Enabling Execution of a Legacy CFD Mini Application on Accelerators Using OpenMP

Ioannis Nompelis[1], Gabriele Jost[2], Alice Koniges[3(✉)], Christopher Daley[4], David Eder[5], and Christopher Stone[6]

[1] University of Minnesota, Minneapolis, MN, USA
nompelis@umn.edu
[2] NASA Ames Research Center, Mountain View, CA, USA
Gabriele.Jost@nasa.gov
[3] University of Hawai'i at Mānoa, Honolulu, HI, USA
koniges@hawaii.edu
[4] NERSC/LBL, Berkeley, CA, USA
csdaley@lbl.gov
[5] MHPCC, University of Hawai'i, Kihei, HI, USA
dceder@hawaii.edu
[6] PACE/Georgia Tech, Atlanta, GA, USA
chris.stone@gatech.edu

Abstract. We describe the process and outcome of our efforts to port a legacy Fortran benchmark code to heterogeneous GPU-accelerated computing architectures using OpenMP. The benchmark code is one of the multi-zone NAS Parallel Benchmarks (NPB-MZ) called SP-MZ. This "mini-app" mimics the computation and data movement that is found in popular legacy and modern implicit computational fluid dynamics (CFD) solvers. Our objective was to examine how efficiently legacy Fortran codes can be ported to accelerators by leveraging OpenMP directives. We describe the development and optimization process and demonstrate the performance impact of various code modifications. We show select profiling results from the NVIDIA Visual Profiler (nvpp) to help others diagnose and overcome performance issues in their own applications. We present results for two compute systems endowed with NVIDIA V100 accelerators.

Keywords: Accelerator · Fortran · GPU · OpenMP · Implicit CFD

1 Introduction

The latest computing architectures that are deployed in existing supercomputing installations have very closely followed hardware trends that were propelled by emergent fields of computational science, such as machine learning, data mining and artificial intelligence. These emergent fields use methodologies that can take

P. Sadayappan et al. (Eds.): ISC High Performance 2020, LNCS 12151, pp. 270–287, 2020.
https://doi.org/10.1007/978-3-030-50743-5_14

advantage of large numbers of low-power processors, as they require simple linear algebra operations to be performed on compactly sized chunks of independent data. As a result, high levels of parallelism are possible because concurrency is relatively easy to realize when processing data without the need of frequent exchanges of information. The pipelining that is possible by means of forming and executing small kernels on streams of data can be done very efficiently on large numbers of co-processors, similar to how graphics processing units (GPU) operate to form graphics pipelines. This hardware model was adopted as a means to provide a large number of theoretical floating point operations for the cost incurred; here we note that cost implies powering, packaging, and cooling the processing units.

While these GPU architectures are generally optimal in data-science fields, they are more difficult to exploit for the traditional algorithms of physical problems, which involve the solution to partial-differential equations that have inherent strong coupling of the data structures. However, this shift in hardware is also inevitable due to the exhaustion of raw computational power achievable by increasing processor clock-speeds and physically compacting the processor footprints. As a result, researchers in the physical sciences that require more computational power and performance have to undergo a paradigm shift, where the methods must take advantage of the new architectures. Legacy codes generally have a history of testing, user bases, I/O, and other aspects that make them staples of the HPC landscape. Thus refactoring and porting them to GPU accelerated nodes is critically important. Our work specifically explores a path forward for porting codes written in Fortran that are already OpenMP enabled on the CPU. For reasons of portability, the OpenMP API [3] is an appropriate direction. Alternative APIs such as OpenACC [2] will require similar modifications, with similar syntax, and can benefit from the work presented here.

Other avenues for porting methodologies exist, for example frameworks that act as "middle-writers" for executing code such that porting, data movement and portability are entirely opaque to the programmer. Kokkos [15] and Raja [17] are two such frameworks that have been demonstrated to make simulation software portable. However, this is not a viable option for porting legacy codes, and especially codes written in legacy Fortran.

This manuscript is organized as follows. First, we discuss the OpenMP framework, and in particular the "Target" constructs that are used for programming co-processor accelerators. We then describe our general porting strategy and discuss some related work. Section 2 discusses the parallel benchmark mini-app that we have chosen for this study. Section 3 describes the evaluation systems, compiler directives and run-time execution environment for our experiments. Results are presented alongside the code modifications that were made and are found in Sects. 4 and 5. We provide some concluding remarks in the final section.

1.1 Programming with OpenMP Target Offload

There are various methods of programming co-processor accelerators. Vendor specific libraries, for example, such as NVIDIA^{TM} cuBLAS [4] or cuFFT [1]

usually provide good performance with a small programming effort. However, they limit portability and require that the application kernels match the pattern of the library routines. Low level frameworks such as CUDA™ [5] or OpenCL [16] are suitable for general kernel patterns but require higher programming effort. The use of CUDA also limits portability.

Compiler directive based APIs, such as OpenMP, require only moderate programming effort with acceptable performance for multi-core nodes. OpenMP is a well established programming API for shared memory systems that was first standardized in 1997. It has since been augmented such that it includes directives for support of accelerators. It provides compiler directives, runtime library routines and environment variables. Accelerator programming support has been available since OpenMP 4.0 and the functionality was greatly extended in OpenMP 4.5. Support is provided to

- Identify kernels for offloading to the accelerator device
- Semi-explicitly specify parallelism
- Manage data transfer between the host and accelerator device.

The new OpenMP 4.5 functionality seamlessly integrates into existing OpenMP code and is supported by many compilers such as GNU Fortran (gfortran) [12], Cray™ ftn, and IBM™ xlf.

