

Acceleration of a production Solar MHD code with Fortran standard parallelism: From OpenACC to ‘do concurrent’

Ronald M. Caplan , Miko M. Stulajter , and Jon A. Linker

Predictive Science Inc.

9990 Mesa Rim Road, Suite 170

San Diego, CA 92121

caplanr@predsci.com

Abstract—There is growing interest in using standard language constructs for accelerated computing, avoiding the need for (often vendor-specific) external APIs. These constructs hold the potential to be more portable and much more ‘future-proof’. For Fortran codes, the current focus is on the `do concurrent` (DC) loop. While there have been some successful examples of GPU-acceleration using DC for benchmark and/or small codes, its widespread adoption will require demonstrations of its use in full-size applications. Here, we look at the current capabilities and performance of using DC in a production application called Magnetohydrodynamic Algorithm outside a Sphere (MAS). MAS is a state-of-the-art model for studying coronal and heliospheric dynamics, is over 70,000 lines long, and has previously been ported to GPUs using MPI+OpenACC. We attempt to eliminate as many of its OpenACC directives as possible in favor of DC. We show that using the NVIDIA nvfortran compiler’s Fortran 202X preview implementation, unified managed memory, and modified MPI launch methods, we can achieve GPU acceleration across multiple GPUs without using a single OpenACC directive. However, doing so results in a slowdown between 1.25x and 3x. We discuss what future improvements are needed to avoid this loss, and show how we can still retain close to the original code’s performance while reducing the number of OpenACC directives by a factor of five.

Keywords-accelerated computing; Fortran; OpenMP; OpenACC; do concurrent; standard language parallelism

I. INTRODUCTION

The use of accelerators (such as GPUs) has become ubiquitous in high-performance computing. This is due to their power efficiency and compact performance (i.e. one accelerator can perform as well as multiple CPU compute nodes). However, developing and/or porting codes to run on accelerators continues to be a challenge, especially while maintaining portability. Due to the historical lack of direct support for accelerators in the base standard languages (such as C, C++, and Fortran), a plethora of (often vendor-specific) accelerator APIs, libraries, and language extensions have been created over the years, each with varying degrees of support, portability, and software level (e.g. high-level front ends to low-level direct device programming). These include CUDA, RocM, OpenCL, SYCL, KOKOS, RAJA, DPC++, PROTO, OpenMP, and OpenACC [1]–[4]. For legacy codes, as well as for new codes wanting maximum portability,

longevity, and code readability, directive-based solutions such as OpenMP and OpenACC have often been preferred over other options. This is because the directives appear as special comments in the code, allowing it to continue to compile and run on previous hardware/software setups, as the directives will simply be ignored. Even with the great feature set of directive-based programming models, they can still make source code difficult to read and require too much training for the domain scientists to be able to develop and modify the code without breaking acceleration (often leading to multiple code bases). Also, the specification for the directive models can change frequently, and cross-vendor support can be unpredictable.

Recently, the use of standard language parallelism for accelerated computing has been gaining popularity. This includes C++ parallel algorithms [5], [6], Fortran’s `do concurrent` [7], [8], and drop-in replacements for python’s `numpy`¹. These new language features have the potential to eliminate (or greatly limit) the need for external acceleration APIs, and could be more portable and ‘future-proof’. Future portability is assured since any compiler that supports the base language version, must compile and run the code correctly (although not necessarily efficiently).

Here, we focus exclusively on Fortran and its `do concurrent` (DC) construct for GPU acceleration. DC is an alternative construct to the standard `do` loop which indicates that the loop has no data dependencies and can be computed out-of-order². This can be used to hint to a compiler that the loop is likely (but may not be) parallelizable. How this parallelism is mapped at a low level is left to the compiler to decide, with many using pre-existing directive-based mappings (treating DC as if it were a collapsed OpenACC/MP parallel loop).

While there have been some promising results in using DC for accelerated computing (see the next section), how well they will extend to much larger codes is an open question. To help answer this, in this paper we describe porting our large in-production solar MHD code from OpenACC to DC.

¹<https://developer.nvidia.com/cunumeric>

²<https://flang.llvm.org/docs/DoConcurrent.html>

The paper is organized as follows: Sec. II describes related work on using Fortran standard parallelism for accelerated computing. We then introduce the MAS solar MHD code in Sec. III. In Sec. IV we describe how we successively reduce the number of OpenACC directives using DC and the resulting six code versions used for testing. The test problem, computational environment, and performance results are described in Sec. V. We summarize the results and use them to assess the current status and future potential of Fortran standard parallelism for large HPC codes in Sec. VI.

II. RELATED WORK

The use of DC for accelerated computing is a very recent capability. The first compiler to support it was NVIDIA's HPC SDK in November of 2020³, and the most recent is the Intel IFX compiler in 2022.

Due to the novelty of DC support for GPUs, there are only a few examples of its use in the literature (such as a spherical surface diffusion tool called DIFFUSE [9], and an implementation of the BabelStream benchmark [10]). There are additional works in progress, including a chemistry mini-app called CCSD(T)⁴, a CFD weather mini-app called MiniWeather⁵, a hydrodynamics mini-app called CloverLeaf⁶, and a conjugate gradient solver used in Solar physics called POT3D⁷.

While these implementations are promising (and many yielded on-par performance with other acceleration methods), they are all small-to-medium codes/minis-apps/benchmarks. Here, we explore applying the lessons learned from these first studies to a large, in-production application called MAS.

