Using GPUs to Run Weather Prediction Models

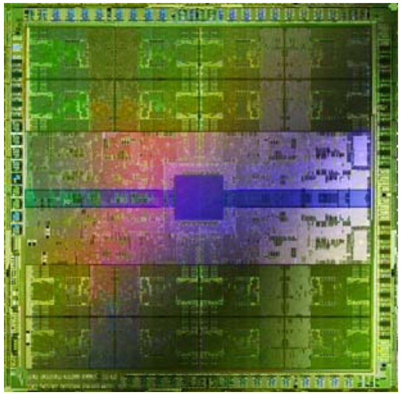
Mark Govett
Tom Henderson, Jacques Middlecoff,
Paul Madden, Jim Rosinski

November 2010

GPU / Multi-core Technology

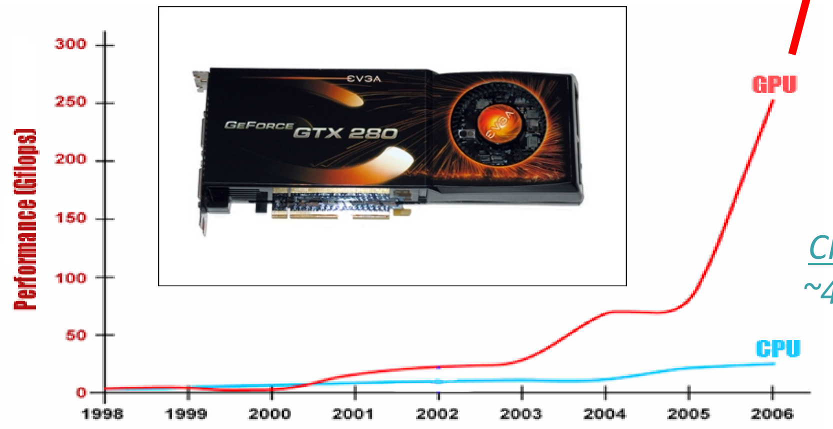
- NVIDIA: Fermi chip first to support HPC
  - Formed partnerships with Cray, IBM on HPC systems
  - #1, #3 systems on TOP500 (Fermi, China)
- AMD/ATI: Primarily graphics currently
  - #7 system on TOP500 (AMD-Radeon, China)
  - Fusion chip in 2011 (5 TeraFlops)
- Intel: Knights Ferry (2011), 32-64 cores

NVIDIA: Fermi (2010)



✧ 1.2 TeraFlops
✧ 8x increase in double precision
✧ 2x increase in memory bandwidth
✧ Error correcting memory

NVIDIA: Tesla (2008)



GPU: 2008
933Gflops
150W

CPU:2008
~45 Gflops
160W

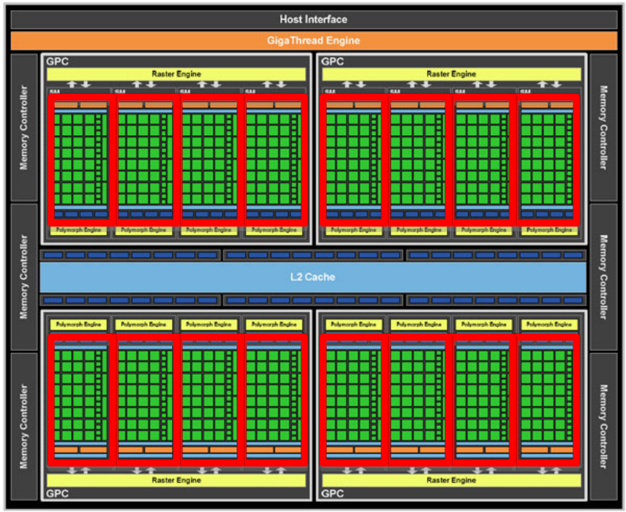
CPU – GPU Comparison

CHIP TYPE	CPU Nahalem	GPU NVIDIA Tesla	GPU NVIDIA Fermi
Cores	4	240	480
Parallelism	Medium Grain	Fine Grain	Fine Grain
Performance Single Precision Double Precision	85 GFlops	933 GFlops 60 GFlops	1040 GFlops 500 GFlops
Power Consumption	90-130W	150W	220W
Transistors	730 million	1.4 bilion	3.0 billion

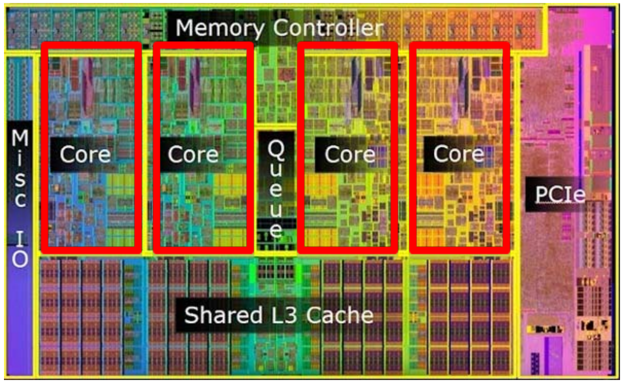
CPU – GPU Comparison

- CPUs focus on per-core performance
  - Chip real estate devoted to cache, speculative logic
  - 4 cores
- GPUs focus on parallel execution
  - Fermi: 512 cores, 16 SM