1.2 Porting Strategy

As alluded to earlier, there are three high-level aspects of porting solvers to heterogeneous accelerated systems that must be evaluated in advance of committing to a given approach:

- performance: the amount of performance that is desirable and is realizable when using accelerators
- portability: what level of abstraction can be maintained to allow for a desired level of portability across systems
- intrusiveness: the amount of alteration to the baseline code and to the data structures that is necessary or is tolerated.

The focus of this work is high performance for the accelerator. We aimed towards portability by employing the OpenMP standard for accessing accelerators. In terms of code alteration, minimal modification to the baseline code was desired and we tried to limit re-factoring of code and changes in data structures. During the development process we documented the code changes and their impact on GPU and CPU performance. In essence, our effort consisted of an assessment of performance gains as a function of the level of alteration made to the mini-app.

1.3 Related Work

Previously, the C implementations of the single zone NAS Parallel Benchmarks FT, LU and BT [10]. Were ported to accelerators using OpenMP 4.5, which

was presented at OpenMPCon 2018 [14]. The focus of the current work differs from the previous work in that we targeted multi-zone codes, and chose SP-MZ implemented in Fortran as an example. Use of OpenACC to parallelize CFD algorithms is described in [22]. Various other work has been done using OpenACC for CFD codes [19]. The focus of the current work differs from previous efforts in that we targeted multi-zone CFD codes implemented in Fortran and we are using OpenMP 4.5 as the API.

The basis of our work was established during a Hackathon event sponsored by the Department of Energy [7].

2 NAS Parallel Benchmark SP-MZ

This effort centered on legacy computational fluid dynamics solvers written in Fortran and their potential for porting to and acceleration with heterogeneous computing systems. In order to avoid the complexity of porting a production solver, we restricted this effort to a mini application that retains the character of the Navier-Stokes equations and employs a common numerical method. While modern CFD solvers employ a mix of structured and unstructured grids, we restricted this work to a structured discretization, which is typical of legacy codes, to avoid the complexity of building kernels out of unstructured data.

We used the multi-zone NAS parallel benchmark suite in this work. The NPB-MZ [23] suite consists of three mini applications. These are multi-zone versions of the well known NAS Parallel Benchmarks BT, SP and LU [10, 21]. SP-MZ supports distributed and shared-memory parallelism with MPI and OpenMP. Zone-level parallelism is exploited using MPI and parallelism within each zone is exploited using OpenMP. Version 3.4 of the SP-MZ mini-app has 3,515 total lines of code (LoC). The mini-app contains only 1809 LoC associated with the implicit integration (i.e. the core arithmetic computation of the CFD solver), which includes MPI function calls and OpenMP directives. The small size of the mini-app allows an end-to-end refactoring and several iterations on offloading strategies. The general execution flow of the original SP-MZ hybrid code is depicted in Fig. 1.

2.1 The Underlying Numerical Method

The mini application is intended to mimic the performance characteristics of CFD applications that use the diagonalized Beam-Warming [11] alternating direction implicit (ADI) algorithm (i.e. the Pulliam-Chaussee algorithm). Major CFD software packages use this algorithm to simulate a wide variety of compressible flows for external aerodynamics applications. SP-MZ uses this implicit integration algorithm and it is designed to model the key performance characteristics of larger CFD applications.

Key features include the formation of the explicit right-hand-side (RHS) vector with finite differences and the factorization of the scalar pentadiagonal (SP) matrices along each grid line in all three directional sweeps. For a mesh

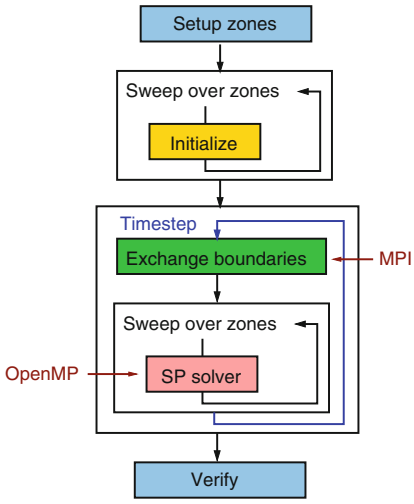

Fig. 1. Execution flow of the SP-MZ Benchmark

with N^3 points, the ADI scheme requires that N^2 SP line matrices (with N rows) be factored and equations solved in each of the sweeps. Fast assembly and factorization of the SP matrices is key to achieving high performance in the mini-app (and similar real CFD applications). In any refactoring effort of similar algorithms, efficient linear solvers on GPUs are needed for scalar and block tri- and penta-diagonal matrices.

Briefly, the SP-MZ benchmark solves a set of discretized nonlinear partial differential equations (PDEs) based on the Navier-Stokes equations. When discretized on a structured grid using 1^{st} or 2^{nd}-order (central) finite-differences for spatial derivatives and an implicit backwards Euler time differencing method, the three-dimensional equations result in a banded, block-matrix with three, non-adjacent sub- and super-diagonal bands. The block-matrix elements are 5×5 submatrices. This block-matrix system is considered too large to solve with either direct or iterative methods. Instead, the discretized PDE system is *approximately factorized* (AF) spatially such that each spatial direction can be solved independently. That is, if the original discretized system is written as:

$$[I - h\{A_x + B_y + C_z\}](U^{n+1} - U^n) = R(U^n) \tag{1}$$

the approximate factorization, following Beam and Warming [11], is:

$$[I - h\ A_x][I - h\ B_y][I - h\ C_z](U^{n+1} - U^n) = R(U^n) \tag{2}$$

Here, A_x, B_y, C_z contain the implicit spatial difference operators, U^{n+1} is the state vector solution at the next (future) time-step, h is the step size, and R contains all forcing terms and explicit terms at the n^{th} time level. Equation 2 can be solved in three sequential steps representing directional solution sweeps:

