Chapter 7

OpenACC and Performance Portability

Graham Lopez and Oscar Hernandez, Oak Ridge National Laboratory

This chapter discusses the performance portability of directives provided by OpenACC to program various types of machine architectures. This includes nodes with attached accelerators: self-hosted multicores (e.g., multicore-only systems such as the Intel Xeon Phi) as well as GPUs. Our goal is to explain how to successfully use OpenACC for moving code between architectures, how much tuning might be required to do so, and what lessons we can learn from writing performance portable code. We use examples of algorithms with varying computational intensities for our evaluation, because both compute and data-access efficiency are important considerations for overall application performance. We explain how various factors affect performance portability, such as the use of tuning parameters, programming style, and the effectiveness of compilers' flags in optimizing and targeting multiple platforms.

7.1 Challenges

Performance portability has been identified by the high-performance computing (HPC) community as a high-priority design constraint for next-generation systems such as those currently being deployed in the Top500 (a list that ranks systems by using the Linpack benchmark)[1] as well as the exascale systems upcoming in the next decade. This prioritization has been emphasized because software development and maintenance costs are as large or larger than the cost of the system itself, and ensuring performance portability helps protect this investment—for example, by ensuring the application's usability if one architecture goes away or a new one becomes available.

Looking forward, we are seeing two main node-architecture types for HPC: one with heterogeneous accelerators (e.g., IBM Power-based systems with multiple NVIDIA Volta GPUs; Sunway accelerator architecture), and the other consisting of homogeneous architectures (e.g., third-generation Intel Xeon Phi-based nodes, Post-K ARM, etc.). At present it is a nontrivial task to write a performance portable application that targets these divergent hardware architectures and that makes efficient use of all the available computational resources (both at the node level and across nodes). It is clear that applications need to be written so that the parallelism can be easily decomposed and mapped with at least three levels of parallelism: across nodes, within the nodes (thread-level parallelism), and vector-level parallelism (fine-grained or SIMD-level parallelism).

The latest OpenACC 2.5 specification defines a directive-based programming API that can target the thread and vector levels, and it accommodates both traditional shared-memory systems and accelerator-based systems. However, we have also learned that performance portability depends on the quality of the implementation of the compilers and their ability to generate efficient code that can take advantage of the latest architectural features on different platforms. Shared-memory programming has been available in, and has been the main focus of, the industry-standard OpenMP specification for more than a decade, but the recent introduction of an offloading model in OpenMP poses the question, Is the accelerator model suitable to target both shared-memory and accelerator-based systems?

In this chapter, we show how OpenACC can be used as a single programming model to program host multicore, homogeneous, and heterogeneous accelerators, and we discuss the potential performance or productivity tradeoffs. We highlight how OpenACC can be used to program a micro-kernel called the Hardware Accelerated Cosmology Code (HACCmk), which is sensitive to vector-level parallelism (e.g., vectorization, warps, SMTs/SIMD, etc.).

We use the PGI and Cray compilers (see Section 7.3.4, "Data Layout for Performance Portability," later in this chapter for versions) to target OpenACC both on CPUs and on NVIDIA GPU hardware platforms. We compare the performance of

1. https://www.top500.org/project/.

the OpenACC versions versus baseline platform-optimized code written in multi-threaded OpenMP 3.1. Another approach to measure performance portability of the code is to compare the performance results to the machine theoretical peak floating-point operations per second (FLOPS) (for compute-bound kernels) or bandwidth (for memory-bound kernels) across the target architectures.

We summarize these experiences to reflect the current state of the art for achieving performance portability using OpenACC.

7.2 Target Architectures

To demonstrate performance portability using OpenACC, we use two target architectures: x86_64 with attached NVIDIA GPUs, and x86_64 only (multicore). The work can also be extended to target self-hosted Intel Xeon Phi KNL processors. At the time of this writing, support for Knights Landing (KNL) was not generally available with the PGI compilers.

7.2.1 COMPILING FOR SPECIFIC PLATFORMS

OpenACC is an open standard that is not tied to a specific architecture or software stack, but in this chapter we focus primarily on the PGI compiler, because it currently provides the most general performance portability. At this time, other available implementations, such as Cray, are more specialized in purpose, and upcoming implementations, such as that in the GNU compilers, are not yet as robust across multiple architectures.