GPU: Fermi (2010)



CPU: Nahalem (2009)



Next Generation Weather Models

- Models being designed for global cloud resolving scales (3-4km)
- Requires PetaFlop Computers

DOE Jaguar System

- 2.3 PetaFlops
- 250,000 CPUs
- 284 cabinets
- 7-10 MW power
- ~ $100 million
- **Reliability in hours**

Equivalent GPU System

- 2.3 PetaFlop
- 2000 Fermi GPUs
- 20 cabinets
- 1.0 MW power
- ~ $10 million
- **Reliability in weeks**

- Large CPU systems (>100 thousand cores) are unrealistic for operational weather forecasting
  - Power, cooling, reliability, cost
  - Application scaling

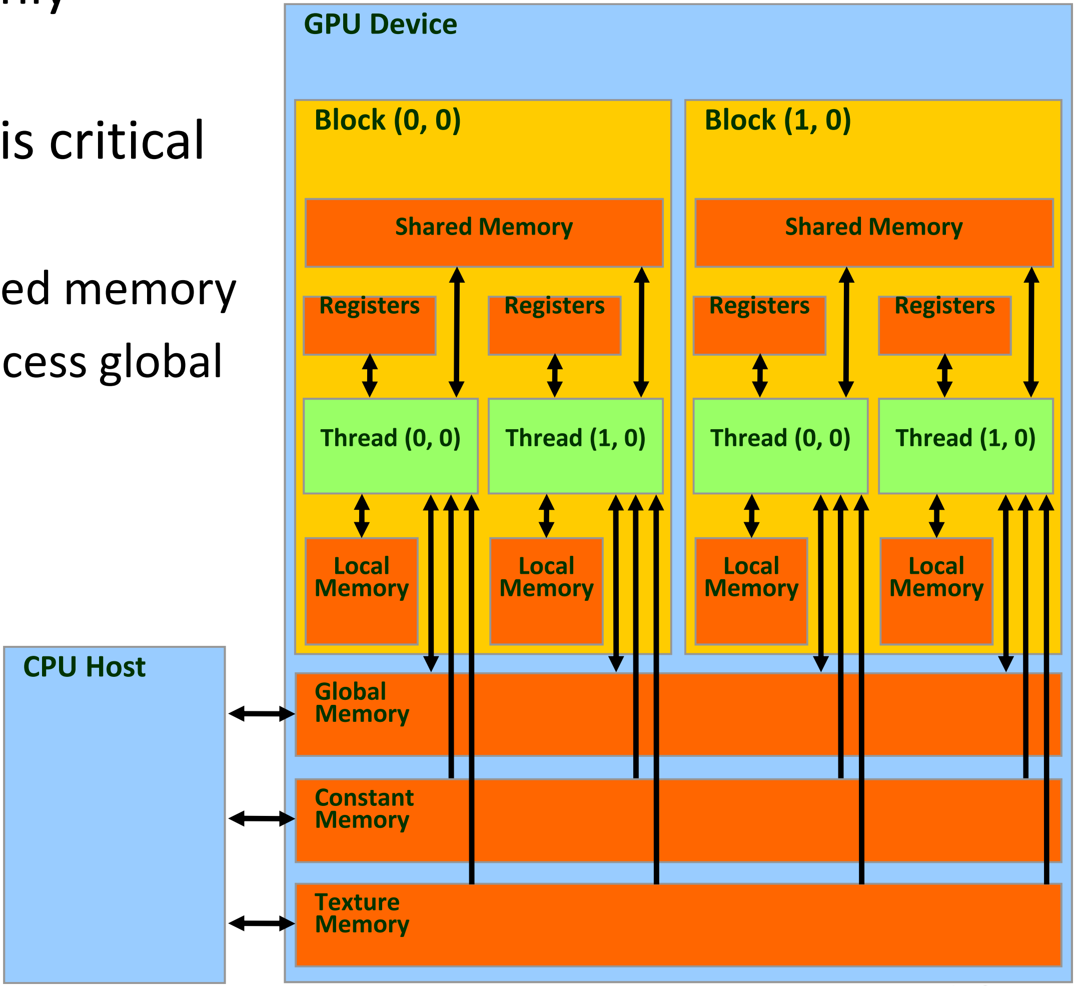
Valmont
Power Plant
~200 MegaWatts
Boulder, CO

Application Performance

- 20-50x is possible on highly scalable codes
- Efficient use of memory is critical to good performance
  - 1-2 cycles to access shared memory
  - Hundreds of cycles to access global memory

