# Experience Applying Fortran GPU Compilers to Numerical Weather Prediction

T. Henderson, J. Middlecoff, J. Rosinski
Cooperative Institute for Research in the Atmosphere
NOAA Earth System Research Laboratory
Boulder, Colorado, USA
thomas.b.henderson@noaa.gov

M. Govett
NOAA Earth System Research Laboratory
Boulder, Colorado, USA

P. Madden
Cooperative Institute for Research in Environmental Sciences
NOAA Earth System Research Laboratory
Boulder, Colorado, USA

***Abstract*—Graphics Processing Units (GPUs) have enabled significant improvements in computational performance compared to traditional CPUs in several application domains. Until recently, GPUs have been programmed using C/C++ based methods such as CUDA (NVIDIA) and OpenCL (NVIDIA and AMD). Using these approaches, Fortran Numerical Weather Prediction (NWP) codes would have to be completely re-written to take full advantage of GPU performance gains. Emerging commercial Fortran compilers allow NWP codes to take advantage of GPU processing power with much less software development effort. The Non-hydrostatic Icosahedral Model (NIM) is a prototype dynamical core for global NWP. We use NIM to examine Fortran directive-based GPU compilers, evaluating code porting effort and computational performance.**

***Keywords-graphics processing units; weather modeling; high-performance computing***

## I. INTRODUCTION

The need for ever-increasing computer processing power to improve Numerical Weather Prediction (NWP) has been well-documented [1]. Since the clock rate of traditional Central Processing Unit (CPU) technology has stalled, some groups are investigating Graphics Processing Units (GPUs) to further speed up computation in NWP codes [2][3]. Most efforts have focused on low-level vendor-specific programming approaches such as NVIDIA's C-language-based Compute Unified Device Architecture (CUDA) which runs only on NVIDIA GPUs. However, most NWP software is written in Fortran and must be completely re-written to use CUDA. It is unrealistic to expect researchers to develop, debug, test, and maintain multiple versions of a single NWP code. Fortran directives are a common solution to this problem. Directives appear as comments in code (similar to the ANSI C "#pragma") allowing developers to specify GPU-related details in code that can still run on CPUs. Directive-based approaches have been successful in similar roles, the most notable example being the widespread use of OpenMP directives to express shared-memory parallelism [4]. A performance-portable directive-based Fortran compiler that permits maintenance of a single source code for GPUs and CPUs would be extremely beneficial for NWP.

When the authors began investigating application of GPU technology to NWP in 2008, no mature commercial directive-based Fortran compilers were available for GPUs. To speed our investigations, we developed "F2C-ACC" [5], an "application-specific" directive-based compiler that automates the Fortran-to-CUDA-C translation. In contrast to a traditional compiler, our application-specific compiler is intentionally limited in scope to support a subset of the Fortran language used by our NWP codes. As commercial directive-based compilers from CAPS [6] and Portland Group (PGI) [7] have become available, we have used CUDA code generated by F2C-ACC as a baseline to evaluate features, ease-of-use, and performance of the commercial compilers.

## II. THE NON-HYDROSTATIC ICOSAHEDRAL MODEL

The Non-hydrostatic Icosahedral Model (NIM) is a new "dynamical core" being developed for NWP. In a NWP model, the dynamical core is responsible for the calculations that solve the equations of motion on explicitly resolved scales. Parameterization of physical processes such as longwave and shortwave radiation, clouds, and planetary boundary layer are not yet included in NIM. Depending on the application of an NWP model, the total computational costs of these "physics" packages may rival or even exceed the cost of the dynamical core.

NIM is designed for use at global cloud-permitting resolutions of 3km (42 million columns) or higher. Each column currently has 96 vertical levels. NIM is a direct descendant of the hydrostatic Flow-following finite-volume Icosahedral Model (FIM) [8]. FIM is scheduled for operational implementation in 2012 as part of the U. S. National Weather Service's global multi-model ensemble. Both FIM and NIM use an icosahedral grid to represent flow on a spherical surface (see Figure. 1). At any resolution, twelve of the grid cells are pentagonal and the rest hexagonal. Unlike traditional latitude-longitude grids, the icosahedral grid is ideal for representing a sphere with minimal distortions due to variations in cell shape, size, or spacing. Also, all adjacent grid cells are connected via sides,

not via vertices [8]. The icosahedral grid has been used successfully in several research and operational NWP models [9][10][11][12][13].

