Journal of Computational Science 9 (2015) 1–6

Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

A case study of CUDA FORTRAN and OpenACC for an atmospheric climate kernel

Matthew Norman[a,*], Jeffrey Larkin[b], Aaron Vose[c], Katherine Evans[a]

[a] Oak Ridge National Laboratory, Oak Ridge, TN, USA
[b] Nvidia, Santa Clara, CA, USA
[c] Cray Seattle, WA, USA

ARTICLE INFO

Article history:
Available online 18 April 2015

Keywords:
OpenACC
Climate
CUDA
GPU
HPC

ABSTRACT

The porting of a key kernel in the tracer advection routines of the Community Atmosphere Model – Spectral Element (CAM-SE) to use Graphics Processing Units (GPUs) using OpenACC is considered in comparison to an existing CUDA FORTRAN port. The development of the OpenACC kernel for GPUs was substantially simpler than that of the CUDA port. Also, OpenACC performance was about 1.5× slower than the optimized CUDA version. Particular focus is given to compiler maturity regarding OpenACC implementation for modern FORTRAN, and it is found that the Cray implementation is currently more mature than the PGI implementation. Still, for the case that ran successfully on PGI, the PGI OpenACC runtime was slightly faster than Cray. The results show encouraging performance for OpenACC implementation compared to CUDA while also exposing some issues that may be necessary before the implementations are suitable for porting all of CAM-SE. Most notable are that GPU shared memory should be used by future OpenACC implementations and that derived type support should be expanded.

1. Introduction

This study considers the viability of using the OpenACC standard as currently implemented in particular by the Cray and PGI compilers from a best/easiest case perspective of an easily ported and highly performant kernel in the atmospheric climate model CAM-SE (Community Atmosphere Model – Spectral Element) [6,14,9], a part of the ACME (Accelerated Climate Model for Energy) coupled climate model code. CAM-SE is a capability-scale, IPCC-class atmospheric climate model intended to simulate climate projections over large time scales. Based on the Spectral Element (SE) method implemented on cubed-sphere topology, it is a highly scalable code with minimal data communication across domain decomposition boundaries. Run on a 14 km mesh, the code scales successfully out to 14 K nodes on the Titan supercomputer at Oak Ridge National Laboratory. Given the acquisition of Graphics Processing Units (GPUs) on each node of Titan, the tracer advection routines of CAM-SE were ported to GPUs using CUDA FORTRAN with good success [4,10]. This study begins the process of considering the use of OpenACC in porting more of CAM-SE's runtime profile to GPUs.

This is an important avenue to investigate because climate simulation is always in need of better efficiency and greater computing power. There has been a strong push to develop Global coupled Climate Models (GCMs), particularly the atmospheric components, with unprecedented horizontal resolution. Doing so has enabled high-resolution simulations that capture and characterize the frequency of tropical cyclones [2], improve the frequency of blocking flow events [3], and improve tropical precipitation and circulation [5,7]. These early results invite additional questions about the interaction between small- and large-scale atmospheric features on long-term, global scales. Some recent so-called "hero runs" (performed at very high resolution) focused on capturing new global-scale atmospheric behavior, and yet there are still plenty of interesting climate questions to answer regarding smaller scales, where extreme events involving precipitation and air quality, for instance, occur.

Improving multi-scale interactions is a primary scientific focus for next generation GCMs. We need to explore model sensitivity at high resolution to better understand regional and global climate variability. Also, multi-ensemble runs (either straightforwardly or via a reduced parameter space exploration [1,11]) help validate and characterize high resolution model results. Ensemble runs introduce additional parallelism since each run can be performed

* Corresponding author.
E-mail addresses: normanmr@ornl.gov (M. Norman), jlarkin@nvidia.com (J. Larkin), avose@cray.com (A. Vose), evanskj@ornl.gov (K. Evans).

http://dx.doi.org/10.1016/j.jocs.2015.04.022
1877-7503/

independently, and this could significantly reduce MPI overheads, better utilize charged core-hours, and increase total utilization of the target platform. More computing also enables so-called "super-parameterization" schemes for greatly improved physical fidelity in resolving the climatic influence of clouds [8]. Being on separate grids, these models could also be run in parallel with the main climate model for improved performance. Also, increasing the number of vertical levels would be enabled by more computing power. Strong vertical sensitivities have led to hesitation in increasing vertical levels while refining in the horizontal, and this has resulted in an inconsistency between the dimensions. Several studies have shown benefits in the mean climate with increased vertical resolution [13,12].

