Early Experiences Writing Performance Portable OpenMP 4 Codes

Verónica G. Vergara Larrea*, Wayne Joubert*, M. Graham Lopez†, Oscar Hernandez†
*National Center for Computational Sciences
†Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN
{vergaravg,joubert,lopezmg,oscar}@ornl.gov

***Abstract*—In this paper, we evaluate the recently available directives in OpenMP 4 to parallelize a computational kernel using both the traditional shared memory approach and the newer accelerator targeting capabilities. In addition, we explore various transformations that attempt to increase application performance portability, and examine the expressiveness and performance implications of using these approaches. For example, we want to understand if the target map directives in OpenMP 4 improve data locality when mapped to a shared memory system, as opposed to the traditional first touch policy approach in traditional OpenMP. To that end, we use recent Cray and Intel compilers to measure the performance variations of a simple application kernel when executed on the OLCF's Titan supercomputer with NVIDIA GPUs and the Beacon system with Intel Xeon Phi accelerators attached. To better understand these trade-offs, we compare our results from traditional OpenMP shared memory implementations to the newer accelerator programming model when it is used to target both the CPU and an attached heterogeneous device. We believe the results and lessons learned as presented in this paper will be useful to the larger user community by providing guidelines that can assist programmers in the development of performance portable code.**

***Keywords*-Performance Portable Programming Models; Shared Memory Programming; Accelerator Programming; OpenMP 4.0**

I. Introduction

As we move towards exascale, we see two general trends in the compute node architecture of high performance computing (HPC) systems: one with heterogeneous nodes containing accelerators, and the other with homogeneous shared memory compute nodes. The prevalence of both trends is clearly reflected in the architectures selected for the CORAL project [1], which include Summit and Sierra, two heterogeneous IBM Power-based systems with multiple NVIDIA Volta GPUs per node [2], [3]; and Aurora, a homogeneous third-generation Intel Xeon Phi-based system [4]. European [5] and Japanese exascale systems plan to use FPGAS and multi-core processors. These architectural classes are different enough from each other (and even within each class in regard to different product generations) to make performance portable programming a nontrivial task.

These developments also come at a time when programmer productivity and the ability to produce portable code have been recognized as major concerns, generating workshops and initiatives to address the application performance portability challenge. The Application Readiness and Portability meetings at the Oak Ridge Leadership Computing Facility (OLCF) and the National Energy Research Scientific Computing Center (NERSC), the Workshop on Portability Among HPC Architectures for Scientific Applications held at SC15 [6], and the DOE Centers of Excellence Performance Portability Meeting [7] are just a few examples of these efforts.

Producing well-performing code on different architectures involves identifying a programming model that both provides an appropriate level of abstraction to expose multiple sources of parallelism and also maps that parallelism to make efficient use of the underlying architecture. To address these challenges, directive-based programming models, including OpenMP and OpenACC, have been proposed to offset the increasing development cost of creating architecture-specific code. Furthermore, the latest OpenMP 4.5 specification allows the programmer to choose between writing code for traditional shared memory execution and using its new accelerator programming model capable of targeting any type of node. However, as compiler support is starting to emerge, the performance portability implications of these models on different architectures are not yet well understood.

It is the view of many that the OpenMP standard offers a way forward to address the need for performance portability of applications. With appropriate application code restructuring to take advantage of the accelerated architecture, as well as minimally invasive directives to guide the code generation process by the compiler, it is hoped that OpenMP will realize the objective of enabling performance portable applications.

Traditionally, the OpenMP 3.1 version of the standard has been a viable means for efficient programming of shared memory architectures, and this standard currently supports some self-hosted accelerated nodes. More recently, the OpenMP 4 standard has added support for the offload model to address the needs of heterogeneous node architectures. Although the OpenMP standard has broad industry support and wide usage, it still embodies two different programming models: the shared memory model and the

offload model, and thus it remains uncertain how to use the standard to develop efficient and portable code for accelerated architectures of each type.

In this paper we examine the effectiveness of the OpenMP 4 offload model as an ubiquitous technique for programming modern node architectures. The offload model is inherently suitable for programming attached accelerator devices for which data transfers between on-node memory spaces is specified explicitly. But beyond this, the use of data directives also holds promise for self-hosted accelerators insofar as it provides a systematic approach to specifying memory access patterns for application code, which is an important consideration for performance.

To execute this study, we employ several modern OpenMP 4-capable compilers on different platforms. Using the simple and well-understood Jacobi computational kernel as an example, we compare the performance of traditional OpenMP shared memory-style and OpenMP 4 offload directives with these compilers to understand their respective performance characteristics. Based on this, we evaluate the effectiveness of OpenMP 4 directives as a potential solution to the performance portability problem of modern architectures. Section II provides a summary of alternative methods to OpenMP and related work, with an overview of the core OpenMP programming models being given in sections III and IV. In sections V and VI we describe our findings on the performance portability of the various ways to use OpenMP, and then we present some conclusions and recommendations for future directions to further improve OpenMP for performance portable programming in sections VII and VIII.

II. Related Work

OpenMP has traditionally played a key role in exposing parallelism for HPC applications running on homogeneous nodes, being used alongside MPI as the "X" in the so-called "MPI+X" configuration. However, OpenMP was originally designed to handle single address space parallelism; thus detached accelerators with separate memory hierarchies requiring explicit data transfer could not be handled by OpenMP until new features were added in OpenMP 4 to support this functionality. In conjunction with these developments in the OpenMP specification and implementations, alternative programming models are being developed to efficiently expose node-level parallelism for a variety of target architectures.

Two projects that have been arisen and were designed from the start to address the performance portability issue are Kokkos [8] and RAJA [9]. These make heavy use of standard C++ features that are able to hide the implementation details necessary to target good performance on multiple architectures from the application developer. However, since these abstractions rely on C++ language features, these programming models are not available to strictly C or Fortran applications without the use of cumbersome foreign function interface designs, which negates some of the convenience and abstraction that the models intend to provide.