Since storage and iteration through this grid do not map naturally onto multi-dimensional arrays found in computer languages such as C and Fortran, we store horizontal fields in one-dimensional vectors and use indirect addressing to locate neighboring cells. The resulting code is extremely

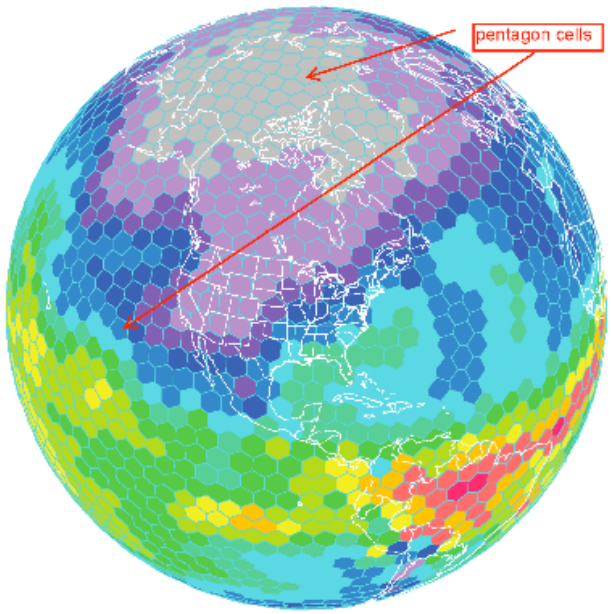


Figure 1. Icosahedral grid used by the Non-hydrostatic Icosahedral Model.

compact compared to equivalent directly-addressed implementations. To overcome the performance penalty of extra memory accesses needed to locate neighbors, we store three-dimensional model state arrays with the directly-addressed vertical dimension fastest-varying (first in Fortran) [14]. This method reduces costs of indirect addressing by a factor equal to the number of vertical levels for computations made on three-dimensional fields such as temperature and winds. Fortunately, the vast majority of computations in NIM occur with three-dimensional fields, so indirect addressing costs less than 1% of total run time. The compactness of the code also makes it easier to parallelize for GPUs.

Since NIM has always been intended to run on very large computers, computational performance has been a consideration from its inception. NIM was designed to present compilers with the simplest possible source code to facilitate optimization. Most computations are organized as simple dot products or vector operations and loops do not include data-dependent conditionals. The Intel "ifort" compiler's vectorization diagnostics are used to ensure that code vectorizes well, which improves performance on CPUs. This approach yields code that tends to run well on GPUs. Also, the algorithms used in NIM require only 32-bit floating-point precision, speeding up computations on both CPU and GPU architectures. Other NWP models, such as the Weather Research and Forecast (WRF) model [15], routinely produce high-quality forecasts using 32-bit precision at NIM's target resolution.

## III. Initial Exploration of GPU Computing

Early on, it was recognized that the goal of running NIM at global 3km resolution would require several hundred thousand CPU cores. NIM designers were intrigued by the potential of GPUs to significantly reduce this number, following pioneering work by Michalakes [2] on portions of the WRF model [15]. Initial work with NIM began in 2008 using NVIDIA "Tesla" GPUs programmed with CUDA-C. Fortran code was translated using an early version of F2C-ACC. Generated CUDA code "kernels" were then hand-tuned, achieving speedups averaging 34x over a single Intel Harpertown CPU. CPU-GPU transfer time was not included in these early speedup calculations [5]. Newer versions of F2C-ACC do not require hand tuning of the automatically generated CUDA code.

Although F2C-ACC has been useful for our NIM dynamics code, many of the codes we will incorporate into future versions of NIM (NWP "physics" packages to model radiation, clouds, etc.) will come from other organizations. These codes use a wider range of Fortran features and we do not plan to extend F2C-ACC into a fully-functional Fortran compiler. In the long run, we prefer to use commercial compilers because compiler vendors have the resources to support most Fortran language features.

Fortunately, two commercial directive-based Fortran compilers for GPUs are now available and are beginning to mature. Following our initial modest success with individual NIM kernels, we began investigating CAPS HMPP Workbench [6] and PGI Accelerator Fortran [7]. At the same time, we expanded our investigation to include the entire NIM dynamical core and addressed data transfers between CPU and GPU.

## IV. Investigation of Commercial Directive-Based GPU Fortran Compilers

Our first step was to create an automated test that compares NIM output with a CPU baseline, printing the number of digits of accuracy for worst-case differences. Model resolutions of 2562 and 10242 columns with 96 vertical levels were tested to simulate the 8000-10000 columns each GPU is expected to be responsible for in an eventual 3km simulation. Our test cases run an idealized simulation for 50 time steps with output written to disk at the beginning and end. This output frequency is much higher than that planned for production configurations (likely every ~1000 time steps) but allows reasonable run-times on a single CPU. We then ran the test cases on CPUs using different compiler versions and optimization options to discover how results differ due to re-ordering of floating-point operations. We require that differences between GPU and CPU results do not exceed differences between CPU runs made with different compiler optimizations. This effectively accomplishes the tests of error growth in

atmospheric models first described for the Community Atmosphere Model [16].

