6 Heterogeneous Architectures

Specialized accelerator processors, which dramatically improve the performance of some computations, are proliferating, and general-purpose processors are now very often connected to some type of accelerator. The popularity of these heterogeneous architectures across all types of computing has had a noticeable impact on the development of software.

To exploit these systems, developers must write software that executes various regions of code on different types of devices. There are many reasons for wanting to do this but very often the motivation is to accelerate computationally intensive loop nests.

However, the programming models for these systems are difficult to use. Often code modules are written twice, once for the general-purpose processor and then again for the accelerator. The accelerator version is often written in a lower-level, accelerator-specific language. The result is the undesirable software maintenance problem of keeping two versions of code, which implement the same algorithm, synchronized.

The OpenMP Language Committee recognized the need to make it easier to program heterogeneous architectures and set about to extend OpenMP to support these types of systems [3]. The results of this work were initially released in OpenMP 4.0 and updated in OpenMP 4.5.

Software developers can use OpenMP to program accelerators in a higher-level language and maintain one version of their code, which can run on either an accelerator or a general-purpose processor. In this chapter, we present the syntax for and describe how to use the OpenMP *device constructs* and related runtime functions that were added to support heterogeneous architectures.

6.1 Devices and Accelerators

Typically, the motivation for running code on a heterogeneous architecture is to execute parts of a program on an *accelerator*. As the name implies, the desire is to dramatically improve the performance of a program by leveraging the specialized hardware capabilities of accelerator devices.

OpenMP provides the means to distribute the execution of a program across different devices in a heterogeneous architecture. A device is a computational resource where a region of code can execute. Examples of devices are GPUs, CPUs, DSPs, FPGAs or other specialized processors. OpenMP makes no distinction about the

```
1 #pragma omp target map(a,b,c,d)
2 {
3   for (i=0; i<N; i++) {
4     a[i] =  b[i] * c + d;
5   }
6 } // End of target
```

Figure 6.1: **Code fragment with one target region** – The target region is executed by a thread running on an accelerator.

specific capabilities or limitations of a device. Devices have their own threads which cannot migrate across devices. Program execution begins on the *host device*. The host device offloads the execution of code and data to accelerator devices.[1] Devices have access to memory where variables are stored. The memory may or may not be shared with other devices.

As shown in the code fragment in Figure 6.1, the `#pragma omp target` directive defines the target region spanning lines $1 - 6$. When a host thread encounters the `target` construct on line 1, the target region is executed by a new thread running on an accelerator.

By default, the thread that encounters the `target` construct waits for the execution of the target region to complete before it can continue executing the code after the `target` construct.

Before the new thread starts executing the target region, the variables `a`, `b`, `c`, and `d` are *mapped* to the accelerator. Mapped is the concept that OpenMP uses to describe how variables are shared across devices.

Very often the code that we wish to accelerate already includes OpenMP pragmas. We can place a `target` directive before a structured block that contains OpenMP constructs. In the code fragment shown in Figure 6.2, the target region is executed by a new thread on an accelerator. However, the new thread immediately encounters a `parallel for` construct and a team of threads is created that work together to execute the iterations of the subsequent loop.

The heterogeneous features of OpenMP fall into two general categories: program execution and data management. In the following sections, we will cover each of these categories in more detail.

[1]OpenMP uses the term *target* devices.

```
omp target map(a,b,c,d)
{
  #pragma parallel for
  for (i=0; i<N; i++) {
    a[i] =  b[i] * c + d;
  }
} // End of target
```

Figure 6.2: **Augmented code fragment with a parallel region** – The parallel region is executed by a team of threads running on an accelerator.

6.2 Heterogeneous Program Execution

This section describes the OpenMP heterogeneous program execution model. The device constructs, clauses, and new environment variable listed below are used to determine where (on which device) and how regions of a program are executed on a heterogeneous architecture:

- Target Construct
- Target Teams Construct
- Declare Target Construct
- Distribute Construct
- Device and Nowait Clauses
- `OMP_DEFAULT_DEVICE` Environment Variable

Of these, the `target` and `target teams` constructs are the most important as they are used to select which parts of a program are run on an accelerator. When a function name appears in a `declare target` construct, it indicates that the function is expected to be called from code executing on an accelerator, thus causing the compiler to generate a device-specific version of the function.