Perhaps the most closely related alternative to OpenMP that is currently available is OpenACC [10], which is another directive-based programming model specifically built to handle accelerator offload computation. The OpenACC specification was created to unify all the directive-based approaches to program accelerators including CAPS HMPP, PGI, and Cray accelerator directives. The purpose of this multi-organization effort was to develop a portable approach to directive-based programming and to provide early experiences on the use of directives to program accelerators. While OpenACC is designed to support multiple hardware targets, support for host CPU execution is quite recent, and implementations for offloading to the Xeon Phi platforms are not supported in production environments. These limitations have prevented OpenACC from becoming a universal performance portable option for application developers.

In the ideal case, standardized base language features would be sufficient to achieve good performance on multiple platforms including discrete accelerators. Indeed, the Fortran standard [11] has had parallelization features such as coarrays and `do concurrent` since 2008. Also, the upcoming C++17 standard is expected [12] to include generic features for parallelization. Unfortunately, these features presently remain insufficient for the case of accelerator programming with multiple memory address spaces and complex compute and memory hierarchies.

Finally, runtime systems designed specifically for HPC applications have been proposed as a solution for either node-level parallelism, more sophisticated internode communication than MPI provides, or both. These runtimes can also have other features besides just exposing the raw parallelism, such as fault-tolerance, automatic load balancing, task-based programming abstractions, and data locality-aware compute scheduling.

Before OpenMP 4.0, Intel supported accelerated programming by use of proprietary directives [13] for Intel Xeon Phi coprocessors. These nonstandard directives supported native and offload modes, making it possible either to run applications on the coprocessors locally or to offload compute intensive regions from CPU to coprocessors. Other examples of previous efforts include CAPS HMPP and PGI compiler accelerator directives and hiCUDA [14]. Early studies have explored the performance of the directive-based approach for GPU programming [15] based on the HMPP and PGI compilers. The results show that using directives with additional code transformations can achieve comparable performance to optimized hand-written CUDA codes in many cases. Other research evaluated the effectiveness of directive-based approach with favorable findings [16]. OpenMPC [17] complemented the OpenMP language with directives to assist with the translation of OpenMP code to CUDA.

III. OpenMP programming models

OpenMP is an Application Program Interface (API), jointly defined by a group of vendors, laboratories, users, and academia. OpenMP provides a portable and scalable model for developers of shared memory parallel applications, supporting a directive-based API for C/C++ and Fortran on multiple shared memory systems. The major features of OpenMP include its various constructs and directives for specifying the fork/join of parallel regions, worksharing, thread synchronization, the creation of tasks, and a data environment.

A. Background

Directive-based programming models were developed to reduce the complexity of porting code to accelerators. Their goal is to let the developer simply insert directives into a program that serve as a guide for the compiler to generate underlying code that makes efficient use of accelerators (though some code restructuring may still be needed to improve performance). The two major efforts in the field are OpenACC and OpenMP. As of today, many production codes use directives, for example, the S3D combustion application [18], CAM/SE [19], COSMO [20], [21] and ICON [22] applications, which all were successfully ported to GPUs by using OpenACC and achieved substantial speedups.

After the viability of directive-based programming for accelerators was demonstrated, a larger effort was started to unify shared memory programming with the accelerator model. OpenMP 4.0/4.5 addressed this by extending its original shared memory directives (e.g., parallel regions, worksharing, tasks) to support the accelerator model. With this release, a series of new features and concepts was introduced to improve the interoperability of these models, for example, by using a unified task-based model in which tasks can run on the accelerator or on CPU threads. Liao [23] performed an early evaluation of the OpenMP accelerator model by using an initial implementation of the Heterogeneous OpenMP (HOMP) compiler and showed successes using this approach.

The accelerator model in both OpenACC and OpenMP assumes host-directed execution with an attached accelerator device such as a GPU or a Xeon Phi (the accelerator model can also be executed on a shared memory system or self-hosted system). The application begins execution on the host and accelerated regions of the application code are offloaded to the accelerator device under control of the host. The device executes these parallel regions, which typically contain work-sharing loops, or regions of code, which in turn can contain one or more loops executed as kernels. The host orchestrates the execution by allocating memory on the accelerator device, initiating data transfer, sending the code to the accelerator, passing arguments to the accelerated region, queuing the device code, waiting for completion, transferring results back to the host, and deallocating memory. In most cases, the host can queue a sequence of operations to be executed on the device, one after another.

The memory model for accelerators is based on defining copy-in and copy-out regions for memory that is synchronized with the host. All data movement between host memory and device memory is performed by the host through runtime library calls that explicitly move data between the separate memories, typically using direct memory access (DMA) transfers.

B. Accelerator Programming Model Features

The following section describes some OpenMP 4.0 accelerator programming features that were used in this study.

1) Executing on the device: In OpenMP, the `target` directive begins a region of code to be executed on the accelerator device. The `device` clause can be used to specify the desired device if multiple devices are present. In OpenMP, asynchronous device operations are not supported directly but can be achieved by using the OpenMP `task` construct.

2) Specifying teams: In OpenMP, the concept of `gangs` or `teams` denotes a collection of thread groups satisfying certain properties: for example, it is not possible to synchronize across different teams over the lifetime of their existence. In OpenMP, the `teams` directive creates a league of thread teams that execute in the region. For convenience, this directive can be combined with the `target` construct; adjacent directives can also be combined in different contexts in OpenMP.

In OpenMP a single master thread from each team is active in the structured block. The `teams` directives are generally not used in this manner in isolation, but are combined with other directives for additional parallelism at the thread and SIMD levels.