For every test, code optimized for GPU was run on a GPU while code optimized for CPU was run on one or more CPU cores. In most cases, optimizations that improved performance on one also benefited the other. However, some divergent code sections were needed to achieve the best performance on the different architectures. Divergence was usually due to the need to hand-optimize Fortran code to address limitations in current GPU compilers. These optimizations included loop splitting/jamming, promotion/demotion of arrays to higher/lower rank, and other techniques that mature CPU compilers have done well for years. (In fact, advanced HMPP capabilities address some of these optimizations. We have not tested them but plan to do so.) Maintenance of divergent code sections is expensive so this issue will continue to be a concern.

### *A. CAPS HMPP Workbench*

We began working with CAPS HMPP Workbench in August 2010. The version we tested, HMPP 2.4.3, can translate Fortran code to CUDA-C for NVIDIA GPUs or to OpenCL [17] for either NVIDIA or AMD GPUs. Thus far, we have only investigated HMPP support for NVIDIA GPUs via CUDA.

HMPP provides directives to identify code sections, called "codelets", that will execute on the GPU. It also supports a range of directives that permit flexible control of expensive CPU-GPU data transfers. Users may begin by using simple forms of directives which can automatically generate CPU-GPU transfers allowing correct computational results to be quickly achieved. Performance can then be optimized via more complicated forms of directives that permit minimization of data transfers for optimal performance. Fine-grained loop optimizations (unroll, jam, permute, etc.) are also supported via a set of low-level directives useful for fine-tuning performance. Vendor-specific values for GPU-specific settings can also be adjusted via directives, although optimal settings must be discovered by trial-and-error.

Until data transfer rates between CPU and GPU increase dramatically, it will be necessary to minimize data transfers to fully realize the performance benefits of GPUs. This is especially true for NWP codes which tend to be memory-bandwidth-bound [2]. Data transfers can be minimized by storing the entire model state in GPU "global" memory. Transfers are then needed only for disk I/O and MPI communication. We chose to make storage of all model state variables in GPU global memory the initial goal of our HMPP work starting with a serial (no MPI) configuration of NIM. For the 3km configuration of NIM, it appears that planned GPU products will have enough memory to support this approach, even after required "physics" packages are added. It is not clear that GPUs will have enough memory if NIM is eventually extended to include full atmospheric chemistry.

We took advangtage of the flexibility of HMPP directives to develop incrementally in small steps, running the test suite and fixing errors encountered during each step. The ability to take very small steps and test after each one is critical to making rapid progress on GPUs where software development tools, like debuggers and even the humble print statement, are still relatively immature. We also found it helpful to look at the HMPP-generated CUDA code and to use the CUDA profiler [25] when tuning performance. (The CUDA profiler is a vendor-supported tool that provides post-mortem details of GPU resource allocation and usage on NVIDIA GPUs.) Fortunately, HMPP generates CUDA-C using the same variable names found in the original Fortran code, making it easy to relate information gathered by the CUDA profiler back to the original Fortran code. CAPS user support was effective, providing advice when needed as we learned how to use HMPP and devising work-arounds and patches for the few bugs we discovered.

We began by gradually adding HMPP directives to select subroutines for execution on the GPU. Ultimately, this included all routines executed during a model time step except for the routine that writes output to disk. As each routine was moved to the GPU, the entire test case was reran and validated against the original CPU results. This stepwise method eased discovery of coding errors. HMPP directives can use the Fortran "INTENT" keywords to automatically deduce the direction(s) of CPU-GPU data transfers when generating CUDA-C code. Fortunately, NIM uses INTENT in all dummy argument declarations, allowing the simplest forms of HMPP directives to be used.

Since HMPP automatically generates CPU-GPU data transfers for every subroutine argument, this initial GPU code was quite slow. To address this issue, we added more sophisticated directives to eliminate duplicate storage of arrays on the GPU and to replace automatic data transfers with a minimal set of explicit data transfers. These data transfer optimizations were straightforward, although they did require a detailed knowledge of data dependencies in the NIM code. After completing this step, all model state variables resided on the GPU after model initialization and were only transferred to the CPU when written to disk.

Once data transfer optimizations were complete, we began exploring HMPP directives for performance tuning. We first used simple directives to swap the order of some loops and achieved modest performance gains in a few routines. We then used more advanced directives to control GPU-specific optimizations and to explicitly identify loops as parallelizable in a few cases with complex dependencies where this is not yet done automatically. We repeatedly referred to the F2C-ACC-generated CUDA code for guidance as we optimized our use of the HMPP directives. F2C-ACC was especially useful in identifying opportunities for additional performance enhancements and convincing compiler developers that improvements to their products could be made. We also made extensive use of diagnostic messages from HMPP and CUDA profiler results. As mentioned above, some GPU-specific hand tuning of Fortran code was needed in a few critical areas. Performance results are summarized in Section V.

