Running the NIM Next-Generation Weather Model on GPUs

M.Govett[1], J.Middlecoff[2] and T.Henderson[2]
1 NOAA Earth System Research Laboratory, Boulder, CO, USA
2 Cooperative Institute for Research in the Atmosphere, Ft.Collins, CO, USA

Abstract

We are using GPUs to run a new weather model being developed at NOAA's Earth System Research Laboratory (ESRL). The parallelization approach is to run the entire model on the GPU and only rely on the CPU for model initialization, I/O, and inter-processor communications. We have written a compiler to convert Fortran into CUDA, and used it to parallelize the dynamics portion of the model. Dynamics, the most computationally intensive part of the model, is currently running 34 times faster on a single GPU than the CPU. We also describe our approach and progress to date in running NIM on multiple GPUs.

Key Words: Weather Prediction Model, Graphics Processors, GPU, High Performance Computing

1. Introduction

Weather and climate prediction have always driven demand for ever-larger computing systems. Global prediction models are now being developed to run at cloud-resolving scales (4 km or less) to: (1) better forecast hurricane track and intensity, (2) produce more accurate week-to month-long forecasts, and (3) understand and predict global scale weather phenomena that have significant weather impacts such as the Madden-Julian Oscillation (MJO) [8]. Using the general rule that operational weather models must produce forecasts at 2 percent of real-time, a 4km global model will require 100,000 to 200,000 CPUs using current technologies. Even if model code could scale to such a degree, building operational systems of this size are unrealistic for reasons of power, cooling, cost and reliability[1].

GPU processors are an attractive, highly scalable alternative to CPUs and appear to be the next big advance in high performance computing. NVIDIA, the leading manufacturer of GPUs, recently announced the release of its latest chip, called Fermi, and stated a long-term commitment to scientific computing in the future. In addition to high performance, the Fermi chip provides two requirements necessary to support scientific computing: error-correcting (ECC) memory, and full support for high-speed double precision calculations.

NOAA's Earth System Research Laboratory (ESRL) began exploring GPUs in 2008. We purchased a Tesla-based system (1 TFlop / chip) and have begun using it to run a next-generation weather model called the Non-hydrostatic Icosahedral Model (NIM).

This paper details our efforts to develop and parallelize NIM for GPUs. A team, composed of scientists, model developers, parallel programmers, and GPU researchers has been developing the atmospheric model and these joint efforts are key to the ambitious scientific goals planned for this model. NIM is slated to run at 4km global scale resolution within 3 years. Factoring in expected GPU performance improvements, we estimate only 5000 GPUs will be needed to produce operational weather forecasts with the 4KM NIM.

The atmospheric portion of NIM can be divided into two basic components: physics and dynamics. We have completed the GPU parallelization of dynamics, the most computationally intensive part of the NIM. Parallelization required conversion of the model code, written in Fortran 90, into CUDA, a high level programming language used on NVIDIA GPUs. To speed and simplify translation, we developed a compiler called F2C-ACC that is described in Section 2. A description of the code, optimization, and serial performance of dynamics is given in Section 3. The Scalable Modeling System (SMS) is being used run the parallel CPU version of the model [4]. We describe the CPU performance, GPU parallelization approach, and development efforts to date, in Section 4.

Commercial compilers from Portland Group International (PGI) and CAPS, which support Fortran, are now available [2][12]. We have begun evaluating these compilers for the GPU parallelization of NIM model physics, and the Hybrid Coordinate Ocean Model (HYCOM) into NIM that is planned [1]. We also plan to compare GPU performance of select dynamics routines using the commercial compilers to those parallelized using F2C-ACC. The oral presentation will report on these activities and describe the current status of this work.

2. The Fortran-to-CUDA Compiler

To speed the conversion of Fortran code into

[1] NOAA National Weather Service computer systems used to produce operational weather forecasts are required to have 99.9 percent system reliability [10].