3) Distributing loop iterations to teams: The OpenMP `distribute` directive specifies that the iterations of one or more loops will be executed by the active thread teams. In the absence of further specifications, elements of the iteration space are each assigned to the master thread of each team and only these master threads are deployed. In OpenMP, it is possible in some cases to apply loop directives to multiple nested loops by flattening the iteration space using the `collapse` clause. Also, the OpenMP `dist_schedule` clause can be used to control how loop iterations are mapped to teams.

4) Distributing loop iterations to teams with multiple threads: The `distributed parallel loop` directive instructs OpenMP to distribute iterations of a loop across teams and threads within a team.

5) Distributing loop indices to teams, threads and SIMD units: In OpenMP the `simd` directive indicates that a loop should be vectorized for the targeted platform. When used in conjunction with the distribute and parallel loop directive the

compiler will schedule the iterations across teams, threads within the teams and SIMD units.

6) Creating a data region: OpenMP has a mechanism for specifying a data region, which is a period of execution time with a distinct beginning and end for which the residence of a data object on the device can be defined. This region can be identical to the parallel execution region or can be specified independently, as described here. The relevant operations that can be selected are: to allocate memory for the object on the device, to copy the host data object to the device on region entry, to copy the data object back to the host, and to delete the device object on exit. In OpenMP 4.0, scalars are copied in and out (`defaultmap(tofrom: <scalar>)`) of the device by default.

The OpenMP `target data` directive is used to specify data transfers between host and accelerator within a code region. The same effects can be obtained for a `target` region by using these clauses to control data behavior on entry into and exit from the parallel region. OpenMP also provides the `enter data` and `exit data` clauses which allow a more flexible specification of data regions not associated with a single code block.

7) Updating data objects: Within the data region it may be necessary to refresh the host data object with the corresponding data on the device or vice versa. The OpenMP `target update` directive is an executable directive which performs a refresh of a host data item from the device copy of the item with the `from` clause or vice versa with the `to` clause.

IV. OpenMP programming models in practice

In order to compare the two programming models, the shared memory (OpenMP 3.1) and accelerator styles, we performed an in-depth study of a representative application kernel. The performance of this kernel was evaluated on two different platforms: Chester, a Cray XK7 test system available at the OLCF, and Beacon, a Cray CS300-AC Linux cluster available at the National Institute for Computational Sciences (NICS). In addition, two compilers were used: the Intel Compiler 16.0.1, and the Cray Compiler Environment (CCE) 8.4.5. Both were chosen because of the level of support they provide for OpenMP 4.0.

Several different experiments were conducted to evaluate the performance of the programming models on a given architecture. First, the application kernel written in standard shared memory OpenMP 3.1 was considered. Second, this kernel was modified to include `target` directives in order to compare the performance of the shared memory model when offloaded to an accelerator device. Finally, the original OpenMP 3.1 code was fully ported to the accelerator model using OpenMP 4.0 directives. Table I shows the different versions of the code that were used and the nomenclature used to distinguish each one.

App. Kernel Version	Abbrev.	Executed On	Offloading To
`shared memory`	SM	CPU	n/a
		Xeon-Phi	n/a
`shared+target`	SM+t	CPU	CPU
		CPU	Xeon Phi
		CPU	GPU
		Xeon Phi	Xeon Phi
`accelerator`	accel	CPU	CPU
		CPU	Xeon Phi
		CPU	GPU
		Xeon Phi	Xeon Phi

Table I
Experiments by Programming Model and Platform

A. Platforms

To test the performance of the OpenMP-based programming models, we make use of two HPC computing platforms: the OLCF Chester Cray XK7 system, and the NICS Beacon Intel Phi Knights Corner system [24].

Chester is a Cray XK7 system architecturally identical to the OLCF Titan system but consisting of a single cabinet. It is used here since its compilers and software are slightly newer than Titan's, it being an early deployment system for new software releases. A detailed description of the architecture can be found in [25]. Of particular importance to the present study are the following specifications: a compute node consists of an AMD Interlagos 16-core processor with a peak flop rate of 140.2 GF and a peak memory bandwidth of 51.2 GB/sec, and an NVIDIA Kepler K20X GPU with a peak double precision flop rate of 1,311 GF and a peak memory bandwidth of 250 GB/sec. For this platform, Cray compilers are used, with version 8.4.5. As shown in Fig. 1 two different execution modes are possible on Chester: *standard* using only the CPU, and *offload* running the executable on the CPU and offloading to the GPU.

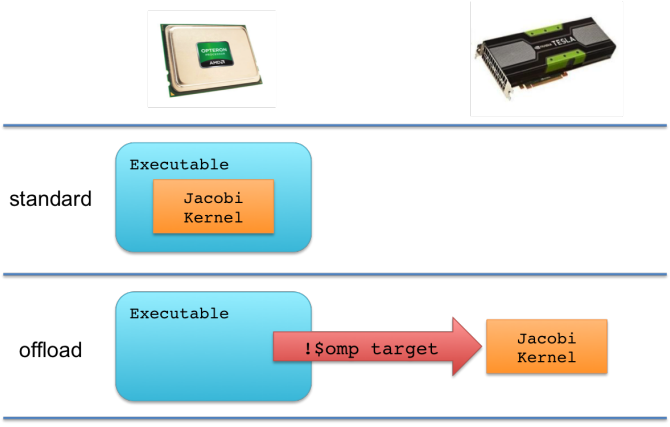

Figure 1. Execution modes on Titan

Beacon is an Intel Phi Knights Corner cluster, each compute node containing two 8-core Xeon E5-2670 processors and four 5110P Intel Phi processors. Each Intel Xeon

processor has a peak flop rate of 165 GF and a peak memory bandwidth of 51.2 GB/sec, yielding aggregate peak rates for the two CPUs of 330 GF and 102.4 GB/sec. Each Intel Xeon Phi processor has peak double precision performance of 1,011 GF and a peak memory bandwidth of 320 GB/sec. For this platform, Intel compilers are used, with version 16.0.1 from the Intel XE Compiler suite version 2016.1.056. Fig. 2 depicts the three different execution modes possible on Beacon: *standard* running only on the CPU, *offload* running the executable on the CPU and offloading to the Intel Xeon Phi, and also *native* or *self-hosted* mode running on the Intel Xeon Phi directly.