7.2.2 X86_64 MULTICORE AND NVIDIA

In the PGI compiler, in addition to the `–acc` switch to enable general OpenACC support, there are flags that can further direct the generation of target code. Here we explain an example of how to use the PGI compiler to generate an executable that is suitable for various types of host CPU and NVIDIA accelerator platforms.

The default PGI behavior for the `–acc` flag is to create a unified "fat" binary that includes both host serial CPU and multiple targets of varying compute capabilities (cc20, cc35, cc50, etc.). This same behavior can be obtained by using, for instance, the flag `–ta=tesla,host`. At run time, the default OpenACC `device_type` will be `NVIDIA` unless no NVIDIA targets are available, in which case the `device_type` will be `HOST`.

Users can modify the default by either compiling to a specific target (such as `–ta=tesla:cc60` for the Pascal architecture, or `–ta=multicore` for a parallel CPU version); calling `acc_set_device_type()` in the program; or setting the `ACC_DEVICE_TYPE` environment variable to have an effect on the default device type.

To take advantage of recent unified memory capabilities in the NVIDIA architecture, you can add the flag `–ta=tesla:managed`.

In the near future, OpenACC parallelization for KNL will be supported by the PGI compiler, furthering its capabilities toward performance portability.

7.3 OpenACC for Performance Portability

When targeting multiple architectures you must be aware of how the programming model maps to the target architecture. One important factor for high performance is efficient use of the memory systems. In this section, we examine how the OpenACC memory model can be mapped to the various memory architectures.

7.3.1 THE OPENACC MEMORY MODEL

OpenACC uses a copy-in and copy-out data regions memory model to move data to local (otherwise known as **affine**) memories of the accelerator. These data regions can be thought of as user-managed caches. The interesting property of this memory model is that it can be mapped efficiently to a variety of architectures. For example, on shared memory, the data regions can be either ignored or used as prefetching hints. On systems that have discrete memories, data regions can be translated to data transfer APIs using a target runtime (e.g., CUDA, OpenCL, etc.). On partially shared memory systems, the `data` directives can be either used or ignored, depending on whether the thread that encounters the data region can share data with the accelerator (e.g., unified memory, managed memory, etc.).

OpenACC data regions can be synchronized with the host memory. All data movement between host memory and device memory is performed by the host through runtime library calls that explicitly move data between the memories (or ignored in shared memory), typically by using direct memory access (DMA) transfers.

7.3.2 MEMORY ARCHITECTURES

Following is a list of various types of system memory.

- **Discrete memories.** These systems have completely separate host and device memory spaces that are connected via, for example, PCIe or NVLINK. This memory architecture maps well to the OpenACC data regions' copy-in and copy-out memory model.
- **Shared memory.** In these systems, all of the cores can access all of the memory available in the system. For example, traditional shared multicore systems (Intel KNL self-hosted, etc.) fit into this category. This model maps well to OpenACC, because the data region directives can be ignored or used as hints to the compiler to do prefetching.
- **Partially shared memories.** In this scenario, some of the system memory is shared between the host and the accelerator threads. Each device may have its own local memories but accesses a portion of memory that is shared. Future OpenACC specifications will support this by allowing data regions to be optionally ignored or used for prefetching if the thread of a host shares the same address space as the thread of the device. Currently, this is supported in the PGI compiler using the `–ta=tesla:managed` flag for dynamic allocated memory, and on systems where sharing is possible, as in the case of NVIDIA managed memory supported by software (e.g., CUDA Managed Memory over PCIe, or NVLINK).

7.3.3 CODE GENERATION

The best way to generate performance portable code is to use and tune the OpenACC `acc loop` directive, which can be used to distribute the loop iterations across gangs, workers, or vectors. The `acc loop` directive can also be used to distribute the work to gangs while the loop is still in worker-single and vector-single mode. For OpenACC, it is possible in some cases to apply `loop` directives such as `tile` to multiple nested loops to strip-mine the iteration space that is to be parallelized.

