

# Evaluation of the FV3 Dynamical Core

Mark Govett and Jim Rosinski  
NOAA Earth System Research Laboratory  
July 8, 2016

## Summary and Recommendations

FV3 is a global atmospheric dynamical core, adaptable to both climate and forecast applications. It is written almost exclusively in the Fortran language. The underlying grid is a cubed sphere, which has six faces encompassing the entire globe. The code is quite well-written, though new users may be confused by the mixed use of preprocessing “ifdef” conditionals alongside Fortran conditionals. There was no Users Guide, but limited documentation was available in the scripts and text files.

Computational parallelism is implemented via hybrid OpenMP (threading) and MPI (message-passing). This is a common and well-tested approach to achieving effective utilization of multiple nodes which each contains multiple processor cores. Partitioning of the domain between MPI tasks and OpenMP threads is done at run-time, which simplifies the process of tuning application performance for the underlying architecture. The code scales well to very high processor counts. The test case provided to us invokes vertical remapping once every 11 dynamics time steps. As a result, vertical remapping consumes less than 10 percent of the total run time. However, remapping could require as much as 50% of the execution time if it were called every model time step.

The code requires a minimum of six MPI tasks to run, one for each face of the cube-sphere grid. MPI communication is implemented via a sophisticated locally-developed library infrastructure (GFDL’s FMS, for “flexible modeling system”) which wraps all the MPI calls. Similar to most models, MPI message packing and unpacking are done as part of the communications. Some overlapping of asynchronous communication with computation is done, reducing the overall cost of communication. However, communications can consume 30% of the runtime in some configurations, which suggests further potential improvements should be explored.

Porting the code to fine-grain architectures was trivial. The full model was ported to the MIC KNC, as well as the KNL processor. Performance of an extracted kernel using a 2016 generation, early release KNL (64 cores) was compared to an Intel Haswell and gave a 2X benefit favoring the KNL processor. The performance boost with KNL is largely due to improved, high-speed MCDRAM memory packaged on the chip.

Due to a compiler bug, we were unable to run the full model on the CPU or GPU using the PGI (Portland Group International) compiler, so we extracted an important kernel to evaluate GPU performance. GPU-CPU comparisons were made using 2013 generation chips that showed a 60 percent performance benefit (1.6X) favoring the K40 GPU over the IvyBridge processor. We believe further improvements are likely. In addition, the latest generation NVIDIA Pascal chip is expected to give a

large boost in application performance on the GPU due to a projected 3X increase in memory bandwidth.

## Recommendations

- Improve documentation, build and run scripts for general use as a community model.
- Modify the code so it can be run in serial mode (w/o MPI).
- Investigate potential load imbalance due to edge and corner cell calculations on the cubed-sphere grid.
- Explore communications optimizations including elimination of message packing/unpacking along dimensions where data are already contiguous.
- Explore the use of compile-time constants in array declarations and loop bounds to improve performance on all architectures.
- Adapt code and scripts to use non-Intel compilers (eg. Cray, PGI, IBM, gfortran, NAG), to improve model portability.
- Evaluate the performance of the IBM (OpenMP) and Cray (OpenACC) compilers on GPUs.

## Background

The FV3 code provided to us by GFDL (NVIDIA.tar.gz) was their entry into the NGGPS phase 1 competition. This included the dycore only, with no physical parameterizations. We ported the full dycore to a variety of platforms. The CPUs contained in these platforms spanned a range of Intel-based cores, including SandyBridge, Haswell, and Broadwell. We also ported the FV3 code to both current and next-generation MIC architectures. These included the Knights Corner system at TACC named "stampede", as well as early access to a Knights Landing system (KNL) courtesy of Intel Corporation.

The FV3 team provided us with the latest version of their model in mid May 2016, but we lacked the time and resources to use it for performance evaluation. However, aside from the vertical remapping, we were assured by GFDL's Rusty Benson that the code was essentially the same as the NGGPS Phase I version. Therefore, results from this evaluation are valid and useful.

The analytic test case provided with the code (GFDL test case #13) has 127 vertical levels, and transports 10 constituents. The horizontal resolution we chose for all of our analysis was 192x192 points across the "i" and "j" dimensions of each of six faces on a cubed sphere (approximately 1/2 degrees), or somewhat more than 200,000 horizontal grid points. This configuration is sufficient to allow scaling analysis, but not so large as to require a huge portion of a supercomputer. The default configuration runs in 64-bit precision, but a 32-bit option was also provided. Most of our analysis utilized the 64-bit option. GFDL personnel provided copious scaling data which clearly indicated that increasing processor count alongside an equivalent increase in horizontal resolution ran equivalently fast. This positive "weak scaling" result was not a surprise, and bodes well for extremely high resolution runs in the future. With no physical parameterizations present, a 2-hour simulation proved sufficient to achieve a representative sample of all model routines exercised.