Memory	Tesla	Fermi
Shared	16K	64K
Constant	16K	64K
Global	1-2GB	4-6GB

GPU Multi-layer Memory

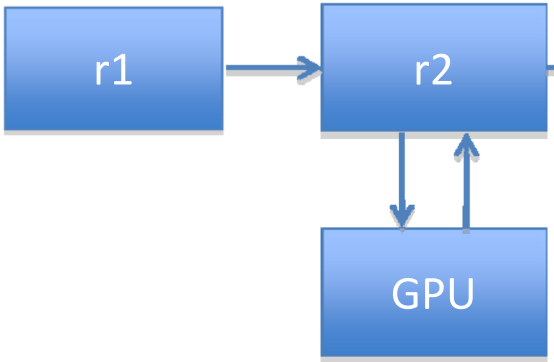


Programming GPUs

- Challenging
- Languages
  - – CUDA-C: available from NVIDIA
  - – OpenCL: industry standard (NVIDIA, AMD, Apple, etc)
  - – Fortran: PGI, CAPS, F2C-ACC compilers
- Fine grain (loop level) parallelism
  - – Code modifications needed to get good performance
  - – Code restructuring may also be required

Execution Flow-control

(Accelerator Approach)



r1

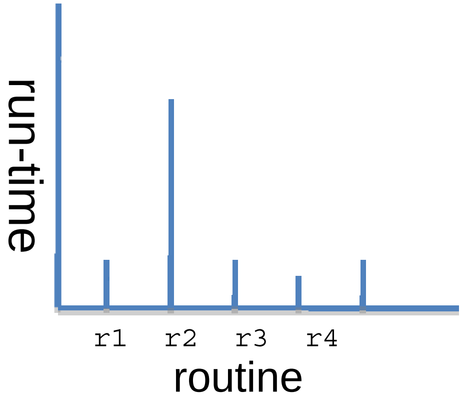
r2

r3

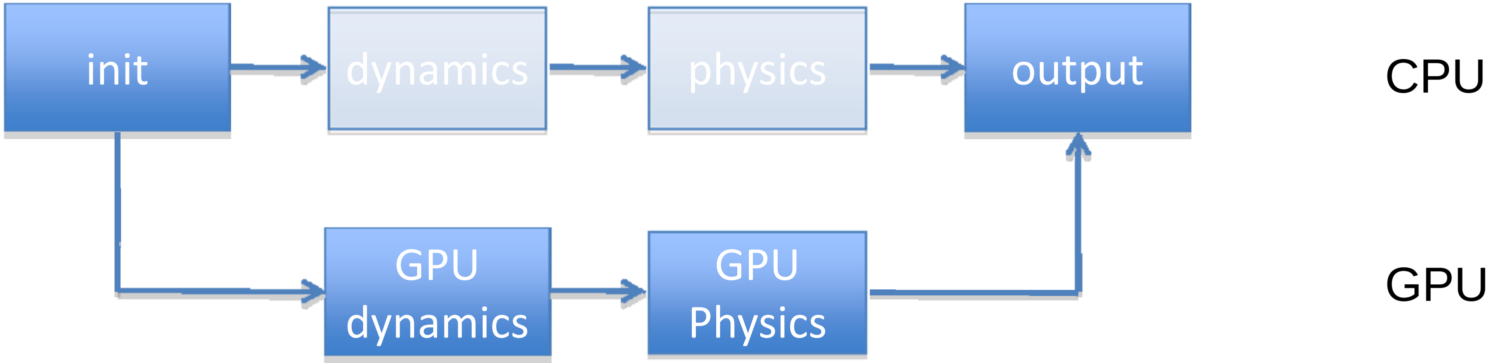
r4

GPU

- Copy between CPU and GPU is non-trivial
  - Performance benefits can be overshadowed by the copy
  - WRF demonstrated ~6x for one subroutine including data transfers (Michalakes, 2009)
    - ~ 10x without data transfers



Execution Flow-control
(run everything on GPUs)



