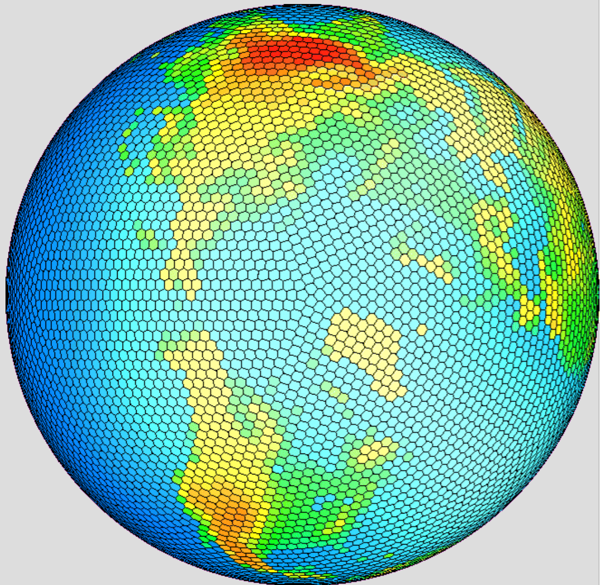
Using GPUs to Meet Next Generation Weather Model Computational Requirements

Mark Govett[1], Leslie Hart[1], Tom Henderson[2], Jacques Middlecoff[2], Craig Tierney[3]

[1]NOAA, ESRL/GSD
[2]CIRA, Colorado State University
[3]CIRES, University of Colorado at Boulder

Contact: craig.tierney@noaa.gov



Introduction

- Desire to improve resolution from 15km to ~1km
  - Weekly to Monthly weather forecasting
  - Improved hurricane prediction
- Moving from 15km to 3.75km => 64x
  - 200,000 cores!!!!
- Standard issues of power, cooling, reliability, scalability apply
- Goal 1 GPU == 10x 1 CPU socket