## Code modifications

Performance data presented in this report are based upon the source code provided by GFDL, with some optimizations implemented by ESRL. The primary optimization was the addition of a compiler flag (-xCORE-AVX2) to take full advantage of vector capabilities in the underlying hardware in Haswell and newer Intel-based architectures. The NOAA machine “theia” is Haswell. FV3 exhibited more than a 12% speedup vs. not including this modification when using 6 nodes on theia. Some additional minor speedup was achieved via modifications involving fusing OpenMP loops where possible, to minimize the combination of thread synchronization overhead and load-induced thread imbalance. In addition to performance enhancements, we instituted the use of a thread-safe timing library (GPTL) for the purposes of estimating percent of peak performance achieved, estimating load imbalance across OpenMP threads and MPI tasks, estimating threading overhead, and assessing MPI performance based on the volume of data exchanged via explicit messages.

## Performance profile and scaling

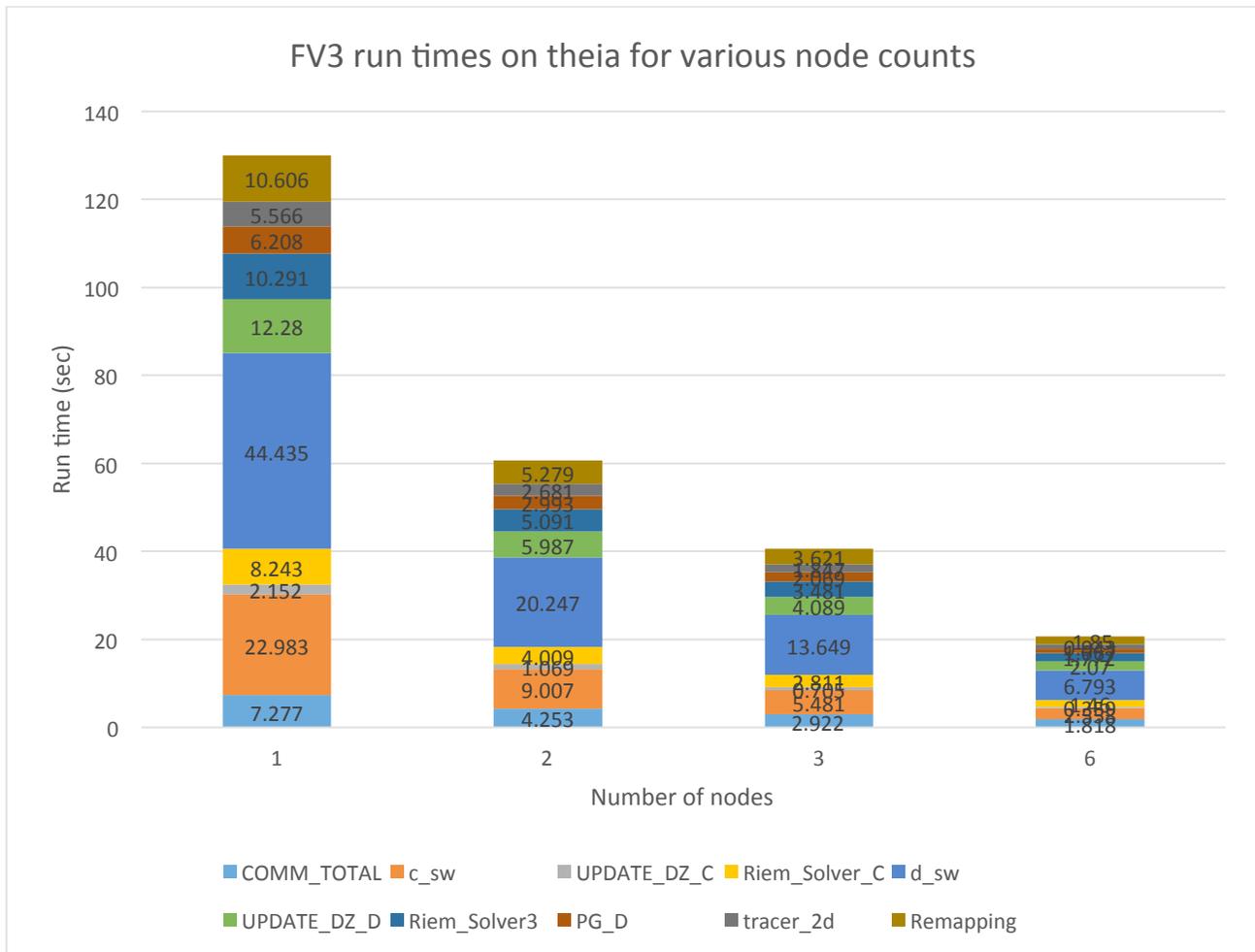


Figure 1. Timing profile for FV3 dynamics for various node counts on NOAA machine “theia” (Intel Haswell processors). In all cases best performance is presented, which turned out to be 2 MPI tasks in the X dimension and 2 MPI tasks in the Y dimension for each of the 6 cubed sphere faces.

Figure 1 profiles important code regions in FV3 at the default 1/2 degree horizontal resolution for various node counts as run on the NOAA Haswell-based machine "theia". Scaling is excellent from 1 to 6 nodes, even showing super-scaling in some cases, especially moving from 1 node to 2. Cache effects are the most likely cause of the super-scaling. Assuming good weak scaling (data provided by GFDL confirms this), we can extrapolate and expect similarly good scaling from 6 nodes at a 1/2 degree resolution to 24 nodes at a 1/4 degree resolution, and so on up to 6144 nodes at a horizontal resolution of 1/32 degree (approximately 3.7 km). Of course this extrapolation assumes a quality interconnect on the machine on which the model is being run. In other words, if the communication time (labeled "COMM\_TOTAL") becomes a much greater fraction of the total run time at much higher node counts and higher resolutions due to a poor interconnect, then the weak scaling expectation will no longer be valid and model run times will be longer than expected.

## Code structure

Computationally, horizontal advection and vertical remapping are done in separate sets of subroutines. For each invocation of vertical remapping, there are 11 executions of the "small-step horizontal dynamics". In this configuration, the relative cost of vertical remapping is minimal. However, if various factors such as wind, terrain, or physical parameterizations embodied in a realistic simulation necessitate more frequent invocation of vertical remapping, the relative cost of remapping can become very significant. The cost of remapping if needed once per small-step could approach 50% of the aggregate computational cost of the FV3 dynamics.

