CHAPTER

Parallel programming with OpenACC 19

Jeff Larkin

CHAPTER OUTLINE

Now that we have learned to design and express parallel algorithms in CUDA C, we are in a strong position to understand and use parallel programming interfaces to rely on the compiler to do the detailed work. OpenACC is a specification of compiler directives and API routines for writing data parallel code in C, C++, or Fortran that can be compiled to parallel architectures, such as GPUs or multicore CPUs. Rather than requiring the programmer to explicitly decompose the computation into parallel kernels, such as is required by CUDA C, the programmer annotates the existing loops and data structures in the code so that an OpenACC compiler can target the

Programming Massively Parallel Processors. DOI: http://dx.doi.org/10.1016/B978-0-12-811986-0.00019-4

code to different devices. For CUDA devices, the OpenACC compiler generates the kernels, creates the register and shared memory variables, and applies some of the performance optimizations that we have discussed in the previous chapters. The goal of OpenACC is to provide a programming model that is simple to use for domain scientists, maintains a single source code between different architectures, and is performance portable, meaning that code that performs well on one architecture will perform well on other architectures. In our experience, OpenACC also provides a convenient programming interface for highly skilled CUDA programmers to quickly parallelize large applications. The main communication channel between the user and the compiler is the set of annotations on the source code. One can think of the user being a supervisor giving directions to the compiler as employees. Just like in any other managerial scenarios, having first-hand experience in the work that an employee does helps the manager to give better advice and directions. Now that we have learned and practiced the work that the OpenACC compiler does, we are ready to learn the effective ways to annotate the code for an OpenACC compiler.

19.1 THE OPENACC EXECUTION MODEL

The OpenACC specification was initially developed by CAPS Enterprise, Cray Inc., The Portland Group (PGI), and NVIDIA with support from multiple universities and national laboratories, but has since grown to include additional vendors, universities, companies, and labs. At the time of writing, the current version of the specification is version 2.5.

OpenACC has been designed to run on modern high-performance computing (HPC) systems, which generally include multicore CPUs and frequently include distinct parallel accelerators, such as GPUs. The programming model assumes that the program execution will begin on a *host* CPU which may offload execution and data to an *accelerator device*. The accelerator may in fact be the same physical device as the host, as is the case with multicore CPUs, or may be an attached device, such as a GPU that is connected to the CPU via the PCI Express (PCIe) bus. Additionally, the programming model allows for the host and device to have physically separate memories or a shared memory. As such, the most portable way to write OpenACC code is to assume a physically distinct accelerator with physically distinct memory, as it is simpler to map these assumptions back onto machines with shared memory or shared compute resources than to do the reverse. Fig. 19.1 shows the abstract machine model assumed by the OpenACC specification.

Fig. 19.2 illustrates this offloading execution model. By default, OpenACC enforces synchronous behavior between the host and accelerator device, where execution and requisite data are migrated from the host to the device and both return to the host upon completion of the OpenACC *region*. At the end of each parallel execution on the device, the host and device performs a synchronization unless the user removes the synchronization with an explicit annotation. This is, in many ways, similar to the fork/join behavior provided by traditional threaded programming models

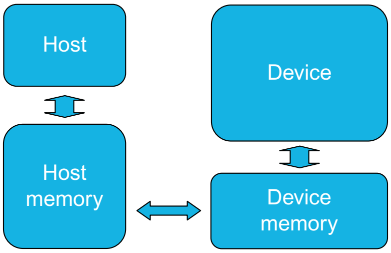


FIGURE 19.1

OpenACC abstract machine model.

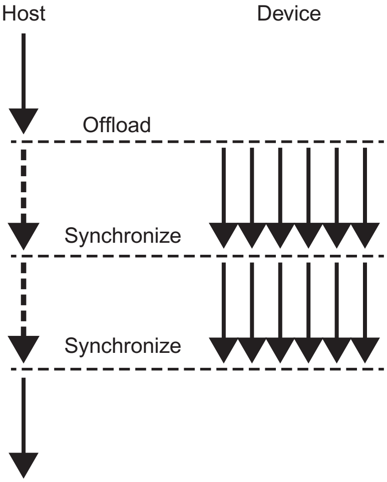


FIGURE 19.2

The OpenACC offloading execution model.

such as posix-threads, except that the forked threads may exist on a different device and the data required by those threads may need to be copied between two physical memories.

Since offloading computation may also require copying of data, which can be time consuming when the host and accelerator have physically separate memories, OpenACC also provides a means for controlling how data is moved between the host and device and how it is shared between different offloaded regions. Conceptually speaking, even though most common accelerator architecture at the time of writing has physically separate memories, OpenACC treats data as if there is always one quintessential copy of the data that lives either on the host or a device, and modifications in one place will at some point in time be reflected on the other too.

In other words, programmers cannot assume that they can modify the same data both on the host and the device at the same time, as the execution model makes no guarantees that the host and device will have physically distinct memories. On machines where the host and device share a memory, the program may behave in an unpredictable or incorrect way if both the host and device are allowed to modify the same memory. Likewise programmers should not assume that all architectures will support shared memory between the host and device; they should use the appropriate directives to synchronize host and device memory when necessary.

OpenACC exposes three levels of parallelism on the accelerator device: gangs, workers, and vectors. Gangs are fully independent execution units, where no two gangs may synchronize nor may they exchange data, except through the globally accessible memory. Since gangs work completely independently of each other, the programmer can make no assumptions about the order in which gangs will execute or how many gangs will be executed simultaneously. A CUDA C programmer should recognize the similarity between gangs and CUDA thread blocks.

Each gang contains one or more workers. Workers have access to a shared cache memory and may be synchronized by the compiler to ensure correct behavior. A CUDA C programmer should recognize the similarity between workers and CUDA threads. Workers operate on vectors of work. A vector is an operation that is computed on multiple data elements in the same instruction; the number of elements calculated in the instruction is referred to as the *vector length*. Additionally OpenACC allows loops to be run sequentially within any of these levels of parallelism.

Imagine a house that is getting its rooms painted. The painting company may send multiple groups of painters, assigning each group different rooms. Each group has its own bucket of paint, and painters within a group can easily talk to each other to plan how to paint their room, but in order for different groups to collaborate they would need to leave their rooms to discuss things with the other teams. Each painter has a roller or brush, the width of which roughly determines how much of the wall he or she can paint per stroke. In this example, the groups of painters represent OpenACC gangs, the painters represent OpenACC workers, the paint brushes or rollers represent OpenACC vectors, and the size of the brushes and rollers is the vector length. Achieving the best time to completion involves balancing the resources correctly among the levels of parallelism. Typically an OpenACC compiler will make a first, educated guess about how to balance these resources based on its knowledge of the target hardware and the information it has about the code, but the programmer is free to override these decisions to optimize the performance.

