Meeting the Challenge of Many-core Architectures in Weather and Climate Models

Dr. Richard Loft
Director, Technology Development
Computational and Information Systems Laboratory
National Center for Atmospheric Research

CESM AMMW03
October 23, 2012
Heterogeneous, many-core co-processor node architecture

One or More Multicore Sockets

PCle Bus

to interconnect

NIC

Accelerator
What do you think of when you think of many-core systems?
In short, where are you on the Gartner Hype cycle?
NCAR Many-Core Workshop Organizers

William Sawyer (CSCS), Mark Govett (NOAA), Rich Loft (NCAR)
Left to right, front row: Ilene Carpenter (NREL), Chris Kerr (NOAA), Bill Putman (NASA)
2nd Programming weather, climate, and earth-system models on heterogeneous multi-core platforms workshop

• We need a shorter name!
• Held at NCAR for two days in mid September
• More weather than climate; mostly GPUs
• 30-40 invitees: we’d like to grow this to 50-60
• We’d love to see you there! Let me know if you’d like an invitation.
Typical NCAR Many-Core Workshop Themes

- Challenges
- Compilers
- Frameworks
- Tools
- Dycores
- Physics
- Communication
**Truth is, lots of groups are working on many-core systems!**

![NCAR Logo](logo.png)

---

Programming weather, climate, and earth-system models on heterogeneous multi-core platforms

September 7-8, 2011 at the National Center for Atmospheric Research in Boulder, Colorado

---

<table>
<thead>
<tr>
<th>GPU related talks (11+) that cover application software such as:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>NIM</td>
<td>WRF</td>
</tr>
<tr>
<td>Successes and Challenges using GPUs for Weather and Climate Models</td>
<td>Mark Govett, NOAA</td>
</tr>
<tr>
<td>Experience using FORTRAN GPU Compilers with the NIM</td>
<td>Tom Henderson, NOAA</td>
</tr>
<tr>
<td>GPU Acceleration of the RRTM in WRF using CUDA FORTRAN</td>
<td>Greg Ruetsch, NVIDIA</td>
</tr>
<tr>
<td>Lessons Learned adapting GEOS-5 GCM Physics to CUDA FORTRAN</td>
<td>Matt Thompson, NASA</td>
</tr>
<tr>
<td>Accelerated Cloud Resolving Model in Hybrid CPU-GPU Clusters</td>
<td>Jose Garcia, NCAR</td>
</tr>
<tr>
<td>Reworking Boundary Exchanges in HOMME for Many-Core Nodes</td>
<td>Ilene Carpenter, NREL</td>
</tr>
<tr>
<td>Performance optimizations for running an NWP model on GPUs</td>
<td>Jacques Middlecoff, NOAA</td>
</tr>
<tr>
<td>Rewrite of the COSMO Dynamical Core</td>
<td>Mueller / Gysi, SCS/CSCS</td>
</tr>
<tr>
<td>Experiences with the Finite-Volume Dynamical core and GEOS-5 on GPUs</td>
<td>Bill Putman, NASA</td>
</tr>
<tr>
<td>Progress in Accelerating CAM-SE</td>
<td>Jeff Larkin, Cray/ORNL</td>
</tr>
<tr>
<td>Porting the ICON Non-hydrostatic Dynamical Solver to GPUs</td>
<td>Will Sawyer, CSCS</td>
</tr>
</tbody>
</table>

---

Courtesy of Stan Posey, NVIDIA
The right and wrong way to measure, report, and think about many-core performance

How many here remember D.H. Bailey’s 1991 paper?
12 Ways to fool the masses with GPUs

1. Quietly rewrite the code in CUDA.
2. Quote only 32-bit performance results, not 64-bit results.
3. Ignore PCIe bus transfer times when reporting performance.
4. Present performance figures for an inner kernel, and then represent these figures as the performance of the entire application.
5. Scale up the problem size to supply the GPU with a massive number of threads, but omit any mention of this fact.
6. Quote performance results projected to a full system ignoring MPI overhead.
7. Compare your GPU results to a single core of an older Xeon.
12 Ways to fool the masses with GPUs