$$[I - h\ A_x]\delta U^x = R(U^n) \tag{3}$$

$$[I - h\ B_y]\delta U^y = \delta U^x \tag{4}$$

$$[I - h\ C_z]\delta U\ \ = \delta U^y \tag{5}$$

with $\delta U = U^{n+1} - U^n$. In this form, which resembles an *alternating direction implicit* (ADI) algorithm, only block *tridiagonal* matrices must be factored. Note that in the spatial decoupling in each directional sweep, multiple block tridiagonal systems must be solved. That is, if the mesh has $N_x \times N_y \times N_z$ points, then $N_y \times N_z$ block-tridiagonal *line* matrix equations must be solved in the x solution direction sweep.

The approximate factorization form requires less storage and less computational time to solve than the nonfactorized Eq. 1. The computational cost and storage can be reduced further by solving the diagonalized form [10,20] of Eq. 5. The block-matrix elements have all real eigenvalues and a complete set of eigenvectors. As such, the coupled systems are cast into a decoupled, diagonal form with some loss of accuracy. The diagonalization recasts the problem from one of solving block-tridiagonal systems (with five unknowns per block) to one of solving five decoupled scalar-pentadiagonal systems. Furthermore, three of the SP systems have the same matrix and can be solved together. The diagonalization requires some additional vector-matrix operations compared to the coupled form, but uses less storage and requires fewer computations. Further details on the diagonalization can be found in [20].

The multi-zone aspect of the SP-MZ relates to the domain decomposition approach. The global three-dimensional mesh is partitioned into the x and y directions. The 2$^{\text{nd}}$-order spatial scheme requires an overlap of ± 1 *ghost* (or *rind*) points. The implicit AF scheme is applied independently within each mesh zone and each MPI process is assigned one or more zones. That is, within each time-step, each zone is integrated independently.

The major computational costs of the SP-MZ benchmark are the evaluation of the right-hand-side terms $R(U^n)$ in Eq. 5, the assembly of the three SP matrices, and solution of the five decoupled equations in the three spatial directions. The SP systems are solved using a variant of the *Thomas algorithm* (TA) for scalar matrices. The TA is inherently sequential, however, many independent mesh-line matrices can be solved concurrently.

3 Testing Architectures

Two types of systems were used in this work: the IBM-built "Ascent" system, which is similar to the "Summit" supercomputer [9] and the Cray-built "Cori" [6] system.

3.1 The Ascent/Summit Compute Node

The Ascent system [8] is located at the Oak Ridge Leadership Computing Facility (OLCF). Ascent is a system that is identical to the Summit supercomputer in terms of compute node hardware, and it serves as a training system. Each

compute node has two banks of 256 GB DDR4 memory, and each bank is connected via a 170 GB/s bus to an IBM Power9 CPU with 21 physical cores. Each core can support up to 4 hardware threads for a total of 84 threads per CPU. The CPUs are connected to each other with a 64 GB/s bus. Each CPU has access to 3 Volta V100 GPUs, and each of the GPU accelerators accesses 16 GB HBM2 memory via a 900 GB/s bus. The 3 GPU accelerators on each bank are connected to the CPU and to each other via a 50 GB/s NVLink2 bus.

One Fortran compiler available on Ascent and the one used in this work is the Fortran compiler "xlf" that is part of the IBM™XL compiler suite. The message passing interface for Ascent is provided by IBM Spectrum MPI. Interactive job submission and run-time support is provided through the IBM Spectrum Load Sharing Facility (LSF).

For our study we used cuda/10.1 and the IBM xlf v16.1 compiler with the following flags:

```
xlf -qfixed -qpreprocess -O3 -g -q64 -qsmp=omp -qoffload
```

3.2 The Cori Compute Node

The Cori system is located at NERSC. Cori is a Cray XC40 supercomputer that comprises of a mix of Intel Xeon "Haswell" nodes and Intel Xeon Phi "Knights Landing" nodes. A small set of "Skylake" nodes (18 in total) with GPU accelerators are accessible via Cori. This will henceforth be referred to as Cori GPU. There are 2 Skylake CPU sockets on each Cori GPU node, each containing 20 cores, and sharing 384 GB DDR4 memory. Each Cori GPU node contains 8 Volta V100 GPUs, each with 16 GB HBM2 memory, and connected to each other in a "hybrid cube-mesh" topology via a NVLink2 interconnect. The Cori GPU nodes are intended to help users prepare for the GPU-accelerated nodes in the Perlmutter supercomputer to be deployed at NERSC in 2020.

The Fortran compilers available on the Cori GPU nodes include GNU-8.1.1, PGI-19.7, Intel-18.0.1.163, Cray-9.0.0 and LLVM/Flang-7.0. Neither PGI nor Intel compilers currently provide OpenMP GPU offload support. We also found that the OpenMP GPU offload support in the GNU and LLVM/Flang compilers failed even in simple benchmark programs. Therefore, we used the Cray compiler in this study. The Cray compiler is accessed using the Cray™ "ftn" MPI and math library wrapper script. A major limitation of the Cray compiler on Cori GPU is that the Cray MPI stack is not supported and so Cray compiler experiments were limited to single process tests only.

For our study we used Cray-9.0.0 with the following flags:

```
ftn -O3 -h omp -h noacc -haccel=nvidia70
                                    -h cpu=haswell -h fp3
```

The reason we optimized for the Haswell processor architecture is that Cray-9.0.0 does not provide optimizations for the Skylake processor architecture.