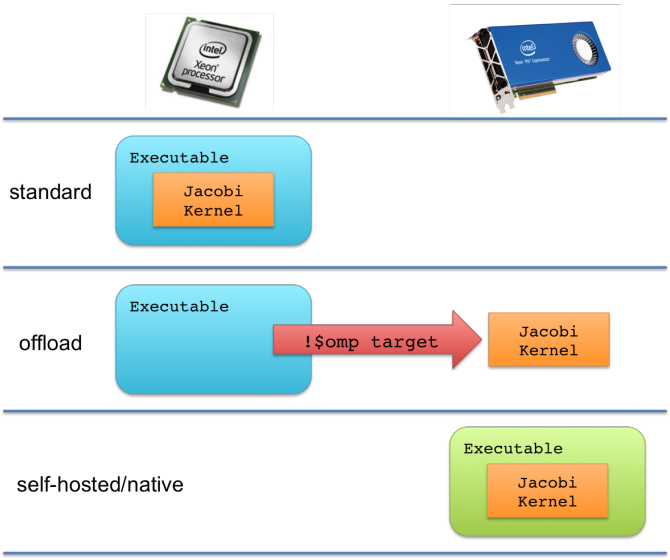

Figure 2. Execution modes on Beacon

B. Test case: Jacobi

To evaluate application portability using OpenMP programming models, it is useful to employ small application kernels that resemble compute-intensive portions of full applications. For this study we focus on a simple Jacobi iterative solver for a 2-D structured grid constant coefficient problem derived from a discretized partial differential equation. This well-understood kernel represents structured grid and sparse linear algebra computational motifs. Its operations resemble those of many application codes, and the kernel itself is used in cases such as implicit grid solvers and structured multigrid smoothers.

Jacobi iteration is a simple linear solver method that can be used, for example, to solve computational fluid dynamics (CFD) problems based on a finite difference grid by updating grid cells with neighbor stencil values. Figure 3 shows a Jacobi iteration applied to a 2-D structured grid with 5-point stencil implemented in Fortran. The kernel was derived from the OpenMP Jacobi example available at [26].

```
do while (k.le.maxit .and. error.gt. tol)
 error = 0.0
 do j=1,m  !---Copy solution
  do i=1,n
   uold(i,j) = u(i,j)
  enddo
 enddo
 do j = 2,m-1 !---Update interior
  do i = 2,n-1
   r = &
       (ax*(uold(i-1,j) + uold(i+1,j)) &
     + ay*(uold(i,j-1) + uold(i,j+1)) &
     - f(i,j))*brecip + uold(i,j)
   u(i,j) = uold(i,j) - omega * r
   error = error + r*r
  enddo
 enddo
 k = k + 1
 error = sqrt(error)/dble(n*m)
enddo
```

Figure 3. Jacobi iteration

To evaluate OpenMP parallelization of this kernel, we compare three approaches which are summarized in Table I:

1) Use of OpenMP 3.1 `parallel` and `do` directives to support traditional shared memory parallelism. This approach is appropriate to CPUs and self-hosted devices for which the host code and the computational kernel both execute on the same device. See Figure 4.
2) Use of OpenMP 4.0 device directives such as `target`, `target data`, `teams`, and `distribute`. This programming method is primarily designed for cases in which the host code launches computational kernels on a separate device. However, as a special case, it can also be used for the host processor to launch a kernel on the host processor itself. See Figure 5.
3) Use of OpenMP 3.1 directives augmented with OpenMP 4.0 `target` and `target data` directives. This case is considered to evaluate whether compilers are able to generate efficient parallel code from existing OpenMP 3.1 code with minor additions to enable wider portability. See Figure 6.

V. Results

A. Experiment details

In this section, we use the nomenclature previously defined in Table I to describe the different experiments conducted.

For each experiment the Jacobi solver is executed for a fixed number (100) of iterations, and timings are reported for the Jacobi kernel excluding the runtime for the initialization and final correctness check steps. The solver is run for a

```
do while (k.le.maxit .and. error.gt. tol)
 error = 0.0
!$omp parallel
!$omp do
 do j=1,m  !---Copy solution
  do i=1,n
   uold(i,j) = u(i,j)
  enddo
 enddo
!$omp do private(r) reduction(+:error)
 do j = 2,m-1 !---Update interior
  do i = 2,n-1
   r = &
       (ax*(uold(i-1,j) + uold(i+1,j)) &
     + ay*(uold(i,j-1) + uold(i,j+1)) &
     - f(i,j))*binv + uold(i,j)
   u(i,j) = uold(i,j) - omega * r
   error = error + r*r
  enddo
 enddo
!$omp enddo nowait
!$omp end parallel
 k = k + 1
 error = sqrt(error)/dble(n*m)
enddo
```

Figure 4. Jacobi iteration with OpenMP 3.1 directives

range of problem sizes of the form $N \times N$, where N denotes the number of grid cells on an edge of the domain, so that the dimension of the matrix and vectors is N^2.

Jacobi is a fundamentally memory-bound algorithm whose performance is primarily dictated by the speed of access to memory. Thus, we report the code performance in terms of attained GB/sec, a scaled inverse of execution time (i.e., a higher value represents better performance).

B. Experiments: Beacon

Due to the nature of the Intel Xeon Phi, it is possible to run a kernel either in offload or native mode. In offload mode, two combinations of experiments are possible, one using the accelerator as the target device, and another using the host, where the code is launched, as the target device. To enable the latter option, the Intel compiler provides the `-qopenmp-offload=host` option.