### B. PGI Accelerator Fortran

We began experimenting with PGI Accelerator version 9.0 in September 2009, focusing on simple stand-alone Fortran code "kernels" extracted from the most computationally intensive parts of NIM. In September 2010, we started testing PGI with the complete NIM code using versions 10.9 and 11.1. Like HMPP, PGI Accelerator provides directives to identify code kernels for execution on a GPU, to transfer data from CPU to GPU and back, and to guide mapping of parallel loops to GPU resources. The versions of PGI Accelerator we tested generate CUDA code (which runs only on NVIDIA GPUs). Unfortunately, generated code is quite obfuscated, making debugging and optimization much more difficult. Obfuscation appears to confer no performance improvement or other benefit from a user's perspective.

As was done with HMPP, we used a step-wise approach to gradually add PGI Accelerator directives to NIM code. At the time of this work, the Portland Group compiler was not yet able to support keeping model state variables in GPU memory between GPU kernel calls. So we concentrated on optimizing performance of individual NIM routines. We were able to measure run times of several individual kernels but do not include any GPU-CPU transfer time in PGI results reported here.

PGI has also been responsive to bug reports and has been able to reproduce issues we have raised. Their planned directives to support minimizing CPU-GPU data transfers appear to be comparable with HMPP in terms of capability and ease-of-use. Recently, PGI compiler developers have used NIM as a test case to improve their compiler. They report that their next release, 11.6, will be able to support maintenance of model state in GPU memory in between kernel calls. We look forward to testing this new release.

## V. GPU Performance

We ran timing tests on an NVIDIA GTX 280 "Tesla" GPU, a newer NVIDIA C2050 "Fermi" GPU, and a single core of an Intel "Nehalem" (2.8 GHz) CPU. The software stack included CUDA 3.2, Intel Fortran Compiler 11.1, HMPP Workbench 2.4.2, and PGI Accelerator Fortran 10.8. Results for both commercial compilers were compared with CUDA-C generated by F2C-ACC using the small (2562-column) test case.

Table I summarizes initial test results using HMPP and PGI compilers with the older "Tesla" GPU and compares with a single core of the Intel Nehalem CPU. Performance statistics were gathered using the General Purpose Timing Library (GPTL). This open source library provides the option to gather wallclock times using a fast register read on x86-style architectures. We used this GPTL option instead of the more ubiquitous 'gettimeofday' because it is more accurate and requires less system overhead. All timer calls were placed in Fortran code immediately surrounding GPU kernel invocations. Since timers are called on the CPU, all times include GPU kernel load and associated overhead.

Each test was repeated ten times. For each case, the run with the minimum total time is recorded in the table. The "Total" row shows total run times for the Nehalem CPU and for each GPU case. Times in the "Total" row include all disk I/O and CPU-GPU data transfer times. Times for several other important NIM routines are also shown to illustrate how performance can vary. These times do not include CPU-GPU data transfers because none are needed for F2C-ACC and HMPP where all arguments to these routines already reside in GPU memory. In the case of PGI, the CPU-GPU data transfers were needed for each routine but transfer times are not included to permit a fair comparison. In a few cases, PGI results are not available due to compiler bugs that will be fixed in their next (11.6) release.

TABLE I. PERFORMANCE OF SMALL TEST CASE

| *NIM Routine* | *Nehalem CPU Time (sec)* | *F2C-ACC Tesla GPU Time (sec)* | *HMPP Tesla GPU Time (sec)* | *PGI Tesla GPU Time (sec)* |
|---|---|---|---|---|
| Total[a] | 106.6 | 10.8 | 10.3 | -- |
| Vdmints | 50.6 | 2.5 | 2.3 | 11 |
| Vdmintv | 23.3 | 0.93 | 0.99 | 25 |
| flux | 10.4 | 1.15 | 1.05 | 24 |
| vdn | 4.6 | 0.58 | 0.73 | -- |
| diag | 4.0 | 0.093 | 0.085 | 34 |
| force | 3.4 | 0.11 | 0.19 | 43 |
| trisol | 2.0 | 1.9 | 1.4 | -- |

a. Total times include disk I/O and CPU-GPU data transfers.

The commercial compilers are producing code that performs as well as the F2C-ACC-generated CUDA code in most cases. The cases where performance is worse have been provided to compiler vendors to assist them in improving their products. The routine that performs worst, "trisol", is a tri-diagonal solver that uses an algorithm that does not parallelize well. We will eventually replace it with a parallel algorithm. For the small test case, "trisol" was run on one GPU thread.