4 From OpenMP 3.1 to OpenMP 4.5

In this section we describe our porting strategy and discuss code transformations that enabled performance gains.

4.1 Identifying Kernels and Describing the Parallelism

In our current implementation we focused on exploiting GPU parallelism within the zones. We demonstrate our approach with the example in the code snippets shown in Fig. 2, part of one of the most time-consuming routines in the module that solves the factored system in the x-direction (running index "i"). This segment implements forward and backward substitution, and portions of the code have been removed to ease presentation. The OpenMP directives indicate how the loops are to be parallelized to perform forward and backward substitution along the i-direction of the data structures.

The refactored code in Fig. 2 uses the `TARGET` construct to offload the code region to the accelerator. The `TEAMS DISTRIBUTE` constructs create a league of teams and distributes the loop iterations across teams. The code transformations of our initial implementation are as follows:

- We manually inlined routines called within OpenMP target regions. An example is routine `lhsinit` in the code listing.
- We transposed some of the arrays to allow for stride one memory access. This is essential for good performance on the GPU because it enables coalesced memory accesses. An example is the array `rhst` in the code listing.
- We used the OpenMP `COLLAPSE` clause to collapse as many loops as possible. The code in the listing permits only 2 loop collapses; collapsing 3 loops was possible in some other routines.
- The original code contains two-dimensional arrays which are declared as `THREADPRIVATE`. However, the effect of an access to a `THREADPRIVATE` variable in a `TARGET` region is unspecified according to the OpenMP Standard. We found that OpenMP `PRIVATE` arrays per thread gave poor performance because of the large size of the arrays. Therefore, we made the array shared. This was accomplished by adding two extra dimensions. An example is the array `lhs4` in the code listing.
- To exploit some of the available zone-level parallelism within the code, we enabled asynchronous kernel execution using deferred OpenMP target tasks with dependencies. This is why the directive in the code listing contains the `NOWAIT` and `DEPEND` clauses. The execution flow of SP-MZ with potential kernel overlap is depicted in the pseudo-code in Fig. 3.

In what follows, we will refer to the implementation described above as our *initial port*. We refer to the original code as the *baseline* version.

4.2 Further Optimizing Parallelism and Data Movement

Details on the performance analysis of our initial implementation are provided in the next section. After profiling, we implemented some additional optimizations.

Implementation optimized for CPUs

```
!$OMP PARALLEL DO DEFAULT(SHARED)
!$OMP&  PRIVATE(fac2,m,fac1,i2,i1,ru1,i,j,k)
!$OMP&  SCHEDULE(STATIC) COLLAPSE(2)
 do k = 1, nz-2
    do j = 1, ny-2

       call lhsinit(lhs, lhsp, lhsm, nx-1)
       ....    !--- operations localized at "i"

       do i = 0, nx-3     !--- Thomas alg. forward elim.
          i1 = i + 1
          i2 = i + 2
          fac1       = 1.d0/lhs(3,i)
          lhs(4,i)   = fac1*lhs(4,i)
          lhs(5,i)   = fac1*lhs(5,i)
          do m = 1, 3
             rhs(m,i,j,k) = fac1*rhs(m,i,j,k)
          end do
          lhs(3,i1) = lhs(3,i1) -
                      lhs(2,i1)*lhs(4,i)
          lhs(4,i1) = lhs(4,i1) -
                      lhs(2,i1)*lhs(5,i)
          do m = 1, 3
             rhs(m,i1,j,k) = rhs(m,i1,j,k) -
                             lhs(2,i1)*rhs(m,i,j,k)
          end do
          ....
          lhsm(4,i1) = lhsm(4,i1) -
                       lhsm(2,i1)*lhsm(5,i)
          ....
       end do
       ....
    end do
 end do
```

Implementation optimized for GPUs

```
!$OMP  TARGET TEAMS DISTRIBUTE
!$OMP&  PARALLEL DO SIMD COLLAPSE(2)
!$OMP&  NOWAIT DEPEND( inout: rhs )
!$OMP&    MAP( ALLOC: lhs4, lhsp4, lhsm4, rhst )
!$OMP&    PRIVATE(fac2,m,fac1,i2,i1,ru1,i)
!$OMP&    PRIVATE( cv_im1, cv_ip1 )
!$OMP&    PRIVATE( rhon_ip1, rhon_im1, rhon_i )
 do k = 1, nz-2
    do j = 1, ny-2

       lhs4 (j,1:5,    0,k) = 0.0d0
       lhs4 (j,1:5,nx-1,k) = 0.0d0
       ....    !--- operations localized at "i"

       do i = 0, nx-3     !--- Thomas alg. forward elim.
          i1 = i  + 1
          i2 = i  + 2
          fac1           = 1.d0/lhs4(j,3,i,k)
          lhs4(j,4,i,k) = fac1*lhs4(j,4,i,k)
          lhs4(j,5,i,k) = fac1*lhs4(j,5,i,k)
          do m = 1, 3
             rhst(j,m,i,k) = fac1*rhst(j,m,i,k)
          end do
          lhs4(j,3,i1,k) = lhs4(j,3,i1,k) -
                           lhs4(j,2,i1,k)*lhs4(j,4,i,k)
          ....
       end do
       ....
    end do
 end do
```

Fig. 2. Code fragments showing base language code and OpenMP directives to execute efficiently on CPUs and GPUs