The heterogeneous execution model concepts are covered in this section. The complete syntax and semantics of the `target`, `target teams`, and `declare target` constructs are covered in detail in Sections 6.4, 6.5, and 6.7, respectively.

On a heterogeneous architecture with multiple accelerators, the `device` clause, `OMP_DEFAULT_DEVICE` environment variable, and runtime functions listed in Section 2.3.4 starting on page 70 are used to choose among and query about the different devices. Selecting a device using these clauses and functions is described in Section 6.10.

By default, the thread that encounters a device construct waits for the construct to complete. However, when a `nowait` clause is added to a device construct, the encountering thread does not wait, but instead continues executing the code after the construct. Task scheduling constructs are used to synchronize with the completion of the device construct's execution. The relationship between the device constructs and tasking is discussed in this section. The `nowait` clause is covered in Section 6.9.

The `target teams` construct starts multiple thread teams running in parallel on an accelerator. The `distribute` construct is a worksharing construct that schedules the iterations of a loop across the teams that are started by a `target teams` construct.

Combined with the `parallel for` and `simd` constructs, the `distribute` construct expresses a three-level hierarchy of parallelism across which loop iterations are spread. Loop iterations are first distributed to teams of threads, then to the threads in each team and, then to the SIMD vector lanes within each thread. This pattern of nested parallelism is executed efficiently by many types of accelerators.

The syntax and details of the `distribute` construct, and its combination with other constructs are covered in Sections 6.5.1 and 6.5.2.

6.2.1 A New Initial Thread

Recall that the thread that starts the execution of a program and executes all of the sequential code outside of any parallel regions is the *initial thread* (see Section 1.2.2). The OpenMP heterogeneous execution model is host-centric. The initial thread that starts the execution of a program is running on the host device. In other words, the program starts running on the host device. Prior to OpenMP 4.0 there was only one initial thread.

After OpenMP 4.0 and the addition of the `target` construct, multiple initial threads could arise during the execution of a program. A *target region* is all of the code that is dynamically encountered during the execution of a `target` construct. As shown in Figure 6.3, the thread that encounters a `target` construct does not

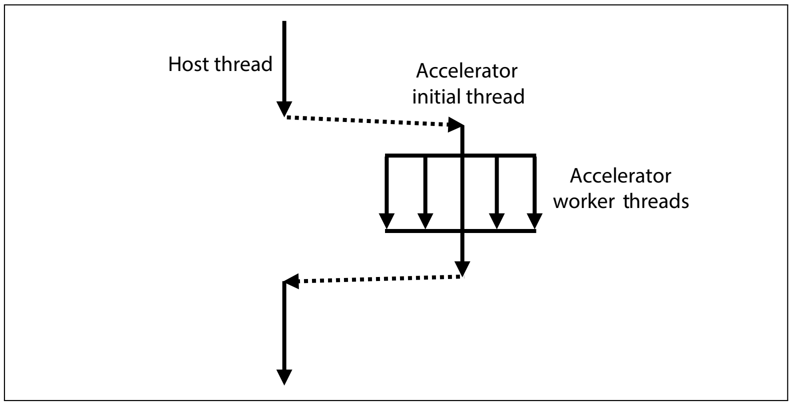

Figure 6.3: **The heterogeneous programming model supported by OpenMP** – Program execution begins on the host device. When a host device thread encounters a `target` construct, a new initial thread executes the target region. When the initial thread encounters a `parallel` construct it becomes the master of a teams of threads.

itself execute the target region. Instead, a new initial thread begins the execution of the target region. Each target region acts as an OpenMP sub-program where an initial thread begins the execution of the sub-program. The initial thread may encounter other parallel constructs and spawn teams of threads.

The initial thread that executes a target region is potentially running on an accelerator. We say potentially because it's possible that the OpenMP program is running on a system that has no accelerators, in which case, the target region is executed by an initial thread running on the host device. Even on systems where accelerators are available, if the `target` construct has an `if` clause whose conditional expression evaluates to *false* then the initial thread executes on the host device (see Section 6.10.2). If there are multiple accelerators available, the `device` clause can be used to select one of them. When a `device` clause is not present, the initial thread executes on the default device specified by the *default-device-var* ICV. By default, the thread that encounters the `target` construct waits for the execution of the target region to complete and then continues executing the code after the `target` construct. Note how this is different from a `parallel` construct where the thread that encounters the construct becomes the master thread in a team of threads that is created to execute the parallel region.