III. THE MAS SOLAR MHD MODEL

The Magnetohydrodynamic Algorithm outside a Sphere (MAS) code is an in-production MHD model with over 20 years of ongoing development used extensively in Solar physics research [11]–[17]. The code is included in the Corona-Heliosphere (CORHEL) software suite [18], [19] hosted at NASA's Community Coordinated Modeling Center (CCMC)⁸ allowing users to generate quasi-steady-state MHD solutions of the corona and heliosphere, as well as simulate solar storms in the form of coronal mass ejections propagating from the Sun to Earth [20], [21]. MAS is written in Fortran ($\approx 70,000$ lines) and parallelized with MPI+OpenACC [22]. It can run simulations containing over three hundred million grid cells [23], [24] and exhibits performance scaling to thousands of CPU cores or dozens of

³<https://developer.nvidia.com/blog/accelerating-fortran-do-concurrent-with-gpus-and-the-nvidia-hpc-sdk/>

⁴<https://www.youtube.com/watch?v=DrvI2gw3tnI>

⁵<https://github.com/mrnorman/miniWeather/tree/main/fortran>

⁶https://github.com/UoB-HPC/cloverleaf_doconcurrent

⁷<https://www.nvidia.com/en-us/on-demand/session/gtcspring22-s41318>

⁸<https://ccmc.gsfc.nasa.gov>

GPUs [22], [25]. The MAS code uses a logically rectangular non-uniform staggered spherical grid and finite-difference discretizations with a combination of explicit and implicit time-stepping methods⁹. The code is highly memory-bound, with its performance typically proportional to the hardware's memory bandwidth.

IV. FROM OPENACC TO DO CONCURRENT

Here we describe our implementations of replacing OpenACC with DC. Our strategy results in six separate code versions, starting with the current OpenACC MAS implementation. These versions arise due to several considerations including (not) avoiding code refactoring, (in)adherence to the current Fortran specification, (not) using experimental compiler features, and trade-offs between reducing OpenACC directives and performance. Among the versions, we were able to achieve a working code with *zero* OpenACC directives (Code 5).

Support for using DC for accelerated computing is in its infancy. Several vendors have announced future plans to support it, but only NVIDIA and Intel have working implementations, with NVIDIA's HPC SDK's nvfortran being the more mature of the two. As our current GPU implementation of MAS uses OpenACC and is tested with NVIDIA GPUs, we exclusively use the nvfortran compiler and NVIDIA GPUs in this work.

The procedure for porting the current OpenACC implementation of MAS (Code 1) to DC is done in several steps. We first try to avoid base code changes and adhere to the current Fortran 2018 specification, incrementally swapping out OpenACC directives with DC, until we arrive at two versions, one with optimized manual data management (Code 2), and one with automatic data management (Code 3). We then use the compiler's Fortran 202X preview features to further reduce the number of OpenACC directives and only use automatic data management (Code 4). By using special compiler flags, alternate code launch scripts, and minimal base code changes, we are able to make a working version that has *zero* OpenACC directives (Code 5). To improve performance, we also create a version of Code 5 with the OpenACC manual data management directives added back in (Code 6).

In Table I we summarize all six code versions that will be discussed in this section.

A. Code 1 [A]: Current OpenACC Implementation

We start with the current GPU production branch of MAS. This is an extension of the code developed in Ref. [22], with added OpenACC directives to accelerate the full set of physics models used in the CORHEL software suite. It is fully portable with CPU compilers as it does not use any OpenACC API calls. It has 73,842 lines of code

⁹<https://www.predsci.com/mas>

Table I
SUMMARY OF ALL MAS CODE VERSIONS DEVELOPED AND TESTED.

Code Version	Code description and nvfortran GPU compiler flags	Total Lines	\$acc Lines
0: CPU	Original CPU-only version	69874	∅
1: A	Original OpenACC implementation -acc=gpu -gpu=cc80	73865	1458
2: AD	OpenACC for DC-incompatible loops and data management, DC for remaining loops -acc=gpu -stdpar=gpu -gpu=cc80, nomanaged	71661	540
3: ADU	OpenACC for DC-incompatible loops, DC for remaining loops, Unified memory -acc=gpu -stdpar=gpu -gpu=cc80, managed	71269	162
4: AD2XU	OpenACC for functionality, DC2X for remaining loops, Unified memory -acc=gpu -stdpar=gpu -gpu=cc80, managed	70868	55
5: D2XU	DC2X for all loops, some code modifications, Unified memory -stdpar=gpu -gpu=cc80 -Minline=reshape, name:s2c, boost, interp, c2s, sv2cv	68994	∅
6: D2XAd	DC2X for all loops, some code modifications, OpenACC for data management -acc=gpu -stdpar=gpu -gpu=cc80, nomanaged -Minline=reshape, name:s2c, boost, interp, c2s, sv2cv	71623	277

including 1,458 !\$acc directive comments. The distribution of OpenACC directive types is summarized in Table. II.

Table II
OPENACC DIRECTIVES IN ORIGINAL GPU BRANCH OF MAS (CODE 1).

OpenACC directive type	# of lines
parallel, loop	997
data management:	
enter, exit, update, host_data, declare	320
atomic	34
routine	12
kernels	6
wait	6
set_device_num	1
continuation lines (!\$acc&) (spread across all directive types)	82
Total	1458

B. Code 2 [AD]: OpenACC for DC-incompatible loops and data management, DC for remaining loops)

In this first pass at replacing OpenACC with DC, we strive to maintain as much performance and portability as possible, with as few code changes as possible (other than converting do loops into DC).