A cause of variation in GPU performance for the other routines is illustrated in Table II. We used PAPI [18] to estimate computation rates for each NIM routine on the Nehalem CPU (after validating PAPI floating-point operation counts by manually counting floating-point operations in one routine) and used these counts to estimate computation rates for each of the GPU runs. These computation rates, in GFLOP/second (GFLOPS) are shown in the second, third, and fourth columns of the table. PAPI also provides an estimate of "computational intensity", defined as number of floating-point computations per memory access. This is shown in the last column.

The 2.8GHz Nehalem CPU is theoretically capable of executing four single-precision floating-point operations per clock cycle for a maximum single-precision peak rate of 11.2 GFLOPS. The GTX 280 GPU is theoretically capable of 936 single-precision GFLOPS. From the "Total" row of Table II, the Nehalem CPU is achieving ~29% of its peak performance. This is a good result for an NWP model and

indicates that the NIM code is well tuned for CPU execution. NIM is achieving ~3% of peak performance on the GTX280 GPU using HMPP with the most efficient routine, "vdmints", achieving ~10% of peak.

TABLE II. ESTIMATED COMPUTATION RATES FOR SMALL TEST CASE

| *NIM Routine* | *Estimated Nehalem CPU GFLOPS* | *Estimated F2C-ACC Tesla GPU GFLOPS* | *Estimated HMPP Tesla GPU GFLOPS* | *Estimated Computational Intensity* |
|---|---|---|---|---|
| Total[a] | 3.2 | -- | 32 | 1.68 |
| vdmints | 3.8 | 77 | 85 | 1.96 |
| vdmintv | 3.9 | 99 | 95 | 1.85 |
| flux | 2.3 | 21 | 23 | 1.11 |
| vdn | 1.0 | 9 | 6 | 0.89 |
| diag | 1.3 | 57 | 62 | 1.12 |
| force | 1.9 | 61 | 35 | 1.41 |
| trisol | 2.3 | -- | 3 | 1.10 |

a. Total rates include disk I/O and CPU-GPU data transfers.

There appears to be some correlation between computational intensity and performance on CPUs. This correlation appears to be even stronger on the GPU (excluding "trisol" which does not yet have a good parallel implementation). Routines with low computational intensity generally perform poorly while routines with high computational intensity generally perform well. NWP codes that have lower computational intensity than NIM may not perform efficiently on GPUs. Others have also observed this correlation across a wide range of GPU applications [21]. This result is useful when selecting new algorithms for implementation on GPUs.

We repeated the small test case on the newer Fermi GPUs and observed nearly identical total run time which is not surprising since its peak 32-bit floating-point performance of 1000 GFLOPS is close to Tesla. Analysis via the CUDA profiler seems to indicate that the small test case cannot support enough concurrent threads to keep the GPU hardware busy. Since the large test case (10242 columns) is much closer to expected GPU workload for a global 3km resolution, we repeated the above tests with this case. Recognizing that a GPU consumes much more power than a single CPU core, we made a "socket-to-socket" comparison of CPU and GPU technology by comparing the newer NVIDIA "Fermi" GPU with a six-core Intel "Westmere" (2.66GHz) node. MPI is used to implement parallelism on the CPU. We do not intend to imply that power consumption of each "socket" is equivalent (full system power consumption can only be accurately measured under load so vendor product data sheets have limited utility). However, "socket-to-socket" comparison is clearly more "fair" than comparison of a GPU to a single CPU core. Table III summarizes CPU and GPU run times for the large case.

For the large test case, we used a partially parallelized version of the tri-diagonal solver ("trisol") that yielded considerable performance improvement on the GPU. We also ran the model for 1000 time steps, yielding a much more realistic output frequency. We regard 4.6x speedup of Fermi over six-core Westmere as an encouraging result that justifies further investigation of GPU technology.

TABLE III. PERFORMANCE OF LARGE TEST CASE

| *NIM Routine* | *Westmere 1-CPU Time (sec)* | *Westmere 6-CPU Time (sec)* | *F2C-ACC Fermi GPU Time (sec)* |
|---|---|---|---|
| Total[a] | 8654 | 2068 | 449 |
| Vdmints | 4559 | 1062 | 196 |
| Vdmintv | 2119 | 446 | 91 |
| Flux | 964 | 175 | 26 |
| Vdn | 131 | 86 | 18 |
| Diag | 389 | 74 | 42 |
| Force | 80 | 33 | 7 |
| Trisol | 119 | 38 | 31 |

a. Total times include disk I/O, initialization, and CPU-GPU data transfers.