FV3 is a hybrid MPI/OpenMP code. Partitioning of the horizontal domain across OpenMP threads and MPI tasks is done at run-time. The horizontal grid is a cubed sphere, which contains six "faces" mapped across the globe. This grid specification is one approach to alleviating the notorious "pole problem" associated with converging meridians in earlier numerical models set on a latitude/longitude grid. Currently there must be at least a single MPI task operating on each face of the cubed sphere. Thus a minimum of six MPI tasks must be employed in an FV3 run.

Data layout in FV3 is (i,j,k) (Fortran ordering). Computationally expensive routines `d_sw` and `c_sw` both drive "shallow water" aspects of the simulation, which means independence of the "k" portion of the calculations. This dimension of parallelism is exploited via OpenMP parallelism. With 127 vertical levels, this is more than adequate to saturate all available cores on current CPU-based hardware. The amount of work performed at each vertical level in these routines is perfectly load balanced. Though some induced load imbalance will probably exist on any real machine due mainly to the fact that the number of cores most likely does not evenly divide into the number of vertical levels (or vice versa)

The FV3 code is highly vectorized. Interior loops in both the shallow water dynamics and the vertical remapping contain independent calculations which can easily vectorize without any additional prodding via compiler directives or hints. Any realistic horizontal resolution or MPI task count enables utilization of the full 512-bit length of the vector register on the latest MIC platform (KNL).

For vertical remapping, data layout remains (i,j,k), as in the shallow water portions of the model. But due to many dependencies in the vertical dimension, OpenMP threading is applied over the "j"

dimension rather than "k". Loop order within remapping is therefore OpenMP over "j" on the outside, followed by "k", and then vectorizable loops over "i" on the inside. Since loop order doesn't exactly match array order, ("j" and "k" are reversed) there will be some cache inefficiencies which are not present in the shallow water parts of the code. But given stride-1 vectorized inner loops, with a threaded loop outside of that, performance of the vertical remapping portion of FV3 is quite good. Due to data dependencies in the vertical, most "k" loops within remapping cannot be threaded or vectorized. But vectorizing over "i" and threading over "j" provide sufficient parallelism for good performance on fine-grain MIC KNC processors, which contain 512-bit vector registers on each core, and up to 244 thread contexts on each card.