6.2.2 Contention Groups

A *contention group* is the set of all threads that are descendants of an initial thread. An initial thread is never a descendant of another initial thread. Each dynamically encountered `target` construct starts a new contention group.

Threads in different contention groups cannot synchronize with each other. This means that threads that arise from different target regions cannot synchronize with each other. Further, the threads in the contention group formed by the initial thread that started the execution of the program cannot synchronize with any threads that arise from target regions. This restriction effectively limits how threads in contention groups (often threads on different devices) can interact with each other.

When threads from different contention groups execute in parallel, only variables[2] written to atomically (using an `atomic` construct) by a thread in one contention group can be read by a thread in another contention group, and only if both contention groups are executing on the same device.

6.2.3 A League of Teams

The `target teams` construct starts a *league* of teams executing on an accelerator. Each of these teams is a single initial thread executing in parallel the subsequent code statement. This is similar to a `parallel` construct but different in that each thread is its own team: a team of one. Threads in different teams are in different contention groups and, therefore, restricted in how they can synchronize with each other.

When a `parallel` construct is encountered by a league, each initial thread in the league becomes the master of a new team of threads. The result is a league of teams where each team has one or more threads. Each team is a contention group. Each team of threads then concurrently executes the parallel region.

Leagues are used to express a type of loosely connected parallelism where teams of threads execute in parallel but with very limited interaction across teams. We will explore this more later in Section 6.5.1 when we discuss how leagues are used in accelerated worksharing.

[2]The size of these variables must be less than or equal 64 bits.

6.2.4 The Target Task

Sometimes we don't want the host thread that encounters a target region to wait for the target region to complete. We want the target region to execute asynchronously so that the host thread can go off and do other work. OpenMP already has tasks that provide capabilities for launching and coordinating the asynchronous execution of code regions. Leveraging these features, the device constructs are formulated as OpenMP task generating constructs.

We have been talking in terms of threads up to this point, but recall that threads are the entities that do work; the actual work is a task. There is always a task (implicit or explicit) that a thread is executing. Tasks are executed only by threads running on the device where the tasks were generated.

The `target` construct is a task-generating construct. When a thread encounters a `target` construct, it generates an explicit task that manages the execution of the target region. The OpenMP 4.5 specification refers to this task as the *target task*. This is an unfortunate name as it seems to imply that the target task is running on an accelerator, but it is an explicit task generated on the host. The target task is complete when the enclosed target region is complete.

When the target task executes, the target region executes in the context of an implicit task, called an *initial task*, on the accelerator. The initial task is executed by the initial thread. Before OpenMP 4.0 there was only one initial task; the implicit task that enclosed the whole program. However, now each time a target region executes, a new initial task is generated on the target device. The target task is complete when the initial task, and thus the target region, is complete.

The task that the host thread is executing when it encounters the `target` construct is called the *generating task*. It generates the target task. Because the `target` construct results in a task, we now have available all of the asynchronous execution features from OpenMP tasking.

As shown in Figure 6.4, the target task is executed immediately by the thread that is executing the generating task. The thread suspends executing the generating task and begins executing the target task. The target task is by default an *included task*. It is a feature of the OpenMP tasking model that the task that generates an included task cannot be scheduled to execute until the included task is complete. For our purposes, the effect is that execution cannot continue after a `target` construct until the target region is complete.

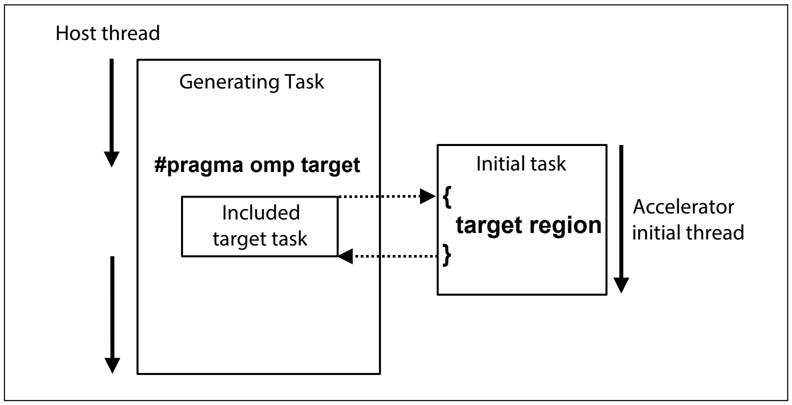

Figure 6.4: **The target task as an included task** – By default, the target task is an included task. The generating task cannot resume until the included target task is complete. The target task completes when the implicit task that contains the target region is completed by the initial thread running on an accelerator.