PAPI-estimated GFLOP rates for one-core Westmere, six-core Westmere, and Fermi are 3.2 GFLOPS, 13.5 GFLOPS, and 62.4 GFLOPS respectively. On Fermi, the large test case is achieving about 6% of peak performance. While this is significantly better than the small test case, there is much room for further improvement.

Unfortunately, we have not been able to successfully run the large case with either commercial compiler; cause(s) are still under investigation. We do not expect the PGI compiler to work until the 11.6 release, as discussed above. We are currently pursuing suspected cause(s) of failure in the HMPP version. Thus far, we have demonstrated that each individual kernel works properly in isolation. And we have verified that data transfers between CPU and GPU appear to be occurring at expected times. The most likely cause is an error in our use of the HMPP directives for minimizing CPU-GPU data transfer. Kernel run times are very close to F2C-ACC and relative differences are very similar to those observed for the small test case. However, until the model results can be validated, we cannot be certain that these HMPP performance results are accurate.

## VI. EASE OF USE

Due to the relative immaturity of the commercial compilers and of the supporting tools provided by GPU vendors, GPUs are still much more difficult to use than CPUs. Qualitatively, the directives provided by CAPS HMPP provide much of the functionality needed to move an entire application from CPU to GPU, provided that the application is already well structured for GPU acceleration. Although PGI Accelerator Fortran lags behind somewhat, its planned directives appear to be equally easy to adopt.

As has been the case with every new compiler technology, many common Fortran language features are either not yet supported by HMPP and PGI or are not yet supported by the underlying GPU software stack. Examples

include full use of newer Fortran90 features (modules, user-defined data kinds and types, pointers, etc.) and language features that interact with the operating system (print statements, file I/O). Some of these features are not required to address performance issues. And the technology is still new enough that users will sometimes encounter bugs or lack of needed features either in the commercial Fortran compilers or in the underlying GPU software stack. While we believe that GPU directives will eventually rival mature directive-based approaches like OpenMP in ease-of-use, GPU users will have to live with the difficulties typically associated with new technology for some time.

Ease-of-use is greatly enhanced when a commercial compiler can generate CUDA-C (or OpenCL, etc.) code that mimics the variable names and code structure of the original Fortran code where practical. This makes it far easier to use GPU profiling tools to guide performance optimization in the original Fortran code. It also makes it much easier for sophisticated users to find and report correctness and performance bugs, leading to stronger compiler products. Both HMPP and F2C-ACC do this well. PGI has long-term plans to eliminate support for CUDA and instead generate assembler-level code (such as PTX [24] for NVIDIA). Until then, we would very much like to see PGI support generation of human-readable CUDA at least as an option.

## VII. Distributed Memory Parallelism

In order to run at 3km global resolution, NIM must scale to roughly ten thousand GPUs. Since NIM is still under development, large test cases at sub-10km resolution have not yet been created. Fortunately NIM's ancestor, FIM, is much more mature and exhibits good scaling when run at high resolutions on CPUs using tens of thousands of MPI tasks. The dynamical cores of both FIM and NIM share the same data structures and implementation details (such as indirect addressing). They both use the Scalable Modeling System (SMS) [19], a directive-based approach to implementing distributed-memory parallelism via MPI message passing. Furthermore, FIM is a fully functional NWP model that includes a complete package of NWP physical parameterizations (clouds, convection, solar radiation, planetary boundary layer, etc.) from the National Weather Service's operational Global Forecast System (GFS) NWP model [20]. Because the implementations of their dynamical cores are so similar, it is reasonable to expect that NIM will also scale well on tens of thousands of CPUs after addition of physics packages.

SMS has been successfully used to create a distributed-memory parallel version of NIM that produces correct results when either CPUs or GPUs are used. Multi-CPU output files exactly match files written by a single-CPU run. Runs made with single and multiple GPUs also produce identical output files. Scaling is reasonable on CPUs for the small test cases currently available. However, when multiple GPUs are used, MPI communication time becomes a much larger fraction of total run time (because GPU computation is much faster) so NIM scaling is not as good. NIM scaling on CPUs and GPUs is illustrated in Figure 2 by curves marked "CPU" and "GPU". In this figure, the ordinate plots total run time in inverse-seconds while the abscissa plots number of GPU or CPU (six-core Westmere) sockets. For reference, "CPU linear" and "GPU linear" lines illustrate perfect scaling. The curves identified as "CPU w/o exchange" and "GPU w/o exchange" illustrate total time with MPI communication time removed. Work is now under way to reduce communication time for multi-GPU runs, initially by using asynchronous message-passing to perform communication and computation concurrently. Once this issue is addressed, we are hopeful that NIM runs made on tens of thousands of GPUs will incur less MPI overhead than NIM runs made on hundreds of thousands of CPUs.