## MPI Communication

The horizontal dynamics requires MPI communication every time step. The communication is across horizontal boundaries of the regions owned by each MPI task. The code is written to allow overlapping of communication with computation if the underlying architecture supports it. This is done by using asynchronous communication (MPI\_Isend/MPI\_Irecv), followed by some amount of computation which can be done alongside the communication, followed by MPI\_Wait which defines the point at which boundary data being communicated must be present before computation can proceed.

Like many numerical models, FV3 employs its own wrapping communication layer(s) on top of calls to intrinsic routines in whatever MPI library is being used. One of the tasks of the higher level routines is to "pack" data destined for communication into user-space buffers prior to sending via MPI messages. Once received, the data are then "unpack"ed from user-space buffers into their appropriate internal array locations. This packing and unpacking can introduce significant overhead, depending on the size of the horizontal regions owned by each MPI task. We note that in an (i,j,k) model such as FV3, packing and unpacking of data being communicated across the northern and southern boundaries of each MPI task adds unnecessary overhead. Since "i" is the innermost dimension, the data are already contiguous and ready to be sent/received in blocks of reasonable size. Packing and unpacking does allow for fewer, larger messages, but we suspect that this slight advantage is not offset by the penalty imposed by employing the CPU to pack and unpack the data. Of course this argument does not apply along the lateral ("j") dimension, since the data are not contiguous in this dimension. Thus packing and unpacking is required along these boundaries.

MPI communication can be expensive. In FV3, the cost of communication can be up to 30% of the model run time at task counts which are high relative to the size of the horizontal domain. Therefore, minimizing the amount of packing and unpacking of data, and overlapping computation with communication wherever possible, is important if one is to achieve good scaling and overall time to solution. We found that comparing FV3 to NIM (a global forecast model developed at NOAA/ESRL which was also part of the NGGPS phase 1 evaluation), FV3 requires less volume of data to be communicated per time step, but that the overall cost of communication was somewhat larger. Some of this discrepancy is no doubt due to the fact that FV3 packs and unpacks its messages while NIM does not. Higher communication cost may also be due to the extent to which overlapping of computation with communication is possible in each model. More investigation is required.

## Suitability for fine-grain architectures

In this section we describe results from porting both the full FV3 dynamic core, and an extracted computationally expensive kernel, to fine-grain architectures from NVIDIA (GPU) and Intel (MIC).

### A. Full dynamical core

In early work, we ported the FV3 model to the “stampede” system at the Texas Advanced Computing Center (TACC). This machine utilizes previous-generation Intel MIC chips (KNC) which require a CPU-based host. Getting the full FV3 dycore compiled and running was straightforward. Just a matter of building required libraries and adding the appropriate compiler flags. Performance (Figure 2) was similar to and possibly somewhat better than that achieved with other atmospheric models such as WRF. Specifically, per-card performance on the MIC compared favorably to a single socket of the SandyBridge (circa 2013) host, but was slower than a full (2-socket) SandyBridge node. Communication time (labeled “COMM\_TOTAL” in the figure) stands out as a major performance issue on the KNC. But MPI communication problems are a known issue on KNC devices, and Intel expects these to be addressed in the next release of MIC hardware (Knights Landing or KNL), expected in June 2016. If we don’t consider communication performance, on the MIC (stampede KNC) FV3 runs about 64% as fast per card as on the SNB host.

In April 2016, Intel announced availability of its newest entry in the MIC series of fine-grain machines, KNL (for “Knight’s Landing”). Intel provided us access to an early version of this processor, which had fewer processor cores than the released product will have (64 instead of 72). This machine has several advantages over its predecessor KNC architecture. Primary among the advantages is KNL is a self-booting Linux system, with no supporting CPU host required. This simplifies porting, and makes MPI communication faster because messages no longer need to go through the host. Another exciting feature of KNL is the availability of so-called MCDRAM, a form of main memory with much faster bandwidth than traditional main memory.