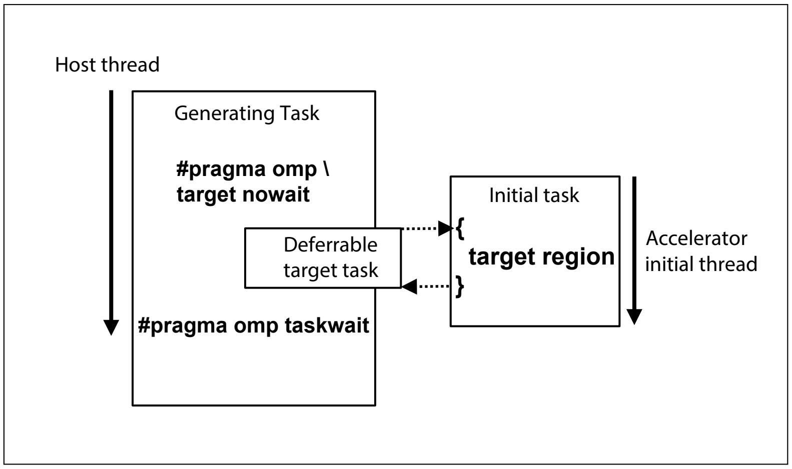

Figure 6.5: **The target task as a deferrable task** – The nowait clause makes the target task a deferrable task. The generating task may now be scheduled to execute before the target task is complete. The effect is that the generating task may execute in parallel with the target task.

However, sometimes we want the host device to do useful work in parallel with the accelerator device. Figure 6.5 shows how the `nowait` clause solves this problem. The `nowait` clause changes the default behavior of the target task so that it is no longer an included task With a `nowait` clause, the target task is like any other deferrable task.

Once a thread suspends execution of a target task, it is available to execute other tasks, including the original task that generated the target task. The effect is that execution of the generating task may continue past the `target` construct and before the associated target region has completed. The generating task is not stuck waiting for the target task (and thus the target region) to complete. The OpenMP task synchronization features, introduced in Chapter 3, may be used to determine when the target task is complete.

For example, in Figure 6.6 the thread that encounters the `target` construct generates a task and then continues after the construct to execute the function `F()`. The target task and the function `F()` are potentially executed in parallel. The host thread then waits at the `taskwait` construct to ensure that the target task has completed.

```
#pragma omp target map(a,b,c,d) nowait // Generate target task
{
  #pragma parallel for
  for (i=0; i<N; i++) {
    a[i] =  b[i] * c + d;
  }
} // End of target

F(b); // Execute in parallel with target task

#pragma omp taskwait // Wait for target task to finish
```

Figure 6.6: **Code fragment with a target nowait region** – The encountering thread generates a target task and then continues past the target construct to execute the function *F()*.

6.3 Heterogeneous Memory Model

This section provides an overview of the OpenMP heterogeneous memory model. The device constructs, clauses, and runtime functions that control how data is shared between threads executing on the host and an accelerator device are listed below:

- Map and Defaultmap Clauses
- Target Data Construct
- Target Enter and Exit Data Constructs
- Target Update Construct
- Declare Target Directive
- Use_device_ptr and Is_device_ptr Clauses
- Device Memory Functions

Of these, by far the most important is the `map` clause. Recall from Chapter 1 that variables are shared or private. As of OpenMP 4.0, variables can also be *mapped*, which is the concept that OpenMP uses to describe how data is shared across devices. The `defaultmap` clause can change the default rules for determining if certain variables are either private or mapped. The general concepts of mapped variables are discussed later in this section. The syntax and mechanics of the `map` and `defaultmap` clauses are covered in Section 6.6.