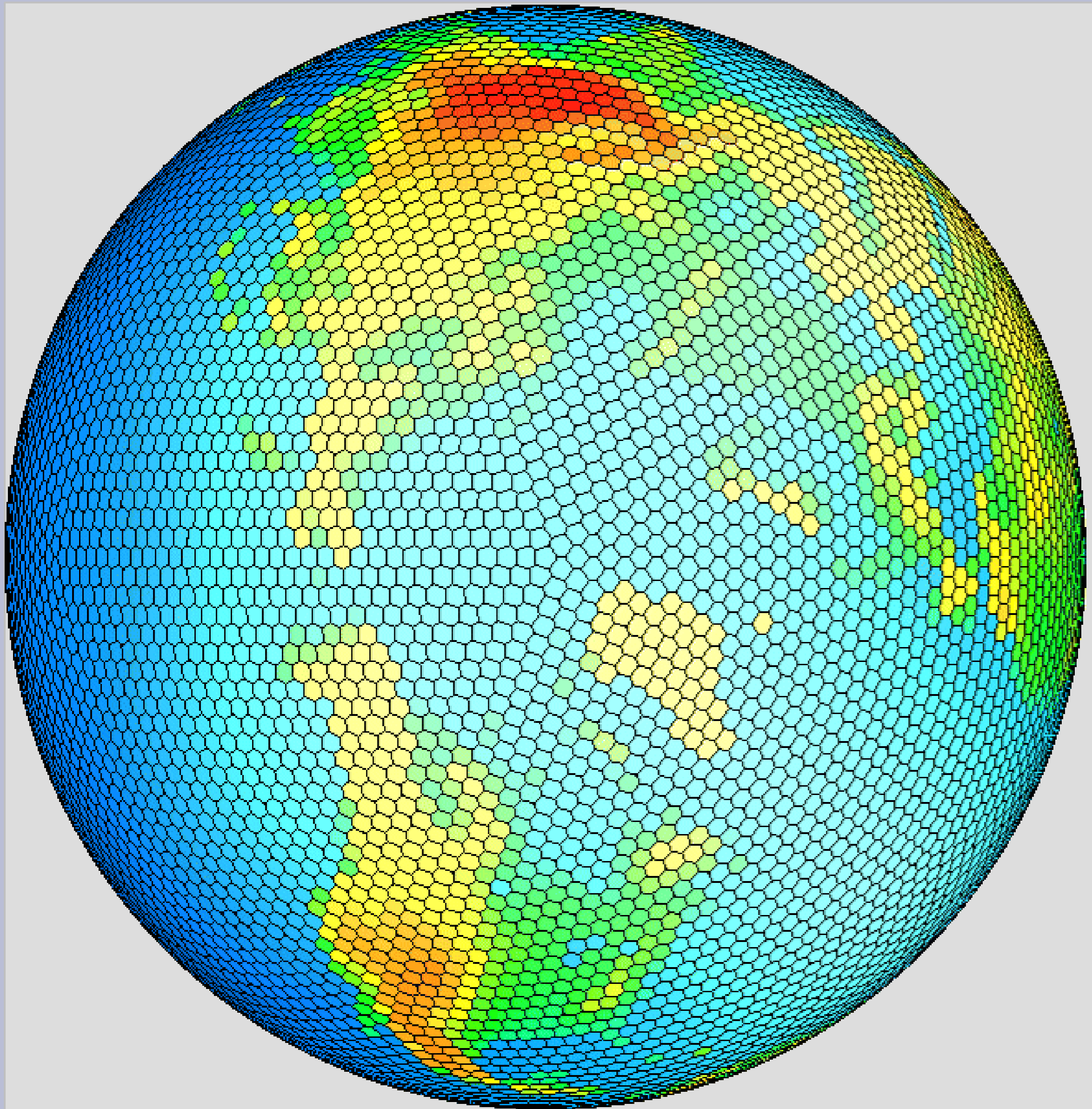
FIM

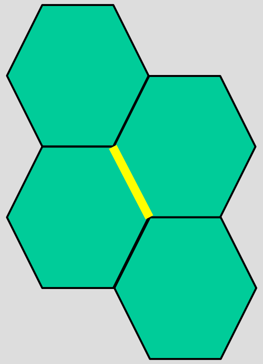
Flow-following finite-volume Icosahedral Model

- Icosahedral grid advantages
  - Small differences in cell areas
  - No singular “pole” cells
- Icosahedral grid disadvantages
  - Complex indexing
  - Hexagonal grid cells except for 12 pentagons
- Indirect addressing yields flexible compact code
  - Minimize performance impact with vertical-on-inside

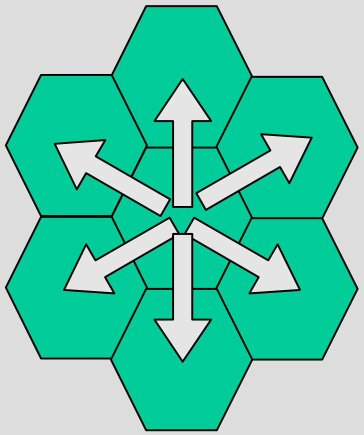
http://fim.noaa.gov

Icosahedral Grid





Side depends on four neighbors



First-order stencil

Working Assumptions

- Most code streams data through processor
  - Bandwidth limited, not computationally limited
- Data should be organized to maximize locality
  - Coalesced load reduce accesses to global memory
  - Indirect addressing considered not good structure for GPU applications
- For full model implementation, all working data on GPU, minimizing data transfers to/from host.

NOAA / ESRL Test System

- CPU
  - Dual socket, quad-core 2.8 GHz Intel Harpertown
  - 16 GB RAM
- GPU
  - Dual GTX280
- Compilations options
  - -O3 for all source
  - Using vectorization on CPU (-xT) provided no measurable benefit

FIM Performance Kernels

- Evaluate performance impact of FIM indirect addressing scheme on CPUs and GPUs
  - Test entire subroutines: load_ls, fctprs, force, getscl
    - 20-30% of total FIM run time at G=8 (r273)
      - Includes all horizontal dependences
  - Test direct and indirect addressing schemes:
    - “r1” (indirect), “ij” (direct), “ijk” (direct with vertical-on-outside), “r1swisn” (indirect with side loop-on-inside), various versions with loop fusion, etc.