Before considering all aspects of OpenACC porting, it needs to be studied and documented whether OpenACC implementations in current compilers are capable of competing with CUDA implementations. The benefits of OpenACC include better portability between compilers and devices as well as greater ease in porting. The ease in porting is due to many factors. First, the management of host and device memories, including allocations, deallocations, and PCI-e data movement between the two, can be managed more seamlessly via directives. Also, low-level, device-specific considerations such as shared memory usage and banking conflicts are removed in OpenACC and placed in the compiler's responsibility for management rather than the user's. Many inconveniences such as partitioning work across GPU grids and blocks are made simpler as well during kernel launches. When mature, OpenACC shows promise for greatly easing the porting effort as well as increasing the available compilers to choose from. Given that CUDA FORTRAN is implemented only by PGI, OpenACC slightly improves portability since it is currently implemented by PGI as well as Cray compilers, with more compilers promised in 2015.

However, the standard is not mature in compiler implementation at this point. For instance, both Cray and PGI have certain difficulties with transferring parts of FORTRAN derived types over PCI-e using directives. The Cray compiler, through "deep copy" support, is more adept at correctly managing derived types onboard the GPU, but it still has significant flaws in the implementation. For instance, one cannot currently pass only part of a derived type with either PGI or Cray implementations. The attempt at doing such typically results in the transfer of the entire derived type rather than just the subset one wishes to transfer, even if that subset is a contiguous block of memory whose bounds are known at compile time. These issues should be resolved in coming implementations, but they mark key issues for OpenACC to resolve for many modern FORTRAN codebases. Note that this is not necessarily an OpenACC standard issue, as the standard itself does not currently address the nuances of partial derived type transfers, the syntax involved, or the expected behavior from an implementation. CUDA Unified Memory may simplify the handling of complex data structures by allowing them to be available both on the host and device, greatly simplifying these data structures for both the compiler and the user. Additionally, the OpenACC technical committee is in the process of designing directives for better handling of complex data structures in a future version of the specification. At the time of writing, however, neither are readily available.

The purpose of this paper, though, is to look past many of the unresolved issues for a full OpenACC port and to ask a more bounded and pointed question. How does OpenACC compare against an existing CUDA FORTRAN implementation for a key kernel in CAM-SE's tracer transport routines in terms of runtime and in terms of ease of porting? Granted, neither the OpenACC nor the CUDA implementations may be entirely optimal, but they are both attempts at porting from an experienced user of CUDA and OpenACC with the help of vendor advice and support. Successful handling of this kernel is a necessary but not sufficient step in the path to a full port. It is something worth investigating in depth.

2. Code and compiler discussion

The machine used in this study is Oak Ridge Leadership Computing Facility's Titan supercomputer, which has a 16-core AMD interlagos processor and a Kepler K20× GPU on each node. CAM-SE is a FORTRAN 90 codebase implemented with several mechanisms of OpenMP parallelism (coarse grained or fine grained threading) and MPI domain decomposition. Each MPI task operates on `nelemd` columns of elements, and in the most widely used OpenMP implementation, the columns are chunked into parts, and each thread operates on a separate chunk within an "`omp parallel`" clause. Each column consists of `nlev` vertical levels, and each vertical level contains `np×np` basis function coefficients to be updated. There are `qsize` independent tracers to solve for. The kernel chosen for this study is the beginning part of a routine called `euler_step`, which performs the variational SE advection operator. The dimension sizes are `np`=4, `nlev`=30, `qsize`=50, and `nelemd`=32. 32 columns of elements per node represents the usual scaling limit for CAM-SE when run in production mode at scale – the point at which MPI waiting begins to dominate the runtime profile. In all, for this kernel, there are `np×np×nlev×qsize×nelemd`=768,000 independent indices over. The CPU version of the kernel uses the coarse-grained OpenMP implementation, which achieves an 11.5× speed-up on Titan's AMD Interlagos processors using 16 cores versus 1 core. So the CPU code is a competitive comparison point when comparing against the Kepler GPU performance.