– Eliminates copy every model time step

– CPU-GPU copies only needed for input /output, inter-process communications

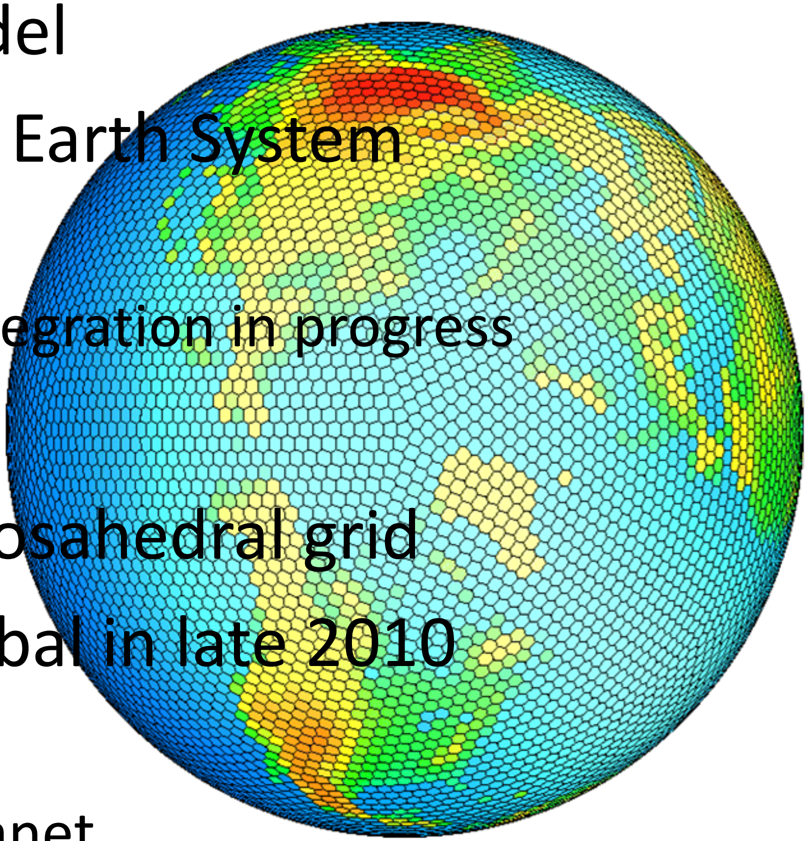
– JMA: ASUCA model, reported a 70x performance improvement

  - Rewrote the code in CUDA

Non-hydrostatic Icosahedral Model (NIM)

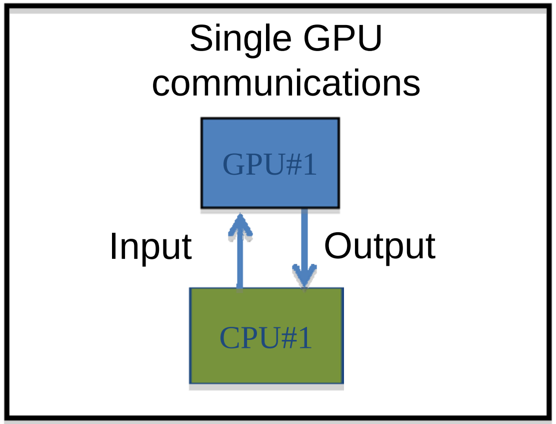
(Lee, MacDonald)

- – Global Weather Forecast Model
- – Under development at NOAA Earth System Research Laboratory
  - Dynamics complete, physics integration in progress
- – Non-hydrostatic
- – Uniform, hexagonal-based, icosahedral grid
- – Plan to run tests at 3.5km global in late 2010
  - Cloud resolving scale
  - Model validation using AquaPlanet



Code Parallelization (2009)

- Developed the Fortran-to-CUDA compiler (F2C-ACC)
  - Commercial compilers were not available in 2008
  - Converts Fortran 90 into C or CUDA-C
  - Some hand tuning was necessary
- Parallelized NIM model dynamics
  - Tesla Chip, Intel Harpertown (2008)
  - Result for a single GPU
  - Communications only needed for I/O

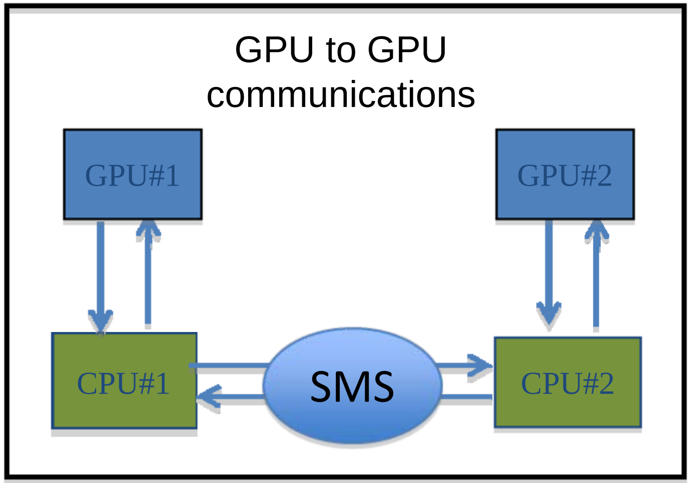


NIM Dynamics (version 160)

Resolution	HorizPts	Harpertown	Tesla	Nehalem	Fermi
G4-480km	2562	2.13	0.079 (26.9)	1.45	0.054 (26.7)
G5-240km	10242	8.81	0.262 (33.5)	5.38	0.205 (26.2)

Model Parallelization (2010)

- Updated NIM Model Parallelization
  - Active model development
  - Code optimizations on-going
- Run with Fermi
- Evaluate Fortran GPU compilers
  - Use F2C results as benchmark
- Run on Multiple GPUs
  - Modified F2C-ACC GPU compiler
  - Uses MPI-based Scalable Modeling System (SMS)
  - Testing on 10 Tesla GPUs