The OpenACC compilers also accept `gang`, `worker`, or `vector` clauses to pick the correct level of parallelism. If you specify only `acc loop`, the compiler decides how to map the iteration space to the target architecture based on its cost models and picks the right type of scheduling across gangs, 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, and that helps generate performance portable code while taking advantage of the target accelerator architecture.

However, in some cases the compiler cannot do the best job in generating the correct loop schedules. For these cases, you can improve the loop scheduling by adding clauses such as `gang`, `worker`, or `vector` to the OpenACC loops. In cases of perfectly nested parallel loops, OpenACC also supports the use of the `tile` clause to schedule nested loops to a given level of parallelism (e.g., gang or vector).

7.3.4 DATA LAYOUT FOR PERFORMANCE PORTABILITY

It's important to decide how the layout of data structures affects performance portability. Structures of arrays are in general more suitable for GPUs as long as the data access to the arrays is contiguous (memory coalescing). This is a good layout optimization for throughput-driven architectures. On the other hand, arrays of structures are also good for caching structure elements to cache lines, a layout that is important for latency-driven architectures common to host CPUs. Interestingly, we have noted that in some cases, improving data layouts on GPUs can also benefit the multicore case, but not as commonly the other way around.

We note that high-level frameworks for data abstractions (e.g. Kokkos,[2] SYCL,[3] etc.) can be useful to explore these types of issues related to data structure layouts, but they come at the cost of making the compilation process and compiler analysis more complex.

7.4 Code Refactoring for Performance Portability

To study the performance portability of accelerator directives provided by OpenACC, we use a kernel from the HACC HPC cosmology application. This kernel is part of the CORAL benchmarks suite.

2. http://www.sciencedirect.com/science/article/pii/S0743731514001257.
3. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0236r0.pdf.

7.4.1 HACCMK

HACC is a framework that uses N-body techniques to simulate fluids during the evolution of the early universe. The HACCmk[4] 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.

The HACCmk kernel as found in the CORAL benchmark suite has shared memory OpenMP 3.1 implemented for CPU multicore parallelization, but here we convert it to OpenACC 2.5. The kernel has one parallel loop over particles that contain a function call to the bulk of the computational kernel. This function contains another parallel loop over particles, resulting in 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 generate performance portable code by decomposing the parallelism of the two nested loops across threads and vector lanes (fine-grained parallelism). For vector (or SIMD-based) architectures, the compiler should aim at generating code that exploits vector instructions having long widths. For multithreaded or SMT architectures, it should exploit thread-level parallelism and smaller vector lanes.

As shown in Listing 7.1, the OpenACC version of the HACCmk microkernel, we parallelize the outer loop level using the `acc parallel loop` directive. The inner loop is marked by using an `acc loop` with `private` and `reduction` clauses. We intentionally do not specify any loop schedule in both `acc` loops to allow the compiler to pick the best schedule for the target architecture (in this case the GPU or multicore). We did this both to test the quality of the optimization of the OpenACC compiler and to measure how performance portable OpenACC is across architectures.

Listing 7.1 OpenACC version of the HACCmk microkernel

```
#pragma acc parallel private(dx1,dy1,dz1) \
                     copy(vx1,vy1,vz1) \
                     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;
```

4. https://asc.llnl.gov/CORAL-benchmarks/Summaries/HACCmk_Summary_v1.0.pdf.

Listing 7.1 OpenACC version of the HACCmk microkernel (*continued*)

```
        int j;
        xi = 0.; yi = 0.; zi = 0.;
#pragma acc loop  private(dxc, dyc, dzc, r2, m, f) \
                  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;
}
```

HACCmk is an extremely interesting case study, because to work with performance portable codes, the compiler must successfully vectorize all the statements of the inner procedure (generate vector instructions) or generate efficient multithreaded or SMT code. For this code, performance portability depends on the quality of the compiler implementation and its ability to vectorize code or to generate multi-threaded (SMT) code for GPUs. To get good performance on the CPU and Xeon Phi, we also need to make sure that there is a vector implementation of the `powf`, which belongs to the C math library `math.h`.