The CPU code is given in Fig. 1. A CUDA kernel taken from the existing implementation in the ACME codebase is also given in Fig. 2. The derived types have been changed into single arrays by placing `nelemd` as the last index. Note that the optimized CUDA code is hardly recognizable, and only then because of common variable names. This is one of the disadvantages of using an optimized CUDA code. When changes are made in the main codebase, it is difficult to mirror them in the CUDA code. It also took a very large effort to get the CUDA kernels into forms like this for best performance, and this level of effort for large sections of the code is not very feasible. Therefore, we are looking to OpenACC as a more viable and portable option going forward for porting to Titan's GPUs.

Finally, the OpenACC code is given in Fig. 3. Many transformations to the CPU code were attempted, and the most impactful are distilled here. There were five main modifications required to get good performance out of the OpenACC code. First, the inner loops could not loop over so small an index of just `np×np`. Rather, since on the GPU the inner loops will form threads run within an SM, it was found that looping over `np×np×nlev` gave enough threads for good performance. Thus, the loops and temporary variables were transformed to loop over `np×np×nlev`. Next, there were variables such as `dinv`, `vstar`, and `gv`, which did not have `np×np×nlev` as the fastest varying array indices. Since these are the indices used for threading on the GPU, this would lead to poor DRAM coalescing over the bus. Therefore, these variables were transformed. Third, the last two loops of the CPU code were transformed to remove the synchronization between them. There were two reasons this improved OpenACC performance. First, the variables `dp_star` and `vvtemp` are not placed in shared memory by OpenACC. Thus, removing these temporary arrays removes communication to and from DRAM for a big improvement. Also, as implemented, there's an implied `__syncthreads()` call between each successive trip through the inner loops, and the removal of this improved performance. Fourth, some temporary variables were added for the first loop to explicitly reduce the frequency of access to arrays located in DRAM. This improved performance for the PGI OpenACC

```
do ie = nets , nete
  do q = 1 , qsize
    do k = 1 , nlev
      do j = 1 , np
        do i = 1 , np
          gv(i,j,1) = elem(ie)%metdet(i,j) * &
                      ( elem(ie)%Dinv(1,1,i,j)*Vstar(i,j,1,k,ie) +   &
                        elem(ie)%Dinv(1,2,i,j)*Vstar(i,j,2,k,ie) ) * &
                      elem(ie)%state%Qdp(i,j,k,q,n0_qdp)
          gv(i,j,2) = elem(ie)%metdet(i,j) * &
                      ( elem(ie)%Dinv(2,1,i,j)*Vstar(i,j,1,k,ie) +   &
                        elem(ie)%Dinv(2,2,i,j)*Vstar(i,j,2,k,ie) ) * &
                      elem(ie)%state%Qdp(i,j,k,q,n0_qdp)
        enddo
      enddo
      do j = 1 , np
        do l = 1 , np
          dudx00 = 0.0d0
          dvdy00 = 0.0d0
          do i = 1 , np
            dudx00 = dudx00 + deriv%Dvv(i,l)*gv(i,j,1)
            dvdy00 = dvdy00 + deriv%Dvv(i,l)*gv(j,i,2)
          enddo
          dp_star(l,j) = dudx00
          vvtemp (j,l) = dvdy00
        enddo
      enddo
      do j = 1 , np
        do i = 1 , np
          dp_star(i,j) = ( dp_star(i,j) + vvtemp(i,j) ) * ( elem(ie)%rmetdet(i,j)*rrearth )
        enddo
      enddo
      elem(ie)%state%Qdp(:,:,k,q,np1_qdp) = elem(ie)%spheremp(:,:) *               &
                                            ( elem(ie)%state%Qdp(:,:,k,q,n0_qdp) - &
                                              dt * dp_star(:,:) )
    enddo
  enddo
enddo
```

Fig. 1. Original CPU kernel code. The variables `gv, metdet, dinv, vstar, qdp, dudx00, dvdy00, Dvv, dp_star, vvtemp, spheremp, rmetdet, dt`, and `rrearth` are all double precision floats, whose dimensions can be inferred by the indexing except for `qdp`, which has the dimensions (`np,np,nlev,qsize,2,nelemd`). `nets` and `nete` are per-thread indices so that each thread operates independently on a different column of elements, and this is run in an OpenMP parallel region.

kernel, implying that it was not doing this by default. The fourth transformation did not appreciably affect Cray OpenACC kernel performance. Finally, the OpenMP decomposition was removed, and the master thread launched the OpenACC kernels.