Porting the full FV3 code to early KNL systems was trivial, easier than the KNC port described above mainly because a host compilation alongside the KNL compilation was not necessary. Overall performance on six nodes of a pre-release system containing 64 cores per node was roughly equivalent to the Haswell-based performance described in Figure 1 above, when utilizing the high-performance MCDRAM memory. A figure describing the breakdown of performance by routine is not shown here, for the following reason. In the other figures we enabled artificial MPI barriers at strategic points in order to avoid having any load imbalance misinterpreted as time spent in another region. Adding these barriers did not have a significant impact on CPU-based machines or KNC. But for some as yet unknown reason adding the artificial barriers did have a significant impact on overall performance on the KNL. Further study is needed.

On average, FV3 executes over 16 OpenMP loops each model time step. We were concerned that this many thread dispatches and synchronizations (“threading overhead”) each time step might cause performance problems, especially on fine-grain machines which have more, slower, cores than

traditional CPU-based systems. So we used the GPTL timing library to measure load-induced imbalance as well as threading overhead in a few of the expensive OpenMP loops in FV3. We define “load-induced imbalance” as the difference in computation cost between slowest and fastest thread for a given loop. This imbalance is generally unavoidable, unless the number of available hardware thread contexts happens to divide evenly into the loop iteration count, and the work per iteration is the same. We measured threading overhead by timing the execution of the entire loop, and subtracting the cost of the slowest thread for the work done inside the loop. Surprisingly, even on the stampede MIC system the threading overhead cost was minimal. Thus we conclude that even 16 or more OpenMP loops per FV3 model time step will not be a bottleneck on MIC systems.