19.2 OPENACC DIRECTIVE FORMAT

The main difference between OpenACC and CUDA C is the use of compiler directives in OpenACC. OpenACC provides directives (pragmas in C and C++ or comment directives in Fortran) for offloading parallel execution, management data offloading, and optimizing loop performance. OpenACC programmers can often

```
// C or C++
#pragma acc <directive> <clauses>
{ … }

! Fortran
!$acc <directive> <clauses>
…
!$acc end <directive>
```

FIGURE 19.3

Basic format for OpenACC directives.

start with writing a sequential version and then annotate their sequential program with OpenACC directives. They leave most of the heavy lifting to the OpenACC compiler. The details of data transfer between host and accelerator memories, data caching, kernel launching, thread scheduling, and parallelism mapping are all handled by OpenACC compiler and runtime. The entry barrier for programming accelerators becomes much lower with OpenACC.

Fig. 19.3 illustrates the basic format for the OpenACC directives. In C and C++, the *#pragma* keyword is a standardized method to provide to the compiler information that is not specified in the standard language. This mechanism is used by many directive-based language extensions including OpenACC. In C and C++, OpenACC directives start with the sentinel "acc". The use of sentinels allows each compiler to only pay attention to the directives intended for that compiler. Fig. 19.3 also shows that in Fortran, the OpenACC directives start with "!$acc" to indicate that the directive is only of interest to an OpenACC compiler.

An OpenACC directive specifies the type of directive and sometimes additional clauses to provide more information. Several OpenACC directives and clauses will be demonstrated in the sections that follow. Most OpenACC directives are applied to blocks of code, often referred to as OpenACC regions, depicted as the code surrounded by the curly brackets. Some directives, particularly data management directives, are standalone, behaving much like a function call.

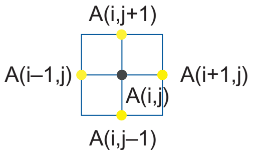
By supporting the use of directives on an existing code, OpenACC provides an incremental path for moving existing code to accelerators. This is attractive because adding directives disturbs the existing code less than other approaches. Some existing scientific applications are large and their developers cannot afford to rewrite them for accelerators. OpenACC lets these developers keep their applications looking like the original C, C++, or Fortran code, and insert the directives in the code where they are needed one place at a time.

The code of an OpenACC application remains correct when a compiler ignores the directives. Because OpenACC directives are implemented in pragmas, which are treated as comments by compilers that do not support these directives, the code can be compiled by other compilers and be expected to work correctly. This allows the application code to remain as a single source that can be compiled by various compilers. The ability to maintain a single source code with and without OpenACC is frequently one of the key reasons programmers choose OpenACC. OpenACC also specifies runtime API functions that can be used for device and memory management above and beyond what is possible in directives.

19.3 OPENACC BY EXAMPLE

OpenACC is best taught by example and for that reason this chapter will present the directives by applying them to a benchmark code. The benchmark code implements a Jacobi iterative method that solves the Laplace equation for heat transfer. As we have seen in Chapter 18: Programming a heterogeneous cluster, the Jacobi iterative method is a means for iteratively calculating the solution to a differential equation by continuously refining the solution until the answer has converged upon a stable solution or some fixed number of steps have completed and the answer is either deemed good enough or unconverged. The example code represents a 2D plane of material that has been divided into a grid of equally sized cells. As heat is applied to the outer edges of this plane, the Laplace equation dictates how the heat will transfer from grid point to grid point over time. Fig. 19.4 shows the problem that the example code solves. To calculate the temperature of a given grid point for the *next* time iteration, one simply calculates the average of the temperatures of the neighboring grid points from the current iteration. Once the *next* value for each grid point is calculated, those values become the current temperature and the calculation continues. At each step the maximum temperature change across all grid points will determine if the problem has converged upon a steady state.

Fig. 19.5 shows the example code that will be used in this chapter. The example code consists of a while loop (line 53) that carries out the Jacobi iteration. This loop will end if either the maximum change reaches below a set threshold (i.e., convergence) or a fixed number of iterations have completed. All performance results in this chapter were obtained by running for 1000 iterations. The while loop contains two loop nests (lines 55 and 64), the first of which calculates the Laplace equation to determine each cell's next temperature and the second copies the next values into the working array for the next iteration. The array copy loop is frequently replaced with pointer manipulation (ping-pong buffering, Chapter 12: Parallel patterns, graph search) when implemented in production science codes, but writing it as a data copy simplifies the example code.



$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

FIGURE 19.4

A Laplace equation example.

```
  while ( err > tol && iter < iter_max ) {
    err=0.0;
    for( int j = 1; j < n-1; j++) {
      for(int i = 1; i < m-1; i++) {

        Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                          A[j -1][i] + A[j+1][i]);

        err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

FIGURE 19.5

Jacobi Iterative Method example code.

The reader might notice that the structure of the example code is actually quite similar to that of the while-loop of the `BFS_sequential` function in Fig. 12.7. Indeed, the while-loop in Fig. 12.7 checks for a convergence condition. The roles of the `c_frontier` and `p_frontier` arrays are actually switched through pointer manipulation. Interested readers can apply the same idea of pointer manipulation to the current code example to make it more efficient.

THE OPENACC KERNELS DIRECTIVE

The simplest method for accelerating loops with OpenACC is the *kernels* directive. This directive informs the compiler of the programmer's desire to accelerate loops within a given region, but places the responsibility on the compiler to identify which loops can be safely parallelized and how to do so. Simply put, it instructs the compiler that the region that follows contains interesting loops that should be transformed into one or more accelerator kernels. As discussed in previous chapters, a kernel is a function that performs independent operations on different parts of the data, thus allowing these operations to be run in parallel. In a more explicit programming paradigm, such as CUDA or OpenCL, it would be the programmer's responsibility to decompose the work into parallel operations in each kernel, but OpenACC places this burden on the compiler and allows the programmer to maintain the existing loop structure.

Notice in Fig. 19.6 that a *kernels* directive is added at line 55, which informs the compiler that the code block from lines 56 to 72 contains loops that should be analyzed and considered for acceleration. The compiler will analyze this region of code,

```
53.     while ( err > tol && iter < iter_max ) {
54.       err=0.0;
55.   #pragma acc kernels
56.   {
57.       for( int j = 1; j < n-1; j++) {
58.         for(int i = 1; i < m-1; i++) {
59.
60.           Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
61.                                   A[j-1][i] + A[j+1][i]);
62.
63.           err = max(err, abs(Anew[j][i] - A[j][i]));
64.         }
65.     }
66.
67.     for( int j = 1; j < n-1; j++) {
68.       for( int i = 1; i < m-1; i++ ) {
69.         A[j][i] = Anew[j][i];
70.       }
71.     }
72.   }
73.     iter++;
74.   }
```

FIGURE 19.6

Example code with OpenACC kernels directive.

looking for loops that are free of data dependencies (one loop iteration that depends upon the results of another) to parallelize, and determining what arrays would need to be transferred if run on a device with a discrete memory. In addition to determining which loops are candidates for acceleration and how to decompose the loops into parallel kernels, it is the compiler's responsibility to identify the data used within those loops and to migrate the data to and from the accelerator device if necessary.

The code in Fig. 19.6 can be built with any OpenACC compiler, but for the purpose of this example the PGI compiler, version 16.4 will be used, targeting an NVIDIA Tesla GPU. OpenACC acceleration is enabled with the −*ta* = *tesla* compiler option, and because we would like to understand how the compiler transforms the code for the accelerator, the −*Minfo* =*all* compiler option is added. The compiler output is in Fig. 19.7.

The compiler output informs us that the compiler found parallelizable loops at lines 57, 59, 67, and 69 of the example code. Additionally it tells us that accelerator kernels were generated for the two loop nests, even showing that the loops at lines 57 and 67 were distributed to gangs and the loops at lines 59 and 69 were distributed across gangs and vectorized with a vector length of 128. Note the lack of a worker loop implies that each gang has just one worker. Lastly the output shows that at line 55 the compiler implicitly generated directives to offload the A and Anew arrays to

