Evaluation of Directive-based Performance Portable Programming Models

M. Graham Lopez, Wayne Joubert, Verónica G. Vergara Larrea, Oscar Hernandez

Computational and Computer Sciences Directorate,
Oak Ridge National Laboratory,
Oak Ridge, TN, USA
E-mail: {lopezmg,joubert,vergaravg,oscar}@ornl.gov

Azzam Haidar, Stanimire Tomov, and Jack Dongarra

Innovative Computing Laboratory,
University of Tennessee,
Knoxville, TN, USA
E-mail: {haidar,tomov,dongarra}@icl.utk.edu

Abstract: We present an extended exploration of the performance portability of directives provided by OpenMP 4 and OpenACC to program various types of node architectures with attached accelerators, both self-hosted multicore and offload multicore/GPU. Our goal is to examine how successful OpenACC and the newer offload features of OpenMP 4.5 are for moving codes between architectures, and we document how much tuning might be required and what lessons we can learn from these experiences. To do this, we use examples of algorithms with varying computational intensities for our evaluation, as both compute and data access efficiency are important considerations for overall application performance. To better understand fundamental compute vs. bandwidth bound characteristics, we add the compute-bound Level 3 BLAS GEMM kernel to our linear algebra evaluation. We implement the kernels of interest using various methods provided by newer OpenACC and OpenMP implementations, and we evaluate their performance on various platforms including both x86_64 and Power8 with attached NVIDIA GPUs, X86_64 multicores, self-hosted Intel Xeon Phi KNL, as well as an X86_64 host system with Intel Xeon Phi coprocessors. We update these evaluations with the newest version of the NVIDIA Pascal architecture (P100), Intel KNL 7230, Power8+, and the newest supporting compiler implementations. Furthermore, we present in detail what factors affected the performance portability, including how to pick the right programming model, its programming style, its availability on different platforms, and how well compilers can optimize and target multiple platforms.

Keywords: OpenMP 4, OpenACC, Performance Portability, Programming Models.

Biographical notes:

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

This paper is authored by an employee(s) of the United States Government and is in the public domain. Non-exclusive copying or redistribution is allowed, provided that the article citation is given and the authors and agency are clearly identified as its source.

1 Introduction and Background

Performance portability has been identified by the U.S. Department of Energy (DOE) as a priority design constraint for pre-exascale systems such as those in the current CORAL project as well as upcoming exascale systems in the next decade. This prioritization has been emphasized in several recent meetings and workshops such as 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 [1], and the DOE Centers of Excellence Performance Portability Meeting [2].

Copyright © 2017 Inderscience Enterprises Ltd.

There are two main node-architecture types being considered in the road to exascale as part of the DOE CORAL project architectures [3]: one with heterogeneous accelerators represented by IBM Power based systems with multiple NVIDIA Volta GPUs per node [4, 5]; and the other with homogeneous third-generation Intel Xeon Phi based nodes [6]. With both of these hardware "swimlanes" for applications to target, writing performance portable code that makes efficient use of all available compute resources in both shared and heterogeneous memory spaces is at present a non-trivial task. The latest OpenMP 4.5 specification defines directives-based programming models that can target both traditional shared memory execution and accelerators using new offload capabilities. However, with growing support from compilers, the degree to which these models are successful is not yet clear, particularly in the context of the different node-architectures enumerated above. While shared memory programming has been available in, and the main focus of, the industry-standard OpenMP specification for more than a decade, the recent 4.0 and 4.5 versions have introduced support for offloading to heterogeneous accelerators. While the shared memory model can support some types of self-hosted accelerators, the offload model has been introduced to further support heterogeneous accelerators with discrete memory address spaces.

This study is a continuation of our efforts to understand if there is a single programming model (and which programming style) that can be used to program host multicore, homogeneous, and heterogeneous accelerators, and what the potential performance or productivity trade-offs might be. Here, we extend some previous work [7, 8] by porting algorithms of various computational intensities, ranging from BLAS level 3 GEMM to bandwidth-bound Jacobi and BLAS level 1 AXPY, to each of the shared memory and offload style of OpenMP as well as OpenACC with both host and accelerator targeting. We use various compilers on both homogeneous and heterogeneous hardware platforms and compare the performance of the directives variants to platform-optimized versions of the algorithms where available and as provided in 3rd-party libraries, and use these as a "baseline" for the best-case performance. Otherwise, we compare to machine theoretical peak FLOPS (for compute-bound kernels) or bandwidth (for memory-bound kernels). We also discuss our experiences of using OpenMP 4.5 and OpenACC and the issues that we identified which can affect performance portability. We summarize these experiences to reflect the current "state of the art" for achieving performance portability using directives.

2 Related Work

Perhaps the most portable option for developers is to use standardized language features such as co-arrays and 'do concurrent,' present in the Fortran standard [9] since 2008. Recently, new parallelization features have been proposed [10] for inclusion in the C++17 standard. However, due to the slow-moving nature of standardization and implementations, these features presently remain inadequate for accelerator programming with heterogeneous compute capabilities and memory hierarchies.

Two new efforts that have gained notoriety for performance portability are Kokkos [11] and RAJA [12]. Both rely on C++ language features that allow the application developer to target multiple architectures with a single implementation. However, these solutions are not available to C or Fortran applications unless foreign function interfaces are used, undoing some of the convenience that these projects try to provide.

Another programming model specifically designed for performance portability across architectures is OpenCL. Previous work has evaluated OpenCL in the context of performance portability [13, 14, 15]. However, OpenCL is a lower-level model than directives, requiring explicit representation of the computational kernels in a style similar to CUDA. While using programming models like OpenCL can benefit performance, some application developers find it difficult to maintain or optimize the code for multiple architectures, specially since some of its optimizations are not portable.

Directives-based programming has been supported [16] on the Intel Xeon Phi accelerator platform even before OpenMP 4.0. This model supported both "native" and "offload" modes which, respectively, run code locally on the device or send isolated kernels for execution in a manner similar to GPUs. Additional directives-based models have included PGI compiler accelerator directives, CAPS HMPP, OpenMPC [17], and hiCUDA [18]. Previous studies [19] of the directives-based approach for GPU programming showed that, with additional code transformations, performance comparable to that of hand-tuned CUDA can be achieved in some cases [20].

Other Performance Portability Studies using Accelerator Directives. As interest increases in performance portable programming models for accelerators, there has been an increase in published studies of the same. Some recent examples include a thorough head-to-head comparison of the OpenACC and OpenMP programming styles [21], but no direct performance evaluations were provided. Another examination of both OpenMP and OpenACC was undertaken when the SPEC ACCEL [22] benchmarks suite was ported [23] from OpenACC to OpenMP. However, at this stage, only the Intel Xeon Phi architecture was targeted by the OpenMP offload directives, so there was no sense of performance portability across multiple architectures provided. A more comprehensive effort has been provided by Martineau et al. [24], where the TeaLeaf mini-app was ported to many programming models and covered host CPU, NVIDIA GPU, and Intel Xeon

Phi architectures. This was a broad effort to examine programming models and architectures, but it only used a single application example and did not focus on the suitability of directives specifically for performance portability.

3 Directive-based Programming Models for Accelerators

3.1 OpenMP and OpenACC

OpenMP is the de-facto standard API for shared memory programming with widespread vendor support and a large user base. It provides a set of directives to manage, synchronize, and assign work to threads that share data. Recently, with the adoption of OpenMP 4.0 and 4.5, the OpenMP shared memory programming model was extended to support accelerators, and this substantially changed the programming model from previous versions of the API. The OpenMP "fork-join" model was extended with the introduction of device constructs for programming accelerators. These constructs allow compute-intensive code regions to be offloaded to accelerator devices where new OpenMP environments can be instantiated. For this, OpenMP 4.5 uses the `target` construct to create a data environment on the device and then execute the code region on that device. OpenMP 4.5 provides `target data` directives that can map data to/from the accelerator and update that data on both the host and accelerator within the `target data` regions. In addition, OpenMP 4.0 added the `teams, distribute` and `SIMD` directives that can be used to describe different types of parallelism.

OpenACC is another specification focused on directive-based ways to program accelerators. The OpenACC programming model is similar to OpenMP 4.5, but its directives focus on accelerator programming and are more "descriptive" than "prescriptive" in nature. The idea is that it is best for the user to describe the parallelism and data motion in a more general way via directives so that the OpenACC compiler can have more freedom to map the parallelism to the hardware. The goal of this approach is to be able to map the parallelism of an application to targets with different architectural characteristics using the same source code in a performance portable style. The user can also provide additional clauses (or hints) to the compiler to further specify and improve this mapping and code generation.

The OpenACC `acc loop` directive can distribute the loop iterations across gangs, workers, or vectors. It is also possible to use the `acc loop` directive to distribute the work to gangs while still in worker-single and vector-single mode. For OpenACC (and OpenMP), it is possible in some cases to apply loop directives like `collapse` to multiple nested loops to flatten the loop iteration space that we want to parallelize. We can also specify a `workers` or `vectors` clause to distribute the iterations across `workers` or `vectors`. If we only specify `acc loop`, the compiler has the option to decide how to map the iteration space across gang, workers, or vectors. This is an important feature of OpenACC because it gives the compiler the freedom to pick how to map the loop iterations to different loop schedules that take advantage of the target accelerator architecture.

3.2 OpenACC and OpenMP 4.5 differences

There are still significant stylistic differences between these two specifications, but they are converging in terms of features and functionality. One of the most significant differences is their philosophy: the "descriptive" philosophy of OpenACC vs. the "prescriptive" approach of OpenMP 4.5 that may impact the way code is written and the performance portability of the resulting parallelism. For example, OpenMP 4.5 has no equivalent for the `acc loop` directive in OpenACC. In OpenACC, the developer can specify that a loop is parallel and the compiler will determine how to distribute the loop iterations across gangs, workers, or vectors. In OpenMP 4.5, the programmer has to specify that a loop is parallel but also how the work in the loop should be distributed. This also applies to any loop in a loop nest that is marked with `acc loop parallel`. The only way to accomplish this in OpenMP is to use the combined directive `omp teams distribute parallel for simd` with a `collapse` clause, which collapses the iteration space across perfectly nested loops. The final schedule used is implementation defined. For example, Intel compilers that target SIMD instructions will pick one team and several parallel threads with SIMD instructions. Compilers that target GPUs will pick parallelizing over teams and parallel threads or SIMD regions (executed by threads). This works well for perfectly nested parallel loops, however, it does not work when the loop nests are imperfect or have function calls. At the time of this writing, it seems likely that future OpenMP specifications will provide a directive equivalent to `acc loop`.