– We eliminated some small host-to-device ("HtoD") data transfers by using the `DECLARE TARGET` construct to declare some of the constants on the device. Furthermore, we declared some subroutine arguments to have the `VALUE` attribute. We found that the xlf compiler did not transfer the associated data as part of the kernel launch if they were passed by reference. However, passing the data by value circumvented the issue.

```
Partition domain into zones
For all zones:
    initialize zone
For all time-steps:
    Communication rind data
          For all zones:
              Pack zone's face points into buffer.
          Call MPI Send/Recv to exchange buffers with neighbors.
          For all zones:
              Unpack buffer and update zone ghost points.
     For all zones:
         Enqueue SP/ADI solver for zone. Async execution on device.
     Wait for all zones to complete async execution.
Verify results
```

Fig. 3. Pseudo-code for asynchronous execution in SP-MZ

- We improved the parallelism in the small loop over the two-dimensional slabs of all zones to accelerate the copy in/out kernels. Instead of a kernel for each rind copy in/out in "exch-bc" for each zone, we created a target region over the zone loop so all the copy in/out operations run within the same kernel. The drawback is that the number of zones shrinks since we end up with just a few or even one gang/team/threadblock.
- Using preallocated `lhs` scratch memory for all zones improved performance. The original implementation used dynamic allocation for each zone inside of the x, y, and z solvers. This required frequent device allocations with global barriers. We pre-allocated a large temporary array for all the `lhs` structures on all zones and passed them to the x, y and z solver routines.
- We merged asynchronous kernels with communication optimizations. This removed some unnecessary HtoD and DtoH transfers. We also combined the `PARALLEL DO` directives with `TARGET TEAMS DISTRIBUTE` since this enables the compiler to generate simpler code where all GPU threads execute the same computation on different data. It allows the compiler to generate code that maps better to GPU hardware [18].

In what follows, we refer to the implementation containing these optimizations as the *optimized version*.

We monitored the incremental performance changes during the development process on the Ascent system, which is shown in Fig. 4. The host configuration used 21 OpenMP threads bound to 1 CPU socket and the device configuration used 1 process bound to 1 CPU socket, which offloaded work to 1 GPU. The intention of this graph is to show the performance in the absence of MPI.

The Roman numerals correspond to the following code changes.

(I) Baseline
(II) Transposed arrays to support coalesced memory operations; replaced dependencies on one-dimensional, thread-private (host) scratch arrays with four-dimensional scratch arrays.
(III) Changed index ordering in x-solve to improve memory coalescing.
(IV) Reduced or eliminated unnecessary host-to-device transfers.
(V) Merged target, teams distribute, and parallel do directives into one combined directive and additionally used loop collapsing.

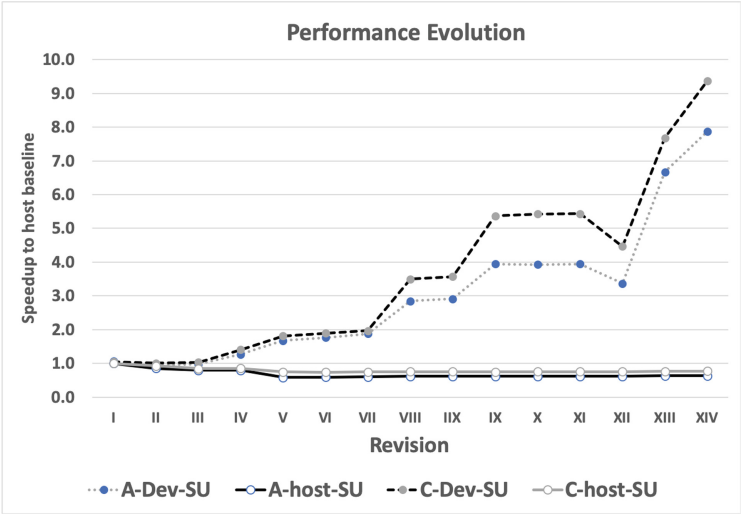

Fig. 4. Incremental performance change during the development process on the Ascent system: Dev-SU and host-SU stands for speed-up on device and host respectively. A and C indicate the Benchmark Class. Note the Dev-SU increases but host-SU decreases during the development process

(VI) Enabled kernel task dependencies to allow asynchronous execution.
(VII) Improved efficiency of ghost/rind data buffer fill kernels.
(VIII) Enabled asynchronous rind fill kernels.
(IX) Added loop scalars to private clauses to avoid unnecessary host-to-device transfers before kernel execution.
(X) Added CUDA-aware MPI option to bypass host transfers during MPI communication.
(XI) Improved parallelism and occupancy of rind fill in/out routines.
(XII) Added `SIMD` to the `PARALLEL DO` directive to improve performance when using the Cray compiler.
(XIII) Change argument passing from by-reference to by-value to allow the IBM OpenMP v4.5 compiler to avoid unnecessary host-to-device transfers.
(XIV) Combined all rind/ghost cell fill in/out routines to execute in one kernel instead of one per zone. That is, changed parallelism from within each zone to all zones at once.

The chart also shows that code changes that improved GPU performance, at times decreased CPU performance. There are several reasons for this. Most important is the fact that there is poorer cache locality. Furthermore, the IV change introduced additional arrays, which hurt host performance. The XII change modified the IBM compiler code generation in a negative way. Figure 4 shows IBM performance, although the XII change describes an optimization for the Cray compiler.

5 Performance Studies

In this section we show timings on our different evaluation systems and discuss performance analysis results. We collected performance results for 3 different benchmark classes.

- Class A: 4 × 4 zones, 128 × 128 × 16 grid points
- Class C: 16 × 16 zones, 480 × 320 × 28 grid points
- Class D: 32 × 32 zones, 1632 × 1216 × 34 grid points.