```
$ pgcc -fast -ta=tesla -Minfo=all laplace2d.c
main:
     40, Loop not fused: function call before adjacent loop
         Generated vector sse code for the loop
     51, Loop not vectorized/parallelized: potential early exits
     55, Generating copyout(Anew[1:4094][1:4094])
         Generating copyin(A[:][:])
         Generating copyout(A[1:4094][1:4094])
         Generating Tesla code
     57, Loop is parallelizable
     59, Loop is parallelizable
         Accelerator kernel generated
         57, #pragma acc loop gang /* blockIdx.y */
         59, #pragma acc loop gang, vector(128) /* blockIdx.x
threadIdx.x */
         63, Max reduction generated for error
     67, Loop is parallelizable
     69, Loop is parallelizable
         Accelerator kernel generated
         67, #pragma acc loop gang /* blockIdx.y */
         69, #pragma acc loop gang, vector(128) /* blockIdx.x
threadIdx.x */
```

FIGURE 19.7

Compiler output from example kernels code.

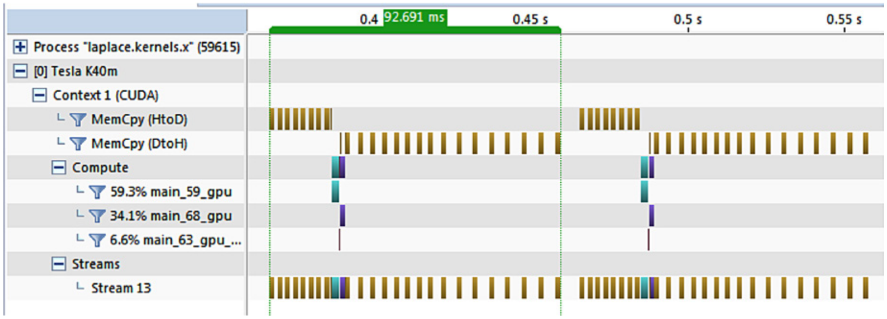


FIGURE 19.8

GPU timeline of kernels directive example code.

the device and back to the host. More information about this data offloading follows. Executing this code on a benchmark machine containing an Intel Xeon(R) CPU E5-2698 v3 CPU and NVIDIA K40 GPU, we see in Fig. 19.11 that even though our loops are now running as kernels on the GPU, the runtime benchmark actually slowed down. This is due to the compiler being overly cautious about the movement of the two arrays, something that we will correct later in this chapter. The PGProf profiler that comes with the PGI compiler can be used to generate a timeline of the program execution, which shows that at each iteration of the method our arrays are copied to and back from the GPU (MemCpy (HtoD) and Memcpy(DtoH)), requiring more time than the actual kernel execution, as shown in Fig. 19.8.

THE OPENACC PARALLEL DIRECTIVE

OpenACC provides an alternative, more programmer-driven approach to writing parallel code: the *parallel* directive. Whereas the programmer only declares a desire for loops to be accelerated when using the *kernels* directive, when the parallel directive is used the programmer declares that the compiler should generate parallelism and when combined with the *loop* directive, makes assertions about the feasibility of loops for acceleration without requiring detailed analysis by the compiler.

Compilers are still required to determine the data requirements for the loops and make decisions about how best to parallelize the loop iterations to the targeted hardware, but it is the programmer's responsibility to determine and assert that the loops are able to be parallelized; if the programmer asserts incorrectly, then it is his/her fault if wrong answers result. The parallel directive is usually paired with the loop directive, with the former indicating that the compiler should generate parallelism on the device and the latter specifying that the iterations of the loop that follow should be mapped to that parallelism.

Unlike the *kernels* directive, which encloses a region that may hold lots of loops to be accelerated, the *parallel* and *loop* directives are added to each *loop* nest that is to be accelerated. The *parallel* directive generates a parallel kernel, which redundantly executes the containing code until a *loop* directive is reached, which will then parallelize the affected loop. These two directives are frequently used together on the same pragma. Fig. 19.9 shows the same benchmark code using the parallel and loop

```
  while ( err > tol &&  iter < iter_max ) {
    err=0.0;
#pragma acc parallel loop reduction(max:err) collapse(2)
    for( int j = 1; j < n -1; j++) {
      for(int i = 1; i < m -1; i++) {

        Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                             A[j-1][i] + A[j+1][i]);

        err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }
#pragma acc parallel loop collapse(2)
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
}

  iter++;
}
```

FIGURE 19.9

Jacobi Iterative Method code using parallel directive.

directives. The directives at lines 55 and 65 generate parallelism on the accelerator, which will be turned into accelerator kernels.

Additionally, by using the *collapse* clause the programmer has declared that, not only the outer loop is free of data races and available to parallelize, but the directive should apply to the inner loop too. It should be noted that the collapse clause can only be used on tightly nested loops (nested loops with no code in between, see Chapter 14: Application case study - non-Cartesian MRI), but the loop directive can be added to individual loops to declare the independence of loop iterations when collapse cannot be used. Whenever collapse can be used, the compiler can potentially generate a multi-dimensional kernel like the one generated for the calculation of electrostatic potential energy at the 2D energy grid in Chapter 15, Application case study—molecular visualization and analysis.

The first loop nest has one small complication that needs to be addressed, however: the calculation of the maximum error. Each iteration of the loop will calculate its own error value based on the difference between the value at the current iteration and the next iteration. Not all values of error are needed, however, only the maximum of all errors is needed. This is known as a *reduction*, meaning the `(n-2)*(m-2)` different values for error are reduced down to one by using the max operation to choose which value to return (see Chapter 5: Performance considerations). While some compilers will detect the existence of this reduction, it is best that the user specify the reduction to be sure.

In Fig. 19.10, we once again see that the compiler has generated accelerator kernels and data motion for the code. Careful examination shows that the compiler has