3.3 Writing OpenMP 4.5 using a Performance Portable style

To write OpenMP in a "performance portable" style we need to exploit certain behaviors of the OpenMP 4.5 accelerator model execution that are implementation-defined, and as such, are left for the compiler to optimize the code for specific architectures. This is described in [25]. For example, when a `teams` construct is executed, a league of threads is created, where the total number of teams is implementation-defined but must be less than or equal to the number of teams specified by the `num_teams` clause. If the user does not specify the `num_teams` clause, then the number of teams is left completely to the implementation.

Similarly, the maximum number of threads created per team is implementation defined. The user has the

option to specify a `thread_limit` clause that gives an upper bound to the implementation defined value for the number of threads per team. The purpose of this implementation defined behavior is to allow the compiler or runtime to pick the best value for a given target region on a given architecture. If a parallel region is nested within a `teams` construct, the number of threads in a parallel region will be determined based on Algorithm 2.1 of the OpenMP 4.5 specification [26]. A user can request a given number of threads for a parallel region via the `num_threads` clause.

For work-sharing constructs such as `distribute` and `parallel for/do`, if no `dist_schedule` or `schedule` clauses are specified, the schedule type is implementation defined. For a `SIMD` loop, the number of iterations executed concurrently at any given time is implementation defined, as well. The preferred number of iterations to be executed concurrently for `SIMD` can be specified via the `simdlen` and `safelen` clauses, respectively.

An example of writing OpenMP in "performance portable" style can be seen when using the Intel 16.2 compiler, which sets the default value for `num_teams` to one and attempts to use all the number of threads available on the host. When using an Intel Xeon Phi as an accelerator in offload mode, the Intel compiler reserves one core on the co-processor to manage the offloading and uses all the remaining threads available on the Intel Xeon Phi (Knights Corner) for execution. On the other hand, the Cray 8.4.2 compiler, by default, uses one team and one thread when running on the host. When running on the GPU, however, if there is a nested parallel region within a team, it defaults to one thread per parallel region. Writing OpenMP in a "performance portable" style might require the user to force the compiler to use a specific number of teams (e.g. `num_teams(1)`).

Another example of an implementation-dependent behavior can be observed in the LLVM compiler, which defaults to `schedule(static,1)` for the parallel loops when executed inside a target region that is offloaded to a GPU. The OpenMP 4.5 Cray compiler implementation picks one thread to execute all parallel regions within a `target teams` region (the equivalent of `num_threads(1)`). Due to the slightly different interpretations of the OpenMP specification, it is crucial to understand how the specific compiler being used implements a particular feature on different platforms, and more studies are needed to understand this.

3.4 Representative Kernels

In order to study the performance portability of accelerator directives provided by OpenMP 4 and OpenACC, we chose kernels that can be found in HPC applications, and we classified them loosely based on their computational intensity.

Dense Linear Algebra (DLA): DLA is well represented on most architectures in highly optimized libraries based on BLAS and LAPACK. As benchmark cases, we consider the daxpy vector operation, the dgemv dense matrix-vector product operation and the dgemm dense matrix-matrix product operation. For our tests we compare our OpenMP 4 and OpenACC implementations against Intel's MKL implementation on the Xeon host CPU and Xeon Phi platforms, and we compare against CUBLAS for the GPU-accelerated implementation on Titan.

Jacobi: Similarly to previous work [8] studying various OpenMP 4 offload methods, we include here data for a Jacobi iterative solver for a discretized constant-coefficient partial differential equation. The 2-D case represented here is a well-understood kernel for structured grid and sparse linear algebra computational motifs. Its behavior is similar to that of many application codes, and the Jacobi kernel itself is used, for example, as part of implicit grid solvers and structured multigrid smoothers.

HACCmk: The Hardware Accelerated Cosmology Code (HACC) is a framework that uses N-body techniques to simulate fluids during the evolution of the early universe. The HACCmk [27] microkernel is derived from the HACC application and is part of the CORAL benchmark suite. It consists of a short-force evaluation routine which uses an $O(n^2)$ algorithm using mostly single-precision floating point operations.

3.4.1 Parallelization of DLA kernels

Here we study two linear algebra routines that are representative of many techniques used in real scientific applications such as Jacobi iteration, Gauss-Seidel methods, Newton-Raphson, among others. We present an analysis of the daxpy, dgemv and the dgemm routines. The daxpy and dgemv kernels are well-understood by compilers, specified in the BLAS standard, and implemented in all BLAS libraries. daxpy takes the form of $y \leftarrow y + \alpha x$ for vectors x and y and scalar α. dgemv takes the form $y \leftarrow \beta y + \alpha A x$ or alternatively $y \leftarrow \beta y + \alpha A^T x$ for matrix A, vectors x and y, and scalars α and β. These are referred to as the non-transpose ("N") and transpose ("T") forms, respectively. The two routines are *memory bound* and their computational patterns are representative of a wide range of numerical methods. The dgemm is a compute-bound kernel that is hard to implement for a subsequently adequate optimization by a compiler. We note that good implementation requires a complex design of its loops in order to increase data reuse, vectorization, and contiguous data accesses that minimize cache misses. Moreover, in order to reach a high fraction of the hardware peak, the dgemm routine often must be written in assembly or intrinsics. It takes the form $C \leftarrow \beta C + \alpha AB$ where A, B and C are matrices, α and β are scalars.

The main differences between them are:

- daxpy is a single loop operating on two vectors of contiguous data that should be easily parallelizable by the compiler; daxpy is the more memory

```
double alpha, beta, *A, *B, *C;
int m,n,k;
for( int i = 0; i < m; ++i) {
    for( int j = 0; j < n; ++j) {
        double sum = 0.0;
        for( int l = 0; l < k; ++l) {
            sum += A(i,l) * B(l,j);
        }
        C(i,j) = beta * C(i,j) + alpha * sum;
    }
}
```

Listing 1: ijk loop implementation of dgemm routine

```
double alpha, *x, *y;
int n;
#pragma omp target data map(to:x[0:n]) &
  map(tofrom:y[0:n])
{
  int i;
  #pragma omp target teams
  {
    #pragma omp distribute parallel for
    for (i=0; i<n; i++)
      y[i] += alpha * x[i];
  }  // teams
} // data
```

Listing 2: OpenMP 4 version of daxpy; OpenACC similar

bandwidth-bound operation (than dgemv) with unit stride data access; daxpy on vectors of dimension n requires $2n$ element reads from memory and n element writes;

- dgemv is two nested loops that can be parallelized row- or column-wise, resulting in data accesses that are contiguous or not, and where reductions are required or not (depending on the transpose option as well).

- dgemm reads three matrices and writes one. Thus, for square matrices of size n, it reads/writes $4n^2$ data, while performing $2n^3$ operations. Listing 1 shows its basic simple ijk-loop implementation. The ijk-loop implementation is not cache-aware and is hard to be vectorized and parallelized by a compiler.

Listing 2 shows the code used for the daxpy operation with OpenMP 4 directives; the OpenACC version used similarly straightforward directives based on the equivalent OpenACC syntax. The code consists of a data region specifying arrays to transfer to and from the accelerator and a parallel region directing execution of the DAXPY loop to the device. For the OpenMP 4.5 version we did not specify an OpenMP SIMD directive in the inner loop since this vectorization pattern was recognized by all the tested compilers (Intel, PGI, and Cray).

Listing 3 shows the code for the dgemv operation, N case. The code has a similar structure including a data region but also several loops including a doubly nested loop and if statement. Additionally, the non-stride-1 access poses a challenge for compilation to efficient code.

```
double alpha, beta, *x, *y, *A;
int m, n;
#pragma omp target data map(to:A[0:m*n]) &
  map(to:x[0:n]) map(tofrom:y[0:m]) &
  map(alloc:tmp[0:m])
{
  int i, j;
  double prod, xval;
  #pragma omp target teams
  {
    #pragma omp distribute parallel for &
      private(prod, xval, j)
    for (i=0; i<m; i++) {
      prod = 0.0;
      for (j=0; j<n; j++)
        prod += A[i+m*j]*x[j];
      tmp[i] = alpha * prod;
    }
    if (beta == 0.0) {
      #pragma omp distribute parallel for
      for (i=0; i<m; i++)
        y[i] = tmp[i];
    } else {
      #pragma omp distribute parallel for
      for (i=0; i<m; i++)
        y[i] = beta * y[i] + tmp[i];
    } // if
  } // teams
} // data
```

Listing 3: OpenMP4 version of dgemv/N

In listing 4, we show the OpenACC equivalent code to allow a direct comparison for this accelerator-portable use case.

```
double alpha, beta, *x, *y, *A;
int m, n;
#pragma acc data pcopyin(A[0:m*n]) &
  pcopyin(x[0:n]) pcopy(y[0:m]) &
  create(tmp[0:m])
{
  int i, j;
  double prod, xval;
  {
    #pragma acc parallel loop gang &
      private(prod, xval, j)
    for (i=0; i<m; i++) {
      prod = 0.0;
      for (j=0; j<n; j++)
        prod += A[i+m*j]*x[j];
      tmp[i] = alpha * prod;
    }
    if (beta == 0.0) {
      #pragma acc parallel loop gang
      for (i=0; i<m; i++)
        y[i] = tmp[i];
    } else {
      #pragma acc parallel loop gang
      for (i=0; i<m; i++)
        y[i] = beta * y[i] + tmp[i];
    } // if
  } // teams
} // data
```

Listing 4: OpenACC version of dgemv/N

The ijk-loop implementation of the dgemm kernel, as mentioned above, is not cache-aware and cannot be vectorized. This can be observed for example by examining performance counters, available through libraries like PAPI, with which we conducted an extensive study. Our analysis concludes that in order to achieve an efficient execution for such computation, we need to maximize the CPU occupancy and minimize the data traffic while respecting the underlying hierarchical memory design. Unfortunately, today's compilers cannot introduce the needed highly sophisticated cache/register

based loop transformations and, consequently, this kind of optimization should be studied and implemented by the developer. This includes techniques like reordering the loop or the data so that it can be easily vectorized, prefetching the data that will be reused in registers, unrolling most inner kernel or maybe writing it explicitly in assembly or intrinsic, and using an optimal blocking strategy. However, the goal of this paper is not to provide the most optimized `dgemm` routine, but rather provide an implementation that introduces some of the techniques mentioned above while preserving an acceptable level of complexity. For that, we first developed different loop reordering versions and show the most representatives of them (jik, kji). The parallelization is challenging, in particular for the kji and kij loops, since the data over the k direction must be summed at the end. Thus, careful attention is required for the parallelization. The ijk loop implementation is not cache-aware and cannot be vectorized. The jik can be vectorized over the i direction and might have better locality. However, it becomes quickly cache unaware when the matrix sizes exceed the cache size since it loads the whole row of A and the whole column of B to compute one element of C. The kji implementation can be vectorized over the i direction, but similarly to ijk is cache unaware and requires to load and store the whole matrix C for every step k.