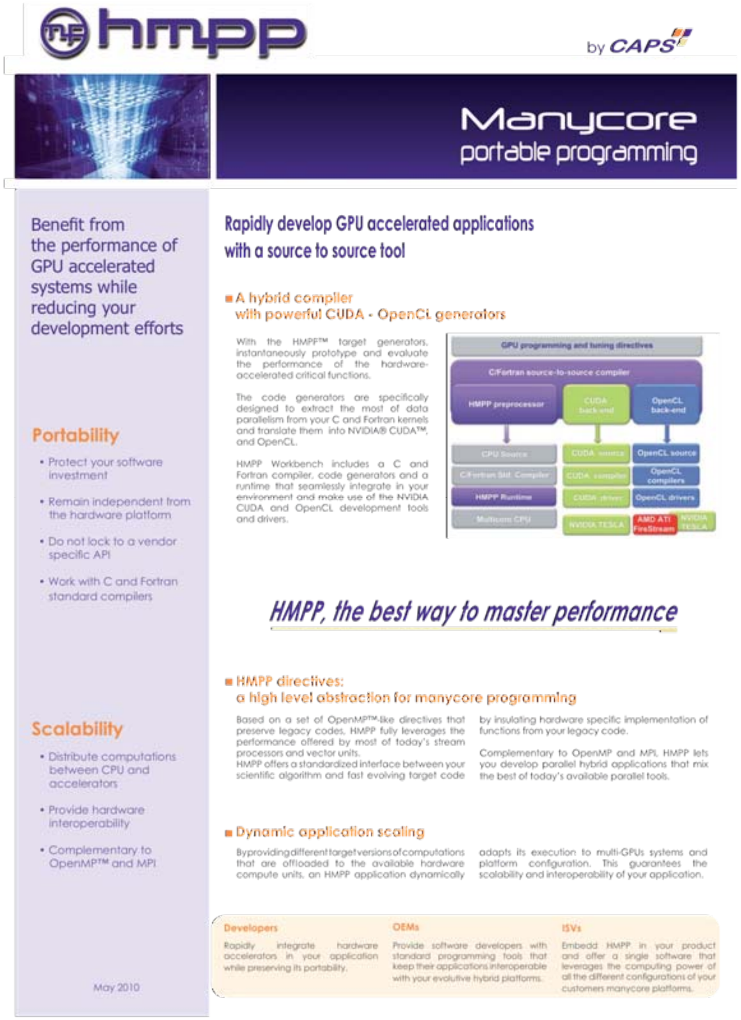

GPU to GPU communications

Fortran GPU Compilers

- General Features
  - Do not support all Fortran language constructs
  - Converts Fortran into CUDA for further compilation
- CAPS – HMPP
  - Extensive set of parallelization directives to guide compiler analysis and optimization
  - Optionally generates OpenCL
- PGI
  - **ACCELERATOR** – directive-based accelerator
  - **CUDA Fortran** – Fortran + language extensions to support Kernel calls, GPU memory, etc
- F2C-ACC
  - Developed at NOAA for our models
  - Requires hand tuning for optimal performance

CAPS-HMPP Compiler

- Multi-core Fortran
  - CUDA, OpenCL code generation
- Extensive set of directives
  - Parallelization
  - Optimization
- A minimal set of directives to get a working code
  - Compiler will do what it safely can and provide diagnostic information
- Our evaluation began ~4 months ago
  - Good documentation & support



hmpp by CAPS

Manycore portable programming

Benefit from the performance of GPU accelerated systems while reducing your development efforts

Portability

- Protect your software investment
- Remain independent from the hardware platform
- Do not lock to a vendor specific API
- Work with C and Fortran standard compilers

Scalability

- Distribute computations between CPU and accelerators
- Provide hardware interoperability
- Complementary to OpenMP™ and MPI

May 2010

Rapidly develop GPU accelerated applications with a source to source tool

A hybrid compiler with powerful CUDA - OpenCL generators

With the HMPP™ target generators, instantaneously prototype and evaluate the performance of the hardware-accelerated critical functions.

The code generators are specifically designed to extract the most of data parallelism from your C and Fortran kernels and translate them into NVIDIA® CUDA™, and OpenCL.

HMPP Workbench includes a C and Fortran compiler, code generators and a runtime that seamlessly integrate in your environment and make use of the NVIDIA CUDA and OpenCL development tools and drivers.

HMPP, the best way to master performance

HMPP directives: a high level abstraction for manycore programming

Based on a set of OpenMP™-like directives that preserve legacy codes, HMPP fully leverages the performance offered by most of today's stream processors and vector units.
HMPP offers a standardized interface between your scientific algorithm and fast evolving target code by insulating hardware specific implementation of functions from your legacy code.

Complementary to OpenMP and MPI, HMPP lets you develop parallel hybrid applications that mix the best of today's available parallel tools.