8. Quote performance in terms of peak flops, flops/$, or flops/watt, anything but time to solution.
9. Mutilate the algorithm used in the GPU implementation to match the architecture.
10. Don’t port code bits with lots of conditionals.
11. Sprinkle jargon like warps, blocks, waves into answers to questions.
12. If all else fails, show pretty pictures and animated videos, and don't talk about performance.
RRTMG is a correlated $k$-distribution radiative transfer technique and a two-stream method for multiple scattering.

### Results

<table>
<thead>
<tr>
<th></th>
<th>Previous Study</th>
<th>Current Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X5440</td>
<td>X5550</td>
</tr>
<tr>
<td></td>
<td>Tesla C1060</td>
<td>Tesla M2050</td>
</tr>
<tr>
<td></td>
<td>-fast</td>
<td>-fast</td>
</tr>
<tr>
<td>Overall time (ms)</td>
<td>703</td>
<td>479</td>
</tr>
<tr>
<td></td>
<td>Baseline</td>
<td>Baseline</td>
</tr>
<tr>
<td>% time in gasabs()</td>
<td>56%</td>
<td>52%</td>
</tr>
<tr>
<td></td>
<td>31%</td>
<td>21%</td>
</tr>
<tr>
<td>% time in rtrn()</td>
<td>28%</td>
<td>27%</td>
</tr>
<tr>
<td></td>
<td>46%</td>
<td>48%</td>
</tr>
</tbody>
</table>

**Tesla M2050/M2090 Baseline**: code from previous study, recompiled with 11.8
**Tesla M2050/M2090 Modified**: items in Recent Changes implemented

Factor of 14X improvement, right? No! Comparing 14 CPU’s to one!
Nice Speed-ups!

But compared to what?

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>GWD</td>
<td>17.6x</td>
</tr>
<tr>
<td>TURBULENCE</td>
<td>11.9x</td>
</tr>
<tr>
<td>CLOUD</td>
<td>23.7x</td>
</tr>
<tr>
<td>IRRAD</td>
<td>26.7x / 36.4x</td>
</tr>
<tr>
<td>SORAD</td>
<td>45.6x / 62.5x</td>
</tr>
</tbody>
</table>
OK, so how do we really program heterogeneous systems efficiently?
Spectrum of Execution Models

MultiCore-centric

- "Pure" MultiCore computing
  - Multi-core Hosted
  - Offload

- Codes with accelerated phases

- Coupled components

- Host as Coprocessor

ManyCore-centric

- "Pure" ManyCore computing
  - Reverse Offload
  - Many-Core Hosted

Main() { do { Dynamics(); MPI_*(); Physics(); } }

Atm() { do { Dynamics(); Physics(); Couple(); } }

μPhysics();

PCIe

Dynamics();

Ocn() { do { Dynamics(); Physics(); Couple(); } }

Main() { do { Dynamics(); MPI_*(); Physics(); Couple(); } }

Main() { do { Dynamics(); MPI_*(); Physics(); } }

Main() { do { Dynamics(); Physics(); } }

GPU direct

GPU direct
The PCIe straw: how CPU’s talk to GPU’s
Living with the straw:
Tianhe-1A overlapping case study

- 7168 nodes, node =
- Two 6-core Intel Westmeres + 1 M2050 Fermi
- 4702 DP PF peak (total) -> 1 petaflops Xeon
- Proprietary network TH-net
Overlapped FV Shallow Water Equations

- 13-point stencil
- Interp. Across patches
  - 1-d linear interpolation
- Discretize using cell-centered finite volume scheme
- Intergrade using second order TVD Runge-Kutta method

Divide each patch into N*N sub-blocks (here 4*4)