There are four separate codes tested in this study. They are labeled "CPU", "CUDA", "OACC", and "OACC2". The first three correspond to Figs. 1–3. The OACC2 code is the same as OACC except with two changes. First, the derived types are all replaced with

```
real(kind=real_kind), shared :: gv_s       (np*np+1,numk_eul,2)
real(kind=real_kind), shared :: deriv_dvv_s(np*np+1           )

ks = int(ceiling(dble(nlev)/numk_eul))
i  = modulo( threadidx%x-1    ,np)+1
j  = modulo((threadidx%x-1)/np,np)+1
kk = (threadidx%x-1)/(np*np)+1
k  = modulo(blockidx%x-1,ks)*numk_eul + kk
q  = modulo((blockidx%x-1)/ks,qsize_d)+1
ie =       ((blockidx%x-1)/ks)/qsize_d+1
ij = (j-1)*np+i

if ( k > nlev .or. q > qsize_d .or. ie > nete ) return

if (kk == 1) deriv_dvv_s(ij) = deriv_dvv(i,j)
qtmp   = Qdp  (i,j,k,q,n0_qdp,ie)
vs1tmp = vstar(i,j,k,1,ie) * metdet(i,j,ie) * qtmp
vs2tmp = vstar(i,j,k,2,ie) * metdet(i,j,ie) * qtmp
gv_s(ij,kk,1) = dinv(i,j,1,1,ie) * vs1tmp + dinv(i,j,1,2,ie) * vs2tmp
gv_s(ij,kk,2) = dinv(i,j,2,1,ie) * vs1tmp + dinv(i,j,2,2,ie) * vs2tmp
divtemp = 0.0d0
vvtemp  = 0.0d0
call syncthreads()
do s = 1 , np
  divtemp = divtemp + deriv_dvv_s((i-1)*np+s) * gv_s((j-1)*np+s,kk,1)
  vvtemp  = vvtemp  + deriv_dvv_s((j-1)*np+s) * gv_s((s-1)*np+i,kk,2)
enddo
Qdp(i,j,k,q,np1_qdp,ie) = spheremp(i,j,ie) * ( qtmp - dt * ( divtemp + vvtemp ) * &
                          ( rmetdet(i,j,ie) * rrearth_d ) )
```

Fig. 2. CUDA kernel code. $numk\_eul = 6$. It is launched with $np \times np \times numk\_eul$ threads per block and $\lceil nlev/numk\_eul \rceil * qsize * nelemd$ blocks.

```
!$acc  parallel loop gang collapse(2) private(gv,dudx00,dvdy00,dp_star,&
!$acc& vvtemp,qtmp,vs1tmp,vs2tmp) vector_length(128) async(1)
do ie = 1 , nelemd
  do q = 1 , qsize
    !$acc loop vector collapse(3)
    do k = 1 , nlev
      do j = 1 , np
        do i = 1 , np
          qtmp = elem(ie)%state%qdp(i,j,k,q,n0_qdp)
          vs1tmp = vstar(i,j,k,1,ie) * elem(ie)%metdet(i,j) * qtmp
          vs2tmp = vstar(i,j,k,2,ie) * elem(ie)%metdet(i,j) * qtmp
          gv(i,j,k,1) = ( dinv(i,j,1,1,ie)*vs1tmp + dinv(i,j,1,2,ie)*vs2tmp )
          gv(i,j,k,2) = ( dinv(i,j,2,1,ie)*vs1tmp + dinv(i,j,2,2,ie)*vs2tmp )
        enddo
      enddo
    enddo
    !$acc loop vector collapse(3)
    do k = 1 , nlev
      do j = 1 , np
        do i = 1 , np
          dudx00 = 0.0d0
          dvdy00 = 0.0d0
          !$acc loop seq
          do l = 1 , np
            dudx00 = dudx00 + deriv%Dvv(l,i)*gv(l,j,k,1)
            dvdy00 = dvdy00 + deriv%Dvv(l,j)*gv(i,l,k,2)
          enddo
          elem(ie)%state%Qdp(i,j,k,q,np1_qdp) =
            elem(ie)%spheremp(i,j) * ( elem(ie)%state%Qdp(i,j,k,q,n0_qdp) - dt * &
                                       (dudx00+dvdy00)*(elem(ie)%rmetdet(i,j)*rrearth) )
        enddo
      enddo
    enddo
  enddo
enddo
```