```
$ pgcc -fast -ta=tesla -Minfo=all laplace2d.c
main:
     41, Loop not fused: function call before adjacent loop
         Loop not vectorized: may not be beneficial
         Unrolled inner loop 4 times
         Generated 3 prefetches in scalar loop
     52, Loop not vectorized/parallelized: potential early exits
     56, Accelerator kernel generated
         Generating Tesla code
         56, Generating reduction(max:error)
         57, #pragma acc loop gang, vector(128) collapse(2) /*
blockIdx.x threadIdx.x */
         59,   /* blockIdx.x threadIdx.x collapsed */
     56, Generating copyout(Anew[1:4094][1:4094])
         Generating copyin(A[:][:])
     67, Accelerator kernel generated
         Generating Tesla code
         68, #pragma acc loop gang, vector(128) collapse(2) /*
blockIdx.x threadIdx.x */
         70,   /* blockIdx.x threadIdx.x collapsed */
     67, Generating copyin(Anew[1:4094][1:4094])
         Generating copyout(A[1:4094][1:4094])
```

FIGURE 19.10

Compiler feedback for Jacobi Iterative Method using parallel directive.

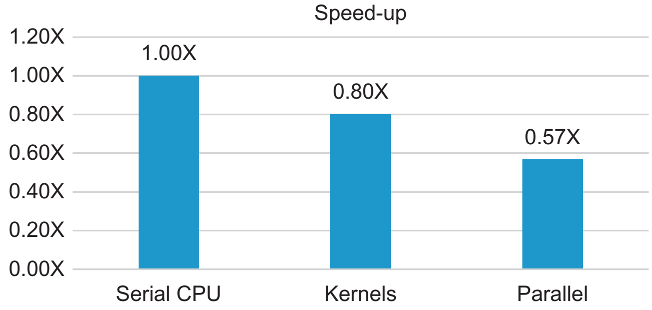


FIGURE 19.11

Performance speed-up from OpenACC kernels and parallel (higher is better).

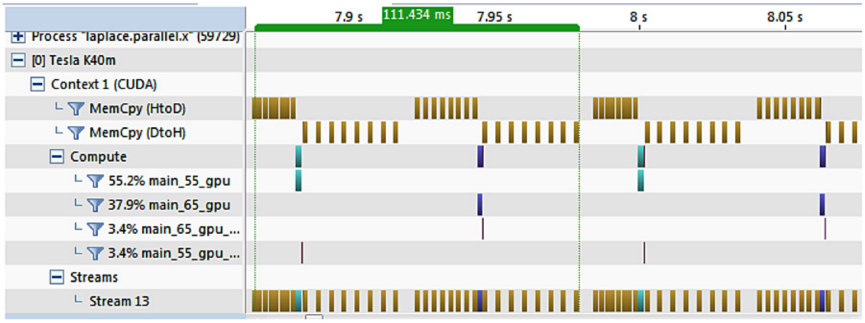


FIGURE 19.12

GPU timeline of parallel loop code.

generated data movement directives at the beginning and end of each parallel loop (lines 56 and 67), resulting in twice as much data motion as the kernels version. Fig. 19.11 shows that the performance of this version of the code is slower than the kernels version, due to the additional data transfers. Fig. 19.12 shows a portion of the GPU timeline for this run. A comparison between Figs. 19.8 and 19.12 shows additional data motion between successive kernel calls in Fig. 19.12. It is obvious that whether the kernels directive or the parallel directive is used to accelerate the loops, the programmer will need to improve the data movement of the code to obtain higher performance.

COMPARISON OF KERNELS AND PARALLEL DIRECTIVES

Some may wonder why OpenACC needs both the kernels and parallel directives, since they are so similar. The kernels directive requires a lot of work from the compiler to determine whether the contained loops are safe and profitable to parallelize. It can also be used around large blocks of codes to generate many compute kernels from the contained loops. The parallel directive, on the other hand, requires analysis by the programmer to be sure that the affected loops are both safe and profitable to accelerate. When the programmer uses the parallel directive the compiler is expected to obey, whether the programmer is correct or not. Parallel also requires the programmer to annotate each loop whose iterations should be executed in parallel by the accelerator.

The loop directive highlights the differences between the two directives fairly nicely. The loop directive has quite a few potential clauses, but two of them in particular highlight the differences between kernels and parallel: *auto* and *independent*. The *auto* clause, which is implicitly added to loop directives used in *kernels* regions, informs the compiler that the loop is interesting, but that the compiler should analyze it to automatically determine whether the loop can and should be parallelized.

The *independent* clause, which is implicitly added to loop directives used within *parallel* regions, asserts to the compiler that all iterations of the loop are independent (free of data dependencies) with regards to each other, so the compiler need not analyze them. The loop independent clause is frequently used to override compiler decisions when a compiler incorrectly thinks that a loop has data dependencies, but because it is implied or assumed on loops within parallel regions, it is the user's responsibility to ensure correct parallelization when the parallel directive is used. If the code gives wrong answers when parallelized using *kernels*, it is a compiler bug, but if it gives wrong answers with the *parallel* directive it very well may be a programmer bug.

So far, we have used the parallel and loop directives together. When the parallel directive is used by itself, one must be aware of an important detail. The statement region that follows will be executed *redundantly in parallel* except for any explicitly marked loop regions. The loops in the loop regions will not be executed redundantly; their iterations will be executed in parallel. This is the behavior when a region is annotated with both parallel and loop.

OPENACC DATA DIRECTIVES

When accelerating loops the compiler will always do what it believes will be necessary to ensure correct results based on the limited information it has, which is typically limited to what it can see in the current function or the current source file. As a general rule of thumb, this means that if a variable appears to the right of an assignment ("=") it will be copied to the accelerator, and if it appears to the left of an assignment it will be copied back. The programmer will generally have a better understanding of the big picture of an application, particularly how data is used between functions, so by providing more information to the compiler it is often possible to substantially reduce the cost of data movement compared to the compiler's choices.

In the above examples we observed that the compiler is copying the A and Anew arrays at the beginning and end of each of the OpenACC regions, since it believes that any variable that is changed on the accelerator may be needed later on the host, and any data used on the accelerator could have been changed on the host, so it must be refreshed to ensure correctness. Looking more closely at the code, the programmer should observe that A and Anew are not changed between successive iterations of the while loop, nor are they changed between the for loop nests. In fact, the only time A needs to be copied to or from the accelerator is at the beginning and end of the while loop. What may not be obvious from the abbreviated code above is that Anew is declared within the scope of this function, meaning that it is really just a temporary array that need not be copied at all; it only needs to exist on the device to be used as a temporary scratchpad. Given this, it is possible for the programmer to reduce data movement significantly by overriding the compiler's data movement with a more optimized scheme.

OpenACC's data directives and clauses enable the programmer to express the appropriate data motion to the compiler. The *data* region works much like the kernels and parallel regions, in that it identifies a block of code and augments the movement of data for the lifetime of that region. Additional information is given to the compiler through the use of *data clauses*. These clauses control the allocation and deletion of space on the accelerator and also the movement of data at the beginning and end of the region. Fig. 19.13 lists the five most common data clauses and their meanings.

It should be added that as of OpenACC 2.5 each of these data clauses first checks whether the variable is already present on the device and only does the specified action for variables that are not already on the device. The OpenACC runtime keeps a *reference count* of each variable, only performing memory actions when the reference count for the variable increases from 0 to 1 or decreases from 1 to 0. The reference count is kept for the base address of the variable, meaning that there is only one reference for an entire array. The reference count atomically increments by 1 at the beginning of data regions and decrements by 1 at the end of data regions. Copies from the host to device only occur when the count for a variable

Create	Allocate space for the listed variable on the accelerator device at the beginning of the region and delete the space at the end.
Copyin	Create the listed variables on the device, then copy the values of that variable into the device variable at the beginning of the region. The space will be deleted at the end of the region.
Copyout	Create the listed variables on the device, then copy the values of that variable from the device variable at the end of the region. The space will be deleted at the end of the region.
Copy	Behaves like a combined copyin and copyout.
Present	Declares that the variables can be assumed to already exist on the device, so no allocation, deletion, or data movement is necessary.

FIGURE 19.13

Five common data clauses and their meanings.

C/C++	clause(start:count), start may be excluded if starting at 0
Fortran	clause(start:end), start or end maybe excluded if they are the beginning or end of the array

FIGURE 19.14

Data clause array size notation.