Experiments on the Beacon system were conducted using all available cores, which, with hyper-threading enabled, is equivalent to 32 threads per node. By default, all possible threads are used by the OpenMP runtime. When running experiments on the Intel Xeon Phi, either via offloading or in native mode, the default number of hardware threads was used. In the offload case, one Intel Xeon Phi core is reserved for the runtime, which results in a total of 236 threads. For experiments run natively on the Xeon Phi, all cores are used which is equivalent to 240 threads. In cases

```
!$omp target data map(to:f) map(tofrom:u)
!$omp+ map(alloc:uold)
do while (k.le.maxit .and. error.gt. tol)
 error = 0.0
!$omp target
!$omp teams distribute parallel do
 do j=1,m  !---Copy solution
  do i=1,n
   uold(i,j) = u(i,j)
  enddo
 enddo
!$omp end teams distribute parallel do
!$omp end target
!$omp target
!$omp teams distribute parallel
!$omp+ do reduction(+:error)
 do j = 2,m-1 !---Update interior
!$omp simd private(r) reduction(+:error)
  do i = 2,n-1
   r = &
       (ax*(uold(i-1,j) + uold(i+1,j)) &
     + ay*(uold(i,j-1) + uold(i,j+1)) &
     - f(i,j))*binv + uold(i,j)
   u(i,j) = uold(i,j) - omega * r
   error = error + r*r
  enddo
 enddo
!$omp end teams distribute parallel do
!$omp end target
 k = k + 1
 error = sqrt(error)/dble(n*m)
enddo
!$omp end target data
```

Figure 5. Jacobi iteration with OpenMP 4 directives

where the OpenMP 4.0 `teams` directive is used, by default, only 1 team is created.

Figure 7 shows the results obtained from running experiments in the three different modes described in Table I. The results show that:

- Jacobi executed natively on the Intel Xeon Phi coprocessor performs well in all three cases: SM/Phi, SM+t/Phi/offload=Phi, and accel/Phi/offload=Phi. The best performance is achieved by the SM/Phi case that is able to reach 180.9 GB/s which is equivalent to approximately 56% of the peak memory bandwidth available on the on the coprocessor.
- The shared memory version of Jacobi when executed on the CPU, SM/CPU, achieves approximately 81.7 GB/s, or 79.7% of peak memory bandwidth on Beacon nodes. It is interesting to see that the accelerator model version, accel/CPU/offload=CPU, as well as the shared memory with `target` directives version, SM+t/CPU/offload=CPU, both achieve similar performance for large matrix sizes. The former achieves 80.5 GB/s, whereas the latter obtains 78.4 GB/s, which is

```
!$omp target data map(to:f) map(tofrom:u)
!$omp+ map(alloc:uold)
do while (k.le.maxit .and. error.gt. tol)
 error = 0.0
!$omp target
!$omp parallel
!$omp do
 do j=1,m  !---Copy solution
  do i=1,n
   uold(i,j) = u(i,j)
  enddo
 enddo
!$omp do private(r) reduction(+:error)
 do j = 2,m-1 !---Update interior
  do i = 2,n-1
   r = &
       (ax*(uold(i-1,j) + uold(i+1,j)) &
     + ay*(uold(i,j-1) + uold(i,j+1)) &
     - f(i,j))*binv + uold(i,j)
   u(i,j) = uold(i,j) - omega * r
   error = error + r*r
  enddo
 enddo
!$omp enddo nowait
!$omp end parallel
!$omp end target
 k = k + 1
 error = sqrt(error)/dble(n*m)
enddo
!$omp end target data
```

Figure 6. Jacobi iteration with OpenMP 3.1 and target directives

equivalent to 78.6% and 76.6% of the peak memory bandwidth on the CPU, respectively.

- The results from experiments with the accelerator version of the Jacobi kernel show that running the code natively on the coprocessor achieves the highest bandwidth. The accel/Phi/offload=Phi experiment achieves 170.8 GB/s, or 53% of the peak memory bandwidth available on the Intel Xeon Phi. In the other two accelerator model cases, accel/CPU/offload=CPU and accel/CPU/offload=Phi, the accelerator model when executed on the CPU offloading to itself provides comparable performance to the SM/CPU and SM+t/CPU/offload=CPU cases.
- On the other hand, the accelerator code executed on the CPU and offloading to the Intel Xeon Phi does not perform well for small matrices. It is only with matrices of $12,800 \times 12,800$ elements or more that we start seeing the benefit of offloading to the coprocessor. For large matrices, the accel/CPU/offload=Phi version of the code reaches a memory bandwidth of 70.2 GB/s. It is apparent that overheads pertaining to the offload, e.g., transfer of data, limit performance.
- The SM+t cases show that offloading to CPU itself results in performance similar to that achieved by SM/CPU and accel/CPU/offload=CPU. The SM+t/CPU/offload=CPU obtains a memory bandwidth of 78.8 GB/s. In the SM+t/CPU/offload=Phi case, however, we observe poor performance for matrices with less than $12,800 \times 12,800$ elements.
- On the Intel Xeon processor, performance starts to decrease when the matrix reaches the $25,600 \times 25,600$ element count. On the Intel Xeon Phi, however, performance starts to tail off much earlier, namely when the matrix has $12,800 \times 12,800$ elements or more. This is true in both the offload and native cases.

C. Experiments: Chester

Experiments for Chester using CPU threading are done using the full complement of 16 hardware threads corresponding to the 16 cores of the AMD Interlagos processor, as specified by `OMP_NUM_THREADS`. For GPU cases, default values are used for thread and team counts.

In order to evaluate the performance of the kernel in a similar fashion as done on Beacon, both experiments using the accelerator as target and also the host as target were conducted. The Cray programming environment provides different accelerator modules that allow the user to specify the target device. To target the GPU, the `craype-accel-nvidia35` module was used, and to target the host, the `craype-accel-host` module was used.