A typical nested do loop with OpenACC directives as used in the MAS is shown in Listing 1. The same loop using DC is shown in Listing 2. Besides eliminating OpenACC directives, the DC loop is also much more compact, reducing the number of lines of code overall.

```
!$acc parallel default(present)
!$acc loop collapse(3)
do k=1,n3
    do j=1,n2
        do i=1,n1
            Computation using array(i,j,k)
        enddo
    enddo
enddo
 !$acc end parallel
```

Listing 1. Standard do loop with OpenACC in the MAS code

```
do concurrent (k=1,n3, j=1:n2, i=1:n1)
    Computation using array(i,j,k)
enddo
```

Listing 2. Equivalent loop to Listing 1 using DC

To ensure that the modified code will still work for CPU-only runs with any compiler vendor/version that currently works with the code, we adhere to the Fortran 2018 standard. This excludes the use of the upcoming Fortran 202X reduce clause on DC loops, leaving all reduction loops to use OpenACC. Also, since some major compilers do not yet support DC affinity clauses such as private and shared (and some are good at detecting proper affinity automatically), we do not make use of these clauses. Array reductions in the current MAS code use OpenACC atomic directives to allow for full parallelization as shown in Listing 3.

```
!$acc parallel default(present)
!$acc loop collapse(2)
do j=1,n2
    do i=1,n1
        !$acc atomic update
            sum0(i)=sum0(i)+array(i,j)*...
    enddo
enddo
 !$acc end parallel
```

Listing 3. Example of array reduction loop in MAS

In order to use DC for the loop and remain within specification, we would need to rewrite the loops (as an outer parallel DC loop, with an inner sequential loop). Since at this point we want to avoid as many code changes as possible, and such a change may affect performance, we also leave the array reduction loops as OpenACC loops.

Some of our parallel loops contain calls to functions/routines. In OpenACC, these must be declared at a specified

level of parallelism using the `routine` directive. The Fortran specification requires that function/routine called in a DC loop must be ‘pure’. Those within our loops are pure, but the current NVIDIA compiler does not yet support them in DC GPU offload loops without leaving in the OpenACC routine directives.

Another issue we faced is the use of OpenACC’s `kernels` construct. While `kernels` can be used on regular `do` loops, it can also be used to contain Fortran array-syntax operations and Fortran intrinsics such as `MINVAL`. In these cases, in order to use DC, we would have to expand the operations into explicit loops. In our updated port of MAS to use OpenACC, we have already done this in most cases. In this DC version, we want to avoid additional code changes, so for now we leave in the remaining few `kernels` regions.

There are two features of OpenACC that are not present in DC which could lead to possible performance losses. The first is ‘kernel fusion’. In OpenACC, one can have multiple data independent loops in a `parallel` region, which the compiler can then compile into a single GPU compute kernel. When converting these loops into DC loops, the compiler has to create separate GPU kernels for each loop. Since kernels have launch overheads, this can lower performance (especially for frequently called small kernels). The second feature is asynchronous kernel launches. In OpenACC, one can use the `async` clause to launch a kernel but allow the code to continue (where it can compute things on the CPU, and/or launch other asynchronous kernels). DC currently has no way to tell the compiler to use an asynchronous kernel launch. Therefore, if a code relies on heavy use of `async`, using DC may reduce performance. In the case of the MAS code, we do not make heavy use of `async`, so we do not expect this to be too much of a problem.

From our previous experience in porting OpenACC to DC [7], we learned that using manual data management yields better performance than relying on unified managed memory (UM) capabilities. Unified managed memory is an NVIDIA feature that will automatically page data to and from the GPU and CPU for GPU-accelerated codes¹⁰. This eases the burden of the programmer by not requiring manual management of the data movement. However, using UM has some drawbacks, such as possible performance degradation and using too much GPU memory. While the performance drop from using UM can be quite small, when used with CUDA-aware MPI calls (as done in MAS), they can be higher. Also, UM is not part of either the Fortran or OpenACC specifications. For these reasons, we do not use UM in this version (AD), leaving the OpenACC data management directives intact.

In the end, for this version (AD) of the code, we have reduced the number of OpenACC directives from 1458 to

540, an almost three-fold reduction.

C. Code 3 [ADU]: OpenACC for DC-incompatible loops, DC for remaining loops, Unified memory

For this version (ADU) of the code, we substantially reduce the number of OpenACC directives by using unified managed memory. As discussed above, UM is not part of the Fortran or OpenACC specification, and without manual data movement, each OpenACC and DC kernel would default to GPU-CPU data migration, causing a huge performance drop. Due to this, multiple vendors have created a unified memory management system (UM for NVIDIA, Unified Shared Memory for Intel OneAPI¹¹, and Smart Access Memory for AMD¹²). Since unified memory management systems are ubiquitous, we feel relying on UM is an acceptable decrease in portability for this version (ADU).

In removing the OpenACC data directives, we found that we could not remove all of them. Specifically, a `declare` (and a subsequent `update`) directive was still needed for a data element that is used in a function called inside a GPU kernel region. Additionally, we had to leave in some `enter` and `exit` data movement for derived types. This was because we are still using OpenACC for reduction loops, which utilize the `default(present)` clause. This tells the compiler that all data is already on the device, and is added to avoid programming performance errors since it will fail for any data not on the device. However, when using UM, even though the arrays within the derived type are paged to the GPU, the derived type structure itself is not, as it is static data (not an allocatable or automatic array which is required for UM paging). Therefore, since we wish to retain the use of `default(present)`, we have to manually place the structures on the GPU.

This version (ADU) of the code further reduced the number of OpenACC directives by over a factor of three from 540 to 162. Since the only change in this code was the removal of data directives, its performance is equivalent to running Code 2 (AD) with UM enabled (in which case all data directives are ignored). Thus, this version (ADU) can be used as a test of the effect UM alone has on performance.