```
#pragma acc data create(Anew[:n][:m]) copy(A[:n][:m])
while ( err > tol && iter < iter_max ) {
    err=0.0;
#pragma acc parallel loop reduction(max:error) collapse(2)
    for( int j = 1; j < n-1; j++) {
      for(int i = 1; i < m-1; i++) {

        Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1]+
                             A[j-1][i] + A[j+1][i]);

        err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }
#pragma acc parallel loop collapse(2)
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

FIGURE 9.15

Jacobi Iterative Method with data region.

is incremented to 1 and copies from the device only occur when the count for a variable is decremented from 1 to 0. Frequently the programmer must inform the compiler of the size and shape of array variables, particular in C and C++, where arrays are simply pointers to memory. This is achieved using the syntax shown in Fig. 19.14, where start gives the beginning index of the array and count gives the number of elements in the array.

At times the compiler may be able to determine the size and shape of arrays based on the loop bounds, but it is generally a best practice to provide this information to ensure that the compiler uses the correct information. Fig. 9.15 shows the earlier parallel loop code with a data region applied to the convergence loop. With the data directives, the compiler will simply follow these directives to generate the specified data movements for A and Anew, rather than conducting its own analysis and inserting its own data movements. The same modification can be made to the kernels version, resulting in the same reduction in data movement costs.

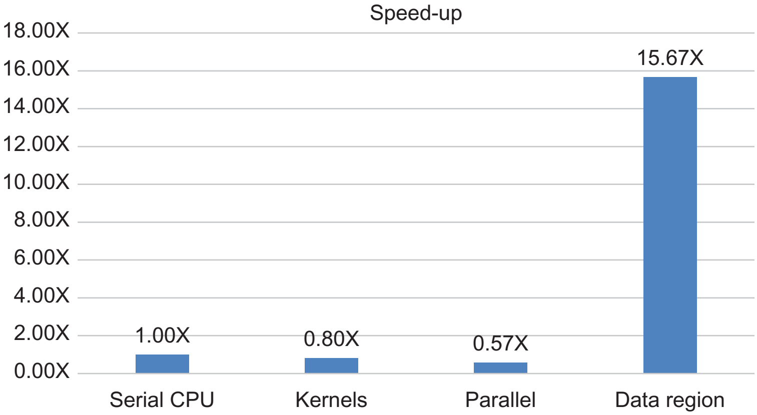


FIGURE 19.16

Speed-up with addition of data directive (higher is better).

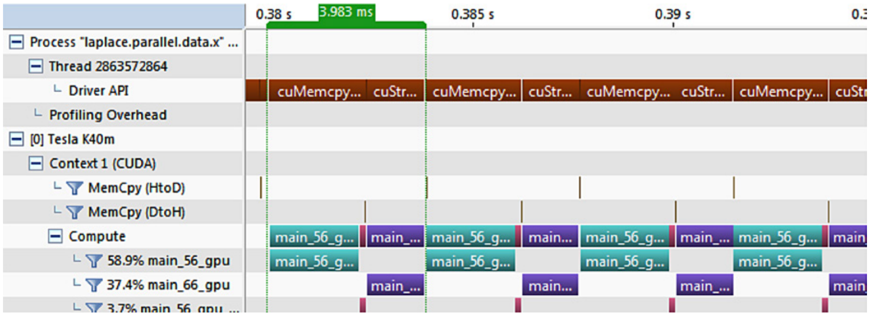


FIGURE 9.17

GPU timeline after optimizing data motion.

Rebuilding and rerunning the code with this optimization added results in the speed-up shown in Fig. 19.16. This result demonstrates the importance of using data regions in OpenACC kernel and parallel regions.

Fig. 9.17 shows the new PGProf GPU timeline, demonstrating that the A and Anew arrays are no longer copied between iterations. Note that there is still a small amount of data copy; the value for error still needs to be copied so that it can be used in evaluating for convergence.

Because data regions can only be applied to structured blocks of code, the *data* directive is not always usable, particularly in the case of C++ classes, where data is frequently allocated in a constructor, deallocated in a destructor, and used elsewhere. In these situations the unstructured *enter data* and *exit data* directives allow data to be managed anywhere in the code. Fig. 19.18 demonstrates use of *enter data* and *exit data* in a C++ class constructor and destructor, respectively.

```
template <class ctype> class Data
{
  private:
/// Length of the data array
     int len ;
/// Data array
     ctype *arr;

  public:
       /// Class constructor
       Data(int length)
            {
              len = length;
              arr = new ctype[len];
       #pragma acc enter data create(arr[0:len])
            }
       /// Class destructor
            ~Data()
            {
       #pragma acc exit data delete(arr)
              delete arr;
              len = 0;
             }
       }
```

FIGURE 19.18

Example of unstructured data directives in C++ class.

Unstructured data directives take data clauses, like structured data regions, but due to their unstructured nature the list of available data clauses is slightly different. Fig. 19.19 shows the data clauses that can be used on each directive and their meanings. As with their structured counterparts, these data clauses implement a reference count, where the reference count for a variable is incremented by *enter data* and decremented by *exit data*. Creation of device variables and copying data to the device only occurs when a reference count is incremented from 0 to 1 and the copying data back from the device to the host and deletion of device variables only occurs when the reference count is decremented from 1 to 0. The one exception to this rule is the delete clause, as explained in Fig. 19.19.

It would be impractical to require users to create or destroy device variables each time it is necessary to perform data copies, so OpenACC also provides an *update* directive for copying data to or from the accelerator device. The update directive is used to make the device and host copies of a variable, when on machines with distinct host and device memories, coherent with each other. On machines with shared memories between the host and device the runtime is allowed to ignore update directives. The update directive is analogous to the various `cudaMemcpy` function calls in CUDA. To update the device memory, the *update device* clause is given, describing

Enter data	Create	Allocate space for the listed variable on the accelerator, but do not initiate any data transfer. Increments reference count.
	Copyin	Create the listed variables on the device, then copy the values of that variable into the device variable. Increments reference count.
Exit data	Copyout	Copy the values of that variable from the device variable and delete the device copy. Decrements reference count.
	Delete	Immediately set the reference count to 0 and remove the device copy of the variable without any data transfer.
	Release	Decrement the reference count for the variable and behave as a delete if the reference count is decremented to zero.