As will be observed during the experimental section, the loop reordering by itself can speedup the computation when the data fits into the cache size, or L2 cache size, but will fail above this sizes. The remedy to this issue is to use double-layer of blocking to increase cache reuse. Here we start discussing a more sophisticated implementation. To simplify the description, we illustrate in Listing 5 and Figure 1 a snapshot of the blocked kernel design. It consists of 3 nested loops that go over the sizes M, N, K by block, and then at the innermost loop, calls the basic simple kernel that itself consists of 3 nested loops. The goal here is to increase the cache reuse. The block size will be tuned to make the blocks of A, B, C fit into the L2 cache at least. We observe in the results section that this implementation is able to achieve acceptable performance.

Our parallelization strategy consisted of giving the minimal information to the compilers via OpenMP 4.5 and OpenACC to give them the freedom to generate good optimized code to target multiple architectures.

3.4.2 Parallelization of Jacobi

The Jacobi kernel utilized for this study was derived from the OpenMP Jacobi example available at [28]. The original code, written in Fortran, is parallelized using OpenMP's shared programming model (OpenMP 3.1). As described in [8], in order to compare different programming models the Jacobi kernel was first transformed to OpenMP's accelerator programming model (OpenMP 4), and then it was compared to the shared programming model when offloaded to the GPU via the `omp target` directive. In this work, the Jacobi kernel was also ported to OpenACC.

```
double alpha, beta, *A, *B, *C;
int M,N,K,i,j,k;
for (int k = 0; k < K; k += BLK_K)
{
    for (int i = 0; i < m; i += BLK_M)
    {
        #pragma omp parallel for schedule(dynamic)
        for (int j = 0; j < n; j += BLK_N)
        {
            int bm = min (M_BLOCK_SIZE, M-i);
            int bn = min (N_BLOCK_SIZE, N-j);
            int bk = min (K_BLOCK_SIZE, K-k);
            dgemm_simple_kernel(bm,bn,bk,...);
        }
    }
}
```

Listing 5: double layer cache blocking implementation of `dgemm` routine

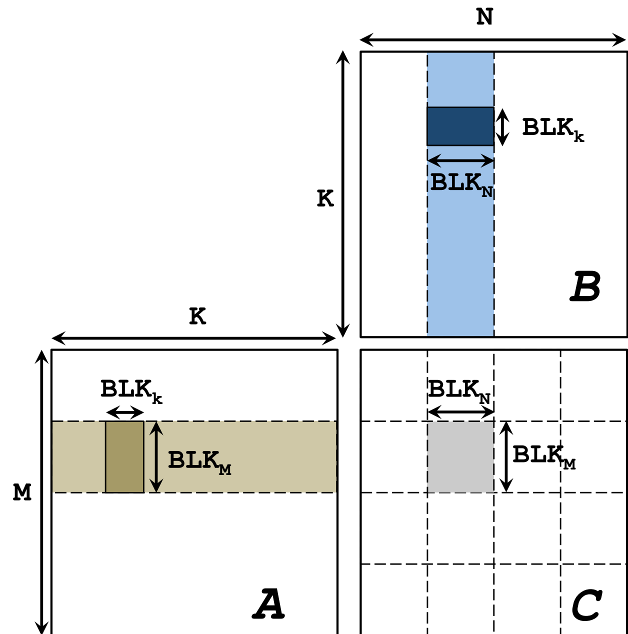

Figure 1: The design of the double layer blocked `dgemm` routine.

To port the shared OpenMP 3.1 code to OpenACC, we added a `parallel loop` construct to each of the two main loops inside the Jacobi subroutine. In addition, we added a `data` construct outside the main `do while` loop to specify which arrays and variables should be copied to the device (`copyin`), which need to be copied back to the host via (`copyout`) and allocated on the device via (`create`). Finally, since the solution computed in the device is needed for each iteration, an `update` clause was added to the `do while` control loop. We did not specify a loop schedule or if the the `acc loop` was `gang`, `worker`, or `vector`, to let the compiler pick the strategy for optimization and performance portability.

Additional optimizations were explored to improve performance. Given that the two main `do` loops

are doubly nested loops, we added a `collapse(2)` to the `parallel loop` directive. We also tested the performance impact of using the `-ta=tesla:pinned` option at compile time to allocate data in CUDA pinned memory, as well as the `-ta=multicore` option to run OpenACC on the CPU host.

3.4.3 Parallelization of HACCmk

The HACCmk kernel [27] as found in the CORAL benchmark suite has shared memory OpenMP 3.1 implemented for CPU multicore parallelization. There is one `for` loop over particles, parallelized with `omp parallel for`, which contains a function call to the bulk of the computational kernel. This function contains another `for` loop over particles to make overall two nested loops over the number of particles and the $O(n^2)$ algorithm as described by the benchmark. A good optimizing compiler should be able to automatically vectorize all of the code within the inner function call to achieve good shared memory performance.

We have observed that the Cray 8.5.0 and Intel 16.0.0 compiler, for example, can successfully vectorize all the statements of the inner procedure. This is the first instance where the amount of parallelization obtained will critically depend on the quality of the compiler implementation.

In order to transform this code to the OpenMP accelerator offload model, we created a target region around the original OpenMP 3.1 parallel region. Since this region contains two main levels of parallelism, we decided to parallelize the outer level across teams and OpenMP threads within the teams using the `distribute parallel for` construct, which allows the compiler to choose the distribution of iterations to two dimensions of threads. In this case, the Cray compiler automatically picked one thread for the `parallel for` as an implementation defined behavior when targeting a GPU. The Intel compiler, in comparison, picked one team when targeting self-hosted Xeon Phi processors. We relied on the `firstprivate` default for scalars in the target region and the `map(tofom:*)` default map for the rest of the variables, except for xx, yy and zz arrays, which are needed only in the accelerator.

We added an `omp declare target` construct to the Step10 subroutine which is called from within the outer loop. For the inner level of parallelism, we explicitly added an `omp simd` construct with a `reduction` clause on the variables `xi`, `yi` and `zi` inside the Step10 subroutine to provide an extra hint to the compiler to vectorize the inner loop. We did this in order to ensure maximum vectorization since most of the performance of this kernel depends on vectorization for multicore architectures.

For the OpenACC version of the HACCmk microkernel, we parallelized the outer loop level with the `acc parallel loop` which calls the subroutine Step10. However, the Cray 8.5.0 OpenACC compiler would not allow a `reduction` clause on the `acc loop vector`

```
[label=code_hacc, basicstyle=\scriptsize]
#pragma omp declare target
void
Step10(int count1, float xxi, float yyi,
       float zzi, float fsrrmax2, float mp_rsm2,
       float *xx1, float *yy1, float *zz1,
       float *mass1, float *dxi, float *dyi,
       float *dzi );
#pragma omp end declare target
int main() {
...
 #pragma omp target teams private(dx1, dy1, dz1)&
 #pragma omp map(to: xx[0:n],yy[0:n],zz[0:n])
 #pragma omp distribute parallel for
  for (i = 0; i < count; ++i) {
    Step10(n, xx[i], yy[i], zz[i], fsrrmax2,
           mp_rsm2, xx, yy, zz, mass, &dx1,
           &dy1, &dz1 );

    vx1[i] = vx1[i] + dx1 * fcoeff;
    vy1[i] = vy1[i] + dy1 * fcoeff;
    vz1[i] = vz1[i] + dz1 * fcoeff;
  }
...
}
void
Step10(...) {
...
#pragma omp simd private(dxc, dyc, dzc, r2, m, f) &
#pragma omp reduction(+:xi,yi,zi)
  for (j = 0; j < count1; j++) {
    dxc = xx1[j] - xxi;
    dyc = yy1[j] - yyi;
    dzc = zz1[j] - zzi;
    r2 = dxc * dxc + dyc * dyc + dzc * dzc;
    m = (r2 < fsrrmax2) ? mass1[j] : 0.0f;
    f =  powf(r2 + mp_rsm2, -1.5 )
         - (ma0 + r2*(ma1 + r2*(ma2
           + r2*(ma3 + r2*(ma4 + r2*ma5)))));
    f = (r2 > 0.0f) ? m * f : 0.0f;
    xi = xi + f * dxc;
    yi = yi + f * dyc;
    zi = zi + f * dzc;
  }
  ...
}
```

Listing 6: OpenMP 4.5 version of HACCmk

construct within an `acc routine gang` region. This required us to manually inline the entire subroutine. This is an OpenACC implementation bug as OpenACC 2.5 allows this. The PGI 16.5 compiler was able to apply the reduction correctly. We left inlined the routine to be able to experiment with both compilers (PGI and Cray) and have a single version of the code. The inner loop was marked with `acc loop` with a private and reduction clause. For the OpenACC version we did not specify any loop schedule in the `acc loop` to allow the compiler pick the best schedule for the target architecture (e.g., in this case the GPU or multicore). We did this to both test the quality of the optimization of the OpenACC compiler and to measure how performance portable OpenACC is across architectures. In listings 6 and 7 below we show the HACCmk kernel implemented using OpenMP 4.5 and OpenACC 2.5.