Dynamic application scaling

Byprovidingdifferenttargetversionsofcomputations that are offloaded to the available hardware compute units, an HMPP application dynamically adapts its execution to multi-GPUs systems and platform configuration. This guarantees the scalability and interoperability of your application.

Developers

Rapidly integrate hardware accelerators in your application while preserving its portability.

OEMs

Provide software developers with standard programming tools that keep their applications interoperable with your evolutive hybrid platforms.

ISVs

Embedd HMPP in your product and offer a single software that leverages the computing power of all the different configurations of your customers manycore platforms.

http://www.caps-entreprise.com

PGI Directives

- Directives are placed directly in the code body
  - Define an accelerated region
    - !$acc region([copy | copyout | copyin]) begin
    - !$acc region end
  - User defines loop level parallelism
    - !$acc do [vector | parallel | unroll]

      vector = thread,   parallel = threadblock
  - Define data resident on the GPU
    - !$acc data region (copy | copyin | copyout)

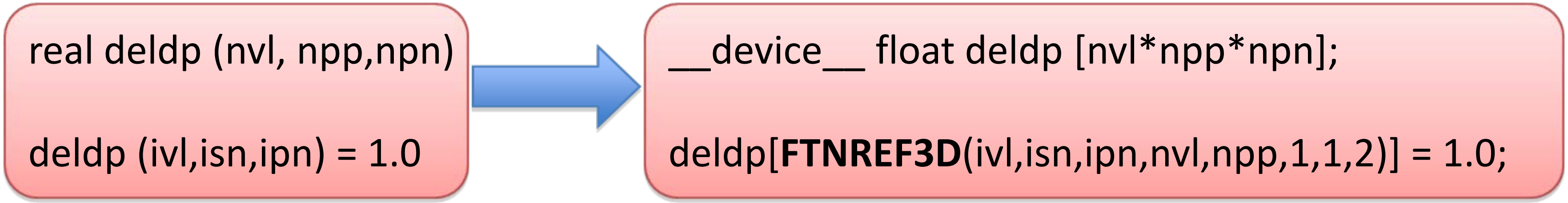
F2C-ACC Compiler

- Developed at NOAA to support our Fortran codes
- Parsing
  - All standard language features for Fortran 77, 90, 95
- Code Translation (added as needed)
  - if, do, assignments, declarations, modules, include files
    - Does not support I/O, common blocks, derived types
  - Handles 98% of code translations (serial & parallel)
    - Support for thread optimizations and GPU memory types needed

```
F2C-ACC ERROR:   “fctprs.f90” line 23:10” Language construct not
currently supported
```

Code Generation

- Uses Macros to resolve array references



```
real deldp (nvl, npp,npn)

deldp (ivl,isn,ipn) = 1.0
```

```
__device__ float deldp [nvl*npp*npn];

deldp[FTNREF3D(ivl,isn,ipn,nvl,npp,1,1,2)] = 1.0;
```

- Generates Fortran-to-C driver routine
  - cudaMemcpy for each variable (intent IN, OUT, INOUT)
  - Block and grid declarations
  - Call to GPU kernel routines
- Generates Kernel Routines for each ACC$REGION BEGIN / END pair
  - Declares each variable used in the kernel
  - Consistency checks between kernels (proper intent)

F2C-ACC Directives

- Directives are required for kernel generation
  - Supports multiple kernels in a routine
- Define Accelerated regions

```
ACC$REGION (< Threads>, < Blocks >) BEGIN
 ACC$REGION END
```

- Define loop level parallelism

```
ACC$DO VECTOR( dim )        - thread level parallelism
ACC$DO PARALLEL (dim )      - block level parallelism
```

- Data movement

```
ACC$REGION (<Threads>, <Blocks>[,<DATA>]) BEGIN
```

Achieving High Performance

#1 Optimize the CPU code

- Modifications often help the GPU performance too
  - Performance Profiling identified a matrix solver that accounted for 40 percent of the runtime
    - Called 150,000 per timestep (G4), 3.6 million for 3.5KM
    - Replaced BLAS routine with hand-coded solution resulting in a 3x performance increase for the entire model
    - Loop unrolling improved overall performance by another 40 percent
      - » Developed test kernels to study CPU & GPU performance
- Organize arrays so the inner dimension will be used for thread calculations
  - Improve loads & stores from GPU global memory
- Re-order calculations to improve data reuse

Achieving High Performance

#2 Optimize GPU Kernels (major issues)

- Use the performance profiler to identify bottlenecks
  - Occupancy
    - Largely determined by registers and shared memory usage
  - Coalesced Loads and Stores
    - Data alignment with adjacent threads can yield big performance gains
      - » Eg. Threading on “k”, for” a(k,I,j)”
  - Use Shared Memory where there is data reuse
    - Must be at least > 3 to have benefit (2 loads & stores are needed to move data from global to local memory