978-0-7695-4039-9 2010
U.S. Government Work Not Protected by U.S. Copyright
DOI 10.1109/CCGRID.2010.106

CUDA, we developed a compiler, called F2C-ACC (Fortran-to-CUDA ACCelerator), using the Eli compiler construction system [5]. Eli is a domain specific programming environment that understands how to solve problems common to compiler development. In contrast to commonly used to tools such as LEX and YACC, which were designed to generate scanners and parsers, Eli was designed to generate the whole compiler from a single, high level specification. The specifications can be written that independently define the grammar, scanning, parsing, name and type analysis, symbol and definition table structures, language analysis, and the generation of the final output code. Eli does the analysis necessary, including determining a computational strategy necessary to satisfy the specifications, to generate a complete compiler. Eli builds the compiler either as a stand-alone executable or generates ANSI C code that can be ported and run on most modern computers.

F2C-ACC can generate either C or CUDA code. C code is useful for testing the generated code and for initial code generation for other accelerators such as IBM Cell [6]. CUDA is based on C with some additional extensions to call GPU routines (kernels), move data between the host (CPU) and device (GPU) and to manage computations and memory on the GPU [11]. The compiler supports all of the Fortran 95 language, however, translation of only the most commonly used constructs are generated. In the event translation of a language construct is not supported, a warning message is placed in the generated code that precedes the line in question.

```
F2C-ACC ERROR: [line number]
     [column number] "Language
     construct not supported."
```

F2C-ACC does not currently generate CUDA code that will run on the GPU; it must be further analyzed, modified, and optimized by hand. Frequent hand-coded modifications are gradually being automated as the compiler is further developed. Two accelerator directives are provided to support GPU kernel creation (ACC$REGION) and to define loop level parallelism (ACC$DO). These directives are similar to those used by the PGI GPU compiler and will simplify the comparison and conversion of NIM to a commercial GPU compiler.

As commercial Fortran GPU compilers mature, we hope many optimizations will be handled automatically. However, we believe the programmer will still need to both understand and modify the code to optimize it fully. For example, the CUDA language exposes four different types of memory available on the GPU: shared, global, constant, and texture memory. Access times to fetch data from memory on the GPU varies from hundreds of clock cycles for global memory (available to all threads and blocks), to a single cycle for shared memory accesses (available to all threads in a single block). Subtle changes in the code can result in large improvements in computational performance. This is largely due to how the programmer utilizes the various types of memory, and how effectively memory latency is hidden or overlapped by computations.

We anticipate F2C-ACC will be adapted to provide analysis, optimization, and debugging support necessary to support the weather and climate community. In this role, we envision F2C-ACC will become a pre-compiler for accelerator architectures, including NVIDIA, and will generate code that can be further optimized by the commercial products.

3. NIM Dynamics

3.1. Model Characteristics

NIM uses an Icosahedral horizontal grid as illustrated in **Figure 1**. Contrary to the traditional latitude / longitude grid used in most global models, this grid uses hexagons to represent the Earth and avoids problems at the poles caused by narrowing of longitudinal lines in a traditional lat-lon grid. The NIM is based on the previous generation Flow-following finite volume Icosahedral Model (FIM) developed at ESRL[3]. The FIM is slated to be part of an operational weather prediction model for the U.S. National Weather Service in 2011. Pre-operational testing will commence in 2010.