7.4.2 TARGETING MULTIPLE ARCHITECTURES

Achieving performance portability with OpenACC depends on how the compiler lowers (translates) and maps the parallelism specified by OpenACC to the target architecture. When we compile HACCmk with the PGI compiler and target a K20x NVIDIA GPU, we get the following output:

```
pgcc -acc -Minfo -O3 -c main.c -o main.o
main:
    203, Loop not vectorized: data dependency
```

```
         Loop unrolled 4 times
         FMA (fused multiply-add) instruction(s) generated
    211, Generated an alternate version of the loop
         Generated vector simd code for the loop
    222, Memory set idiom, loop replaced by call to __c_mset1
    223, Memory copy idiom, loop replaced by call to __c_mcopy1
    239, Generating implicit copyin(mass[:n])
         Generating copy(vx1[:],vy1[:],vz1[:])
         Generating copyin(xx[:n],zz[:n],yy[:n])
         Accelerator kernel generated
         Generating Tesla code
        242, #pragma acc loop gang /* blockIdx.x */
        257, #pragma acc loop vector(128) /* threadIdx.x */
             Generating reduction(+:xi,zi,yi)
```

When you compile HACCmk with the PGI compiler and target an AMD Bulldozer architecture, you get the following compiler output:

```
pgcc -acc -Minfo -O3 -ta=multicore -c main.c -o main.o
main:
    188, Loop not vectorized/parallelized:
         contains a parallel region
    203, Loop not vectorized: data dependency
         Loop unrolled 4 times
         FMA (fused multiply-add) instruction(s) generated
    211, Generated an alternate version of the loop
         Generated vector simd code for the loop
    222, Memory set idiom, loop replaced by call to __c_mset1
    223, Loop unrolled 8 times
    239, Generating Multicore code
        242, #pragma acc loop gang
    257, Loop is parallelizable
         Generated vector simd code for the loop containing
         reductions and conditionals
         Generated 4 prefetch instructions for the loop
         FMA (fused multiply-add) instruction(s) generated
    301, Generated vector simd code for the loop containing
         reductions
         Generated 3 prefetch instructions for the loop
         FMA (fused multiply-add) instruction(s) generated
```

Notice that for these two architectures, PGI translates the outer OpenACC loop to gang-level parallelism (loop 239) and translates the inner loop to vector-level parallelism (loop 257). However, one of the main differences is the `vector_length` used. For the GPU version, the vector length is 128, whereas the CPU version is based on the size of the vector register for AVX (256-bits). In this case the `vector_length` is 8 (for 8 floats vector instructions). Also notice that to efficiently vectorize the inner loop for multicores, generation of vector predicates and intrinsics is needed (e.g., `powf()`).

We specified different vector lengths using OpenACC, but for both cases (GPU and multicore) the PGI compiler always picked 128 (for the GPU) and 4 (for the multicore) architecture. The compiler's internal cost models picked the right length for the different architectures and optionally decided to ignore the clauses provided by the user, as reported in the output from `–Minfo` shown earlier.

To control the number of threads spawned on the multicore platforms, you use the `ACC_MULTICORE` environment variable. However, this variable is a PGI extension and not part of the OpenACC standard. You should use this flag if you want to control the number of threads to be used on the CPU. If the flag is not specified, the CPU will use the maximum number of threads available on the target multicore architecture.

7.4.3 OPENACC OVER NVIDIA K20X GPU

We ran HACCmk on the OLCF Titan Cray XK7 supercomputer, which consists of a cluster of AMD Interlagos host CPUs connected to NVIDIA K20x GPUs. For the Oak Ridge Leadership Computing Facility (OLCF) Titan system, a compute node consists of (a) 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 (b) an NVIDIA Kepler K20x GPU with a peak single- or double-precision flop rate of 3,935/1,311 GF and a peak memory bandwidth of 250 GB/sec. For this platform, Cray compilers were used, with versions 8.5.0, and PGI 16.5 / 17.1.