D. Code 4 [AD2XU]: OpenACC for functionality, DC2X for all loops, Unified memory

To see how much further we can reduce the number of OpenACC directives, in this version (AD2XU) of the code we allow ourselves to take advantage of NVIDIA’s Fortran 202X preview implementation of the `reduce` clause on DC loops. Using `reduce` breaks the portability of this version (AD2XU), making it only currently work with the `nvfortran` compiler (even on the CPU). However, this clause is expected to become part of the Fortran standard

¹¹<https://www.intel.com/content/www/us/en/developer/articles/code-sample/dpcpp-usm-code-sample.html>

¹²<https://www.amd.com/en/technologies/smart-access-memory>

¹⁰<https://www.pgroup.com/blogs/posts/openacc-unified-memory.htm>

within a year, so its use will eventually be portable (as soon as all major compilers recognize it).

Using `reduce` allows us to convert all scalar OpenACC reduction loops into DC, but for array reductions, direct use of `reduce` is not yet supported in `nvfortran`. However, due to the underlying compiler mechanisms, we found that we can use OpenACC atomic directives within DC loops, in the same manner we have been using them in the OpenACC loops. Therefore, we have also converted all array reduction OpenACC loops into DC (while retaining the `atomic` directives within them) as shown in Listing 4.

```
do concurrent (j=1:n2, i=1:n1)
!$acc atomic update
  sum0(i)=sum0(i)+array(i,j)*...
enddo
```

Listing 4. DC version of array reduction from Listing 3

We also were able to convert the few non-reduction loops that used `atomic` directives into DC. After this change, all loops using the derived types are now DC loops, so some of the few remaining data directives were also able to be removed.

This version (AD2XU) of the code further reduced the number of OpenACC directives by more than another factor of three from 162 to 55. The remaining OpenACC directives consist of `atomic`, `declare`, `update`, `set device_num`, `routine`, and `kernels`.

E. Code 5 [D2XU]: DC2X for all loops, some code modifications, Unified memory

Here, we allow ourselves to make use of `nvfortran`-specific compiler flags and some modifications to the base code. This is an attempt to see what is needed to currently achieve an ‘ideal’ code (in terms of eliminating OpenACC directives).

As a first step, we eliminated the few remaining kernels directives by expanding the Fortran intrinsic functions being used into explicit DC reduction loops. We note that for codes that have a lot of `kernels` use, this step could be quite involved.

Next, we re-wrote all array reduction loops, by flipping the loop order and using a standard DC on the outer loop, with a DC reduction on the inner loop as shown in Listing 5.

```
do concurrent (i=1:n1)
  tmp=0.
  do concurrent (j=1:n2) reduce (+:tmp)
    tmp=tmp+array(i,j)*...
  enddo
  sum0(i)=tmp
enddo
```

Listing 5. Modified array reduction from Listing 4, allowing removal of the `atomic` directive

This eliminated the need for the OpenACC `atomic` directives in these loops. We could have also replaced the inner loop with a serial `do` loop or Fortran intrinsic (e.g. `SUM`). Indeed, the `nvfortran` compiler output shows that it serializes the inner DC reduction loop (as it is probably faster to do that than launch a multitude of tiny kernels). Small code modifications were also used in a couple of places to allow the removal of the last few `atomic` directives.

A remaining set of OpenACC directives are the `routine` directives used to declare functions and routines that are called within a GPU kernel loop. Since all such functions in MAS are ‘pure’, they should be supported within DC loops, and further development of the `nvfortran` compiler should remove the need for the `routine` directives. In the meantime, we use a compiler flag `-Minline`, name to specify that those routines need to be in-lined. For one of the routines, we also needed to specify the `reshape inline` option. The compiler refused to inline one of the routines, forcing us to manually inline it. Luckily, it was only one routine and not used very many times, but in other codes, this step could be prohibitive. This step also allowed us to remove the `declare` directive for the data elements used within the in-lined functions.

The last OpenACC directive left is the `set device_num` used to select the GPU device based on the local MPI rank in multi-GPU runs. In order to remove this directive, we modify how we launch the code. Instead of launching the code directly (e.g. `mpirun/mas ...`), we launch a bash script (shown in Listing 6) that uses an MPI run time environment variable to set an NVIDIA environment variable, allowing the MPI process to only see the correct GPU device. The script is invoked by running `mpirun/launch.sh ./mas ...`

```
#!/bin/bash
# Assume 1 GPU per MPI local rank
# Set device for this MPI rank:
export CUDA_VISIBLE_DEVICES="
  $OMPI_COMM_WORLD_LOCAL_RANK"
# Execute code:
exec $*
```

Listing 6. Bash script (`launch.sh`) used to launch Codes 5 (D2XU) and 6 (D2XAd).

While this example is specific to the OpenMPI library (which is bundled with the NV HPC SDK), similar environment variables exist in other MPI libraries.

With the previous step completed, we have achieved our goal of obtaining a version of the code with *zero* OpenACC directives that can be run on multiple GPUs. It also allows us to reduce the size of the code further by removing a series of duplicate routines that were needed in the OpenACC implementation. A number of routines in MAS are called both in the setup phase and the computational portion of the code. Since the setup phase contains a very large amount of

code but accounts for only a negligible amount of run time, we did not want to port all of the setup code to GPUs. This caused a problem since the ported routines assume the data is on the GPU. Now that we are using unified managed memory, we can remove the CPU-only versions of these routines, as the GPU-CPU paging overhead during the small setup phase should not significantly affect the overall performance of the code.