Results for these experiments are given in Figure 8. Several features of these results are evident.

- The standard cases SM/CPU and accel/CPU/offload=GPU execute at a significant fraction of peak memory bandwidth, 51.2 and 250 GB/sec, respectively, indicating that the baseline cases are performing well. Notably, for smaller problem sizes, the CPU overperforms peak bandwidth, since data fits in cache, and the GPU underperforms, insofar as the problem size is too small to hide latencies.
- The CPU self-offload case of accel/CPU/offload=CPU, unfortunately, does not perform well. Comparisons show that this case executes merely at single-thread performance. It is clear that the self-offload feature of the Cray compiler is not implemented for high performance but rather as more of a testing tool.
- The SM+t cases do not point to viable options for developing performant code.

VI. Discussion

During the development phase of this study, we ported the original application kernel to OpenMP 4.0 by using only `target`, `teams distribute`, and `simd` directives. As a next step, we added `target data map` directives to reduce the amount of data transferred to and from the accelerator device. With help from the `OFFLOAD_REPORT`

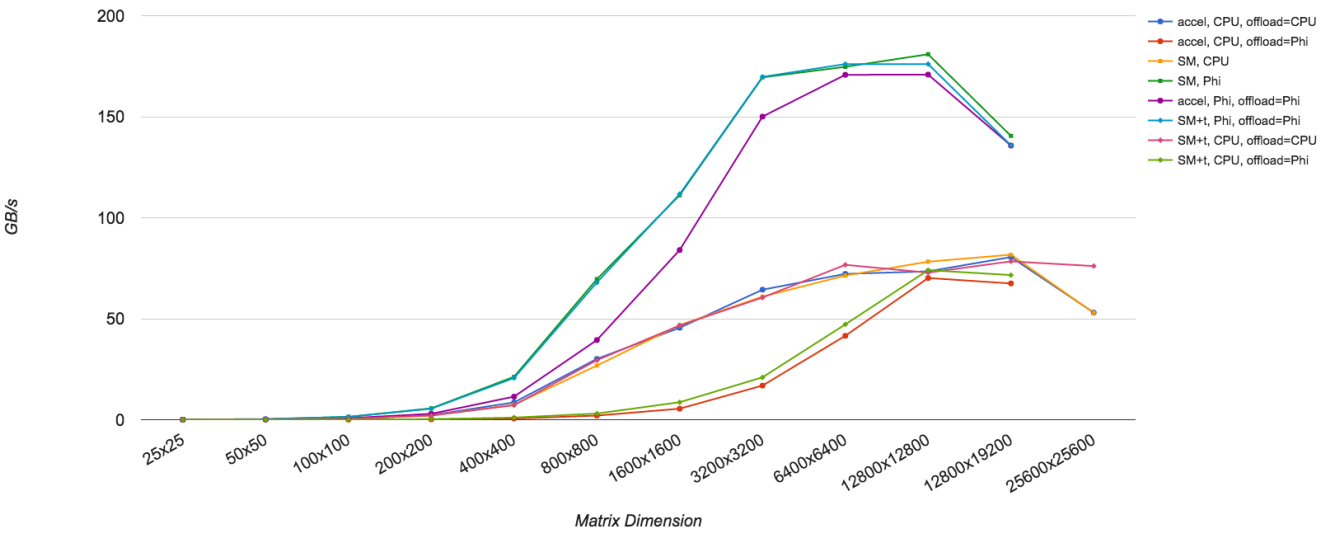

Figure 7. Beacon Results

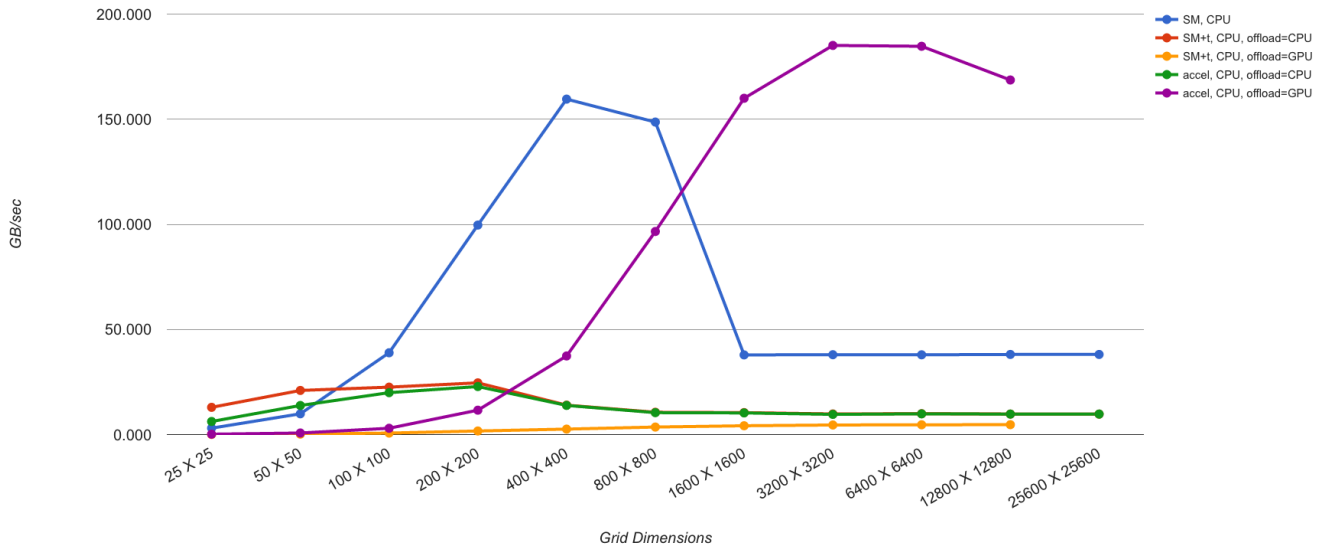

Figure 8. Chester results

reporting feature, we were able to determine that additional arrays were being transferred with every iteration of the Jacobi loop. This resulted in a significant impact in the performance of the accelerator version of the code, both on Beacon and Chester. To address this issue, we included the `target map(alloc:<var>)` directive to allocate the temporary array used to store previous results. This seemingly minor modification resulted in a drastic improvement in performance for large matrix sizes, as shown in Fig. 9.