Fortran -> CUDA translator

- Motivation
  - Open source f2c was not usable
    - Fortran 77, code was not meant to be read / modified
  - Eliminate time to hand translate code into C
- Language Support
  - Limited Fortran 90 (sufficient for FIM)
  - Fixed or Free format files
  - Uses M4 for F90 -> Interface, Macros
  - Generates C (for testing) or CUDA .cu
  - Optionally retains Fortran as comments

Eg. F2C --generate=CUDA --Free --comment   force.F90

Fortran - CUDA translator

- Output Code Generation
  - Uses macros to collapse multi-dim arrays
  - Generates Fortran-to-C routine
    - CUDA malloc, memCpy for each “intent (IN)” variable
    - Block and grid declarations
    - Call to GPU kernel
  - cudaMemcpy for every “intent (OUT)” variable
  - Fixing bugs, extending language support as needed

Table of Initial Results

Kernel	F90	CUDA	Speedup
force	6.5	0.30	23.3
fctprs	9.8	1.59	6.1
load_ls	7.1	0.44	14.9
getscl	1.8	0.20	8.2

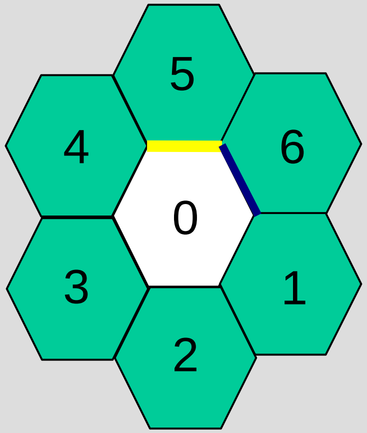
Note: Timings are based on kernel launch to synchronization of kernel completion only. Timings (F90,CUDA) are in milliseconds. Comparison is best CPU case to best GPU case, even if they are different sub-cases (ex: r1 vs. ij).

Load_Ls

Load_ls has horizontal dependencies, no exchanges, no vertical sums.
Only memory reuse is during access of neighboring edges.

	F90	Opt1	Opt2	Opt3
r1	7.2	1.37	1.20	
r1swisn	6.5	1.34	1.05	
r1swisnu	9.2	1.33		
ij	6.9	1.07	0.62	
ijk	7.1	2.15		
r1uisn	7.3	0.93	0.54	
r1uisnf	15.5	0.91	0.56	
r1uisna	15.7	1.01	0.47	0.44

Times are in milliseconds.



Opt1: Basic removal of vertical and horizontal loops
Opt2: Data reuse, store horizontal point mapping in constant memory
Opt3: Combine transformation coefficients into float4 for better data reuse, fuse loops, collapse conditionals

FCTPRS

FCTPRS is more complicated. Includes horizontal dependencies, halo calculations (requiring multiple exchanges), and vertical sums (scans). Divided into multiple kernels to solve inter-block synchronization.

	Opt1	Opt2	Opt3
FCTPRS_r1	5.1	5.6	6.1

Values are relative speed-up to Fortran Code.

Opt1:	Per thread execution of single horizontal and vertical point
Opt2:	Removed single execution for loops
Opt3:	Store/access horizontal mapping arrays from constant memory

Still to do: Eliminate global memory references, optimize scan algorithm, fuse loops, test other cases

Conclusions

"It's all about the memory references"

- Fortran to C translator does save time porting code to CUDA (order of magnitude!)
- K on outside appears to be bad idea
- Cases where structures have side index on inside appear to be fastest (at least for load_ls)
  - How do we pick which to use?
- Eliminate redundant memory accesses
  - Fuse/Unroll loops
  - Use register variables as temporaries

Next Steps

- Finish suite of existing tests
- Create similar test cases for code regions with vertical dependencies
- Recommend memory layout to scientists
- Continue to improve F2C
- Investigate CPU/GPU computational overlap (dynamics on GPU, physics on CPU)
- Investigate developing technologies (OpenCL, PGI Fortran support for GPGPU, etc.)