A characteristic of SP-MZ is that all zones are of equal size. Provided that a high performance MPI library is available and that there are no memory constraints imposed by the problem size, it is best to exploit zone level parallelism via MPI rather than using hybrid MPI+OpenMP. The baseline measurements on Ascent were thus obtained in MPI-only configuration. We could not do this on Cori GPU with the Cray compiler because of the lack of MPI support. Thus we used the nested OpenMP implementation of SP-MZ that is part of the NPB 3.4 distribution for the CPU-only measurements. We ran this version with OpenMP threads on the outer level parallel region to exploit zone parallelism only, making it more of a fair comparison against the Ascent MPI-only experiment on CPUs. We ran the MPI+OpenMP target offload implementation of SP-MZ on Cori GPU by using a single process and a "dummy" MPI library.

The CPU-only configuration on the Ascent system used in Fig. 5 is different from the one used in Fig. 4. The purpose of Fig. 4 is to show the performance in the absence of MPI. In Fig. 5, on the other hand, we want to show the best possible result that can be obtained on the CPU, which is using 42 MPI ranks on zone level. This is the reason why speed-up on CPU versus GPU is not as high in Fig. 5 as in Fig. 4. Using 84 ranks/threads and 168 ranks/threads on the IBM Power9 did not yield a performance gain since the code is memory bandwidth limited.

Figures 5 and 6 show the performance in GFLOP/s that we obtained on Ascent and Cori GPU for different benchmark classes. The Class D problem on a single Ascent GPU failed and thus we only have results for the smaller problem sizes for this case. The results show that the performance is generally highest for Class D, which is the largest problem size. The CPU-only results are approximately a factor of 2 higher on Ascent than Cori-GPU. This is most likely because of NUMA penalties affecting the OpenMP version of SP-MZ. The best single GPU performance we obtained was approximately a factor of 2 slower than the best CPU-only performance. The benefit of GPUs is only really seen when running the Class D problem on all 6 GPUs of an Ascent compute node. Here, performance is approximately 4x higher than the corresponding MPI-only configuration on CPUs.

We used the nvprof profiler to collect performance statistics. We noticed that in the optimized code, the CUDA API overhead decreased significantly in comparison to the initial port. The biggest impact was observed on the `cuMemcpyHtoDAsync` call, which we attribute to the excess HtoD transfers of

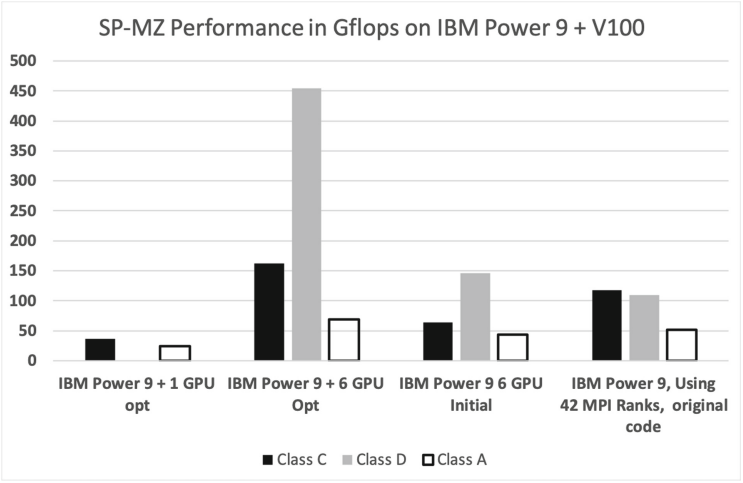

Fig. 5. Single node performance for different classes of the SP-MZ Benchmark on Ascent, using 1 GPU and 6 GPUs on the node. Class D for a single GPU failed.

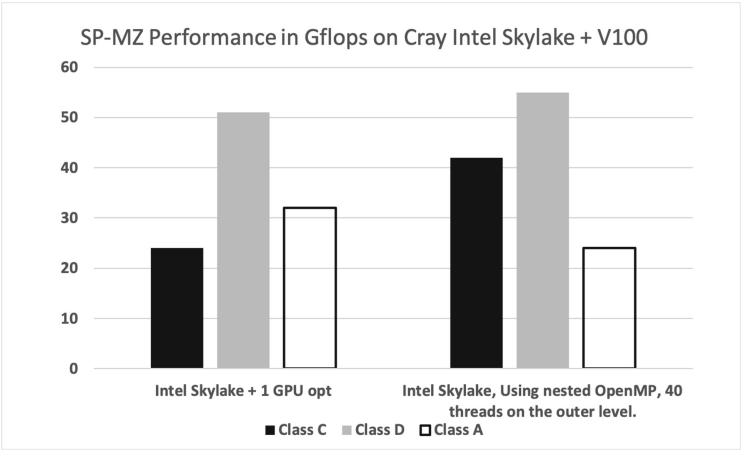

Fig. 6. Performance for different classes of the SP-MZ Benchmark on Cori GPU

the small constants. Those issues were fixed with the OpenMP "declare target" directive to make variables available on the device across target regions. As noted previously, we also declared some subroutine arguments to be passed by value rather than being passed by reference. An example for this are the array dimensions in the calls to the solver routines. Another good performance boost came from using the pre-allocated `lhs` scratch memory for all zones.

As described in Sect. 4 we used the OpenMP tasking mechanism to allow for kernel execution overlap. In principle, this allows one to "implicitly" compute SP-MZ zones in parallel because the work per zone is independent. We used

OpenMP dependencies for each zone to enable computation for different zones to execute concurrently. Figures 7 and 8 display the execution timeline generated by the NVIDIA nvvp visual profiler on Ascent and Cori-GPU, respectively. The results were obtained by running 10 iterations of the Class C benchmark. The "Compute" part of the timeline, shows that there was not much overlap of the kernel execution. We observed that only 2 streams were used for the execution of the kernels. On the Cori system, on the other hand, kernel execution overlapped. This can be clearly seen in the nvvp execution timeline in Fig. 8. Here we observed that 7 streams were used for the kernel execution. Another observation is, that the IBM compiler introduces additional Memcpy (HtoD) because variables are not passed by value as part of the kernel launch.