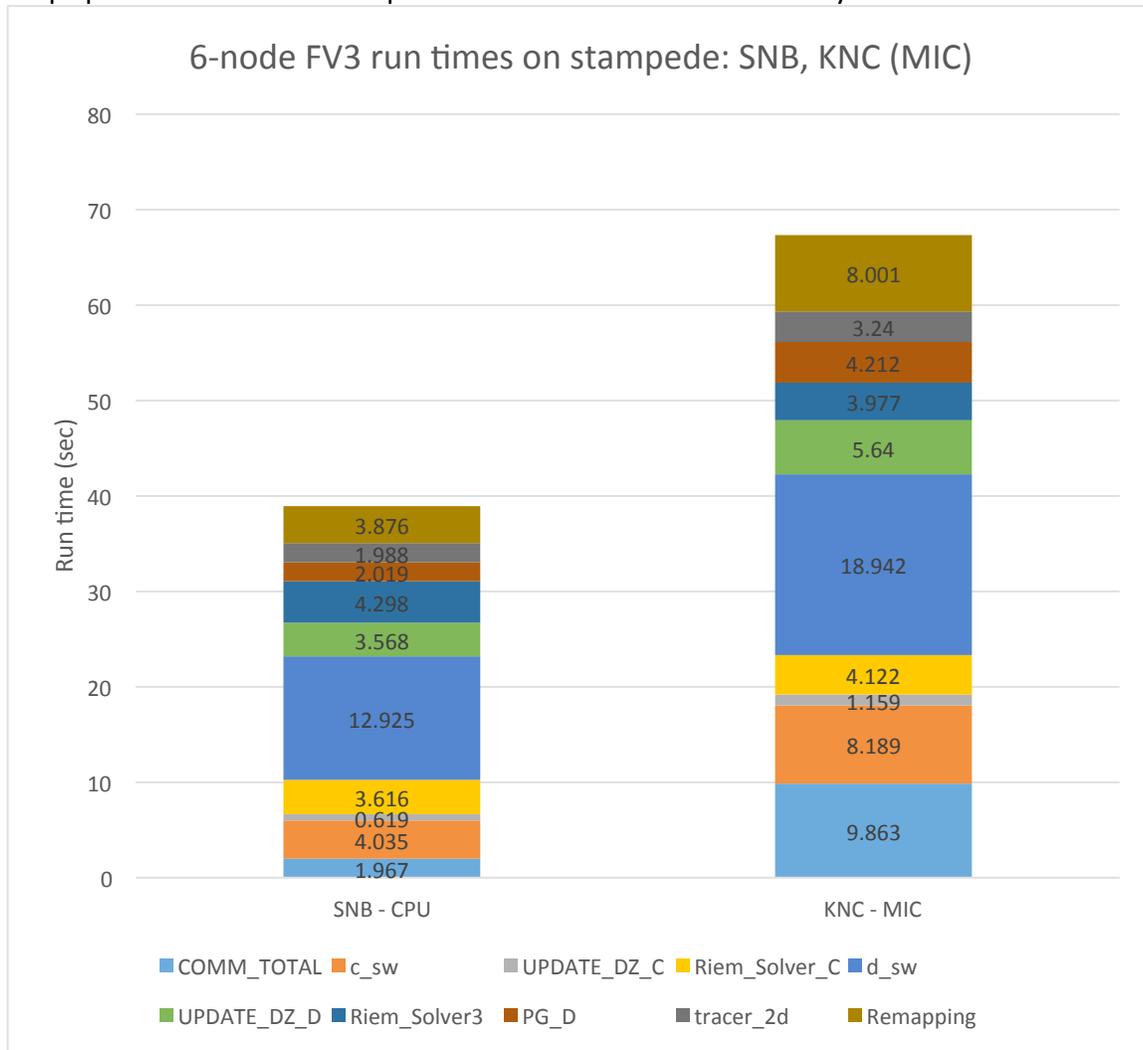


Figure 2: Comparison of FV3 6-node run times on stampede machine using SandyBridge CPU processors (SNB) vs. MIC (KNC) processors. Best configuration on SNB was 2 MPI in X and 2 MPI in Y. Best configuration on MIC was 4 MPI in X and 4 MPI in Y

We did find however, that load-induced imbalance can be significant on the MIC. In FV3 most threading is done over the vertical index. Also, in most cases dividing the MPI domain into 4 equal sized squares for each face (2 in X, 2 in Y) proved optimal. Running 1 node per face, the number of available thread contexts (240 on stampede) does not divide evenly into the total number of vertical levels across all MPI tasks on the card ( $2 \times 2 \times 127 = 508$ ). By reducing the number of vertical levels to 120 so that

240 now divides evenly into the total number of vertical levels on the card ( $2 \times 2 \times 120 = 480$ ), this minor decrease in number of levels (5.5%) resulted in a speedup of over 15% in model run time. It is likely suboptimal to choose vertical resolution based on the machine hardware. Thus a mechanism to mitigate this load imbalance problem on MIC platforms is to ensure that the total number of threaded iterations is sufficiently larger than the number of thread contexts to minimize the load-induced imbalance.

## B. Extracted Kernel

We extracted and built a driver around a computationally expensive portion of the FV3 code in order to isolate and study its performance characteristics, and possible opportunities for data and coding rearrangement for execution on fine-grain architectures (GPU and MIC). The extracted kernel was for a full  $192 \times 192$  horizontal domain of a single face of the cubed sphere, with 127 vertical levels just as in the full model runs described earlier. The routine chosen was `c_sw`, whose performance profile within the full model context is shown above in Figure 2. OpenMP parallelism in this routine is over the vertical (`k`) index. Two variants of the `c_sw` kernel were constructed. The first retained the existing `i-j-k` data ordering, but in order to accommodate the GPU compiler, the vertical “`k`” index was pushed down into the routine being threaded rather than kept outside the routine as in the original code. This modification had little impact on CPU run-times, so comparisons between CPU and GPU and MIC are still valid. The second variant of the `c_sw` kernel swapped the data ordering from `i-j-k` to `k-i-j`. Using the Intel compiler to build and run, we were able to demonstrate bitwise exact results compared to the original code for both the `i-j-k` and `k-i-j` variants.

Fine-grain assessment has focused on using `i-j-k` and `k-i-j` variants of `c_sw` extracted from FV3 to compare performance between CPU, GPU and MIC processors. It should be noted that the `k-i-j` variant required significant modifications to the code, including changing the order of each 3-dimensional (3D) array declaration, promoting 2D arrays to 3D, and reordering loops. This variant was intended to investigate reordering as an option to improve parallelism. A focus was to improve performance of edge and corner calculations inherent in the cube-sphere grid, currently serialized in the `i-j-k` loop ordering.

Successful validation of model output was followed by attempts to get FV3 running on the CPU with the PGI compiler, as a first step toward performance assessments on the GPU. Unfortunately, the PGI compiler was not able to run FV3 on the CPU, due to what appears to be a bug in the compiler. Working with PGI engineers, the bug was traced to a namelist read in one of FV3’s support libraries (called “`mpp`”). PGI’s Dave Norton indicates the bug will be fixed in a future release. As a result, continued work on the full model has been postponed until either the bug is fixed, or the FV3 code is modified. We also plan to try using the Cray and IBM GPU compilers on the full model.