FIGURE 19.19

Data clauses for unstructured data directives.

```
#pragma acc update host(u_new[offset_first_row:m-2],u_new[offset_last_row:m-2])
MPI_Sendrecv(u_new+offset_first_row, m-2, MPI_DOUBLE,
               t_nb, 0, u_new+offset_bottom_boundary, m-2,
               MPI_DOUBLE, b_nb, 0,
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Sendrecv(u_new+offset_last_row, m-2, MPI_DOUBLE,
               b_nb, 1, u_new+offset_top_boundary, m-2,
               MPI_DOUBLE, t_nb, 1,
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
   #pragma acc update device(u_new[offset_top_boundary:m-2],u_new[offset_bottom_boundary:m-2])
```

FIGURE 19.20

Example of update directive with MPI halo exchange.

the variable or subsection of a variable that should be updated. When updating the *host* copy, the *self* clause (formerly called *host*) is used instead. Fig. 19.20 shows the use of update directives around an MPI halo exchange (see Chapter 18: Programming a heterogeneous cluster) of the top and bottom rows of a local array, where the host copies of the halo rows are first updated, then exchanged with neighboring processors, and finally the device copy is updated with the new values.

OPENACC LOOP OPTIMIZATIONS

While the OpenACC compiler will make a best effort to optimize the code for the target device, it is frequently possible for the developer to override the compiler's decisions and obtain higher performance. The OpenACC *loop* directive, which has already been discussed in the context of the *parallel* directive, enables the developer to suggest optimizations for particular loops, such as how to better decompose the loop iterations for the accelerator device. The loop *auto*, *independent*, and *collapse* clauses have been discussed previously, so they will not be discussed further in this section.

The first set of loop optimization clauses are the *gang*, *worker*, *vector*, and *seq* clauses. These clauses inform the compiler that the loop immediately following the loop directive should have the listed forms of parallelism applied to it. For instance, in Fig. 19.21, the *l* loop is distributed to gangs, the *k* loop to workers, the *j* loop is run sequentially, and the *i* loop is vectorized. In general gang loops are found at the outermost levels of loop nests and vector loops at the innermost levels, where data is accessed in a contiguous manner. Worker and vector levels are optionally used as needed in between these levels.

In addition to specifying how loops are decomposed it is sometimes useful to specify the number of gangs or workers or the vector length used. When using a *parallel* directive, these parameters are provided at the beginning of the region on the parallel directive, as shown in Fig. 19.22. When using the *kernels* directive these parameters can be provided on the loops themselves, as shown in Fig. 19.23. OpenACC 2.5 loosens these restrictions to allow either format to be used on both *parallel* and *kernels* regions. Any parameter not specified will be selected by the compiler.

```
#pragma acc parallel loop gang
for (int l=0; l < N; l++)
#pragma acc loop worker
  for (int k=0; k < N; k++ )
#pragma acc loop seq
    for (int j=0; j < N; j++ )
#pragma acc loop vector
      for (int i=0; i < N; i++)
        { … }
```

FIGURE 19.21

Example of loop directive specifying levels of parallelism.

```
#pragma acc parallel loop gang num_gangs(1024) num_workers(32)
vector_length(32)
for (int l=0; l < N; l++)
#pragma acc loop worker
  for (int k=0; k < N; k++ )
#pragma acc loop seq
    for (int j=0; j < N; j++ )
#pragma acc loop vector
      for (int i=0; i < N; i++)
        { … }
```

FIGURE 19.22

Adjusting loop parameters within a parallel region.

```
1.  #pragma acc kernels loop gang(1024)
2.  for (int l=0; l < N; l++)
3.  #pragma acc loop worker(32)
4.    for (int k=0; k < N; k++ )
5.  #pragma acc loop seq
6.      for (int j=0; j < N; j++ )
7.  #pragma acc loop vector(32)
8.        for (int i=0; i < N; i++)
9.           { … }
```

FIGURE 19.23

Adjusting loop parameters within a kernels region.

When specifying loop parameters, which are inherently device-specific, it is generally a best practice to use a device_type clause to specialize the parameters to only a particular device. For instance, to only set the vector length for NVIDIA GPUs, line 7 of Fig. 19.23 could be changed to *acc loop device_type(nvidia) vector(32)*. Using the device_type clause informs the compiler of optimizations for specific devices without making the code less portable to other devices, where the user may not have optimal values.

One more notable optimization clause for loops is the *tile* clause, which specifies the two or more tightly nested loops that follow should be broken into tiles of work to exploit the locality of their data access pattern. As we discussed in Chapter 4, Memory and data locality, tiling is a technique that involves introducing additional loops to a loop nest to change the order of loop iterations to take advantage of localized data access patterns. This transformation could be performed by the developer, but often makes the code less readable and more difficult to maintain, so it is desirable to ask the compiler to perform the transformations instead.

The Jacobi Iterative Method example belongs in the convolution parallel pattern (see Chapter 7: Parallel patterns: convolution) and is a good candidate for tiling, since each iteration accesses its neighbor values, which may already exist in cache or registers. Fig. 19.24 shows the Jacobi Iterative Method code with the two loop nests broken into 32× 4 tiles on NVIDIA devices, which was experimentally determined to be the best value on the benchmark machine, giving roughly a 10% performance improvement over the previous version. Although the *parallel loop* version is shown here, the same optimization can be applied to the *kernels* version for a comparable speed-up.

OPENACC ROUTINE DIRECTIVE

Because the OpenACC compiler parallelizes loops based on the information available to it at compile time, function calls within OpenACC parallel or kernels regions can be problematic for a compiler. In fact, OpenACC 1.0 explicitly disallowed function or subroutine calls within OpenACC code regions unless the compiler was able to inline the function. OpenACC 2.0 removed this restriction, but requires that the

```
      while ( err > tol && iter < iter_max ) {
        err=0.0;
    #pragma acc parallel loop reduction(max:err)
device_type(nvidia) tile(32,4)

        for( int j = 1; j < n-1; j++) {
          for(int i = 1; i < m-1; i++) {

            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                                 A[j-1][i] + A[j+1][i]);

            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
      }
    #pragma acc parallel loop device_type(nvidia) tile(32,4)
      for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
          A[j][i] = Anew[j][i];
        }
      }
    }

      iter++;
    }
```

FIGURE 19.24

Jacobi Iterative Method code with loop tile clause.

```
#pragma acc routine seq
unsigned char mandelbrot(int Px, int Py);}
```

FIGURE 19.25

Example of the routine directive.

programmer update the function declaration with more information about how the function will be used.

The *routine* directive is used to essentially reserve certain levels of parallelism for loops within that function so that the compiler will know the levels of parallelism available to use on loops that call the function. The *routine* directive is placed at the function declaration, typically in a header file in C or C++ codes or module in Fortran codes, and accepts the same *gang*, *worker*, *vector*, and *seq* clauses as the loop directive. In the case of nested function calls, it is the programmer's responsibility to annotate each function in the call tree with the highest level of parallelism use in that function.

Fig. 19.25 shows the declaration for a function *mandelbrot*, which contains no parallelism, thus it is a seq function. By placing this declaration in the header file, the compiler knows when it encounters the source for the mandelbrot function that it must build a sequential version of the function for the target device, and also when it encounters the callsite for the function that it can rely on a sequential device version to be available.

ASYNCHRONOUS COMPUTATION AND DATA

All of the OpenACC directives shown thus far operate synchronously with the host, meaning that the host CPU will wait for the accelerator operation to complete before proceeding. Defaulting to synchronous behavior ensures correctness, but means that at most, one system resource (CPU, Accelerator, PCIe bus) can be busy at any given time. By opting into asynchronous behavior it is possible to concurrently use any or all of the system resources, improving overall application performance.