4 Results

In this section, we present results obtained from porting the previously described kernels to directives-based programming models and then examine some

```
...
#pragma acc parallel private(dx1,dy1,dz1) &
#pragma acc copy(vx1,vy1,vz1) &
#pragma acc copyin(xx[0:n],yy[0:n],zz[0:n])

#pragma acc loop
  for ( i = 0; i < count; ++i) {
        const float ma0 = 0.269327, ma1 = -0.0750978,
        ma2 = 0.0114808, ma3 = -0.00109313,
        ma4 = 0.0000605491, ma5 = -0.00000147177;
        float dxc, dyc, dzc, m, r2, f, xi, yi, zi;
        int j;
        xi = 0.; yi = 0.; zi = 0.;
#pragma acc loop  private(dxc, dyc, dzc, r2, m, f)&
#pragma acc reduction(+:xi,yi,zi)
     for ( j = 0; j < n; j++ )    {
       dxc = xx[j] - xx[i];
       dyc = yy[j] - yy[i];
       dzc = zz[j] - zz[i];

       r2 = dxc * dxc + dyc * dyc + dzc * dzc;
      m = ( r2 < fsrrmax2 ) ? mass[j] : 0.0f;
       f =  powf( r2 + mp_rsm2, -1.5 ) -
       ( ma0 + r2*(ma1 + r2*(ma2 + r2*(ma3
      + r2*(ma4+ r2*ma5)))));
       f = ( r2 > 0.0f ) ? m * f : 0.0f;
        xi = xi + f * dxc;
         yi = yi + f * dyc;
         zi = zi + f * dzc;
        }
        dx1 = xi;
        dy1 = yi;
        dz1 = zi;
  }
      vx1[i] = vx1[i] + dx1 * fcoeff;
      vy1[i] = vy1[i] + dy1 * fcoeff;
      vz1[i] = vz1[i] + dz1 * fcoeff;
}
..
```

Listing 7: OpenACC 2.5 version of HACCmk

issues affecting their performance portability. For the evaluations conducted in this paper, we used the following systems.

The OLCF Titan [29] Cray XK7 contains AMD Interlagos host CPUs connected to NVIDIA K20X GPUs. For the OLCF Titan system, 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 single/double precision flop rate of 3,935/1,311 GF and a peak memory bandwidth of 250 GB/sec. For this platform, Cray compilers are used, with versions 8.4.5, 8.5.0, and 8.5.5. See [8].

The OLCF Summitdev IBM system [30] contains IBM Power8+ host CPUs connected to NVIDIA P100 GPUs. Compute nodes on Summitdev have two IBM Power8+ 10-core processors with a peak flop rate of approximately 560 GF and a peak memory bandwidth of 340 GB/sec, and four NVIDIA Tesla P100 GPUs with a peak single/double precision flop rate of 10.6/5.3 TF and a peak memory bandwidth of 732 GB/sec. For the new experiments on Summitdev, the IBM XL C/C++ 13.1.5, IBM XL Fortran 15.1.5, and PGI 17.1 compilers were used.

Experiments were also conducted on the OLCF Percival [31] Cray XC40 Knights Landing (KNL) system. Each compute node on Percival contains a 64-core Intel Xeon Phi 7230 processor which has a peak flop rate of 2.66 TF and a peak memory bandwidth of 115.2 GB/sec. On this system, the Intel 17.0.0 20160721, the GCC 6.2.0, and the PGI 17.1 compilers were used.

The NICS Beacon Intel Phi Knights Corner system [32] compute nodes contain 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, which translates to combined 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 the results presented here, Intel compilers were used on this system, with version 16.0.1 from the Intel XE Compiler suite version 2016.1.056 and version 16.0.3 20160415.

The Intel Xeon Phi KNC 7210 processor used for the DLA evaluations is composed of 64 processors running at 1.3 GHz (1.5 GHz max turbo frequency) with a maximum memory bandwidth of 102.4 GB/sec; the KNL 7250 uses 68 processors running at 1.4 GHz (1.6 GHz max turbo frequency) and a maximum memory bandwidth of 115.2 GB/sec. The Intel Xeon Haswell processor E5-2650 v3 is composed of 10 cores running at 2.3 GHz (3 GHz max turbo frequency) with a maximum memory bandwidth of 68 GB/sec.

4.1 Dense Linear Algebra

We ran an extensive set of experiments to illustrate our findings. Figures 2-11 illustrate the performance results in double precision (DP) arithmetic for the daxpy, the dgemv "N" and the the dgemv "T" kernels, respectively, for the four types of hardware and in both offload and self-hosted configurations. We use the same code to show its portability, sustainability, and ability to provide close to peak performance when used in self-hosted model, on a KNC 7120, KNL 7250, CPU and when using the offload model on the KNC 7120 and a K20X GPU. We also present the OpenACC implementation using either the Cray or the PGI compiler.

The basic approach to performance comparisons for the DLA kernels is to compare performance for a range of problem sizes, using custom kernels written with OpenMP or OpenACC directives and, when appropriate, comparisons with system libraries (MKL for Intel processors and cuBLAS for GPUs). The figures show the portability of our code across a wide range of heterogeneous architectures. In addition to the portability, note that the results confirm the following observations. Our implementation achieves good scalability, is competitive with the vendor optimized libraries, and runs close to the peak performance. In order to evaluate the performance of an implementation, we rate its performance compared to what we refer to as practical peak which is the peak performance that can be achieved if we consider the computation time is zero. For example, the daxpy routine reads the two vectors x and y and then writes back y. Overall, it reads and writes $3n$ elements (that in

DP equals to $24n$ bytes) and performs $2n$ operations. Therefore, if we consider that the computation time is near zero and the operation is entirely memory bandwidth bound, then the time to perform the daxpy operation will be the time to R/W the $24n$ bytes which is $B/(24n)$ seconds, where B denotes the achievable peak bandwidth measured in Bytes/s. Thus, the peak performance is $P = flops/time = B/12$. To show this, we plot the practical peak performance that each of these routines can achieve based on the achievable bandwidth B. The value of B is noted in the graphs. The roofline bandwidth limit is shown for each case, and the performance is reported in Gflop/s.

Our goal is to perform as many DLA operations as possible on the discrete accelerator between data transfers with the host. For this reason, we present timings for the kernel only, without the transfers. Experiments show that the first kernel, from a sequence of kernel calls, may be slower and thus unrepresentative for the rest; therefore, we perform a "warmup" kernel call after the transfer and prior to the actual "timing call," to eliminate this effect. Also, to ensure the kernel has completed before the final timer call, a single-word update to host is performed to cause a wait for the possibly asynchronous kernel to complete.

4.1.1 daxpy

For the daxpy test case, our implementation demonstrates performance portability across all tested hardware. The code of this routine is simple: it operates on vectors (i.e., contiguous data) and thus the compiler is able to perform an excellent job optimizing it for the target hardware. We also compared the self-hosted model on KNC with the offload model and, as illustrated in Figure 4, both implementations reach the same performance. Similarly, the same code performs very well on the Nvidia K20X GPU (Figure 2), Nvidia P100 GPU (Figure 3), a recent Xeon CPU, and the newer Xeon Phi KNL 7250 (Figure 5). The results shown here are promising and appealing. Our proposed offload model is able to match and even sometimes overcome the optimized routine from the vendors libraries on all of the experimented hardware. Moreover, the proposed one source code reach very close to the theoretical roofline model for this routine.

4.1.2 dgemv

Figures 6-11 show a performance comparison for the dgemv routine of our offload mode (running on a set of different architectures: CPU, Xeon Phi, and GPU) vs. both our code running in self-hosted model on the KNC and either the optimized Intel MKL dgemv from Intel or the cuBLAS dgemv from Nvidia. Our self-hosted results are important since they allow us to understand the performance of our code despite the effect of the offload directives. The comparison with the self-hosted and the MKL native shows the efficiency of the OpenMP 3.1

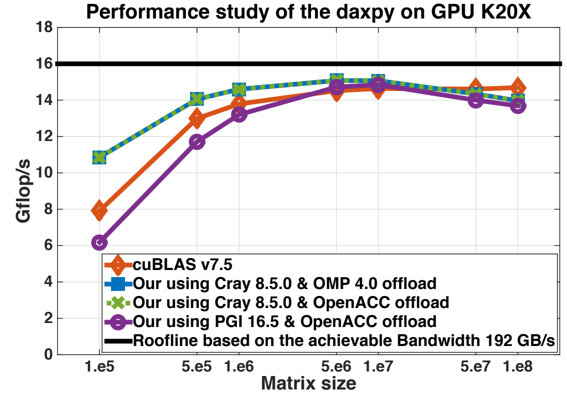

Figure 2: Performance measurement of the daxpy kernel on GPU K20X using three different method of offloading and comparing to the vendor optimized cuBLAS Library.

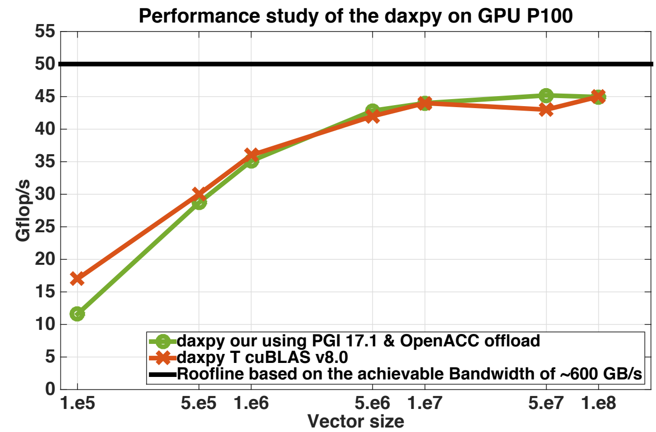

Figure 3: Performance measurement of the daxpy kernel on Nvidia GPU P100 using offloading model with PGI 17.1 and comparing to the vendor optimized cuBLAS Library.

directive into getting performance close to the optimized routine. The comparison between our model (self-hosted vs. offload) shows the portability behavior of the code and the possible overhead that could be introduced by the offload model. As shown, the offload model does not affect the performance of the kernel in any of the dgemv cases. More impressive is that this behavior has been demonstrated across multiple platforms (GPUs, KNC, etc). We note that the lower performance behavior shown for the dgemv non-transpose case on the KNC is due to the fact that the parallelization is implemented in such a way that every thread is reading a row of the matrix. This means that every thread reads data that is not contiguous in memory. On the CPU and KNL, we believe that, due to the size of the L3 level of cache and to the hardware prefetch, the code can still give acceptable results close to the MKL library and about 70% of the achievable peak. Because of the lack of hardware prefetching and the complex memory constraints of KNC, one might propose writing a more complex and parametrized kernel to reach better results.