Fig. 3. OpenACC kernel code.

single arrays in a similar method as used in the CUDA code. This was done because the PGI compiler gives "invalid address" errors during runtime whenever any derived types were used inside the kernel. Second, the PGI compiler still gave "invalid address" errors even when all derived types were removed. This was eventually tracked to the variable, `gv`. As a gang-private variable, the compiler was handling the variable wrongly. When `gv` was removed from the private variable list by appending `qsize×nelemd` indices to the end of the array, this error went away for the PGI compiler, and it was finally giving the correct answer as well. This had no net effect on the performance because `gv` wasn't being placed into GPU shared memory by any of the compilers anyway, even though it was more than small enough to fit. The Cray compiler worked immediately for all cases without trouble.

The CPU code was compiled by PGI with the flags "`-mp -O3-fastsse -tp=bulldozer`" and by Cray with all defaults and no additional flags. The PGI CUDA FORTRAN code was compiled with "`-ta=nvidia,cc35-Mcuda=5.5,cc35 -mp -O3 -tp=bulldozer`". The Cray OpenACC code was compiled with "`-O 3 -h vector3,scalar3,fp4`", and the PGI OpenACC code was compiled with "`-mp -acc -ta=nvidia,cuda5.5,cc35-O3 -tp=bulldozer`". The PGI version was 14.10.0 for all codes. For the OpenACC codes, the Cray version 8.3.4 compiler was used. For the CPU code, the Cray version 8.2.5 compiler was used. The reason is that 8.3.4 was faster for the OpenACC code, but the 8.3.4 compiler produced an executable that was over 50% slower for the CPU code than version 8.2.5. Thus, for a competitive comparison between CPU and GPU codes, Cray 8.2.5 was the fastest and thus serves as the baseline. All codes gave the same output for `qdp` to a relative absolute difference of 10^{-21} or smaller.

3. Results

There are six different runs in all: (1) CPU code run with PGI, (2) CPU code run with Cray 8.2.5, (3) CUDA code run with PGI, (4) OACC code run with Cray 8.3.4, (5) OACC2 code run with Cray 8.3.4, and (6) OACC2 code run with PGI. Two different methods of timing were used. The first, which is used for all of the runs is the `omp_get_wtime()` call, which is wrapped around 1000 invocations of the kernel. After waiting for completion, a second call is run, and the resulting walltime is divided by 1000 to obtain the average time per kernel call. What this does for GPU kernels is effectively removes kernel invocation overheads in the walltime estimation. The second method of timing is the use of the CUDA profiler, which uses CUDA events for accurate timing. In most cases, the two gave nearly identical numbers when run on the GPU. The timings and other performance related data are given in Table 1. The PGI compiler performed nearly 2× slower than the Cray compiler for the CPU code. The CUDA-PGI kernel gained a 2.75× speed-up over CPU-Cray, the highest among the GPU runs. In terms of kernel time, the OACC2-PGI code performed the best among of all the OpenACC kernels, just slightly faster than OACC2-Cray. However, the walltime reported by `omp_get_wtime()` for OACC2-PGI is significantly worse than the kernel time, indicating there may be extra overheads in the asynchronous kernel calls than exist for the other compilers. It was confirmed in profiling that the kernels ran in succession without intermittent data transfers, so the source of the overhead is unknown.