Mesh points (filled) and halos (empty) for a sub-block

Lin Gan
Center for Earth System Science
Adjustable partition CPU-GPU algorithm

- Divide each sub-block into
  - Outer part: \( n \) layers of points \( \rightarrow \) CPU computing
  - Inner part: without halo exchanging \( \rightarrow \) GPU computing
How Overlap Works

After Lin Gan
Overlapping GPU and CPU Computations

- Peta-scalable global SWE simulation, 809TFlops in double precision in 3750 nodes (45,000 CPU cores & 52,500 GPU cores)
- Adjustable partition between CPUs/GPUs
- Effective comm.-comp. overlap to hide comm. cost
- “Pipe-flow” scheme to arrange message-passing
- Systematic optimizations on CPU and CUDA code, 130 speedup

17% of peak!

After Lin Gan
Tian-he-1A Weak Scaling Study

<table>
<thead>
<tr>
<th>Number of nodes</th>
<th>384</th>
<th>1536</th>
<th>2400</th>
<th>3750</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (s)</td>
<td>56.9</td>
<td>14.4</td>
<td>9.4</td>
<td>5.9</td>
</tr>
<tr>
<td>Efficiency</td>
<td>1.00</td>
<td>0.99</td>
<td>0.97</td>
<td>0.98</td>
</tr>
<tr>
<td>Agg. Tflops</td>
<td>84.5</td>
<td>335.1</td>
<td>513.4</td>
<td>809.6</td>
</tr>
</tbody>
</table>

G2C/C2G as the non-overlapping part is very small

Successful comm.-comp. overlap
GPU’s and Dynamics

• Unstructured 3D dynamics solution:
  – thread over z
  – block over x, y

• i.e. write code thusly:
  
  do i = 1, npts
    do k=1, nlev
      a(k,i) = b(k,ia(i,1))+... ! ia encodes structure
    end do
  end do

• Some exceptions to this: vertical summations
**NIM dycore Fermi GPU vs. Single/Multiple Westmere CPU cores, “G5-L96”**

<table>
<thead>
<tr>
<th>NIM routine</th>
<th>CPU 1-core Time (sec)</th>
<th>CPU 6-core Time (sec)</th>
<th>F2C-ACC GPU Time (sec)</th>
<th>HMPP GPU Time (sec)</th>
<th>PGI GPU Time (sec)</th>
<th>F2C-ACC Speedup vs. 6-core CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>8654</td>
<td>2068</td>
<td>449</td>
<td>--</td>
<td>--</td>
<td>4.6</td>
</tr>
<tr>
<td>vdmints</td>
<td>4559</td>
<td>1062</td>
<td>196</td>
<td>192</td>
<td>197</td>
<td>5.4</td>
</tr>
<tr>
<td>vdmintv</td>
<td>2119</td>
<td>446</td>
<td>91</td>
<td>101</td>
<td>88</td>
<td>4.9</td>
</tr>
<tr>
<td>flux</td>
<td>964</td>
<td>175</td>
<td>26</td>
<td>24</td>
<td>26</td>
<td>6.7</td>
</tr>
<tr>
<td>vdn</td>
<td>131</td>
<td>86</td>
<td>18</td>
<td>17</td>
<td>18</td>
<td>4.8</td>
</tr>
<tr>
<td>diag</td>
<td>389</td>
<td>74</td>
<td>42</td>
<td>33</td>
<td>--</td>
<td>1.8</td>
</tr>
<tr>
<td>force</td>
<td>80</td>
<td>33</td>
<td>7</td>
<td>11</td>
<td>13</td>
<td>4.7</td>
</tr>
</tbody>
</table>

*after Tom Henderson, ESRL*
Recall NICAM GPU Acceleration got 4.5X

7 Tflops on 217 nodes

6.3% of peak

Performance, TFLOPS

the number of nodes

11/8/2012

After Satoshi Matsuoka, TI Tech
What can be achieved:
ASUCA NWP on Tsubame 2.0

Tsubame 2.0
Tokyo Institute of Technology

- 1.19 Petaflops
- 4,224 Tesla M2050 GPUs

2.196 DP Petaflops

3990 Tesla M2050s
145.0 Tflops SP
76.1 Tflops DP

3.7% of DP peak

After Satoshi Matsuoka, TiTech
GPUs and physics
Recall 3D dynamics solution: thread over z, block over x, y
i.e. write $a(k,ia(i))$
But physics has k dependencies...