F. Code 6 [D2XAd]: DC2X for all loops, some code modifications, OpenACC for data management

As we will see in the next section, the use of UM has a very bad effect on performance for the MAS code. Therefore, we add one more version of the code to our list. In this version (D2XAd), our goal is similar to Code 5 (D2XU) in trying to get the absolute lowest number of OpenACC directives possible (using experimental features, code modifications, etc.), but here we also require that the performance is on par with Code 1 (A) and Code 2 (AD). To do this, we start with Code 5 (D2XU) and put back in all the OpenACC manual data movement directives to allow us to run without using UM (this also required us to put back all the duplicate CPU-only routines mentioned in the previous section). We then modified the code to use wrapper routines for creating and initializing arrays on the GPU, reducing the number of required data movement directives. The resulting code has 277 OpenACC directives, which is over 5 times fewer than our original Code 1 (A), and almost 50% less than Code 2 (AD), while retaining similar performance (see next section).

V. PERFORMANCE TESTS

A. Test Case

To test the performance of our implementations, we use a production quasi-steady-state coronal background simulation from Ref. [26] that uses a full thermodynamic MHD physics model. The problem is set to a resolution of 36 million cells and run for the first 24 minutes of its 48 hour physical simulation time. A visualization of the resulting solution is shown in Fig. 1. The resolution is chosen to represent a medium-sized case that can also fit into the memory of a single NVIDIA A100 (40GB) GPU. For all test runs, the solutions were validated against that of the original code to within solver tolerances.

B. Hardware and Software Environment

We test the code versions using a single 8xGPU node of NCSA’s Delta¹³ supercomputer. The node is dual-socket, with two AMD EPYC Milan 7763 CPUs and a total of eight NVlink-connected NVIDIA A100 (40GB) GPUs, each of which has a peak theoretical memory bandwidth of 1,555 GB/s. We compile the codes with the NVIDIA HPC SDK’s

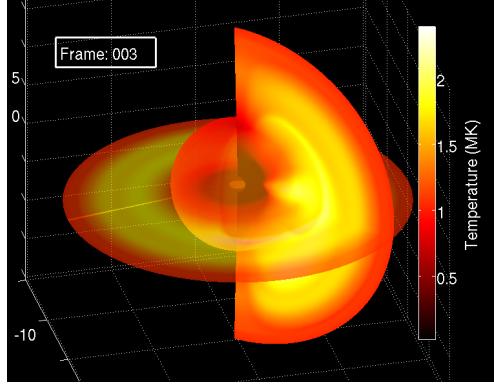


Figure 1. Visualization of the MAS solution for the test case. Cuts of temperature from the last time step are shown and plotted on a stretched grid.

nvfortran compiler version 22.11 and the OpenMPI library version 3.1.3 (we did not use OpenMPIv4 with UCX as its performance and compatibility was not on par with OpenMPIv3). For our baseline CPU tests, we use the dual-socket AMD EPYC 7742 CPU nodes on SDSC’s Expanse¹⁴ supercomputer, each having a maximum theoretical memory bandwidth of 381.4 GiB/s (409.5 GB/s). We use GCC’s gfortran compiler version 10.2.0 with the OpenMPI MPI library version 4.0.4.

C. Performance Results

Here we test the performance of each of the six code versions described in Sec. IV. First, we run Code 1 (A) and Code 2 (AD) on the dual-socket AMD EPYC CPU nodes on Expanse. Besides giving a baseline performance, this also checks that using DC does not negatively affect portability and performance (at least for the specification-compliant version). The results are shown in Table. III and show that the DC version of the code runs equivalently to the original code on the CPUs.

Table III
WALL CLOCK TIME (IN MINUTES) FOR THE TEST PROBLEM RUN ON DUAL-SOCKET AMD EPYC 7742 CPU NODES.

# Nodes	Code 1 (A)	Code 2 (AD)
1	725.54	725.53
8	79.58	79.64

The timings for all six versions of the code run from one to eight NVIDIA A100 (40GB) GPUs on a Delta GPU node are shown in Fig. 2. We see that Codes 1 (A), 2 (AD), and 6 (D2XAd) exhibit ‘super’ scaling at first, and then the scaling dips below the ideal rate. However, due to the initial super scaling, all these codes exhibit better than or close to ideal scaling at 8 GPUs. We also see that Codes 2 (AD) and 6 (D2XAd), which use DC, are both

¹³<https://delta.ncsa.illinois.edu>

¹⁴<https://www.sdsc.edu/services/hpc/expanse>

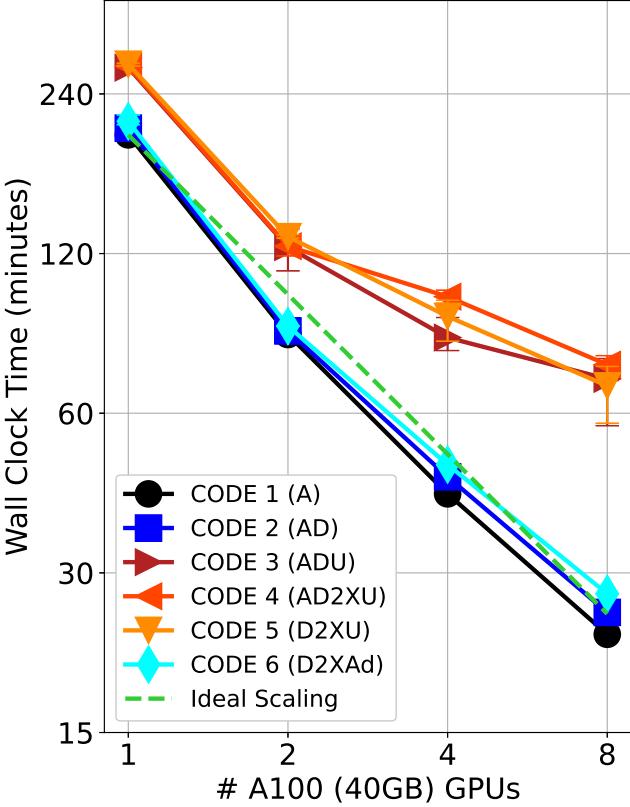


Figure 2. Wall clock time for the test problem run on an 8xA100 (40GB) GPU node. For each result, the average of three runs is shown, with the minimum and maximum time runs shown as an error bar.