The host and accelerator may have different representations for the address of a variable. The `use_device_ptr` and `is_device_ptr` clauses are provided for the instances in which this difference in address representation must be dealt with explicitly. These device pointer clauses are covered in Section 6.11.

Variable's with static storage (for example, global variables) may be mapped for the entire program using the `declare target` directive, which is covered in Section 6.7.

The `target data`, `target enter data`, `target exit data`, and `target update` constructs are used to reduce the performance overhead of copying data between the host and an accelerator. These data-mapping constructs are covered in Section 6.8.

The device memory functions are described in detail in Section 2.3.4 starting on page 70. The `omp_target_is_present` function determines if a variable is mapped. Otherwise, the other device memory functions manage dynamically allocated device memory. Section 6.12 has examples that demonstrate how to use these functions.

6.3.1 Mapped Variables

Threads executing on an accelerator can have private variables. The initial thread that begins the execution of a target region gets a private instance of a variable that appears in a `private` or `firstprivate` clause on the `target` construct. For a `firstprivate` clause, the private variable is initialized with the value of the original variable from the host thread that encountered the construct. Likewise, any automatic (stack) variables that are declared in a scope contained within the construct are private to the initial thread.

OpenMP threads share variables that are stored in a single shared memory. However, heterogeneous architectures do not always have memory that is symmetrically shared between host and accelerator devices. A very common example of a heterogeneous architecture like this is one where the accelerator is a card and communication to the accelerator occurs over a PCIe bus.

As shown in figure 6.7, OpenMP supports heterogeneous architectures with both distributed and shared memory by *mapping* variables from the host to an accelerator. When the host and accelerator device do not share memory, a mapped variable is copied from the host's memory into the accelerator's local memory. Mapping hides whether or not a variable is shared by or copied to a device. Based on a heterogeneous architecture's memory system, the OpenMP implementation does what is required, either sharing or copying a variable when it is mapped.

How does one ensure that threads on different devices see the same value of a mapped variable and when? For the most part, the OpenMP memory consistency model as outlined in Chapter 1.2.3, starting on page 6, is extended to mapped variables.

A mapped variable is similar to a shared variable. Without some type of synchronization, two threads executing on different devices cannot simultaneously access the same mapped variable if either of the threads writes to the variable.

Threads executing on different devices may see a consistent value of a mapped variable at points that are determined by the effects of the `map` clause and the `target update` construct.

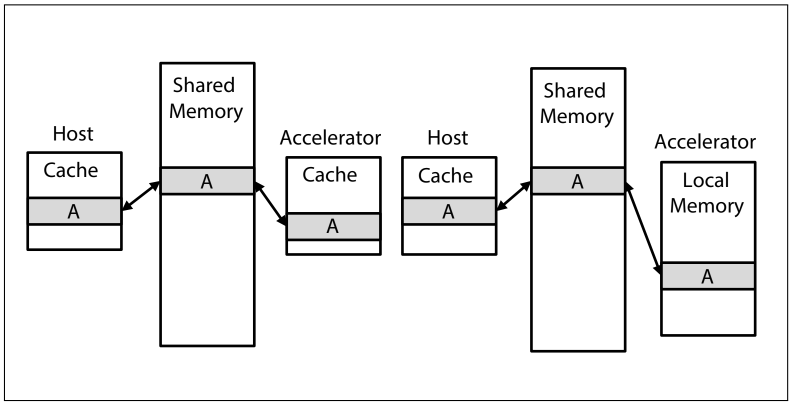

Figure 6.7: **A mapped variable in shared or distributed memory** – A mapped variable may be in either shared or distributed memory. The OpenMP implementation determines if copies are required.

If memory is distributed, then mapping a variable requires memory allocation, copy, and flush operations. The allocation and copy operations are not required (or are trivial) when memory is shared, but the flush operation is still necessary. Although the underlying machinations of variable mapping is handled by the OpenMP implementation, it is important to be aware of the dual-nature of mapped variables to write programs that achieve good performance across different architectures.

Because two devices may not share the same address space, the address of a mapped variable may not be the same on two devices When a pointer variable is mapped, only the pointer is synchronized, not the block of memory it points to. However, array sections may be used to map the pointed-to memory.

6.3.2 Device Data Environments

An accelerator has a *device data environment* that contains the set of all variables currently accessible by threads running on that device. As we discussed in the previous section, host threads share variables with target device threads by mapping them. Mapping a variable ensures that the variable is in the data environment of an accelerator.