The reason the OACC2 kernels performed faster than the OACC kernel is because the OACC kernel is performing additional addressing lookups for the derived types, as evidenced by the total DRAM load requests during kernel execution. Removing these additional lookups gives an additional 1.13× speedup, but it comes at a high cost in terms of development. Removing derived types is a very invasive and intensive change for a codebase like CAM-SE. Thus, it is likely not worth the effort. The OpenACC kernels have a grid size that is 5 times less than the CUDA kernel. The CUDA kernel explicitly handles six vertical levels per CUDA block (specified by `numk_eul`) leaving *nlev*/6 =5 chunks of levels to be placed into the CUDA grid. The difference in grid size between OpenACC and

Table 1
Runtimes and some other performance related data for the CPU and GPU kernels.

	CPU-PGI	CPU-Cray	CUDA-PGI	OACC-Cray	OACC2-Cray	OACC2-PGI
omp_get_wtime (μs)	830	446	165	257	230	382
kernel time (μs)	–	–	162	252	224	218
Kernel speedup vs CPU-Cray	–	–	2.75×	1.77×	1.99×	2.05×
wtime speedup vs CPU-Cray	0.537×	–	2.70×	1.74×	1.94×	1.68×
CUDA grid size	–	–	8000	1600	1600	1600
CUDA block size	–	–	96	128	128	128
Registers per thread	–	–	34	49	51	52
Occupancy	–	–	0.750	0.562	0.562	0.562
Total Shared Memory	–	–	1768 B	0 B	0 B	0 B
DRAM Load Requests	–	–	3420	22,230	17,100	17,100

CUDA kernels shows that the OpenACC kernels are all *looping* over *nlev*/8 =3.75 chunks of levels *within* the kernel rather than placing them into the grid. The CUDA kernel also has significantly fewer DRAM load requests than the OpenACC kernels, and this is most likely due to the use of temporary variables and shared memory. None of the OpenACC kernels in any of the tested implementations (including beyond what is shown explicitly in this study) used statically or dynamically allocated shared memory at any point. This is likely the greatest flaw and difficulty with current OpenACC compiler implementations when comparing with CUDA kernels. Still, given that the OpenACC kernels do not use shared memory, the fact that they get within 35% of the performance of our CUDA kernel is very impressive. The register usage is likely one potential explanation for how this was achieved.

4. Conclusions

In this study, a kernel has been ported to GPUs using OpenACC for comparison against a previous CUDA implementation. The PGI and Cray compilers were used for the OpenACC implementation, and the nuances of the implementations and runs have been specified. In all, the CUDA kernel was surprisingly only 1.35× faster than the best OpenACC implementation. However, that implementation involves transforming data in a manner that is difficult to do for the entire codebase and should be avoided if possible. For the OACC kernel, the one using derived types as exist in the current codebase, the CUDA kernel was 1.56× faster than the OpenACC implementation. Though it did give the fastest overall OpenACC kernel, the PGI compiler was unable to manage derived types and experienced additional difficulties, such as additional "illegal address" errors as well as odd overheads in the asynchronous invocation of OpenACC kernels, which significantly degraded performance in terms of raw walltime. For this kernel in particular, the Cray compiler is the only one mature enough to use in practice at this point.

Both compilers have a way to go before we believe they could be considered useful in the real world for significant portions of the CAM-SE codebase, though the Cray compiler is currently closer to a usable state. Two things are currently high priority: (1) the ability to transfer parts of derived types and (2) the ability to get a stream handle from OpenACC for interoperability with CUDA if great optimization is desired. One thing OpenACC simply has not achieved at this point is portability (at least for this code). Therefore, it appears that the implementation of OpenACC has a way to go before the ideals driving the standard are realized. Also, many of the looping structures were done in such a way as to explicitly benefit a GPU. It is not clear that this type of looping would perform well on, for instance, a MIC device. Also, the transformations performed in the OpenACC code would degrade cache efficiency on the CPU as well. Thus, it is highly unlikely that a literal single source code would suffice for performance portability. Still, the level of branching / divergence in the code is greatly aided by OpenACC, and data and loop restructuring to *expose* threading is a single effort that benefits all architectures.

However, the biggest virtue of OpenACC experienced here is the significant ease of porting to a GPU, and this is no small virtue. OpenACC is still a rapidly evolving technology, probably more implementation-wise than standard-wise. But the standard is, in fact, evolving in important ways as well. This study demonstrates the need to continue this evolution and advancement so that more applications can more easily and portably realize the benefits of GPUs and MICs to advance the capabilities of scientific simulation on capability-scale hardware. Also, even with the shared memory deficiencies, OpenACC did get somewhat close to CUDA performance. This is encouraging and indicates that OpenACC will likely be performant enough to use for production applications when the implementations become more mature.