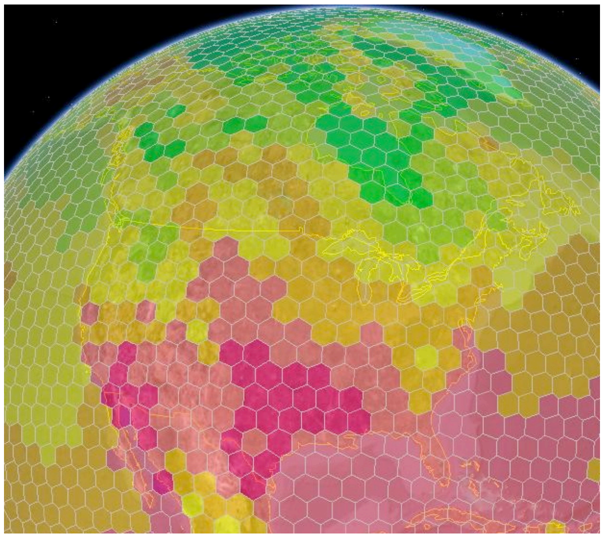

Figure 1: An illustration of the Icosahedral grid projected on the globe over North America. The grid is composed of ten pentagons, and the remaining cells are hexagons.

Like most weather and climate models NIM is a streaming code; data is fetched from memory, a small number of calculations are done on the data, and the result is copied back to memory. NIM uses indirect addressing to access horizontal points on the grid. Earlier studies indicate no performance impact using indirect indexing on the CPU because an inner k-loop can mitigate the access penalty over the vertical levels [7]. We also determined no performance impact on the GPU because the run-time performance is dominated by memory load and store operations.

During GPU parallelization, some code optimizations we identified helped both CPU and GPU performance For example, the most CPU intensive routine consumed over 40 percent of the dynamics runtime. Deep in the code, this routine called a BLAS routine from an inner loop over one million times per model time-step (once per grid point). By replacing this general routine with a hand-coded solution, the CPU code ran 10 times faster. Direct in-lining of this routine into the caller was also beneficial; it led to further, modest improvements on the CPU, and resulted in large gains on the GPU.

3.2. Serial Performance

Efforts to use GPUs thus far by the weather research community have focused on targeting select routines in the model that require the most time to execute [9]. However, this approach must also account for significant data transfer times needed to move data between the CPU and GPU. To avoid this penalty, all of NIM dynamics is run on the GPU; for the serial code, communications between CPU and GPU is only required (1) to copy input data to GPU memory, and (2) to transfer data to be output back to the CPU.

Table 1 compares the performance of NIM dynamics on the CPU (Intel Harpertown) and the GPU (Tesla – GTX280) and measures the impact of the copy. As shown, no performance gain is seen when the copy between CPU and GPU is included. It is likely we could significantly reduce the impact of the data copies by combining routines, eliminating redundant copies, using constant memory, and other means. However, these performance gains would never achieve the robust performance achieved by running everything on the GPU.

Columns 3 and 5 show that serial performance improved as the number of blocks increased from G4 (2562 points) to G5 (10242 points). We are investigating this result and believe it is due to the improved efficiency of the GPU Streaming Multi-processors (SMs) resulting from the increased workload. There are 30 SMs on the Tesla chip (GTX280). Computation on each SM is divided into units of work called warps. A warp contains 32 threads that are executed concurrently by the SM. The number of active warps, queued for execution, is based on the resources they require. Primarily the limiting resources are registers and the amount of shared memory each thread block requires. Active warps means they are queued for execution on the SM and in some run state. If a warp is in a wait state (eg. a memory fetch), the SM immediately moves to another warp in the queue via context switch. If a warp has all the information it needs to perform a computation, the SM will execute the instruction, and then look for the next ready-to-execute warp. As the number of blocks on the GPU increase, the number of warps in the ready queue will increase unless the SM resource limits are reached, or there are no more blocks to execute. We therefore speculate the increased number of warps in a ready state hide the latency of memory fetches by the SM, resulting in better performance.

Table 1: NIM Performance is given for each routine with two model resolutions containing 2562 (G4) and 10242 (G5) blocks or grids. The last column gives run-time and speedup when the CPU-GPU copy is included.