The lines in the "Compute" part of the timeline, show whether kernels were running concurrently. The results show that on Ascent we did not get the amount of overlap we were hoping for, while we could clearly observe overlapping kernels on the Cori system. It is unclear why there was not more overlap with the IBM compiler on the Ascent system. We could work around this manually by adding explicit zone-level parallelism. This would be achieved by lifting the "target teams" region to the sweep over the zones. All ADI functions will need to be target functions that could then be called at the team level. We consider such an implementation as an item for future work.

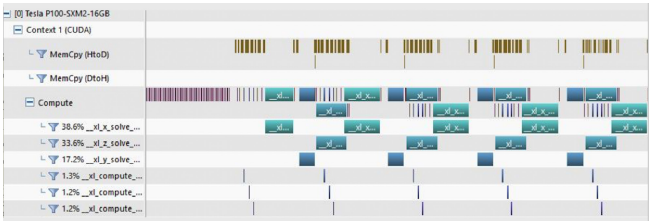

Fig. 7. Execution timeline of SP-MZ Class C on Ascent as seen on the profiler

We mentioned in Sect. 4 that the CPU performance degraded during the development process. While the code is functionally portable between different hardware platforms, its performance is clearly not. We introduced code patterns that improve GPU performance, but degrade performance on the CPU. We noticed, for example, that the solver in z dimension suffered most during the development process. In the refactored code, the stride on the inner, non-vectorized loop is very large. So this is an issue of terrible cache reuse, as it will thrash the L1 cache considerably. A code snippet of the loop in question is displayed in Fig. 9.

While performance portability is very important, this was not the focus of our effort. We plan to address this in future work. The question of performance portability for directive-based programming is addressed in [13], for example.

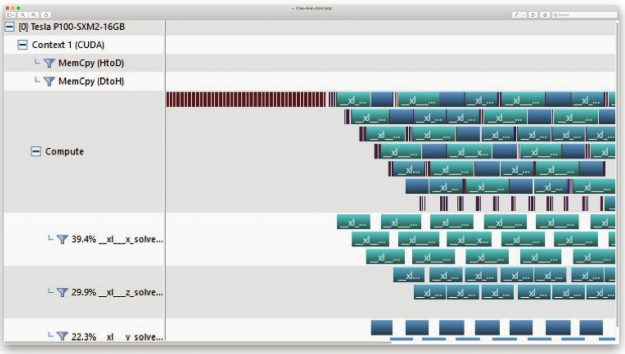

Fig. 8. Execution timeline of SP-MZ Class C on Cori GPU as seen on the profiler

```
! sequential loop
              do   k =  1, nz-2
                 ...
                 ru1 = c3c4*rho_i(i,j,k-1)
                 cv_km1 = ws(i,j,k-1)
                 rhos_km1 = dmax1(dz4 + con43 * ru1,
     >                           dz5 + c1c5 * ru1,
     >                           dzmax + ru1,
     >                           dz1)
                 ...
                 lhs4(i,2,j,k) = -dttz2 * cv_km1 - dttz1 * rhos_km1
                 lhs4(i,4,j,k) =  dttz2 * cv_kp1 - dttz1 * rhos_kp1
                 ...
              end do
```

Fig. 9. Inner loop in refactored z solve routine

6 Summary and Conclusions

In this study we showed that OpenMP 4.5 offers a path forward for achieving reasonable performance on accelerators when adapting Fortran codes that are over 10 years old. We described our experience using the OpenMP 4.5 support for heterogeneous compute nodes. We found that the compiler directives permitted us to port the legacy Fortran CFD mini-application for execution on compute nodes endowed with NVIDIA V100 GPU accelerators. We collected results on two different evaluation systems and conducted performance studies.

As a positive observation, we note that using OpenMP compiler directives permitted us to port the code within one week of programming effort. We also note that significant code changes were required to obtain acceptable performance. What is important to emphasize is that our approach was one-sided, in the sense that we targeted code execution and performance gains only on GPUs; no effort was made to retain performance for execution on CPUs. During the development process, we noticed that code changes that improved the performance on the GPU accelerators actually decreased performance on CPUs. This implies that when porting legacy codes (i.e. codes not originally designed to use co-processors), different aspects of the numerical method at hand may have to be handled differently and distributed across co-processors and the CPUs. An

approach that is designed systematically to leverage GPUs/co-processors and CPUs in a manner tailored to the numerical method can potentially yield much higher performance gains. We envision that such an approach is a suitable path forward.

Another positive aspect was, that we did not find a lack of functionality in OpenMP when comparing it to OpenACC. The lack of a *present* clause did not present a hurdle. Using OpenMP tasking allowed us to implement asynchronous execution expressing the dependences more explicitly than using the OpenACC *async* construct.

The flip side of using the convenience of OpenMP is that we depend very much on compiler support. We have mentioned a number of system specific issues we encountered. Performance optimization is also difficult, as it is often not clear if poor performance is due to poorly chosen directives or bad code generated by the compiler. We plan further studies to investigate such issues.