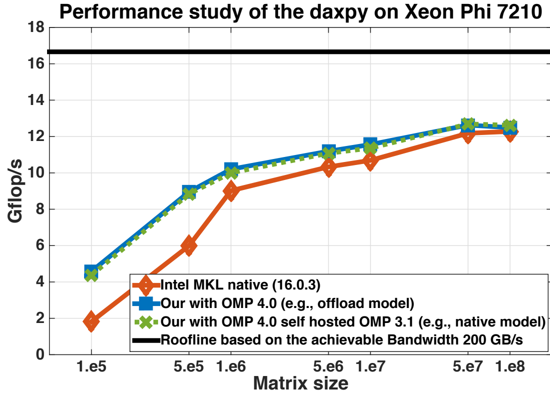

Figure 4: Performance measurement of the daxpy kernel on Xeon Phi KNC 7210 using the offload model and comparing to itself in native model and to the vendor optimized Intel MKL Library.

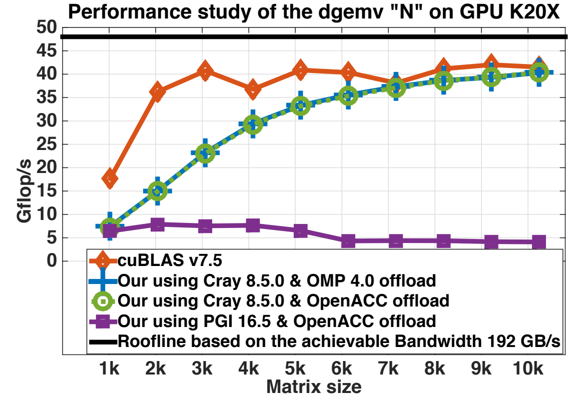

Figure 6: Performance measurement of the dgemv "N" kernel on GPU K20X using three different method of offloading and comparing to the vendor optimized cuBLAS Library.

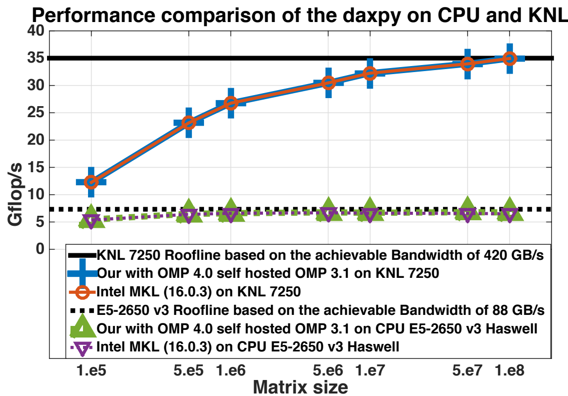

Figure 5: Performance measurement of the daxpy kernel on either a Xeon Phi KNL 7250 or recent CPU E5-2650 v3 running OMP4 as native model and comparing it to the vendor optimized Intel MKL Library.

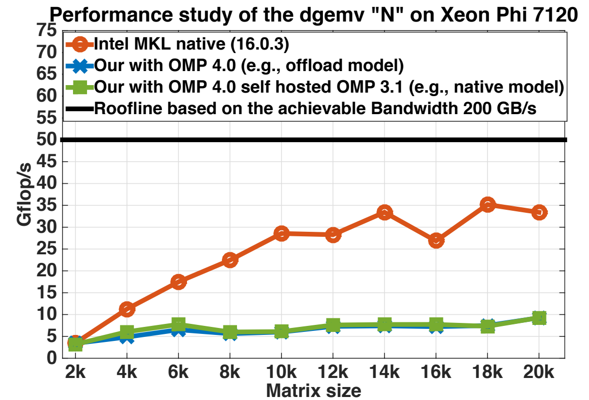

Figure 7: Performance measurement of the dgemv "N" kernel on Xeon Phi KNC 7210 using the offload model and comparing to itself in selfhosted model and to the vendor optimized Intel MKL Library.

For the GPU case, the results are mixed; the Cray compiler, with OpenMP 4 or OpenACC, is able to nearly match the cuBLAS performance, but the PGI compiler with OpenACC generates poorly performing code on both K20x and the P100. The dgemv transpose case is considered more data access friendly where each thread reads a column of the matrix, which means reading consecutive elements. For this case, we can see that our implementation performs as well as the libraries and achieves performance numbers close to the peak on all the considered platforms. Due to the simple structure of the code and stride-1 accesses, all compiler and directive combinations performed near peak performance on the two recent generation of Nvidia GPU.

4.1.3 dgemm

We studied the compute intensive kernel dgemm on the Intel E2650 v3 CPU (Haswell) and Intel KNL. The dgemm is well known to be the most complex kernel to implement. In our design, we propose to order the iterations of the nested loops in such a way that we increase locality and expose more parallelism for vectorization. The matrix-matrix product is an example of perfectly nested loops which means that all the assignment statements are in the innermost loop. Hence, loop unrolling, and loop interchange can be useful techniques for such algorithm [33, 34]. These transformations improve the locality and help to reduce the stride of an array-based computation. We proposed the loop ordering ijk, jik, and kji. Each of these flavor gave us the opportunity to either unroll, vectorize or perform both to the two most inner loops, which also allows us to reorder the computations for continuous access and improved vectorization. Figures 12 and 13 show the performance that can be achieved using any of the basic simple three loop implementations. As expected, the ijk loop ordering is the worst between the three of them. The jik is generally the best and the kji reaches the same or higher performance than the jik for small matrices. We note that for sizes less than 2,000, the kji always achieves higher performance than the jik. Nevertheless, we can easily see that all simple loop

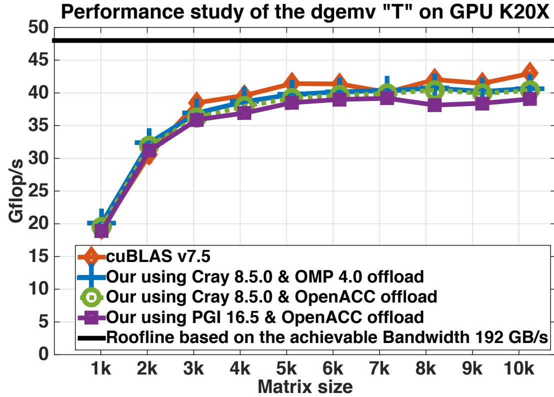

Figure 8: Performance measurement of the dgemv "T" kernel on GPU K20X using three different method of offloading and comparing to the vendor optimized cuBLAS Library.

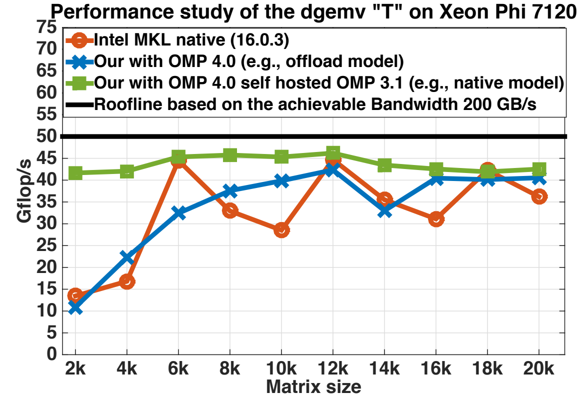

Figure 10: Performance measurement of the dgemv "T" kernel on Xeon Phi KNC 7210 using the offload model and comparing to itself in selfhosted model and to the vendor optimized Intel MKL Library.

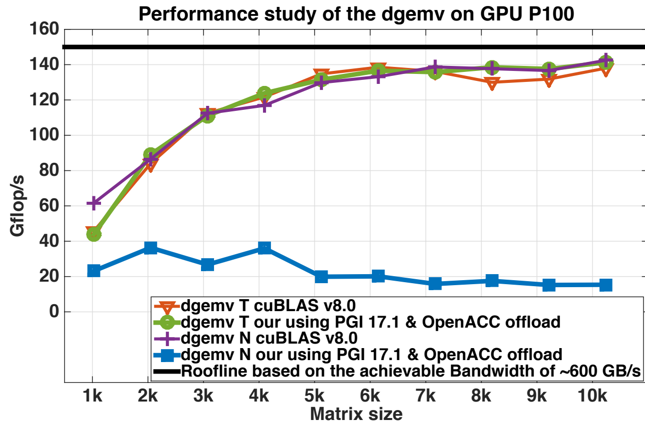

Figure 9: Performance measurement of the dgemv kernels (for both the "N" and "T" cases) on a Nvidia GPU P100 using offloading model with PGI 17.1 and comparing to the vendor optimized cuBLAS Library.