In this study, we evaluated different versions of an application kernel in order to identify which programming model is best suited for multiple architectures without significantly sacrificing performance. Looking at Fig. 7, it is clear that, on Beacon, it is beneficial to execute the kernel natively on the Intel Xeon Phi. Furthermore, out of the three versions of Jacobi that achieve the best performance on Beacon, only *SM+t* and *accel* can take advantage of the accelerators on Chester. However, as Fig. 8 shows, *SM+t* on a GPU based system does not perform well. In addition, the results on Chester show the performance of the *accel* version of Jacobi increases for a wider range of matrix sizes. The performance of all other versions of the Jacobi code executed on Chester begins to tail off at a much smaller matrix size.

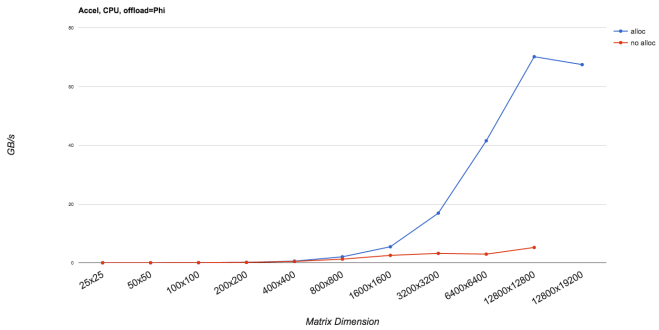

Figure 9. Performance difference between Jacobi accelerator version with and without `target map(alloc:<var>)` directives.

VII. Conclusion

In this paper we have attempted to evaluate the suitability of OpenMP for providing a single programming approach for code that is performance portable across all currently-targeted HPC node architectures, including CPU, CPU+accelerator, and native or self-hosted coprocessor. This programming model takes the form of OpenMP 4.0 directives used to target either the CPU or the accelerator as needed. This approach has the potential of allowing the user to specify the inherent parallelism in the code with a single syntax which can be processed by the respective compilers to adapt to the code to the specific hardware.

Based on experiments with a test kernel, we have shown that for the Intel Xeon Phi 5110P (Knights Corner), OpenMP 4.0 in "offload-to-self" mode performs nearly as well as native OpenMP 3.1, respectively, on both the CPU and the Phi coprocessor. The same code can also be used to effectively offload computation from CPU to the accelerator. Since newer generations of Intel Xeon Phi will be self-hosted, we anticipate that this programming model will be efficient for future Intel Xeon Phi generations.

For the Cray XK7 system, both OpenMP 3.1 for CPU and OpenMP 4.0 for offload from CPU to GPU performed effectively. However, OpenMP 4.0 "offload-to-self" for the CPU, though supported, did not perform well, yielding only single-thread performance. It would be beneficial to the community for Cray to make this option more performant in its compiler since this will improve the performance portability of applications across architectures; accomplishing this objective in the compiler should be feasible insofar as the viability of this technique has been shown on Beacon.

The results presented here indicate that for memory bound kernels, such as the test case used in this study, the *accel* version of the code provides the highest level of performance portability between the two architectures evaluated.

Besides performance considerations, it is important to take into account the effort needed to port a code to a given programming model. When the code is already using OpenMP 3.1, porting it to the accelerator model can be straightforward. However, as described above, because data transfers between the host and the device can easily become a performance bottleneck, a good understanding of the data mapping is crucial when using the accelerator programming model.

OpenMP 4.0 is a relatively new standard, and compilers are only recently starting to support the features of OpenMP 4.0 and beyond. It is hoped that as compiler support for these features becomes more mature, the programming model described herein will become an avenue for universal performance portability across all HPC platforms. The original focus of the OpenMP 4.0 target directives was to meet the needs of codes running on heterogeneous systems with connected accelerator devices. We believe that now it is vitally important for vendors and compiler developers to support the use of OpenMP `target` directives to access all available compute hardware (including both CPUs and accelerators) to allow flexible development of portable code. The present study shows that this should in principle be possible, and some vendors are already addressing this.

In the meantime, the primary focus of application developers is to expose parallelism in their codes, regardless of the specific syntax used. As a stop gap measure, conditional compilation may be needed to adapt parallelism to different APIs and programming models. It is in the best interest of the developer community for standards bodies and vendors to support universally performance portable solutions that allow users to harness the power of multiple emerging accelerated architectures.

VIII. Future Work

This study is, to our knowledge, one of the first to evaluate OpenMP 4.0 on both Intel Xeon Phi coprocessors and GPUs. The application kernel chosen here is representative of one of several types of workloads regularly executed on Titan. We plan to extend this work to include an in-depth analysis of additional application kernels, including compute-bound kernels and mini-applications. Additionally, we intend to expand this analysis to include features introduced in OpenMP 4.5, and eventually OpenMP 5.0, as well as other compilers as they begin supporting those features.

Our team has regular contact with code projects focused on early application readiness for upcoming systems such as OLCF's Summit system [2]. Increasingly, these teams seek to use OpenMP directives to program accelerators, as the compilers become more mature and usable for this purpose. The findings of this work suggest a way forward for these projects to make codes portable across accelerated as well as non-accelerated systems. We intend to work with these teams to explore this possibility and evaluate the effectiveness of this approach for more complex use cases.