### 1) GPU

The NVIDIA PSG cluster was used to run select portions of the FV3 model on NVIDIA GPUs. This cluster was used because it has three generations of GPU chips: K20x (2012), K40 (2013) and K80 (2014). PSG

also has the latest Fortran GPU compilers from PGI used for GPU parallelization and optimization. A Fortran GPU compiler from Cray is also available but was not used for this study. Both compilers adhere to the openACC specification, a directive-based parallelization standard for fine-grain processors (CPU, GPU, MIC). OpenACC compilers are an alternative to OpenMP based compilers. IBM is developing an OpenMP based Fortran GPU compiler that will be available in a few months.

With standalone kernels generated, OpenACC directives were added to the code. There are two general approaches to GPU parallelization using the `!$acc kernels` and `!$acc parallel` directives. The **kernels** directive gives the compiler maximum flexibility to parallelize the calculations. The PGI compiler was able to generate coarse grain (block) parallelism over the k-loop, and fine-grain (thread) parallelism on the inner-most i-loop. The PGI compiler was not able to utilize the j-loop for further parallelism. Use of `!$acc loop collapse` to combine the “k” and “j” loops into a single larger loop was tried, but did not improve performance in most cases. We have not tried using the `!$acc parallel` directive. GPU parallelism using the **parallel** directive is more restrictive because the standard does not allow multi-dimensional block parallelism even though the GPU device supports it.

Using **kernels**, a few changes to the kernels were made to work around PGI compiler bugs that surfaced when compiling and running on the GPU. Parallelization was quite easy, with insertion of a single “`!$acc kernels`” directives at the top of each subroutine (identical placement as the openMP directives). Diagnostic output generated by the PGI compiler showed what the compiler was doing including identifying data movement, loops that would be parallelized, variable dependencies, and loops that could not be parallelized due to dependencies in the code. This information guided parallelization and optimization.

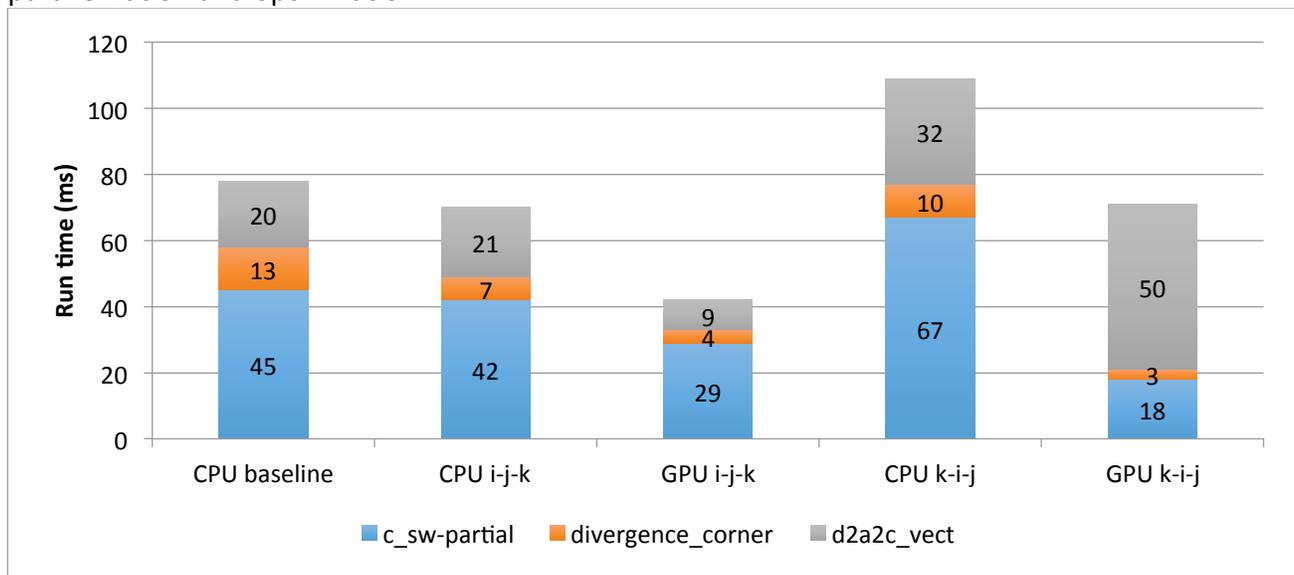


Figure 3: Comparison of performance on a dual-socket IvyBridge CPU and NVIDIA K40 GPU for three variants of the FV3 subroutine “c\_sw”. The main sub-routines are shown, with “c\_sw partial” representing the time needed to run “c\_sw” minus “divergence\_corner” and “d2a2c\_vect”.