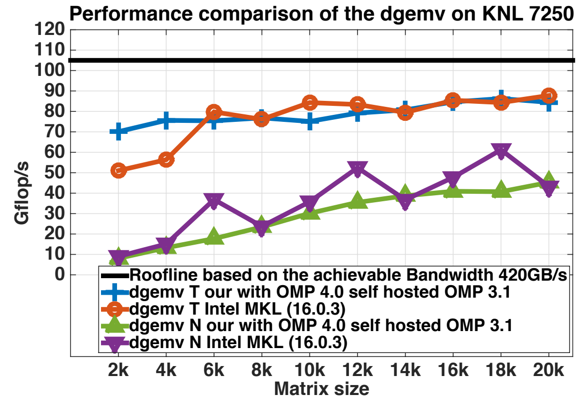

Figure 11: Performance measurement of the dgemv kernels (for both the "N" and "T" cases) on a Xeon Phi KNL 7250 running OMP4 as self-hosted model and comparing it to the vendor optimized Intel MKL Library.

implementations suffer when the matrix size is large, e.g., the matrices do not fit into the L2 cache in the case of CPU computations (see Figure 12).

For that, one of the best practices for numerically intensive operations, is to implement blocking strategies for better cache reuse. We focused on blocking to increase the performance and proposed another blocked implementation that call the simple basic kernel at its most inner loop. Since we had three basic kernels, we provided three flavors of the blocked dgemm routine. Our study concludes that the cache reuse ends up being one of the key factors for performance. The idea is that when data is loaded into L2 cache, it will be reused as much as possible before its replacement by new data. The amount of data that can be kept into L2 cache becomes an important tuning parameter and is hardware dependent. We note that the blocking size for the CPU was different from the one for the KNL. We performed a set of auto-tuning processes in order to determine the best combination. It turned out that 64, 64, 256 is the best one for the E2560 v3 CPU for BLK_M BLK_N, BLK_K, respectively, while 32, 32, 1024 is best for the KNL. The results obtained by the blocking implementation can reach half of what the vendor-optimized library can achieve, and about 40-50% of the theoretical peak of the machine. This is a very attractive result, and in particular the fact that by introducing a little bit of complexity to the code without breaking the proposed offload model, we are able to achieve performance that is about half of the vendor-optimized routine, which is mostly written in assembly. Also, since the blocking parameter can be defined in the header of the routine, we consider that this design is acceptable and portable. Otherwise, by running only the basic simple loop, one can never achieve high performance for any compute intensive kernel.

4.2 Jacobi

The results obtained from comparing the performance of OpenMP 3.1 (shared memory) with the OpenMP

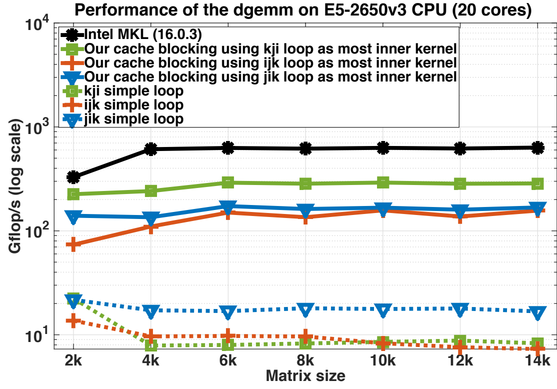

Figure 12: Performance measurement of different implementation of the dgemm kernel on a Intel 2650 v3 CPU (Haswell) running OMP4 as self-hosted model and comparing it to the vendor optimized Intel MKL Library.

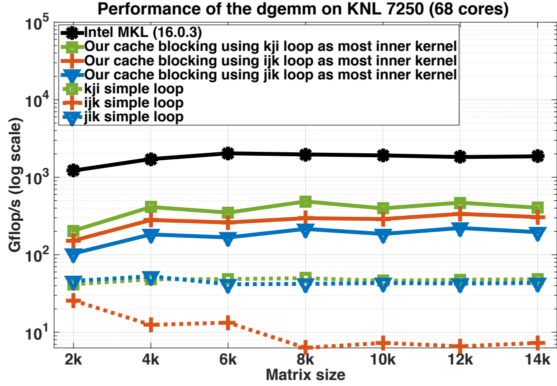

Figure 13: Performance measurement of different implementation of the dgemm kernel on a Xeon Phi KNL 7250 running OMP4 as self-hosted model and comparing it to the vendor optimized Intel MKL Library.

4.0 (accelerator) and OpenACC versions of the Jacobi kernel are shown in Figure 14. The OpenACC version of the kernel achieves the highest performance for both the PGI 16.5 and CCE 8.5.0 compilers. The OpenMP 4 (accelerator) version when offloaded to the GPU results in better performance than the OpenMP 3.1 (shared) version executed natively on the Intel Xeon Phi, and than the OpenMP 4 (accelerator) version when executed on the Intel Xeon Phi and offloaded to itself. When running the OpenMP 3.1 (shared) and OpenMP 4 (accelerator) version in native mode, we see similar performance results, though the shared version results in slightly higher performance. For this kernel, the lowest performance was observed when running the OpenMP 4 (accelerator) version on the host and offloading to the Intel Xeon Phi.

Ref. [8] explains in detail the two different execution modes possible on Titan and Beacon. For Titan, there is *standard* using only the CPU, and *offload* running the executable on the CPU and offloading to the GPU. For Beacon, three different execution modes are possible:

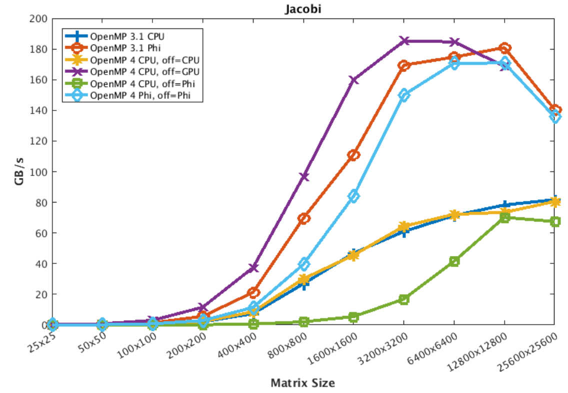

Figure 14: Jacobi kernel. Memory bandwidth of OpenMP 3.1 (shared memory), OpenMP 4 (offload) and OpenACC versions of Jacobi when running on Beacon and Titan. The OpenMP 3.1 (shared memory) model was measured on a 16-core AMD Opteron processor using 16 threads.

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. The additional platforms included in this study, Summitdev and Percival, are also capable of similar execution modes. On Summitdev, the *standard* and *offload* modes are available, whereas on Percival, only *standard* is available.

In the case of OpenACC version, the optimizations described in 3.4.2 did not result in a significant performance improvement. The results shown here include the `collapse(2)` directive for the first loop in the Jacobi subroutine. In order to run the OpenACC Jacobi kernel in different platforms, we used attempted to use the `-ta=host` and the `-ta=multicore` compiler flags for PGI 17.1. As expected, we found that using the `-ta=host` flag resulted in sequential code. The `-ta=multicore` flag, however, was useful to run the OpenACC Jacobi kernel on the AMD Interlagos, and the Intel KNL processors. Unfortunately, although PGI 17.1 was able to compile Jacobi on the Power8+ system, the application terminated abnormally without executing the full number of iterations. Those results are not included here.

Two distinct binaries were built with the PGI 17.1 compiler to run the OpenACC Jacobi kernel on Titan: one built using the multicore target architecture, and another using the NVIDIA specific one. The highest performance on the K20x was obtained for the OpenACC kernel using the PGI compiler, followed by the OpenMP 4 kernel code compiled with the Cray 8.5.5 compiler. When running only on the CPU, we observed that the best performance was achieved by using the OpenMP 3.1 (shared) kernel. On Summitdev, the OpenACC kernel built with PGI 17.1 was able to achieve approximately 90% of the peak memory bandwidth on the P100 GPU for medium size matrices. Results from Titan and Summitdev experiments are

summarized in Fig. 15. Note that we do not have a Cray platform with P100 GPUs installed, and so do not report P100 results for using the Cray compiler.

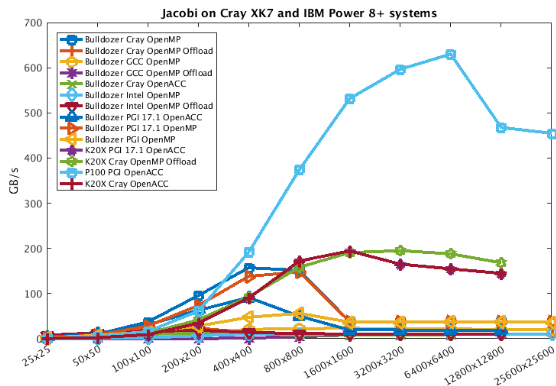

Figure 15: Jacobi kernel. OpenMP 3.1 (shared, OpenMP 4 (offload), and OpenACC versions of Jacobi when running on Titan (Bulldozer/K20x) and Summitdev (P8+/P100).

On Percival, the OpenACC Jacobi kernel compiled using the `-ta=multicore` flag achieved the highest performance. Our experiments show that changing the `ACC_NUM_CORES` variable had a significant impact in performance. The best performance was observed when setting `ACC_NUM_CORES` to 64, to match the number of real cores on a Percival compute node. For matrices larger than 160,000 elements, the PGI OpenACC multicore kernel performed better than the rest, followed closely by the OpenMP 3.1 (shared) kernel with GCC 6.2.0. Fig. 16 summarizes the results from the different kernels ran on Percival.

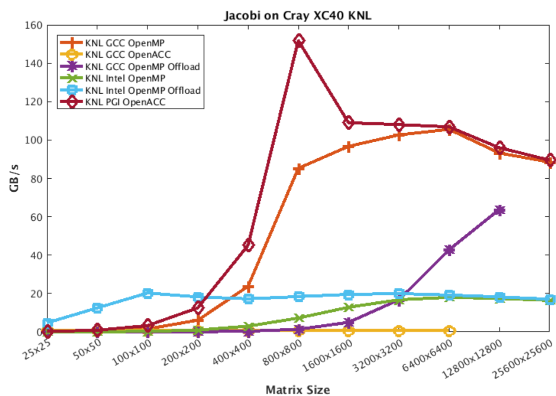

Figure 16: Jacobi kernel. OpenMP 3.1 (shared), OpenMP 4 (offload), and OpenACC versions of Jacobi when running on Percival (KNL Intel Xeon Phi 7230).