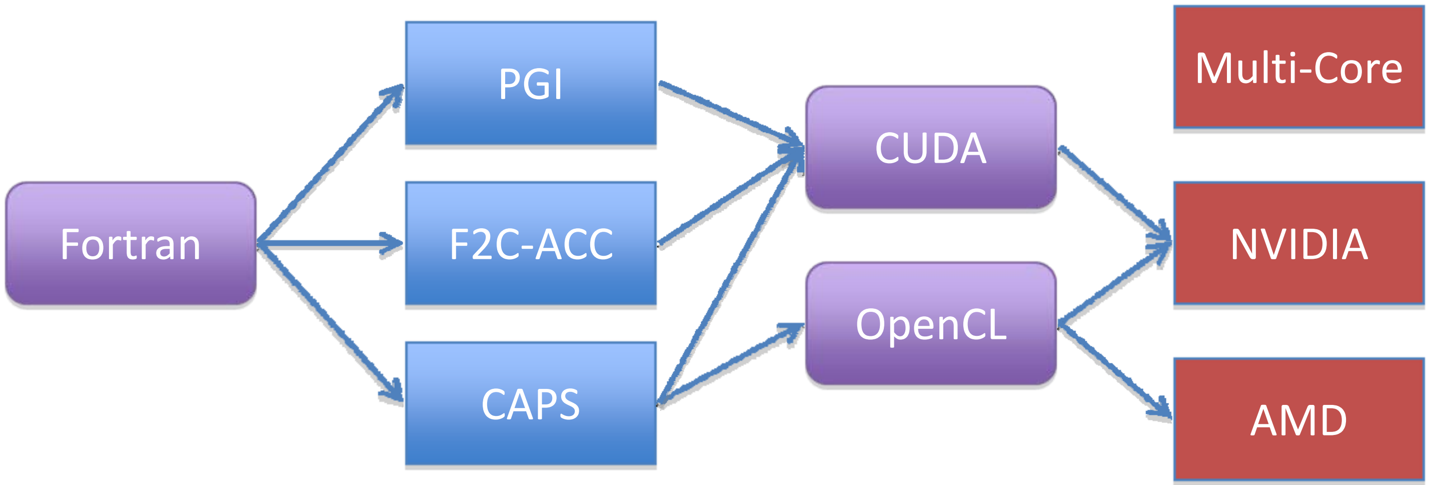
Run Times for Single GPU vs. Single Nehalem CPU Core (2562 points)

Results for F2C-ACC generated code, no optimizations

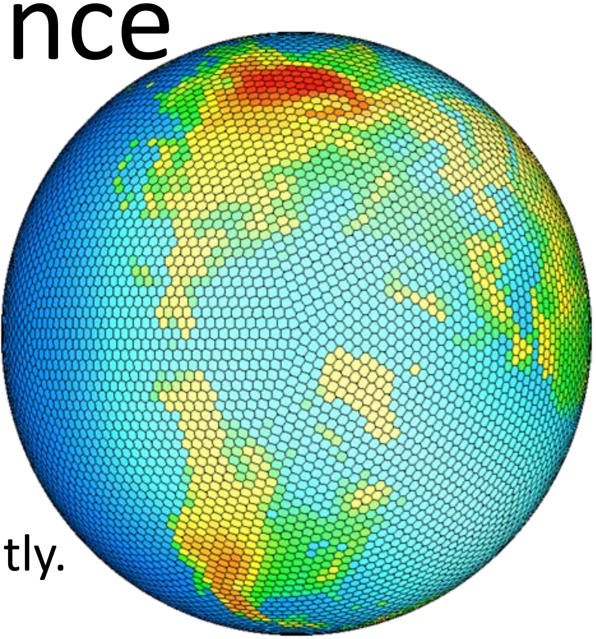
	Harpertown CPU Time	F2C-ACC CUDA-C Tesla GPU Time	Nahalem CPU Time	F2C-ACC CUDA-C Fermi GPU Time
Total	190.5	--	106.6	--
vdmints	89.2	4.28 (20.8)	50.6	2.66 (19.0)
vdmintv	38.8	2.48 (15.6)	23.3	1.02 (22.8)
flux	16.0	0.75 (21.3)	10.4	0.35 (29.7)
vdn	12.6	0.56 (22.5)	4.6	0.56 (8.2)
diag	4.0	0.09 (44.4)	4.0	0.09 (44.4)
force	5.3	0.11 (48.2)	3.4	0.11 (30.9)
trisol	8.6	1.45 (5.8)	2.0	1.28 (1.5)
input/init	1.0	1.0	1.0	1.0
output	0.2	0.2	0.2	0.2

GPU Compilers

- Reliance on NVIDIA, AMD compilers
  - Register allocation inefficient
    - Fermi Cache helps offload register pressure
  - data re-use limiting performance
  - “normal” optimizations not done
  - Requires code changes to achieve good results