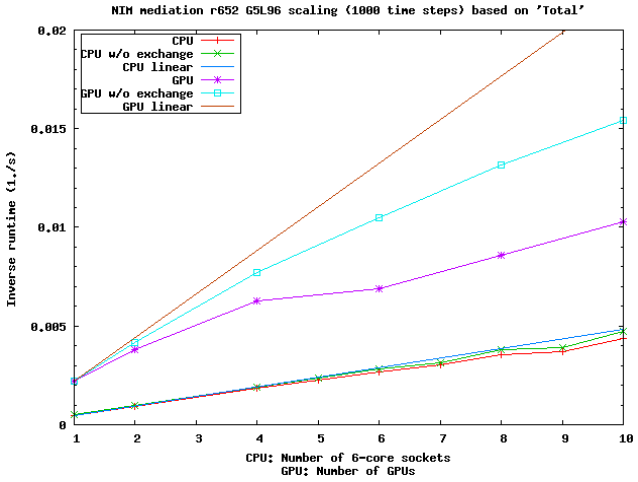


Figure 2. NIM scaling on CPUs and GPUs.

Most importantly, this result shows that SMS directives and F2C-ACC directives can be safely mixed allowing a single source code to be used for serial CPU, serial GPU, multi-node CPU, and multi-node GPU execution. We believe that the directives provided by the commercial Fortran compilers will also mix easily with SMS directives. At this point, we are optimistic that it will eventually be possible to maintain a single source code for all desired modes of CPU and GPU execution.

## VIII. Conclusions and Future Work

As CPU clock rates stall, GPUs offer an attractive alternative. Like any emerging technology, GPUs are now relatively difficult to use for high-performance scientific computing. Our work with the NIM dynamical core shows that the performance of commercial Fortran directive-based GPU compilers is becoming competitive with our application-specific F2C-ACC compiler for smaller test cases. Performance results for the large test case with F2C-ACC are encouraging with the Fermi GPU considerably out-performing a six-core Westmere CPU node. Percentage of peak performance achieved for NIM shows that CPU computational resources are well utilized. GPU utilization is not as good but it is reasonable to expect improvement as the technology and our understanding of it matures. It is also reasonable to expect that ease-of-use will improve. To speed improvement of PGI and HMPP compilers, we have

provided NIM source code and F2C-ACC-generated CUDA code to both vendors and both have used it to improve their products. Finally, F2C-ACC has proven to be extremely valuable as a means to create correctness and performance baselines for evaluating commercial compilers and convincing vendors that improvements are necessary and possible.

We will continue this work with immediate focus on reduction of communication time to improve performance of multi-GPU runs and further performance tuning of the NIM dynamical core using more sophisticated HMPP features, more mature versions of the HMPP and PGI GPU compilers, and the new commercial GPU compiler being developed by Cray [22]. We will also continue to tune performance on CPUs (including addition of OpenMP directives as an additional parallelization option) and will investigate CPUs with larger core counts.

The next challenge will be integration of physics packages into NIM. Many of these physics packages do not share some of the characteristics of NIM (such as compact, vectorizable code) that made it relatively easy to parallelize for GPUs. Candidate physics packages for NIM have been selected and one has already been parallelized for GPUs [3]. The best approaches for obtaining good GPU performance when these packages are called from NIM are under investigation. Physics schemes typically have complex data dependence in the vertical dimension and it is likely that arrays stored with the vertical dimension fastest-varying will not be optimal for GPU computational efficiency. We will evaluate whether optimal performance is achieved by computing in-place or by transposing data between "dynamics" and "physics". Preliminary results indicate that transpose time is small compared to physics computation time.

We will then broaden our investigation to include AMD GPUs and the new Intel Many Integrated Core (MIC) devices as they become available. As the NIM dynamical core matures, we will test larger cases on large numbers of GPUs. We will continue to use F2C-ACC to provide feedback to GPU compiler vendors until their compilers can consistently beat it.

## ACKNOWLEDGMENT

We would like to thank several contributors who assisted us in key areas. Guillaume Poirier and the support staff at CAPS repeatedly provided valuable and timely assistance as we learned to use the HMPP directives. Dave Norton of High Performance Computing Associates helped with issues encountered using the PGI GPU compiler. We are grateful to the NOAA HPCC program office for supporting this work.

## REFERENCES

[1] P. Lynch, "The origins of computer weather prediction and climate modeling," J. Computational Physics, vol. 227, pp. 3431–3444, 2008.

[2] J. Michalakes and M. Vachharajani, "GPU Acceleration of Numerical Weather Prediction," Parallel Processing Letters, vol. 18, no. 4, pp. 531–548, 2008.