Forthcoming pre-exascale systems will deliver increasing node complexity, such as complex NUMA domains and memory hierarchies. Application development teams require

that standards bodies, vendors and tool developers agree to standardize universally-applicable methodologies for exploiting the capabilities of these new hardware features.

In future work we intend to explore ways to use OpenMP 4.0 and later features to address exascale needs such as resilience (by use of task-based programming), efficient use of multiple resources on heterogeneous nodes and use of multiple accelerators per node.

ACKNOWLEDGMENT

This research used resources of the Oak Ridge Leadership Computing Facility at the 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.

This material is based upon work supported by the National Science Foundation under Grant Number 1137097 and by the University of Tennessee through the Beacon Project. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the University of Tennessee.

REFERENCES

[1] "CORAL fact sheet." [Online]. Available: http://www.anl.gov/sites/anl.gov/files/CORAL%20Fact%20Sheet.pdf

[2] "Summit: Scale new heights. discover new solutions." [Online]. Available: https://www.olcf.ornl.gov/summit/

[3] "Sierra advanced technology system." [Online]. Available: http://computation.llnl.gov/computers/sierra-advanced-technology-system

[4] "Aurora." [Online]. Available: http://aurora.alcf.anl.gov/

[5] T. Trader, "EU projects unite on heterogeneous ARM-based exascale prototype," http://www.hpcwire.com/2016/02/24/eu-projects-unite-exascale-prototype, 2016.

[6] "2015 workshop on portability among HPC architectures for scientific applications," in *The International Conference for High Performance Computing, Networking, Storage, and Analysis*, November 2015. [Online]. Available: http://hpcport.alcf.anl.gov/

[7] "DOE centers of excellence performance portability meeting," https://asc.llnl.gov/DOE-COE-Mtg-2016/, 2016.

[8] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257

[9] R. D. Hornung and J. A. Keasler, "The RAJA portability layer: Overview and status," https://e-reports-ext.llnl.gov/pdf/782261.pdf, 2014.

[10] CAPS, CRAY and NVIDIA, PGI, "The OpenACC application programming interface," http://openacc.org, 2013.

[11] J. Reid, "The new features of fortran 2008," *SIGPLAN Fortran Forum*, vol. 27, no. 2, pp. 8–21, Aug. 2008. [Online]. Available: http://doi.acm.org/10.1145/1408643.1408645

[12] J. Hoberock, "Working draft, technical specification for c++ extensions for parallelism," http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4071.htm, 2014.

[13] J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor High Performance Programming*. Burlington, MA: Morgan Kaufmann, 2013.

[14] T. D. Han and T. S. Abdelrahman, "hi CUDA: a high-level directive-based language for GPU programming," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2009, pp. 52–61.

[15] O. Hernandez, W. Ding, B. Chapman, C. Kartsaklis, R. Sankaran, and R. Graham, "Experiences with high-level programming directives for porting applications to GPUs," in *Facing the Multicore-Challenge II*. Springer, 2012, pp. 96–107.

[16] S. Lee and J. S. Vetter, "Early evaluation of directive-based GPU programming models for productive exascale computing," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 23.

[17] S. Lee and R. Eigenmann, "OpenMPC: extended OpenMP programming and tuning for GPUs," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. IEEE Computer Society, 2010, pp. 1–11.

[18] J. M. Levesque, R. Sankaran, and R. Grout, "Hybridizing S3D into an exascale application using OpenACC: an approach for moving to multi-petaflops and beyond," in *Proceedings of the International conference on high performance computing, networking, storage and analysis*. IEEE Computer Society Press, 2012, p. 15.

[19] M. Norman, J. Larkin, A. Vose, and K. Evans, "A case study of CUDA FORTRAN and OpenACC for an atmospheric climate kernel," *Journal of Computational Science*, vol. 9, pp. 1 – 6, 2015, computational Science at the Gates of Nature. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877750315000605

[20] B. Cumming, C. Osuna, T. Gysi, M. Bianco, X. Lapillonne, O. Fuhrer, and T. C. Schulthess, "A review of the challenges and results of refactoring the community climate code cosmo for hybrid cray hpc systems," *Proceedings of Cray User Group*, 2013.

[21] O. Fuhrer, C. Osuna, X. Lapillonne, T. Gysi, B. Cumming, M. Bianco, A. Arteaga, and T. Schulthess, "Towards a performance portable, architecture agnostic implementation strategy for weather and climate models," *Supercomputing frontiers and innovations*, vol. 1, no. 1, 2014.

[22] W. Sawyer, G. Zaengl, and L. Linardakis, "Towards a multi-node openacc implementation of the icon model," in *EGU General Assembly Conference Abstracts*, vol. 16, 2014, p. 15276.

[23] C. Liao, Y. Yan, B. R. de Supinski, D. J. Quinlan, and B. Chapman, "Early experiences with the OpenMP accelerator model," in *OpenMP in the Era of Low Power Devices and Accelerators*. Springer, 2013, pp. 84–98.

[24] R. G. Brook, A. Heinecke, A. B. Costa, P. Peltz Jr., V. C. Betro, T. Baer, M. Bader, , and P. Dubey, "Beacon: Deployment and application of Intel Xeon Phi coprocessors for scientific computing," *Computing in Science and Engineering*, vol. 17, no. 2, pp. 65–72, 2015.

[25] W. Joubert, R. K. Archibald, M. A. Berrill, W. M. Brown, M. Eisenbach, R. Grout, J. Larkin, J. Levesque, B. Messer, M. R. Norman, and et al., "Accelerated application development: The ORNL Titan experience," *Computers and Electrical Engineering*, vol. 46, May 2015.

[26] J. Robicheaux, "Program to solve a finite difference equation using Jacobi iterative method," http://www.openmp.org/samples/jacobi.f.