Figure 3 compares performance for the three variants of “c\_sw” on a dual-socket 20 core IvyBridge CPU and the same 2013 generation NVIDIA K40 GPU. Execution times (in milliseconds) had little run-to-run

variability, increasing our confidence in the performance results. Execution time for the main routines are shown, with “c\_sw-partial” representing the total “c\_sw” runtime minus the two main subroutines it calls: divergence\_corner and d2a2c\_vect.

Performance results for the i-j-k variant show the NVIDIA K40 processor was 60% faster than the Intel IvyBridge (70 versus 42 ms). This is significantly better than early work by NVIDIA on a similarly coded kernel from FV3 (subroutine d\_sw). Additional performance improvement may be possible using the **parallel** directive instead of **kernels**. The CPU results relied on direct pinning of OpenMP execution threads to the CPU cores. Comparing CPU performance, it is not clear the reason the i-j-k variant is faster than the k-i-j variant; further investigation is needed. Given that significant code changes were required for the k-i-j variant and it gave no performance benefit on the GPU, most of the optimization work focused on the i-j-k variant.

### 1) MIC

Figure 4 profiles the same c\_sw kernel described above (i-j-k version) on a 24-core Haswell CPU and pre-release 64-core KNL system (courtesy of Intel Corp.). We note that production versions of the KNL architecture are scheduled to contain 72 cores, but none were available as of this writing. Most striking in the comparison is how much value is added by utilizing the high-bandwidth memory (MCDRAM) available on the KNL. Other weather models and kernels (e.g. NIM, MPAS) show similar 2-3X speedup when employing MCDRAM vs. standard DDR memory. Though without MCDRAM the KNL is a bit slower than its CPU-based Haswell brother.

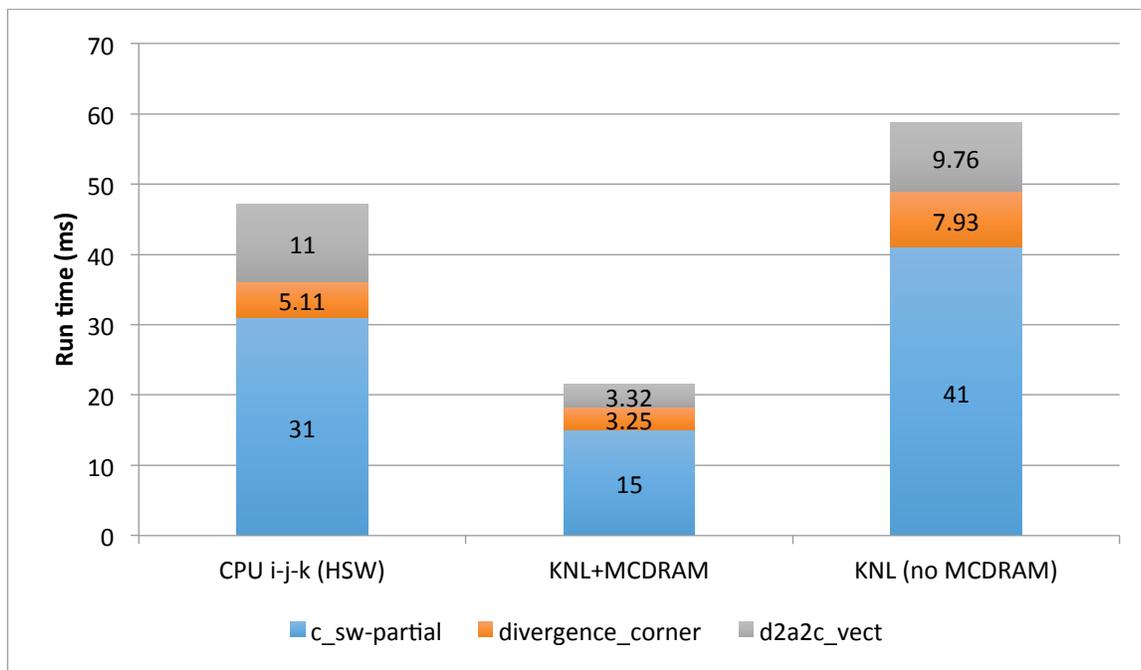


Figure 4: c\_sw kernel performance on CPU (Haswell-LHS), KNL with MCDRAM (center), and KNL without MCDRAM (right). Compare to Figure 3 for GPU and earlier generation CPU results.