Routine	CPU G4	GPU G4	CPU G5	GPU G5	G5 + copy
Vdm	1.551	0.054 (28.4)	6.344	0.170 (37.3)	9.651 (0.65)
Flux	0.293	0.012 (24.0)	1.219	0.046 (26.5)	1.168 (1.04)
Grad	0.102	0.006 (15.7)	0.469	0.026 (18.4)	0.324 (1.44)
Force	0.094	0.003 (26.9)	0.391	0.013 (29.8)	0.373 (1.04)
Timedif	0.031	0.001 (38.7)	0.125	0.003 (43.1)	0.151 (0.83)
Sponge	0.001	0.001 (05.0)	0.016	0.002 (06.6)	0.038 (0.42)
Diag	0.059	0.001 (59.0)	0.234	0.003 (75.5)	0.157 (1.49)
TOTAL	**2.130**	**.0079 (26.9)**	**8.814**	**0.263 (33.6)**	**11.86 (0.74)**

Thread-level parallelism is defined in the vertical dimension, since dynamics contains almost no vertical data dependencies. Key factors in achieving good performance over the optimized CPU run-times were:

- **Eliminate data transfers**: Data transfers, required for model initialization and output. are amortized over the entire run.
- **Coalesce loads and stores**: Coalesced loads and stores occur when adjacent threads need to load data that is contiguous and aligned in memory. Thread blocks in NIM are defined over the vertical levels and data is contiguous in memory over the vertical levels ("k" dimension). In some

cases, increasing the rank of arrays, to include the vertical dimension (" k ") in some routines, improved coalesced loads and stores from global memory and led to big performance gains.

- **Maximize GPU utilization**: NIM contains 96 vertical levels. Since 96 is a multiple of a *warp* (32 threads), good utilization of the GPU hardware was achieved. Scaling tests indicate 96 threads per block, while fewer than the 128 or 256 threads NVIDIA recommends, were sufficient for good performance.
- **Minimize shared memory and register use**: The use of registers and shared memory has a direct impact on performance. If either resource is high, the number of thread blocks available for execution will be reduced, thus limiting performance. The programmer has limited control over the use of registers; the only technique we have found useful is to break large routines, where register usage is high (> 50 registers per thread), into multiple kernels. Shared memory declarations are controlled by the programmer. Use of this fast memory is only beneficial when there is data reuse that reduces global memory accesses.

Due to limitations in GPU global memory, we were unable to scale to NIM G6 (40K points) for the serial tests. The Tesla chip only provides 1 GByte of Global memory. This constraint will be eased when the Fermi chip, with 6 GBytes of global memory, is available.

4. Parallel Performance

4.1. CPU Performance

The Scalable Modeling System (SMS) is being used to parallelize and run NIM on multiple CPUs. SMS is a directive-based tool, developed at ESRL, to handle most aspects of parallel programming including data decomposition, reductions, inter-processor communications, I/O, and parallel debugging [4]. Both FIM and NIM were designed to be highly scalable, with an emphasis on minimizing inter-process communications, since these operations incur large performance penalties.

Table 2 illustrates CPU performance and scaling of the FIM model and can be used as a guide for NIM performance. Domain decomposition used in both FIM and NIM is in the horizontal dimension, so all inter-processor communication is the dynamics portion of the model. FIM requires five communications per time-step, while the NIM requires only three. This reduction was achieved by better code design and doing redundant calculations in the halo regions in lieu of increased communications. Since GPU calculations are cheap, and communications are expensive, this is expected to be beneficial for GPU performance and scaling.

Table 2: FIM parallel performance and scaling are given for NIM G6 (120km) running on an Intel Nahalem-based cluster.

CPU cores	Points / Proc	Time (sec)	Scaling	Efficiency
10	9216	12743	1.0	100
20	4608	6338	2.01	101
40	2304	3235	3.94	98
80	1152	1674	7.61	95
160	576	912	13.97	87

4.2 GPU Parallelization

To date, there is no support for direct inter-GPU communications; GPUs must communicate via the parent CPU. Therefore, inter-GPU communications becomes a three step process: (1) copy data to be communicated to the CPU, (2) send data to the target CPU/GPU pair via MPI or similar, and (3) copy data received by each CPU back to the GPU.