For instance, the earlier example saw a significant performance boost by reducing data copies to a bare minimum, but what if the time spent copying data could be reduced further by overlapping the data transfer with other, unrelated computations? Eventually data must be copied, but data copied while computation is also occurring is essentially free.

OpenACC *parallel*, *kernels*, and *update* directives accept an async clause, which informs the runtime that the operation should be sent to the accelerator, but the host CPU should continue working as soon as this has happened. This means that the CPU can either enqueue more work for the accelerator, placing the operations in an *asynchronous work queue* or even perform its own calculations on other data. When operating on an NVIDIA device, work queues directly map to CUDA streams. Before the CPU uses data that has been sent to the device asynchronously, it will need to synchronize using the *wait* directive. Fig. 19.26 shows an example using the *async* and *wait* directives.

While being able to perform work on the host and accelerator concurrently is a powerful feature, it becomes even more powerful when using multiple asynchronous work queues to overlap independent data transfers and computation on the accelerator as well. Just like when working with CUDA streams (see Chapter 18: Programming a heterogeneous cluster), work placed in the same queue is processed sequentially in the order it was enqueued, but work placed in different queues can be overlapped. On a high-end NVIDIA GPU machine, this means that the PCIe bus can be copying data in each direction while the host CPU and GPU are both performing

```
#pragma acc data create(A[N])
{
#pragma acc parallel loop async
for (int i=0; i<N; i++) A[i] = 1;
#pragma acc update host(A[:N]) async
for (int j=0; j<N; j++) B[j] = 2;
#pragma acc wait
for (int k=0; k<N; k++) C[k] = A[k] + B[k];
}
```

FIGURE 19.26

Example of async and wait.

```
#pragma acc data create(A[WIDTH*HEIGHT])
for(int block = 0; block < num_blocks; block++ ) {
  int start = block * (HEIGHT/num_blocks),
  end  = start + (HEIGHT/num_blocks);
#pragma acc update
device(A[block*block_size:block_size]) async(block%3)
#pragma acc parallel loop async(block%3)
  for(int y=start;y<end;y++) {
    for(int x=0;x<WIDTH;x++) {
      A[y*WIDTH+x]=x*y;
    }
  }
#pragma acc update
self(A[block*block_size:block_size]) async(block%3)
  }
 #pragma acc wait
```

FIGURE 19.27

Example of pipelining with async and wait.

computations. Such a process requires significant care by the developer to implement, but can result in significant performance gains.

In order to exploit different work queues, both the async and wait keywords accept an optional integer parameter to denote the queue number. If the async clause does not have a parameter, work will go into the default queue. If wait does not have a parameter, it will wait on all previously submitting asynchronous work on the current device. Fig. 19.27 demonstrates using three queues to pipeline blocks of work, thus overlapping all by the first and last data transfer.

19.4 COMPARING OPENACC AND CUDA

Since both OpenACC and CUDA can be used to accelerate applications on GPUs it is natural to wonder why both approaches are necessary. CUDA is a low-level approach to parallelizing a code for GPUs, which requires the developer to explicitly decompose the work into parallel parts and map the parallel parts to the GPU resources. OpenACC, on the other hand, is designed to express the parallelism of the code at a high enough level that compilers can parallelize the application to any parallel hardware. Each of these approaches has its own tradeoffs.

PORTABILITY

In terms of portability, OpenACC is generally considered the more portable approach to writing parallel code. CUDA is supported on only NVIDIA GPUs and thus requires maintaining both a host CPU and GPU version of the code. Any bug fixes or new capabilities need to be implemented both in the CPU and CUDA versions of the application. OpenACC on the other hand requires just one version of the code, which can be built for the CPU, GPU, or any other architecture supported by the compiler without changes. The ability to run a single source code across a wide range of architectures is OpenACC's most important feature to many HPC software developers, as it greatly reduces software development and maintenance costs and allows the code to run at any supercomputing center. Additionally, through use of the *device_type* clause, optimizations made for particular architectures do not affect portability, since they do not affect other architectures. In contrast, CUDA may require differently optimized kernels for different generations of GPUs.

PERFORMANCE

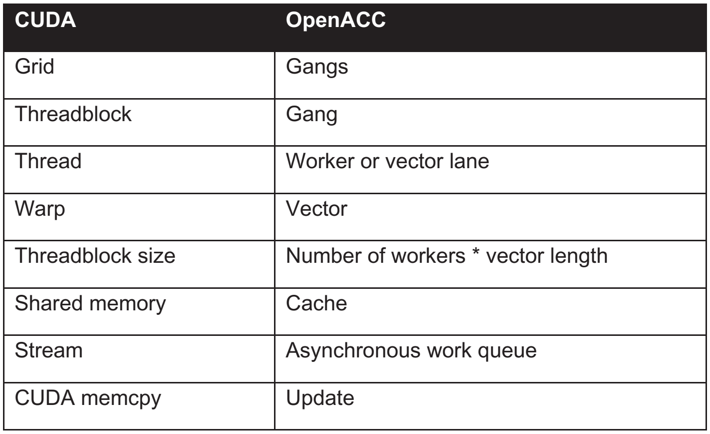
Because OpenACC is designed to run across a variety of architectures, it represents only architecture characteristics that are common everywhere. As such, there are certain optimizations that simply cannot be applied by the developer when using OpenACC. For instance, many shared memory optimizations that are commonly applied in CUDA are difficult or even impossible to express using OpenACC directives. CUDA however is a low-level approach to programming that closely follows new features in NVIDIA GPUs. Experienced programmers can achieve near assembly-level performance when writing CUDA kernels. When absolute performance on a given GPU is critical, CUDA is the more appropriate programming model of the two.

SIMPLICITY

OpenACC's primary target audience is domain scientists, many of whom have learned only enough computer programming to express their algorithms in code. Frequently these developers do not have the programming background and/or time required to explicitly parallelize their algorithms using a lower-level programming model, such as CUDA. OpenACC enables these users to maintain the familiar coding style of loops and arrays while still parallelizing the code for modern GPUs. By simplifying data management to eliminate the need for device and host arrays and transforming loops automatically into GPU kernels, OpenACC is often simpler for new users and domain scientists to learn. Nevertheless, having learned the concepts of the optimizations in CUDA often helps an OpenACC user to be much more effective.

As is always the case when choosing a programming model, it is up to the developer to choose which programming model best fits their project and skillset. Fortunately, both programming models are able to co-exist in the same application, as discussed in the next section.

When evaluating different programming models on the same hardware, developers often find it necessary to translate the concepts and terminology of each model to the other. Some concepts, such as gangs and workers, have clear 1:1 correspondence, while others, such as vectors, can be a bit murkier. Fig. 19.28 presents a commonly accepted translation between CUDA and OpenACC terminology.



CUDA	OpenACC
Grid	Gangs
Threadblock	Gang
Thread	Worker or vector lane
Warp	Vector
Threadblock size	Number of workers * vector length
Shared memory	Cache
Stream	Asynchronous work queue
CUDA memcpy	Update

FIGURE 19.28

Table of CUDA and OpenACC terminology.