4.3 HACCmk

Figure 17 shows the HACCmk speedup of the OpenMP 4.5 (offload) and OpenACC versions when running on an NVIDIA K20x GPU as compared to the OpenMP

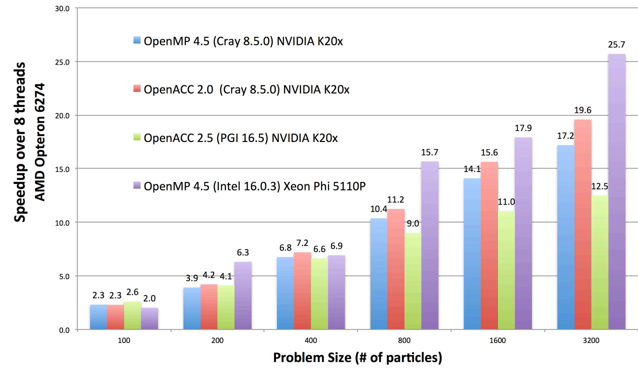

Figure 17: HACCmk kernel. Speedup of OpenMP 4.5 (offload) and OpenACC running on GPUs when compared to OpenMP shared memory running on a Bulldozer AMD using 8 threads.

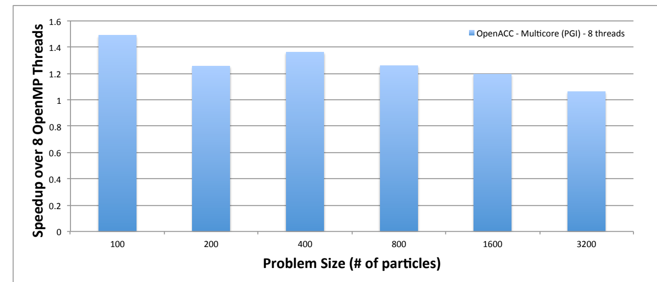

Figure 18: HACCmk kernel. Speedup of OpenACC (multicore) compared to OpenMP shared memory running on a Bulldozer AMD using 8 threads.

shared memory running on a Bulldozer AMD using 8 host CPU threads since each floating point unit is shared between 2 of the 16 physical cores. The OpenMP 4.5 and OpenACC versions always outperform the shared memory version running on the CPU. This is what we would expect given the K20x compute capabilities. Both OpenMP and OpenACC produced the same loop schedules using (327,0,0) grid blocks for the i loop and (128,0,0) threads for the threadblock for the j loop using both Cray and PGI compilers. For the Cray compiler, the speedups achieved when using OpenMP 4.5 offload and OpenACC are similar for small problem sizes, but there is a difference for the largest program size. This due to the inlining of Step10 (see Sec. 3.4.3) which helped the OpenACC version achieved a 12% improvement over OpenMP 4.5 using the Cray compiler. We observed less OpenACC speedup when using the PGI 16.5 compiler. The PGI compiler generated code that produced more thread divergence for the inner `acc loops` (e.g. control flow instructions) achieving only 49.4% of the occupancy when compared to 79.5% from the OpenACC code compiled with the Cray compiler.

Figure 18 shows the HACCmk speedup of OpenACC (multicore) over OpenMP 3.1 using 8 threads when running on a Bulldozer AMD using 8 cores using PGI 17.1. We used the OpenACC environment flag `ACC_NUM_CORES=8` to specify 8 OpenACC threads. The

OpenACC version outperforms the OpenMP 3.1 version. The main reason is the inlining of the subroutine in the OpenACC version The PGI compiler only parallelize the code at the gang level for multicores, which is equivalent to the OpenMP 3.1 version. At the time of the writing, when we tried to use OpenMP 4.5 offload to run on the the AMD processor, we did not see good speedups. The Cray implementation serialized the OpenMP 4.5 version and it did not support the multicore mode for OpenACC.

When HACC OpenMP 4.5 self-hosted on the Xeon Phi system and running on 240 OpenMP threads, we saw significant improvements over the baseline. The Intel compiler was able to vectorize all the instructions in the Step10 routine. As the problem size increased, we see significant improvements in performance because it is able to exploit the long vector units on the Xeon Phi.

5 Discussion and Conclusions

Directives like OpenMP 4 and OpenACC are designed to give programmers a way to express parallelism in their applications so that compilers can map this parallelism to significantly different hardware architectures. Because this is a difficult expectation, compiler implementations are still determining the optimal ways to fulfill these performance portability requirements. This means that application developers must be aware of general differences that arise from using directives on different platforms and compilers.

There are several examples of "lessons learned" that applied to all of the kernels that we studied. All of these apply to the compiler versions used on this study. The value for OpenMP `teams` will be supplied by the Cray compiler, and a single thread is supplied for `parallel for`. However, the Intel compiler will choose 1 team and multiple threads for `parallel for`. Similarly, the Cray compiler maps `SIMD` parallelization to a GPU threadblock, while the Intel compiler converts `SIMD` in actual vector instructions for the Xeon Phi. These types of differences are of course necessary to map to varying architectures, but to achieve optimal portability, the application developer must be aware of them.

During the development of this study, we made other small discoveries that turned out to be critical to achieving performance portability and the results that we have presented. In the following section, we explain some of these in the context of the application kernel that exposed the finding.

Lessons from DLA

We found with the DLA that performance portability is possible using directives across multiple architectures when we have good optimizing compilers. For DAXPY, all programming approaches for all tested platforms (OpenMP4/Phi, OpenMP3.1/Phi, OpenMP4/GPU, OpenACC/GPU) were able to roughly track the performance of the respective optimized linear algebra libraries (MKL, cuBLAS). For DGEMV/T the same was true. However, for DGEMV/N, for some cases this was true (OpenMP3.1/Phi 7250, OpenMP4/GPU/Cray compiler, OpenACC/GPU/Cray compiler), but for other cases (OpenMP4/Phi 7210, OpenMP3.1/Phi 7210, OpenACC/GPU/PGI compiler) the user-written code highly underperformed the optimized libraries due to the difficulty of the compiler optimizing the non-stride-1 arithmetic as well as slightly more complex code logic having to do with multiple parallel loops.

For the well-performing cases, the code for the respective methods was written in a natural way without requiring excessive effort for manual code transformations. Since the kernels are very fundamental in nature, it seems likely that compiler implementors would include these or similar cases in their performance regression suites, hence the good performance. For DAXPY and DGEMV/N, this is a success story for the compilers and directives-based methods since they were all able to generate nearly-optimal code. For DGEMV/T however, the added complexities created challenges which for some compilers were unsurmountable. We also learned that the `omp simd` directive is not needed to achieve performance portability when compilers have good automatic vectorization capabilities.

However, for the compute intensive `dgemm` kernel, performance portability and compiler-based optimizations are not as straightforward as for the memory bound kernels; here performance depends critically on the kernel design. This was illustrated by quantifying the performance for a number of simple implementations. The results clearly show that the compiler can not accomplish much for the simple loop implementations, and that algorithmic changes are needed. We note that even simple loop transformations can help the compilers to significantly improve performance. In order to get another level of improvement, we demonstrated that the design must include hierarchical communications and optimized memory management. In particular, we show that only after we (1) developed algorithmic designs that feature multilevel blocking of the computations that increase cache reuse and use different loop reordering to improve vectorization, and (2) introduced autotuning, our implementations reached acceptable performance (around 50% of the peak of the machine).

Lessons from Jacobi

Jacobi is another example where we were able to achieve good performance portability across architectures using the OpenMP 4.5 accelerator programming model. Performance depends on OpenMP programming style and on which mode is being in the hardware. The best performance was achieved when we ran OpenMP 4.5 in Xeon Phi self hosted mode and when OpenMP 4.5 was using in the offloading for GPUs. We also learned that OpenACC as a programming model is easier to use than OpenMP. Both the PGI and CCE implementations of OpenACC were able to achieve good performance with minimal effort. When converting the code to OpenACC, the PGI compiler was able to automatically insert the loop schedules and

levels of parallelism such as `gang` and `vector` directives on both of the main loops inside the kernel. One of the goals of OpenACC directives is to give this flexibility to compiler. The drawback of using OpenACC is its lack of or poor implementations on multiple architectures (such as Xeon Phi or CPU). This impacts its usability for performance portability as of now.

When we tried offloading both OpenACC and OpenMP 4.5 to CPU multicores, this resulted in poor performance due to implementations not being optimized yet on the Cray and PGI compiler. We were able to successfully compile OpenMP 4.5 on Xeon Phi. The performance is good when running in OpenMP 4.5 accelerator model on self-hosted mode vs offloading target regions to the Xeon Phi from CPU. OpenMP 4.5 accelerator model was able to achieve comparable performance to OpenMP 3.1 in self-hosted mode, which supports the idea that the OpenMP 4.5 accelerator model is performance portable. This was very encouraging as we can use the OpenMP accelerator model as a performance portable programming style that can achieve good performance across multiple architectures.

We were able to use GCC for both the OpenACC version of the Jacobi kernel and the OpenMP 4.5 version. Using the `ACC_DEVICE_TYPE` environment variable, we were able to also compare the performance of the OpenACC kernel when run on the CPU vs. when offloaded to the GPU.

With the PGI 17.1 compiler, we observed better performance of the OpenACC kernel when the `-ta=tesla:cc60` and the `-ta=tesla:cc35` compiler flags were used on the Power8+ and the Cray XK7 system, respectively. As expected, our results show that using the `-ta=host` compiler flag results in sequential code. To run OpenACC code on the CPU, we explored using the `-ta=multicore` compiler flag and found that on the Power8+ system, incorrect results were produced. However, on both the Intel KNL system and the Cray XK7 AMD CPU, we were able to run the OpenACC kernel successfully. On the Intel KNL system, setting `ACC_NUM_CORES` to 64 produced the best results, which represents the total number of cores on a compute node. By default, all visible hardware threads were being used.