somewhat slower than Code 1 (A). Possible reasons for this are kernel fission, loss of asynchronous kernels, and different compiler offload parameters between the OpenACC and DC kernels. Code 6 (D2XAd) is also seen to be a bit slower than Code 2 (AD), which is likely due to additional array initialization kernels in the wrapper routines in places where the original code did not initialize the arrays to zero. Codes 3 (ADU), 4 (AD2XU), and 5 (D2XU) have greatly reduced performance and scaling. In Fig. 3, we show the results for 1 and 8 GPUs, highlighting the MPI overhead time. The MPI time is greatly increased in the codes that use UM, and the non-MPI time is increased as well (but to a much smaller degree). All the codes that exhibit worse performance have similar timings, and all use UM. Since Code 3 (ADU) is equivalent to running Code 2 (AD) with UM enabled, these results indicate that the UM is the cause of the performance drop, not DC. We confirmed this by running Code 1 (A) and Code 2 (AD) with UM and got similar timings to Code 3 (ADU). To explore why UM is causing such a large drop in performance, we ran the NVIDIA NSIGHT Systems profiler for the 8-GPU run of Code 1 (A) with both manual memory management and UM. The results are shown in Fig. 4. We

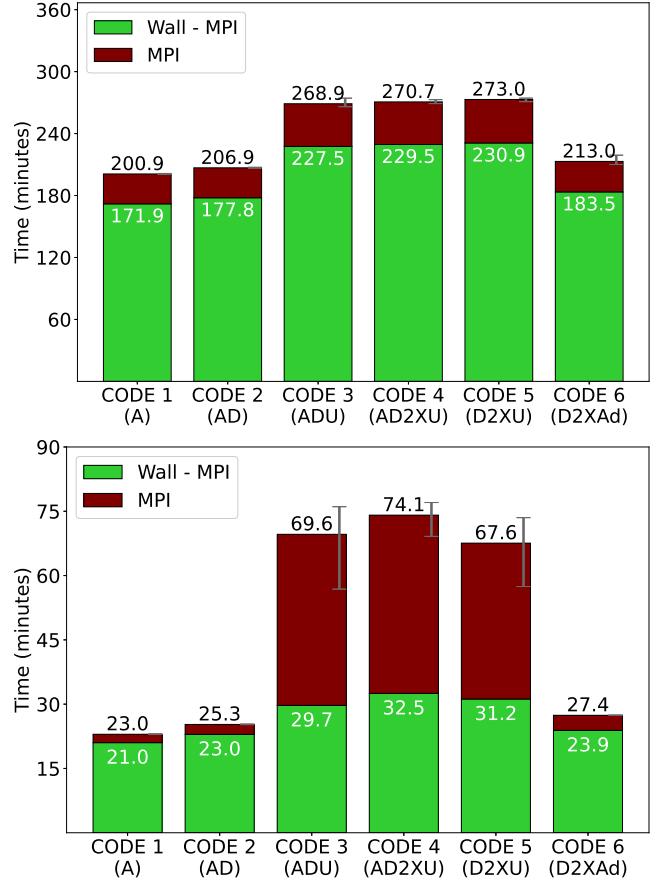


Figure 3. Run times for the test problem on 1 (top) and 8 (bottom) A100 (40GB) GPUs. For each result, the average of three runs is shown, with the minimum and maximum wall clock times shown as an error bar. The MPI time (including all MPI calls, buffer initialization/loading/unloading, and MPI waiting caused by load imbalance) is shown in maroon, while the remainder of the wall clock time is shown in green.

see that the manually managed memory allows for GPU peer-to-peer data transfers in the MPI halo exchanges, while the UM performs multiple CPU-GPU transfers, leading to a slowdown. The UM run also had more overhead as seen in the larger gaps between kernel launches. Overall, the manual memory management run completes almost three full iterations in the same time it takes the UM run to complete one. The developers at NVIDIA have recognized this problem, and are working on a way to resolve it¹⁵.

VI. SUMMARY AND OUTLOOK

Out of the six code versions we tested, we highlight three. First, Code 1 (A, our original OpenACC code) is the best performing version. This is due to manual memory management, and OpenACC's ability to have asynchronous kernel launches, and kernel fusion. The second code we