OK, so how do you do physics on GPU’s?
Solution to GPU-izing Physics

• Transpose and “chunking”: \( a(i, nc, k, ...) \)

```fortran
!ACC$REGION(<chunksize>,<nchunks>,<dynvars,physvars:none>) BEGIN
  do n=1,nvars_dyn2phy ! Loop over dyn vars needed in phy
    do k=1,nz ! Vertical loop
      !ACC$DO PARALLEL (1)
      do nc=1,nchunks ! Chunksize*nchunks >= nip
        !ACC$DO VECTOR (1)
        do i=1,chunksize ! 128 is a good # for chunksize
          ipn = min (ipe, ips + (c-1)*chunksize + (i-1))
          physvars(i,nc,k,n) = dynvars(k,ipn,n)
        end do
      end do
    end do
  end do
!ACC$REGION END
```

After Jim Rosinski, ESRL
Dyn ⇐⇒ Phys
Transpose Performance in NIM

After Jim Rosinski, ESRL
Physics and conditionals

• NVIDIA GPUs use conditional execution to handle branch divergence within the SIMD group ("warp").

• In an if..else example, both branches get executed by every thread in the diverging warp, but those threads which don't follow a given branch are flagged and perform a null op instead.

Solution: Gather/scatter or offload to CPU
Typical Porting Status: GEOS-5

GEOS-5 GCM Physics Converted

Due to Bill Putman, NASA
Viability of the code refactoring strategies for many-core processors

Five requirements of a viable porting strategy:

- Identify parallelization paradigm
- Single source – especially in volatile sections
- Portability – avoid/isolate vendor specifics
- Verifiability – bit-for-bit CPU-GPU results ideal
- Readability -
Current Pathways from Fortran to Many-core Hardware

Due to Mark Govett, NOAA
GPU Fortran Compilers

• **Commercial directive-based compilers**
  – CAPS HMPP
    • Generates CUDA-C and OpenCL
    • Supports NVIDIA and AMD GPUs
  – Portland Group PGI Accelerator
    • 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

• **Directive-based OpenACC compilers**
  – Cray, PGI, HMPP all support
  – Gotcha: calling a function from inside a parallel region
F2C-ACC Source-to-Source Translator to CUDA (Input is Fortran Source)

- Mark Govett (NOAA)
- Directive-based
- Freely available to for download
- Used for NIM model dynamics
- "Kinda supported" until vendors catch up

```fortran
!ACC$REGION(<nz>,<ipe-ips+1>,<ur, vr, ..> BEGIN
!ACC$DO PARALLEL(1)
do ipn=ips, ipe
!ACC$DO VECTOR(1)
do k=1,nz
  urs(k,ipn) = ur(k,ipn)
  vrs(k,ipn) = vr(k,ipn)
 trs(k,ipn) = trp(k,ipn)
  rps(k,ipn) = rp(k,ipn)
end do !k loop
!ACC$THREAD(0)
  wrs(0,ipn) = wr(0,ipn)
!ACC$DO VECTOR(1)
do k=1,nz
  wrs(k,ipn) = wr(k,ipn)
end do !k loop
end do !ipn loop
!ACC$REGION END
```
OpenACC SW-DG Gradient Calculation

Initialize GPU data ->
$\texttt{acc data copy(flx,fly) create(grad) copyin(gw,der,delta,dt)}$
$\texttt{do it=1,nit}$

OpenMP directive ->
$\texttt{omp parallel do shared(flx,fly,grad)}$
$\texttt{acc kernels}$
$\texttt{acc loop gang(ngangs) vector(neblk)}$
$\texttt{do ie=1,nelem}$
$\texttt{acc loop vector(npts) private(s1, s2, i, j, k, l)}$