Acknowledgements

This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] J.D. Annan, J.C. Hargreaves, Efficient estimation and ensemble generation in climate modelling, Philos. Trans. R. Soc. A 365 (1857) (2007) 2077–2088.
[2] J. Bachmeister, R. Neale, A. Gettleman, C. Hannay, P.H. Lauritzen, J. Caron, J.E. Truesdale, M. Wehner, Exploratory high-resolution climate simulations using the community atmosphere model (CAM), 2014, pp. 3073–3099.
[3] J. Berckmans, T. Woollings, M.-E. Demory, P.-L. Vidale, M. Roberts, Atmospheric blocking in a high resolution climate model: influences of mean state, orography and eddy forcing, Atmos. Sci. Lett. 14 (2013) 34–40.
[4] I. Carpenter, R. Archibald, K.J. Evans, J. Larkin, P. Micikevicius, M. Norman, J. Rosinski, J. Schwarzmeier, M.A. Taylor, Progress towards accelerating HOMME on hybrid multi-core systems, Int. J. High Perform. Comput. Appl. (2012), http://dx.doi.org/10.1177/1094342012462751
[5] T. Delworth, A. Rosati, W. Anderson, A.J. Adcroft, V. Balaji, R. Benson, K. Dixon, S.M. Griffies, H.-C. Lee, R.C. Pacanowski, G.A. Vecchi, A.T. Wittenberg, F. Zeng, R. Zhang, Simulated climate and climate change in the GFDL2.5 high-resolution coupled climate model, J. Climate 25 (2012) 2755–2781.
[6] J.M. Dennis, J. Edwards, K.J. Evans, O. Guba, P.H. Lauritzen, A.A. Mirin, A. St-Cyr, M.A. Taylor, P.H. Worley, CAM-SE: a scalable spectral element dynamical core for the community atmosphere model, Int. J. High Perform. Comput. Appl. 26 (1) (2012) 74–89.
[7] T. Jung, M.J. Miller, T.N. Palmer, P. Towers, N. Wedi, D. Achuthavarier, J.M. Adams, E.L. Altshuler, B.A. Cash, J.L. Kinter III, L. Marx, C. Stan, K.I. Hodges, High-resolution global climate simulations with the ECMWF model in project athena: experimental design, model climate, and seasonal forecast skill, J. Climate 25 (2012) 3155–3172.
[8] M. Khairoutdinov, D. Randall, C. DeMott, Simulations of the atmospheric general circulation using a cloud-resolving model as a superparameterization of physical processes, J. Atmos. Sci. 62 (2006) 2136–2154.
[9] P.H Lauritzen, C. Jablonowski, M. Taylor, R. Nair, Numerical Techniques for Global Atmospheric Models, Springer, 2011.
[10] J. Larkin, R. Archibald, I. Carpenter, V. Anantharaj, P. Micikevicius, K.M. Evans Norman, Porting the Community Atmosphere Model – Spectral Element Code to Utilize GPU Accelerators, Cray User Group, 2012.

[11] V. Rao, R. Archibald, K.J. Evans, Emulation to simulate low-resolution atmospheric data, Int. J. Comput. Math. 91 (4) (2014) 770–780.
[12] J.H. Richter, A. Solomon, J.T. Bacmeister, Effects of vertical resolution and nonorographic gravity wave drag on the simulated climate in the community atmosphere model, version 5, J. Adv. Model. Earth Syst. 6 (2014) 357–383.
[13] E. Roeckner, R. Brokopf, M. Esch, M. Giorgetta, S. Hagemann, L. Kornblueh, E. Manzini, U. Schlese, U. Schulzweida, Efficient estimation and ensemble generation in climate modelling, J. Climate 19 (2006) 3771–3791.
[14] M.A. Taylor, J. Edwards, S. Thomas, R. Nair, A mass and energy conserving spectral element atmospheric dynamical core on the cubed-sphere grid, J. Phys. Conf. Series 78 (2007) 012074.