Figure 7.1 shows the HACCmk speedup of the OpenACC version when running on an NVIDIA K20x GPU, as compared with the OpenMP shared-memory version running on an AMD Bulldozer processor using 8 host CPU threads (because each floating-point unit is shared between 2 of the 16 physical cores). The OpenACC version always outperformed the shared-memory version running on the CPU. This is what we would expect given the K20x compute capabilities. When we compare the results using different compilers, we observed less OpenACC speedup when using the PGI 16.5 compiler. These results highlight the fact that performance portability of code also depends on the quality of the compiler optimizations, because more hints may be needed to generate performance portable code, depending on the compiler.

7.4.4 OPENACC OVER AMD BULLDOZER MULTICORE

Figure 7.2 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

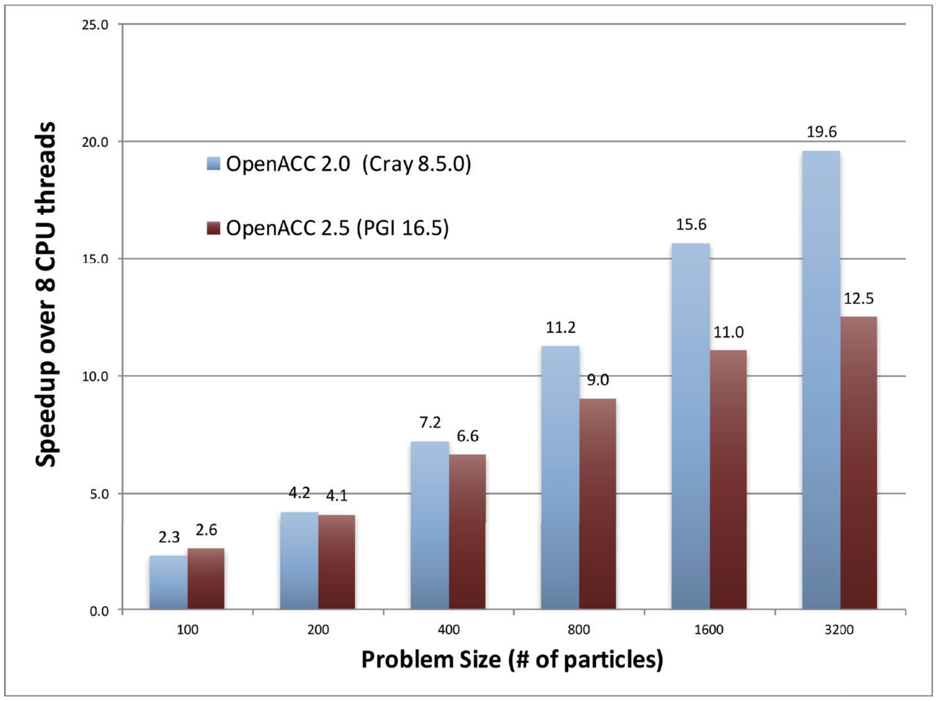

Figure 7.1 Speedup of OpenACC running on NVIDIA K20x GPUs when compared to OpenMP shared memory running on Bulldozer AMD CPU using 8 threads