19.5 INTEROPERABILITY WITH CUDA AND LIBRARIES

As noted in the previous section, choosing to use OpenACC does not preclude the use of CUDA. In fact, the most productive strategy for accelerating an application to a GPU may be to combine the use of accelerated libraries, such as cuBLAS, CUDA, and OpenACC in the same application. This approach gives the best of all worlds, leveraging available libraries, rapid development with OpenACC, and best performance on key kernels with CUDA. For a more complete survey of ways to mix OpenACC with other programming models, refer to the following article on NVIDIA's Parallel Forall developer blog (https://devblogs.nvidia.com/parallelforall/3-versatile-openacc-interoperability-techniques/).

CALLING CUDA OR LIBRARIES WITH OPENACC ARRAYS

The most common example of OpenACC interoperating with other programming models is passing device arrays from OpenACC to CUDA libraries. This is achieved with the *host_data* region. A *host_data* region can be thought of as a reverse data region. A data region exposes arrays from the host onto the accelerator, and a *host_data* region exposes data that is already on the accelerator to the host. The *use_device* clause to the region specifies which arrays should have their device addresses exposed to the host within the region.

Take for instance the code in Fig. 19.29; the *data* region at line 1 creates device copies of the *x* and *y* arrays on the accelerator, which get populated in the *kernels* region at line 3. The *host_data* region at line 12 exposes the device addresses of

```
#pragma acc data create(x[0:n]) copyout(y[0:n])
{
  #pragma acc kernels
  {
    for( i = 0; i < n; i++)
    {
      x[i] = 1.0f;
      y[i] = 0.0f;
    }
  }

   #pragma acc host_data use_device(x,y)
   {
     cublasSaxpy(n, 2.0, x, 1, y, 1);
   }
 }
```

FIGURE 19.29

Example using host_data with NVIDIA cuBLAS.

```
void saxpy(int n, float a, float * restrict x, float * restrict y)
  {
  #pragma acc kernels deviceptr(x,y)
  {
    for(int i=0; i<n; i++)
    {
      y[i] += a*x[i];
    }
  }
 }
```

FIGURE 19.30

Example of deviceptr clause.

x and *y* to the host to be passed into the *cublasSaxpy* function, which comes from the NVIDIA cuBLAS library. This allows developers to implement the majority of their code using OpenACC, but use accelerated libraries or CUDA functions selectively for the best performance.

USING CUDA POINTERS IN OPENACC

It is also possible to expose CUDA pointers to OpenACC regions in cases where CUDA is already being used in part of the application, but OpenACC is used in another. In this case, the deviceptr data clause can be used on data, kernels, or parallel directives to inform the compiler that any time the listed variables are seen, they are device pointers. For instance, if a developer wanted to provide an OpenACC version of the same SAXPY routine as above, one could use the code in Fig. 19.30, which accepts device pointers for *x* and *y* and passes them directly to the OpenACC region.

CALLING CUDA DEVICE KERNELS FROM OPENACC

Lastly, it is even possible to use CUDA device kernels within OpenACC compute regions to hand-tune the performance for critical functions within the region. In this case the previously discussed *routine* directive can be used to inform the compiler that a copy of the declared function already exists for the device and at what level of parallelism it was built. When mixing OpenACC and CUDA in this way it is generally simplest to implement a *seq* routine, which will be called from each loop iteration. Fig. 19.31 demonstrates both the declaration of the device function, typically in a separate header file, at line 2 and the implementation of the device kernel, typically in a source file, beginning at line 6. Fig. 19.32 then shows the device kernel being called from a *parallel loop* at line 15.

```
1.    // Declaration from header file
2.    #pragma acc routine seq
3.    extern "C" float saxpy_dev(float, float, float);
4.
5.    // Implementation from source file.
6.    extern "C"
7.    __device__
8.    float saxpy_dev(float a, float x, float y)
9.    {
10.     return a * x + y;
11.    }
```

FIGURE 19.31

Example using OpenACC routine directive with CUDA device kernel.

```
1.     #pragma acc data create(x[0:n]) copyout(y[0:n])
2.     {
3.       #pragma acc kernels
4.       {
5.         for( i = 0; i < n; i++)
6.         {
7.           x[i] = 1.0f;
8.           y[i] = 0.0f;
9.         }
10.       }
11.
12.    #pragma acc parallel loop
13.       for( i = 0; i < n; i++ )
14.       {
15.         y[i] = saxpy_dev(2.0, x[i], y[i]);
16.       }
17.    }
```

FIGURE 19.32

Example calling CUDA device kernel from OpenACC.

The interoperability features of OpenACC make it a part of a much larger ecosystem of accelerated and parallel computing. There are additional interoperability features that are not shown in this chapter. As such, developers should remember when choosing a programming model that their choice isn't "OpenACC or" but rather "OpenACC and" the other available tools.

19.6 THE FUTURE OF OPENACC

OpenACC began its life as a unification of emerging and competing compiler-based solutions that existed at the time, particularly from CAPS, Cray, and PGI targeting NVIDIA GPUs. First implementations of OpenACC focused on GPUs from NVIDIA, but with an eye toward the trend of increasingly parallel processor architectures. As a result, OpenACC is frequently mistaken as a GPU programming model, when in fact it is designed as a modern parallel programming model that builds on programming models that came before it. OpenACC is not designed to address all forms of parallelism or replace all other programming models, but instead is focused on loop-level data parallelism that is commonly found in HPC applications.

The most significant challenge that the OpenACC committee is still working to solve is the representation data structures that are more complex than simple arrays, such as C++ classes, Fortran derived types, and C structures containing pointers. These data structures are a significant challenge to the compiler, since they may not fit completely in device memory, may be shared between the host and the device, and may not contain sufficient information for the compiler to understand how to manage them effectively.

So-called deep copy, that is the copying of not only the pointers contained within a structure but what they point to, remains a topic of active discussion in the OpenACC community and is considered the most important feature to be added to OpenACC 3.0. As many supercomputing centers have adopted multi-device nodes, it will also be necessary for the OpenACC community to suggest new and better ways to manage multiple devices. With these larger compute nodes also come richer and more complex memory hierarchies, yet another challenge the technical committee intends to address. With the changing landscape of computing, there will be no shortage of challenges for the OpenACC specification to address.

There are some who believe that with the addition of offloading features to the more established OpenMP specification, OpenACC is no longer necessary. Others believe that OpenACC, as the more modern specification, provides value above and beyond what is available in OpenMP. It is the author's belief that programmers are used to choosing programming models based on the needs of their project, availability of tools, and personal preference and that both specifications provide developers with value while pushing each other forward through both collaboration and competition.

19.7 EXERCISES

1. The code below implements a simple matrix copy routine. Parallelize these loops using either OpenACC kernels or parallel loop such that the inner loop is a vector loop with a length of 128 and the outer loop is a gang loop of 1024 gangs.

```
for( int j = 1; j < n-1; j++ ) {
  for( int i = 1; i < m-1; i++ ) {
    B[j*m+i] = A[j*m+i];
  }
}
```

2. List two differences between the kernels and parallel constructs.

This page intentionally left blank