Both FIM and NIM use SMS for inter-processor communications. The directive SMS$EXCHANGE is used to transfer data between CPUs. For example, SMS$EXCHANGE (A) will communicate the halo region of the variable A, to its neighboring processors. We are adding support to SMS to permit data to be exchanged between GPUs in the following way:

1. From the CPU, launch a GPU routine to pack data to be communicated. If multiple arrays are to be exchanged, their data will be aggregated into a single data structure.
2. From the CPU, copy the data to be exchanged from the GPU and use SMS to send it to the appropriate CPU. Each CPU node will wait to receive its data.
3. From the CPU, copy data to the GPU and launch a GPU routine to unpack and distribute the data to the arrays as appropriate.

We anticipate GPU parallel performance will depend largely on how effectively communications latency can be hidden. Table 2 indicates latency between processors does not adversely affect CPU performance. The additional latency resulting from communications between CPU and GPU and will likely change the number of points per GPU that yields optimal efficiency.

5. Conclusion

The NIM model is being developed to run on GPUs. Our approach is to run the entire model on the GPU, and avoid copying data between CPU and GPU for every kernel invocation. A Fortran-to-CUDA compiler was described and used to parallelize NIM dynamics. Using our compiler to generate CUDA, plus hand-coded optimizations, we have demonstrated a 34x performance improvement over the CPU.

Work on NIM is continuing in three areas: run dynamics on multiple GPU nodes, parallelize GFS physics for GPUs, and incorporate the HYCOM Ocean Model into NIM. We are using SMS to run NIM on multiple CPU / GPU nodes. SMS is being modified to support inter-GPU communications. We are planning to use a commercial GPU Fortran compiler to parallelize GFS physics that will be used in NIM for initial testing. We are monitoring the development of an icosahedral grid version of HYCOM, and are influencing the design as necessary to be sure it will run efficiently on GPUs.

References

[1] R.Bleck, "An Oceanic General Circulation Model Framed in Hybrid Isentropic Cartesian Coordinates", *J. Ocean Modeling*, 2001.

[2] CAPS Web Page http://www.caps entreprise.com/.

[3] FIM Web Page: http://fim.noaa.gov/.

[4] M.Govett, L.Hart, T.Henderson, J.Middlecoff, and D.Schaffer, "The Scalable Modeling System: Directive-Based Code Parallelization for Distributed and Shared Memory Computers", *Journal of Parallel Computing*, August 2003, pp 995-1020.

[5] R.Gray, V.Heuring, S.Levi, A.Sloane, and W.Waite, "Eli, A Flexible Compiler Construction System", *Communications of the ACM*, 1992, Volume 35, pp121-131.

[6] IBM Cell Web Page, http://www.research.ibm.com/cell/.

[7] A.E.MacDonald, T.Henderson, J.Middlecoff and J.Lee, "An Indirect Addressing Scheme for Global Icosahedral Grids", submission to *The International Journal of High Performance Computing Applications*, Dec 2009.

[8] Madden-Julian Oscillation Web Reference, http://en.wikipedia.org/wiki/Madden–Julian_oscillation

[9] Michalakes, J. and M. Vachharajani: GPU Acceleration of Numerical Weather Prediction. *Parallel Processing Letters* Vol. 18 No. 4. World Scientific. Dec. 2008. pp. 531--548. http://wwww.worldscinet.com/ppl

[10] NOAA High Performance Computing Strategic Plan for FY2011-FY2015, *http://*www.cio.**noaa**.gov/**HPC**C/pdfs/**HPC**_Strategic_Plan.pdf.

[11] NVIDIA CUDA Reference Manual, http://developer.download.nvidia.com/compute/cuda/2_0/docs/CudaReferenceManual_2.0.pdf.

[12] Portland Group International GPU Web Page, http://www.pgroup.com/resources/accel.htm.