Block elements ->
$\texttt{do ii=1,npts}$
$\texttt{k=MODULO(ii-1,nx)+1}$
$\texttt{l=(ii-1)/nx+1}$
$\texttt{s2 = zero}$
$\texttt{do j = 1, nx}$
$\texttt{s1 = zero}$
$\texttt{do i = 1, nx}$
$\texttt{s1 = s1 + (delta(l,j)*flx(i+(j-1)*nx,ie)*der(i,k) + &}$
$\texttt{delta(i,k)*fly(i+(j-1)*nx,ie)*der(j,l))*gw(i)}$
$\texttt{end do ! i loop}$
$\texttt{s2 = s2 + s1*gw(j)}$
$\texttt{end do ! j loop}$
$\texttt{grad(ii,ie) = s2}$
$\texttt{end do ! i1 loop}$
$\texttt{end do ! ie}$
$\texttt{acc end kernels}$
$\texttt{omp end parallel do}$

Warp element pts (16) ->

I’d love to tell you this gives a 10x speed up! over a single core of a Xeon Westmere, but I won’t.
CESM Many-Core Evaluation

CESM – Fortran+MPI+OpenMP
Works well on 4-6 cores/socket

Pipeline not meant to suggest that architectural investigations must occur sequentially…

Refactor code for higher thread parallelism…

Blue Gene/Q
Test optimal mixture of processes and threads per socket

Two Memory Spaces

Cuda Fortran/OpenACC Directives

Xeon Phi
Introduce overlap + vectorization :

Nvidia GPU
Evaluate overlap + directive based acceleration :

Pipeline not meant to suggest that architectural investigations must occur sequentially…
I hope I’ve shown you that there is something important going on here... we just don’t know what it is yet!

Thanks!
Let’s look at the current candidates...

IBM BG/Q
Cores: 16 + 2
Multithread: 4-way
Coprocessor: no
Boot Linux: yes

Intel Knights Corner
Cores: NDA
Multithread: NDA
Coprocessor: yes
Boot Linux: yes

NVIDIA Fermi->Kepler
DP Cores: 512->960
Multithread: 32-way
Coprocessor: yes
Boot Linux: no
Blue Gene/Q: System on a Chip

Vital Statistics
- 45 nm technology
- clocked @ 1.6 GHz
- Quad FPU’s (8 flops/clock tick)
- 16 user cores +
  - 1 system core
  - 1 spare core
- 128 flops/clock tick
- 204.8 GFLOPS/chip
- 55W/chip
- 268 pJ/flop
- 42.7 GB/sec

This could be considered the Prius of modern day multi-processor systems!
NVIDIA Kepler GK110 Architecture

- 15 SMX units
- Each SMX has 64 DP; 192 SP; 32 load/store units
**Changes NVIDIA’s Kepler relative to Fermi**

### SMX Balance of Resources

<table>
<thead>
<tr>
<th>Resource</th>
<th>Kepler GK110 vs Fermi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating point throughput</td>
<td>2-3x</td>
</tr>
<tr>
<td>Max Blocks per SMX</td>
<td>2x</td>
</tr>
<tr>
<td>Max Threads per SMX</td>
<td>1.3x</td>
</tr>
<tr>
<td>Register File Bandwidth</td>
<td>2x</td>
</tr>
<tr>
<td>Register File Capacity</td>
<td>2x</td>
</tr>
<tr>
<td>Shared Memory Bandwidth</td>
<td>2x</td>
</tr>
<tr>
<td>Shared Memory Capacity</td>
<td>1x</td>
</tr>
</tbody>
</table>
Hyper-Q: GPU resource sharing

Hyper-Q Helps to Better Utilize the GPU

- Streams from multiple CPU processes can execute concurrently
- Use as many MPI ranks as in CPU-only case => smaller impact of CPU work
- Particularly interesting for strong scaling