¹⁵Jeff Larkin, NVIDIA Corp. personal correspondence

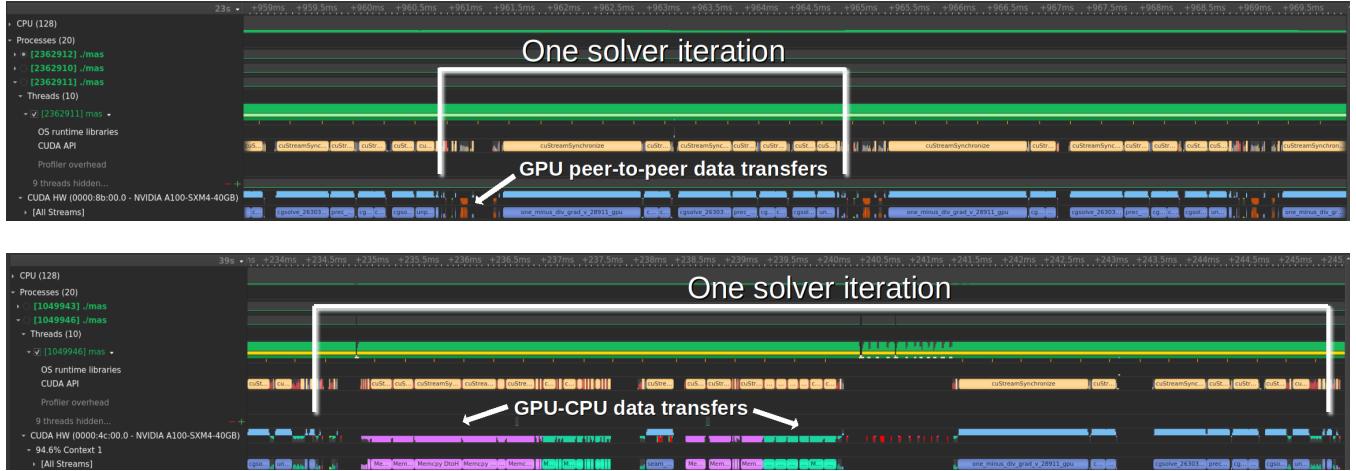


Figure 4. NVIDIA NSIGHT Systems time profile of viscosity solver iterations in MAS using OpenACC manual memory management (top) and unified managed memory (bottom) for Code 1 (A) on 8 A100 GPUs. We see that the manually managed memory results in GPU peer-to-peer data transfers within the MPI halo exchanges, while the unified memory performs multiple CPU-GPU transfers, leading to a slowdown. This, along with more overhead, makes computing a solver iteration three times slower with unified memory management than with manual memory management.

highlight is Code 5 (D2XU). This code shows the potential of standard languages as it can run on multiple GPUs without using a single OpenACC directive. Although it suffers from performance degradation due to UM, we expect this to be a temporary setback which will be solved with further development of the compilers, MPI libraries, and system integration improvements. Code 5 (D2XU) also required us to inline some functions by hand, use inlining compiler flags, and use an alternative method for launching the code with MPI. Some of these requirements are expected to be avoided with future compiler releases as well. The last code we highlight is Code 2 (AD). It adheres to the current Fortran specification, has performance nearly as good as Code 1 (A), can still compile with all major CPU compilers, and has greatly reduced the number of OpenACC directives from 1458 to 540. It shows that the performance of the DC loops are competitive with OpenACC as long as one uses manual GPU-CPU data movement. Adding in some of the Code 5 (D2XU) and Code 6 (D2XAd) modifications (as well as others) can bring the number of OpenACC directives even lower, and such a modified version has become our new production GPU version.

With further development and cross-vendor support, we hope to eventually have a single code base capable of running on multiple vendors' accelerator hardware without the need for directives at all. This will be extremely valuable for maintaining a high-performance, portable, accelerated code, while also allowing domain scientists to develop it in standard Fortran. Although directives can be more straightforward to use than low-level APIs, they still require domain scientists to learn an unfamiliar syntax (and how to properly utilize it), hindering productivity. With a standard

language approach, they can focus on their implementations, comfortable in the language (with minor adjustments) they have been using for decades.

In the meantime, for our codes, we expect to need some amount of OpenACC support for years to come. We highly encourage vendors to maintain and/or add such support (as it is often used instead of OpenMP for GPU applications), until the standard languages are ready for full cross-vendor, cross-platform use.

ACKNOWLEDGMENTS

This work is supported by NSF awards AGS 2028154 and ICER 1854790, and NASA grants 80NSSC20K1582 and 80NSSC22K0893. It also utilized the Delta system at NCSA and the Expanse system at SDSC through ACCESS allocation TG-MCA03S014. We thank Jeff Larkin at NVIDIA Corp. for his useful insights and the anonymous reviewers for their suggestions.

REFERENCES

- [1] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "Evaluating attainable memory bandwidth of parallel programming models via babelstream," *Int. J. Comput. Sci. Eng.*, vol. 17, pp. 247–262, 2018.
- [2] E. C. Davis, C. R. Olschanowsky, and B. Van Straalen, "A structured grid solver with polyhedral+ dataflow representation," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2021, pp. 127–146.
- [3] S. Chandrasekaran and G. Juckeland, *OpenACC for Programmers: Concepts and Strategies*. Addison-Wesley Professional, 2017.