Parallel Performance

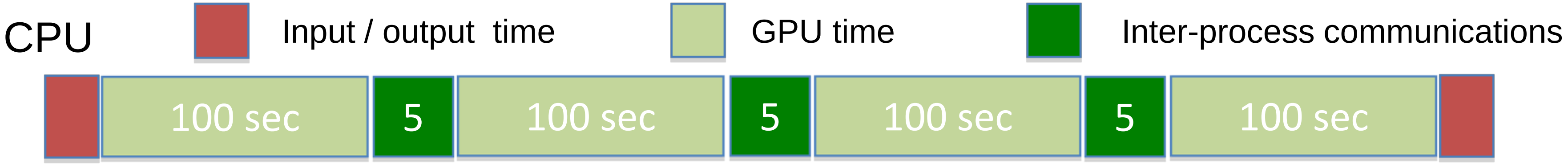


- A doubling of model resolution implies:
  - 4x increase in horizontal points
  - 2x increase in model time step
  - 4x increase in memory
- GPU memory is the limiting factor
- Number of horizontal points per GPU is 10K currently.

	G4	G5	G6	G7	G8	G9	G10	G11
resolution	480KM	240KM	120KM	60KM	30 KM	15 KM	7 KM	3.5 KM
horizontal points	2.5K	10K	40K	160K	640K	2560K	10,000K	40,000K
memory	.25GB	1GB	4GB	16GB	64GB			
tesla	26x	33x						
fermi	26x	26x	??					
# GPUs	1	1	4	16	64	256	1000	4000

Communication Bottleneck

- Application scaling will be limited by the fraction of time spent doing inter-process communications

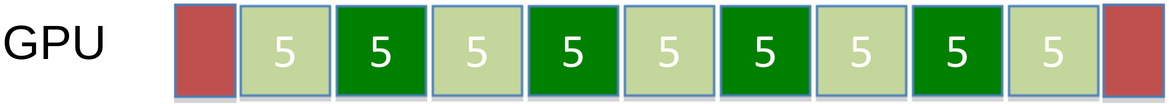


CPU

🟥 Input / output time 🟩 GPU time 🟩 Inter-process communications

🟥	100 sec	5	100 sec	5	100 sec	5	100 sec	🟥

- CPU communications time is about 5% of compute time
- Using GPUs, if we get a ~20x speedup in computation time, communications now becomes 50 percent of the runtime.



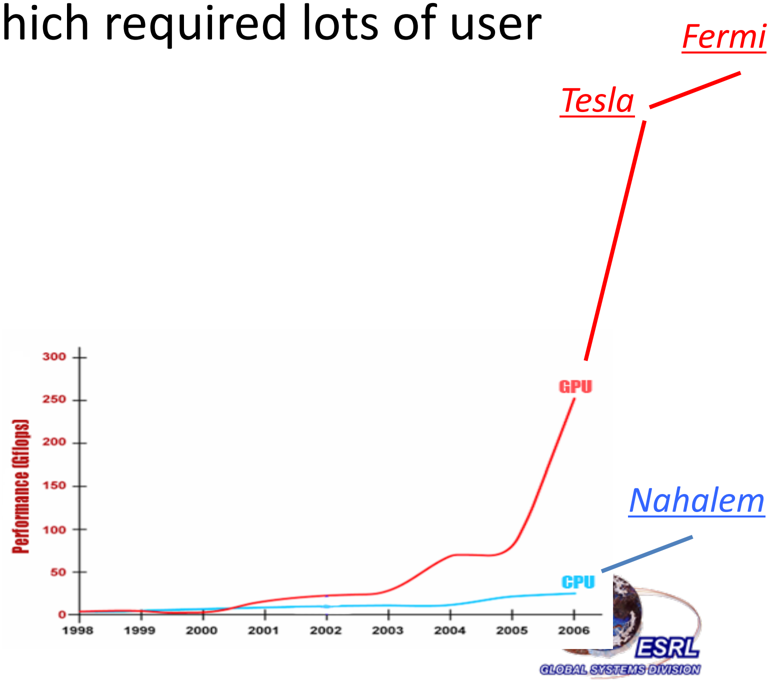
GPU

🟥	5	5	5	5	5	5	5	5	5	🟥

- Will need to reduce communications time
  - Minimize data volume and frequency
  - Overlap communications with computations
    - Move inter-process communications from just before needed to just after data is computed

Conclusion

- HPC transitions about every decade
  - Vector -> MPP -> COTS Clusters -> GPUs
    - 20-50x cost savings: hardware, power, infrastructure
- Tools and compilers are not mature
  - CUDA register allocation policies can restrict performance
  - Commercial Fortran compilers need to do more analysis
    - Similar to early vector compilers, which required lots of user involvement
- GPU Roadmaps look strong
    - Focus on power, performance
- Ready for operations?
- Useful for research?


Fermi
Tesla
Nahalem
GPU
CPU
Performance (Gflops)