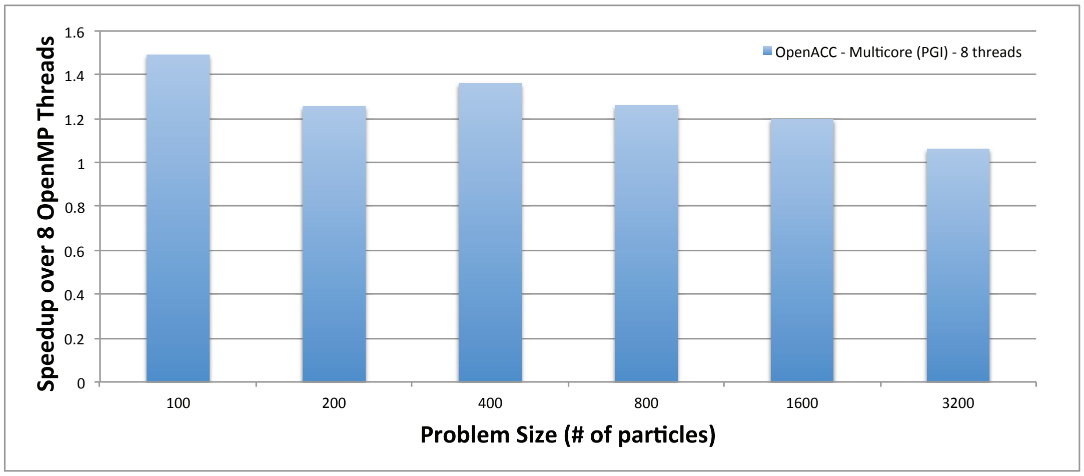

Figure 7.2 Speedup of OpenACC running on a Bulldozer AMD CPU when compared to OpenMP shared memory running on a Bulldozer AMD CPU using 8 threads

threads on the CPU. The OpenACC version outperformed the OpenMP 3.1 version. One of the reasons is that our OpenACC version inlines the inner loop (compared to the OpenMP 3.1 version) and provides more information to the vectorization phase of the compiler, including information about reductions. We also notice that when the problem size increases, the OpenACC improvement in terms of speedup becomes less profitable (from 1.49 to 1.06) as we saturate the memory bandwidth. The Cray 8.8.5 compiler did not support the mode of targeting OpenACC to multicore.

7.5 Summary

It is possible to achieve improved performance portability across architectures when you use OpenACC. Performance of OpenACC depends on the ability of the compiler to generate good code on the target platform. For example, we observed a significant performance variation when we compiled OpenACC with Cray 8.5.0 compared with 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 in the level of performance portability of a programming model. To write performance portable code, it is important to specify where the parallelism in the code is (e.g., via `#pragma acc loop`) without specifying clauses that affect how the parallelism is mapped to the architecture. For example, specifying hints such as the OpenACC `vector_length` can help achieve good performance on an architecture, but, at the same time, it can possibly hinder performance on another one. However, sometimes these hints are necessary when the compiler cannot efficiently map the parallelism to the target platform.

When we compiled OpenACC to multicore, which in this case corresponded to the shared-memory host CPU, the PGI compiler ignored the data region directives. These included any `#pragma acc data` directives or clauses that move data to or from the accelerator.

Not only is the `#pragma acc loop` directive extremely useful for performance portability (to specify parallelism for offloading to accelerators), but also, depending on the implementation, it can be critical if you are to achieve good levels of multithreading and vectorization. It can help compilers identify the parallelism in the code when they cannot figure it out automatically, and this is important for optimizations such as automatic vectorization. Compilers cannot yet consistently identify these opportunities in all cases, so hints must be used to ensure that vectorization is used where appropriate. Although GPUs do not have vector units, the `#pragma` `acc loop` directive can be used to help identify parallelism that can be mapped to grids, thread blocks, and potential very fine-grained parallelism that can be executed by SMT threads (e.g., GPU warps).

We noticed better performance when using OpenACC (multicore) versus OpenMP 3.1 baseline when running in an AMD Bulldozer processor using 8 cores using PGI 17.1. One of the possible reasons for this behavior is that OpenACC provides more information to the vectorization phase of the compiler, including information about reductions. Being able to specify another level of parallelism in OpenACC that maps to vector instructions was an advantage. At the time of this writing, the Cray compiler didn't allow the ability to generate OpenACC code that targets multicore processors.

7.6 Exercises

1. How many levels of parallelism can be specified using OpenACC?
2. Which of the following directives is the most performance portable?
   a. `#pragma acc parallel loop`
   b. `#pragma acc parallel loop gang`
   c. `#pragma acc parallel loop vector`
   d. `#pragma acc parallel loop vector vector_length(N)`
3. Which of the following memory models does OpenACC support? Describe any limitations that apply to those models that are supported.
   a. Shared memory
   b. Discrete memory
   c. Partially Shared Memory
4. Which of the following clauses can be used to tune for a specific architecture?
   a. Specifying a `vector_length()`
   b. Specifying the number of gangs, workers, or vectors
   c. `acc copyin` and `copyout`

   d. All of the above
   e. None of the above

5. Is an OpenACC compiler allowed to
   a. Ignore directives specified by the user
   b. Change values in directives provided by the user
   c. Both
   d. Neither