- [4] R. Van der Pas, E. Stotzer, and C. Terboven, *Using OpenMP: The Next Step: Affinity, Accelerators, Tasking, and SIMD*. MIT press, 2017.
- [5] B. A. L. David Olsen, Graham Lopez. (2021) Accelerating standard C++ with GPUs using stdpar. [Online]. Available: <https://developer.nvidia.com/blog/accelerating-standard-c-with-gpus-using-stdpar>
- [6] J. Latt and C. Coreixas, “C++ parallel algorithms for gpu programming: A case study with the lattice boltzmann method,” 2021. [Online]. Available: <http://rgdoi.net/10.13140/RG.2.2.20168.24328>
- [7] M. Stulajter, R. M. Caplan, J. Larkin, and J. Linker. (2022) Using Fortran standard parallel programming for GPU acceleration. [Online]. Available: <https://developer.nvidia.com/blog/using-fortran-standard-parallel-programming-for-gpu-acceleration/>
- [8] I. Corporation. (2022) The next chapter for the Intel Fortran compiler. [Online]. Available: <https://community.intel.com/t5/Blogs/Tech-Innovation/Tools/The-Next-Chapter-for-the-Intel-Fortran-Compiler/post/1439297>
- [9] M. M. Stulajter, R. M. Caplan, and J. A. Linker, “Can fortran’s ‘do concurrent’ replace directives for accelerated computing?” in *Accelerator Programming Using Directives*. Springer International Publishing, 2022, pp. 3–21. [Online]. Available: https://doi.org/10.1007/978-3-030-97759-7_1
- [10] J. R. Hammond, T. Deakin, J. Cownie, and S. McIntosh-Smith, “Benchmarking fortran DO CONCURRENT on cpus and gpus using babelstream,” in *Proceedings of the 13th IEEE International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. ACM, 2022.
- [11] Z. Mikić, J. A. Linker, D. D. Schnack, R. Lionello, and A. Tarditi, “Magnetohydrodynamic modeling of the global solar corona,” *Phys. Plasmas*, vol. 6, no. 5, p. 2217, 1999.
- [12] J. A. Linker, Z. Mikić, R. Lionello, P. Riley, T. Amari, and D. Odstrcil, “Flux cancellation and coronal mass ejections,” *Physics of Plasmas*, vol. 10, no. 5, pp. 1971–1978, May 2003.
- [13] R. Lionello, J. A. Linker, Z. Mikić, and P. Riley, “The latitudinal excursion of coronal magnetic field lines in response to differential rotation: Mhd simulations,” *The Astrophysical Journal Letters*, vol. 642, no. 1, p. L69, 2006.
- [14] R. Lionello, J. A. Linker, and Z. Mikić, “Multispectral Emission of the Sun During the First Whole Sun Month: Magnetohydrodynamic Simulations,” *The Astrophysical Journal*, vol. 690, no. 1, pp. 902–912, Jan. 2009.
- [15] J. A. Linker, R. Lionello, Z. Mikić, V. S. Titov, and S. K. Antiochos, “The Evolution of Open Magnetic Flux Driven by Photospheric Dynamics,” *The Astrophysical Journal*, vol. 731, no. 2, p. 110, Apr. 2011.
- [16] C. Downs, J. A. Linker, Z. Mikić, P. Riley, C. J. Schrijver, and P. Saint-Hilaire, “Probing the Solar Magnetic Field with a Sun-Grazing Comet,” *Science*, vol. 340, no. 6137, pp. 1196–1199, Jun. 2013.
- [17] Z. Mikić, C. Downs, J. A. Linker, R. M. Caplan, D. H. Mackay, L. A. Upton, P. Riley, R. Lionello, T. Török, V. S. Titov *et al.*, “Predicting the corona for the 21 august 2017 total solar eclipse,” *Nature Astronomy*, p. 1, 2018.
- [18] P. Riley, J. A. Linker, R. Lionello, and Z. Mikić, “Corotating interaction regions during the recent solar minimum: The power and limitations of global MHD modeling,” *Journal of Atmospheric and Solar-Terrestrial Physics*, vol. 83, pp. 1–10, Jul. 2012.
- [19] X. Feng, *Current Status of MHD Simulations for Space Weather*. Singapore: Springer Singapore, 2020, pp. 1–123. [Online]. Available: https://doi.org/10.1007/978-981-13-9081-4_1
- [20] R. Lionello, C. Downs, J. A. Linker, T. Török, P. Riley, and Z. Mikić, “Magnetohydrodynamic Simulations of Interplanetary Coronal Mass Ejections,” *The Astrophysical Journal*, vol. 777, p. 76, Nov. 2013.
- [21] T. Török, C. Downs, J. A. Linker, R. Lionello, V. S. Titov, Z. Mikić, P. Riley, R. M. Caplan, and J. Wijaya, “Sun-to-earth mhd simulation of the 2000 july 14 “bastille day” eruption,” *The Astrophysical Journal*, vol. 856, no. 1, p. 75, 2018.
- [22] R. M. Caplan, J. A. Linker, Z. Mikić, C. Downs, T. Török, and V. S. Titov, “GPU acceleration of an established solar MHD code using OpenACC,” *Journal of Physics: Conference Series*, vol. 1225, p. 012012, may 2019.
- [23] C. Downs, J. A. Linker, V. S. Titov, R. M. Caplan, P. Riley, and R. Lionello, “Linking coronal heating to coronal structure and complexity at global scales,” *The Third Triennial Earth-Sun Summit (TESS)*, vol. 54, no. 7, 2022.
- [24] C. Downs, J. A. Linker, R. M. Caplan, and V. S. Titov, “Linking surface complexity to coronal and solar wind structure,” *In preparation*, 2023.
- [25] R. M. Caplan, Z. Mikić, J. A. Linker, and R. Lionello, “Advancing parabolic operators in thermodynamic MHD models: Explicit super time-stepping versus implicit schemes with krylov solvers,” *Journal of Physics: Conference Series*, vol. 837, p. 012016, may 2017.
- [26] K. K. Reeves, T. Török, Z. Mikić, J. Linker, and N. A. Murphy, “Exploring plasma heating in the current sheet region in a three-dimensional coronal mass ejection simulation,” *The Astrophysical Journal*, vol. 887, no. 1, p. 103, Dec. 2019. [Online]. Available: <https://doi.org/10.3847/1538-4357/ab4ce8>