[3] G. Ruetsch, E.Phillips, and M. Fatica, "GPU Acceleration of the Long-Wave Rapid Radiative Transfer Model in WRF Using CUDA Fortran," http://www.pgroup.com/lit/articles/nvidia_paper_rrtm.pdf.

[4] B. Chapman, G Jost, R. van der Pas, Using OpenMP: Portable Shared Memory Parallel Programming. MIT Press, 2007.

[5] M. Govett, J. Middlecoff, T. Henderson, "Running the NIM Next Generation Weather Model on GPUs," Proc. 10th Intl. Conf. on Cluster, Cloud, and Grid Computing, Melbourne, Australia, pp. 792-796, 2010.

[6] CAPS website, http://www.caps-entreprise.com/index.php, 2011.

[7] Portland Group website, http://www.pgroup.com/, 2011.

[8] J. Lee, R. Bleck. A. MacDonald, J. Bao, S. Benjamin, J. Middlecoff, N. Wang, J. Brown, "FIM: A vertically flow-following, finite-volume icosahedral model," Preprints, 22nd Conf. Wea. Analysis Forecasting/ 18th Conf. Num. Wea. Pred., Park City, UT, Amer. Meteor. Soc., 2008.

[9] D. Williamson, "Integration of the barotropic vorticity equation on a spherical geodesic grid," Tellus, vol 20, pp. 642-653, 1968.

[10] R. Heikes, D. Randall, "Numerical integration of the shallow-water equations on a twisted icosahedral grid," Mon. Wea. Rev. vol. 123 pp. 1862-…, 1995.

[11] T. Ringler, R. Heikes, D. Randall, "Modeling the atmospheric general circulation using a sperical geodesic grid: a new class of dynamical cores," Mon. Wea. Rev., vol. 128, pp. 2471-2490, 2000.

[12] H.Tomita, M. Satoh, K. Goto, "A new dynamical framework of global non-hydrostatic model using the icosahedral grid," Fluid Dyn. Res., vol. 34, pp. 357-400, 2004.

[13] D. Majewski, D. LierMann, P. Prohl, B. Ritter, M. Buchhold, T. Hanisch, G. Paul, and W. Wergen, "The operational global icosahedral hexagonal gridpoint model GME: description and high-resolution tests," Mon. Wea. Rev., vol. 130, pp. 319-338, 2002.

[14] A. MacDonald, J. Middlecoff, T. Henderson, and J. Lee, "A general method for modeling on irregular grids," Intl. J. of High Performance Computing Applications, doi:10.1177/1094342010385019, 2010.

[15] W. Skamarock, J. Klemp, J. Dudhia, D. Gill, D. Barker, W. Wang, J. Powers, "A description of the advanced research WRF version 2," Technical Report NCAR/TN-468+STR, National Center for Atmospheric Research, 2007.

[16] J. Rosinski, D. Williamson, "The accumulation of rounding errors and port validation for global atmospheric models," SIAM J. Scientific Computing, vol. 18, No. 2, pp. 552-564, 1997.

[17] OpenCL website, http://www.khronos.org/opencl/, Khronos Group, 2011.

[18] S. Browne, J. Dongarra, N. Garner, G. Ho, P. Mucci, "A portable programming interface for performance evaluation on modern processors," Intl. J. of High Performance Computing Applications, vol. 14, no. 3, pp. 189-204, 2000.

[19] M. Govett, L. Hart, T. Henderson, J. Middlecoff, D. Schaffer, "The scalable modeling system: directive-based code parallelization for distributed and shared memory computers," J. of Parallel Computing, vol. 29, pp. 995-1020, 2003.

[20] Environmental Modeling Center, "The GFS atmospheric model," NCEP Office Note 442, Global Climate and Weather Modeling Branch, EMC, Camp Springs MD, 2003.

[21] V. Natoli, "Top 10 Objections to GPU Computing Reconsidered," HPCwire http://www.hpcwire.com/hpcwire/2011-06-09/top_10_objections_to_gpu_computing_reconsidered.html, 2011.

[22] M. Feldman, "Cray Unveils its First GPU Supercomputer," HPCwire http://www.hpcwire.com/hpcwire/2011-05-24/cray_unveils_its_first_gpu_supercomputer.html, 2011.

[23] J. Rosinski, General Purpose Timing Library website, http://www.burningserver.net/rosinski/gptl, 2011.

[24] NVIDIA Corp., "PTX Parallel Thread Execution ISA Version 2.3," http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/ptx_isa_2.3.pdf, 2011.

[25] NVIDIA Corp., "CUDA Profiler," http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/visual_profiler_cuda/CUDA_Profiler_3.0.txt, 2011.