Lessons from HACCmk When using the same compiler, the performance gap between OpenACC and OpenMP 4.5 can be small when the OpenMP 4.5 parallelization strategy (e.g. loop schedules, etc) matches the one picked by the OpenACC compiler. Also their difference in performance is small when using the same compiler to compile both versions. One of our findings is that the performance of OpenACC depends on the ability of the compiler to generate good code. We observed a significant performance variation between when OpenACC is compiled with Cray 8.5.0 and the PGI 16.5. Further investigation showed a significant performance variation when we tried PGI 16.7. This tells us that compilers play a significant role on the the level of performance portability of a programming model. When we compiled OpenMP 4.5 to run self-hosted on the Xeon Phi, the Intel compiler ignores the target directives. These includes `omp teams, omp distribute, omp declare target` or any form of a target combined directives. Because of this behavior, we had to transform the combined directive `omp distribute parallel for` to individual directives `omp distribute` and `omp parallel for`.

The `omp simd` directive is extremely useful for performance portability, not only to specify parallelism for offloading to accelerators, but depending on the implementation, it can be critical to achieve good levels of vectorization across compilers when there exist different levels of support for automatic vectorization. Compilers are not yet able to consistently identify these opportunities in all cases, so it must be used to ensure that vectorization is used where appropriate. Although, GPUs do not have vector units, the `SIMD` directive can be helpful to identify potential very fine-grained parallelism that can be executed by SMT threads (e.g. GPU warps) and by using this directive, the programmers can increase the performance portability of the model. We were able to achieve good OpenMP 4.5 performance on GPUs and self hosted Xeon Phi. However, it would be helpful if future Intel compilers support the combined target directives.

We experience better performance when using OpenACC (multicore) vs OpenMP 3.1 baseline when running in an AMD Bulldozer processor using 8 cores using PGI 17.1. One of the reasons is because OpenACC provides more information to vectorization phase of the compiler including information about reductions. Being able to specify another level of parallelism in OpenACC that target vector instructions was profitable. We were not able to do the same with OpenMP 4.5 and target multicore. At the time of the writing, the Cray compiler serialized the OpenMP 4.5 (offload) code on the CPU.

6 Future Work

Compiler implementations and new HPC architectures are currently evolving quickly, and there is a wealth of future study that presents itself in this area. Some straightforward technical tasks to further round out the portability aspect of the exploration with more platforms and implementations. We have been working with implementers to address implementation bugs as we encounter them during this study, and so as the functionality and performance improves, the apparent performance portability of directives could possibly change accordingly.

We observed sensitivity of the performance (and therefore overall performance portability) to not only the choice of a programming model, its programming style, and quality of the compiler implementation, but also the compilation optimizations requested by the user. A thorough parameter space exploration of compiler options, directive clause arguments, and

runtime environment setup is necessary to more fully understand these effects. When such a large parameter space exists, autotuning presents itself as a likely candidate for improvement of the model, both from a performance as well as usability point of view. Such a parameter space exploration could also better inform the SPEC ACCEL committee's prescriptions for writing performance portable OpenMP 4 as described in Sec. 3.3, and if these are sufficient for application kernels.

This paper focused heavily on the performance and portability of expressing the fine-grained parallelism of application kernels using directives. However, various coarse-grained parallelization schemes such as tasking are needed to efficiently address multiple levels of compute and memory heterogeneity. How to productively couple these different models of expressing parallelism and the performance implications of the choices made about granularity, etc., are not yet well-understood, but thought to be critical to achieving exascale performance for real-world applications.

Acknowledgements

This material is based upon work supported by the U.S. Department of Energy, Office of science, and 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 research was also partially supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

References

[1] T. Williams, K. Antypas, and T. Straatsma. 2015 workshop on portability among HPC architectures for scientific applications. `http://hpcport.alcf.anl.gov/`, November 2015 (accessed 10/2016).

[2] J. Reinders, M. Glass, R. Hartman-Baker, J. Levesque, H. Ah Nam, R. Neely, J. Sexton, T. Straatsma, T. Williams, and C. Zeller. DOE centers of excellence performance portability meeting. `https://asc.llnl.gov/DOE-COE-Mtg-2016/`, (accessed 10/2016).

[3] CORAL fact sheet. http://www.anl.gov/sites/anl.gov/files/CORAL(accessed 10/2016).

[4] Summit: Scale new heights. discover new solutions. https://www.olcf.ornl.gov/summit/, (accessed 10/2016).

[5] Sierra advanced technology system. http://computation.llnl.gov/computers/sierra-advanced-technology-system, (accessed 10/2016).

[6] Aurora. http://aurora.alcf.anl.gov/, (accessed 10/2016).

[7] M. Graham Lopez, Verónica Vergara Larrea, Wayne Joubert, Oscar Hernandez, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. Towards achieving performance portability using directives for accelerators. In *Proceedings of the Third International Workshop on Accelerator Programming Using Directives*, WACCPD '16, pages 13–24, Piscataway, NJ, USA, 2016. IEEE Press.

[8] Verónica Vergara Larrea, Wayne Joubert, M. Graham Lopez, and Oscar Hernandez. Early experiences writing performance portable OpenMP 4 codes. In *Proc. Cray User Group Meeting, London, England*. Cray User Group Incorporated, May 2016.

[9] John Reid. The new features of fortran 2008. *SIGPLAN Fortran Forum*, 27(2):8–21, August 2008.

[10] Jared Hoberock. Working draft, technical specification for c++ extensions for parallelism. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4071.htm`, 2014 (accessed 10/2016).

[11] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[12] R. D. Hornung and J. A. Keasler. The RAJA portability layer: Overview and status. `https://e-reports-ext.llnl.gov/pdf/782261.pdf`, 2014 (accessed 10/2016).

[13] E. Calore, S. F. Schifano, and R. Tripiccione. On Portability, Performance and Scalability of an MPI OpenCL Lattice Boltzmann Code. *Euro-Par 2014: Parallel Processing Workshops, Pt Ii*, 8806:438–449, 2014.

[14] S. J. Pennycook and S. A. Jarvis. Developing Performance-Portable Molecular Dynamics Kernels in OpenCL. *2012 Sc Companion: High Performance Computing, Networking, Storage and Analysis (Scc)*, pages 386–395, 2012.

[15] Chongxiao Cao, Mark Gates, Azzam Haidar, Piotr Luszczek, Stanimire Tomov, Ichitaro Yamazaki, and Jack Dongarra. Performance and portability

with opencl for throughput-oriented hpc workloads across accelerators, coprocessors, and multicore processors. In *Proceedings of the 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '14, pages 61–68, Piscataway, NJ, USA, 2014. IEEE Press.

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

[17] Seyong Lee and Rudolf 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*, SC '10, pages 1–11. IEEE Computer Society, 2010.

[18] Tianyi David Han and Tarek 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*, pages 52–61. ACM, 2009.

[19] Oscar Hernandez, Wei Ding, Barbara Chapman, Christos Kartsaklis, Ramanan Sankaran, and Richard Graham. Experiences with high-level programming directives for porting applications to GPUs. In *Facing the Multicore-Challenge II*, pages 96–107. Springer, 2012.

[20] Seyong Lee and Jeffrey 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*, page 23. IEEE Computer Society Press, 2012.

[21] Sandra Wienke, Christian Terboven, James C. Beyer, and Matthias S. Müller. *Euro-Par 2014 Parallel Processing: 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings*, chapter A Pattern-Based Comparison of OpenACC and OpenMP for Accelerator Computing, pages 812–823. Springer International Publishing, Cham, 2014.

[22] Guido Juckeland, William Brantley, Sunita Chandrasekaran, Barbara Chapman, Shuai Che, Mathew Colgrove, Huiyu Feng, Alexander Grund, Robert Henschel, Wen-Mei W. Hwu, Huian Li, Matthias S. Müller, Wolfgang E. Nagel, Maxim Perminov, Pavel Shelepugin, Kevin Skadron, John Stratton, Alexey Titov, Ke Wang, Matthijs Waveren, Brian Whitney, Sandra Wienke, Rengan Xu, and Kalyan Kumaran. *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation: 5th International Workshop, PMBS 2014, New Orleans, LA, USA, November 16, 2014. Revised Selected Papers*, chapter SPEC ACCEL: A Standard Application Suite for Measuring Hardware Accelerator Performance, pages 46–67. Springer International Publishing, Cham, 2015.

[23] G. Juckeland, A. Grund, and W. E. Nagel. Performance Portable Applications for Hardware Accelerators: Lessons Learned from SPEC ACCEL. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pages 689–698, May 2015.

[24] Matt Martineau, Simon McIntosh-Smith, Mike Boulton, and Wayne Gaudin. An Evaluation of Emerging Many-Core Parallel Programming Models. In *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM'16, pages 1–10, New York, NY, USA, 2016. ACM.

[25] Guido Juckeland et. al. From describing to prescribing parallelism: Translating the SPEC ACCEL OpenACC suite to OpenMP target directives. In *ISC High Performance 2016 International Workshops, P3MA*, June 2016.

[26] OpenMP Architecture Review Board. OpenMP Application Program Interface. Version 4.5. `http://www.openmp.org/mp-documents/openmp-4.5.pdf`, November 2015 (accessed 10/2016).

[27] HACCmk. `https://asc.llnl.gov/CORAL-benchmarks/Summaries/HACCmk_Summary_v1.0.pdf`, accessed 03/2016.

[28] Joseph Robicheaux. Program to solve a finite difference equation using Jacobi iterative method. `http://www.openmp.org/samples/jacobi.f`, 2004 (accessed 03/2016).

[29] Wayne Joubert, Rick Archibald, Mark Berrill, W. Michael Brown, Markus Eisenbach, Ray Grout, Jeff Larkin, John Levesque, Bronson Messer, Matt Norman, Bobby Philip, Ramanan Sankaran, Arnold Tharrington, and John Turner. Accelerated application development. *Computers and Electrical Engineering*, 46(C):123–138, August 2015.

[30] Summitdev Quickstart: System Overview. `https://www.olcf.ornl.gov/kb_articles/summitdev-quickstart/#System_Overview`, (accessed 10/2016).

[31] Percival Quickstart: System Overview. `https://www.olcf.ornl.gov/kb_articles/percival-quickstart/#System_Overview_`, (accessed 10/2016).

[32] 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*, 17(2):65–72, 2015.

[33] N. Ahmed, N. Mateev, and K. Pingali. Tiling imperfectly-nested loop nests. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 31–31, Nov 2000.

[34] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, December 1994.