A number of paths to portability (e.g. Kokkos, Raja, and Thrust), can work well for C++ codes, but are not appropriate for codes entirely written in Fortran. OpenMP can play an important role here. In our work we focused on exclusively porting code to GPUs, where we also observed minor degraded CPU-only performance. However, we expect that porting of Fortran codes without such performance decreases will be possible in the future. There are a number of research projects on the way addressing specifically performance portability of Fortran codes as discussed in [13]. The porting process will greatly improve as 5.0 features become available, where one will be able to use different functions and directives for specific vendor hardware, and in that sense OpenMP can undertake the middleware role (like Kokkos, for example), performing appropriate changes based on hardware. Our future work will include enabling explicit zone-level parallelism, compute kernel optimization and multi-node scaling studies.

Acknowledgments. The authors would like to thank all those who helped with valuable advice during the NERSC/OLCF Oakland Hackathon, especially, Kevin Gott and Jack Deslippe from NERSC and Tom Papatheodore from OLCF for facilitation of an excellent hackathon experience, Max Katz and Angela Chen from NVIDIA who taught us a lot about profiling, and Fazlay Rabbi from NERSC who helped us with compiling and benchmarking on the Ascent and Cori systems. Thanks to Joel Bretheim from the Naval Research Lab, who contributed to the code optimizations. Special thanks go to Kelvin Li from IBM for advising on the changes of Fortran value versus by-reference arguments. I. Nompelis would like to thank Prof. G. V. Candler's group from the Department of Aerospace Engineering and Mechanics, Univ. of Minnesota for their support. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231. This research also used resources of the Oak Ridge Leadership Computing Facility (OLCF), which is a DOE Office of Science User Facility supported under Contract No. DE-AC05-00OR22725. We thank OLCF and NERSC for their support during our study. This work was partially funded by NASA Contract No: 80ARC018D001. Work at the hackathon included funding from the Department of Defense High Performance Computing Modernization

Program (HPCMP) under User Productivity, Technology Transfer and Training (PETTT) Contract No. GS04T09DBC0017 and for the University of Hawai'i under Maui High Performance Computing Center (MHPCC) Contract No. N00024-19-D-6400.

References

1. CUDA CuFFT Library (2017). http://developer.download.nvidia.com/compute/cuda/1.0/CUFFT_Library_1.0.pdf
2. The OpenACC application programming interface (2018). https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.7.pdf
3. The OpenMP API specification for parallel programming (2018). https://www.openmp.org
4. cuBLAS Library (2019). https://docs.nvidia.com/pdf/CUBLAS_Library.pdf
5. CUDA toolkit documentation (2019). https://docs.nvidia.com/cuda/
6. NERSC development system documentation (2019). https://docs-dev.nersc.gov/cgpu/
7. *NERSC OLCF GPU Hackathon* (2019). https://sites.google.com/lbl.gov/july2019-gpu-hackathon
8. Summit training system (ascent) (2019). https://www.olcf.ornl.gov/for-users/system-user-guides/summit/summit-user-guide/#training-system-(ascent)
9. Summit user guide (2019). https://www.olcf.ornl.gov/for-users/system-user-guides/summit/
10. Bailey, D., et al.: The NAS parallel benchmarks. Tech. rep. NRN Technical report NRN-94-007, NASA Ames Research Center (March 1994)
11. Beam, R.M., Warming, R.F.: An implicit factored scheme for the compressible Navier-stokes equations. AIAA J. **16**, 393–401 (1978)
12. Brook, P., et al.: GNU Fortran (2019). https://gcc.gnu.org/fortran/
13. Clement, V., et al.: The CLAW DSL: abstractions for performance portable weather and climate models. In: Proceedings of the Platform for Advanced Scientific Computing Conference. PASC 2018. ACM, New York (2018). https://doi.org/10.1145/3218176.3218226
14. Diaz, J.M.M., Jost, G., Chandrasekaran, S., Pino, S.: Is OpenMP 4.5 target off-load ready for real life? - a case study of three benchmark kernels. https://openmpcon.org/wp-content/uploads/2018_Session1_Diaz.pdf
15. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: enabling manycore performance portability through polymorphic memory access patterns. J. Parallel Distrib. Comput. **74**(12), 3202–3216 (2014). https://doi.org/10.1016/j.jpdc.2014.07.003. http://www.sciencedirect.com/science/article/pii/S0743731514001257. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing
16. Group, K.O.W.: The OpenCL specification v2.2-11 (2019). https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf
17. Hornung, R., et al.: ASC tri-lab co-design level 2 milestone report. Tech. rep., Lawrence-Livermore National Laboratories (September 2015)
18. Jacob, A.C., et al.: Efficient fork-join on GPUs through warp specialization. In: 2017 IEEE 24th International Conference on High Performance Computing (HiPC), pp. 358–367 (December 2017). https://doi.org/10.1109/HiPC.2017.00048
19. McCall, A.J.: Multi-level parallelism with MPI and OpenACC for CFD applications. Ph.D. thesis, Virginia Tech (2017)

20. Pulliam, T., Chaussee, D.: A diagonal form of an implicit approximate-factorization algorithm. J. Comput. Phys. **39**, 347–363 (1981)
21. Shan, H., et al.: A programming model performance study using the NAS parallel benchmarks. Sci. Program. **18**(3–4), 153–167 (2010). https://doi.org/10.1155/2010/715637
22. Stone, C.P., Elton, B.H.: Accelerating the multi-zone scalar pentadiagonal CFD algorithm with OpenACC. In: Proceedings of the Second Workshop on Accelerator Programming Using Directives, WACCPD 2015, pp. 2:1–2:7. ACM, New York (2015). https://doi.org/10.1145/2832105.2832110
23. Van der Wijngaart, R.F., Jin, H.: NAS parallel benchmarks, multi-zone versions. Tech. rep., NASA Ames Research Center (July 2003)