An *original* variable in a host thread's data environment is mapped to a *corresponding* variable in the accelerator's data environment. Depending on the availability of shared memory between the host and target devices, the original and corresponding variables are either the same variable allocated in shared memory, or they are allocated in different memories and copy operations are required to keep the original and corresponding variables consistent. Whether a mapped variable uses shared or distributed memory is taken care of by the OpenMP implementation.

There only can be one instance of a variable in a device data environment. The OpenMP implementation keeps track of which variables are mapped. If a variable is already *present* in a device data environment, mapping it again will find the variable is already there and increment a reference count. It will not allocate another instance of the variable.

Minimizing the transfer of data between the host and an accelerator is often critical to getting good performance on heterogeneous architectures. Repetitively mapping a variable that is reused by multiple `target` constructs is potentially inefficient. The `target data`, `target enter data` and `target exit data` constructs amortize data transfers by mapping variables across the execution of multiple `target` constructs. Further, the `declare target` construct can map static and global variables for the whole program. Once a variable is mapped to an accelerator, situations can arise where the value of the variable must be updated from or to the device, and the `target update` construct fulfills this need. The `omp_target_is_present` runtime function is used to test if a variable is mapped.

6.3.3 Device Pointers

Because shared and distributed memory is supported, the OpenMP memory model assumes that the host and accelerator data environments are in different address spaces. However, this assumption creates some restrictions on accessing the address of the variable. With the OpenMP device constructs, the user must be aware of the different address spaces and be careful when using pointers. If the host and accelerator do not share memory, their local memories are in different address spaces. When a variable is mapped to an accelerator's data environment, a copy occurs, and the address of the variable on the accelerator is not the same as the address of the variable on the host.

Memory addresses are stored in pointer variables. A host thread cannot access memory via a pointer variable that contains an accelerator address. Likewise, an accelerator thread cannot access memory via a pointer variable that contains a host address. Further, the host and accelerator may have different representations for

```
1   char *hptr = malloc(N);
2
3   // Error - Accessing a host address on accelerator
4   #pragma omp target map(hptr)
5   for (int i=0; i<N; i++)
6     *hptr++ = 0;
```

Figure 6.8: **Illegal access of a host memory address** – A pointer variable containing a host memory address cannot be de-referenced by an accelerator thread.

```
1   char *dptr;
2   #pragma omp target map(dptr)
3   dptr = malloc(N);
4
5   // Error - Accessing a device address on host
6   for (int i=0; i<N; i++)
7     *dptr++ = 0;
```

Figure 6.9: **Illegal access of an accelerator memory address** – A pointer variable containing an accelerator memory address cannot be de-referenced by a host thread.

the address of a variable. For example, the value of a memory address might require 64 bits on a host and 32 bits on an accelerator.

In Figure 6.8, the host pointer variable `hptr` is assigned the address of a memory location in the host's address space. Mapping `hptr` copies the value of the pointer variable to the accelerator. The access to `hptr` at line 6 in the target region by an accelerator thread is illegal. The accelerator thread is attempting to access a host address.

Likewise in Figure 6.9, the accelerator pointer variable `dptr` is assigned the address of a memory location in the accelerator's address space. The access to `dptr` at line 7 by a host thread is illegal.

A *device pointer* is a pointer variable in the host data environment whose value is an object that contains the address of a storage location in an accelerator's device data environment.

Note that the value of a device pointer is an object. How the value of a device address is represented on a host is not necessarily the same way that it is represented

```
int dev = omp_get_default_device();
char *dptr = omp_target_alloc(dev, n);

#pragma omp target is_device_ptr(dptr)
for (int i=0; i<n; i++)
  *dptr++ = 0;
```

Figure 6.10: **Legal access of an accelerator memory address using a device pointer** – A device pointer variable that appears in an is_device_ptr clause may be de-referenced in a target region.

on an accelerator. When a device pointer is referenced in a target construct, the compiler may need to transform the representation of the device address stored in the device pointer.

In Figure 6.10, the omp_target_alloc function returns a device address. The device pointer dptr must appear in an is_device_ptr clause on the target construct to correctly refer to it in the target region. The variable dptr is private in the target region. On entry to the region, the private dptr variable is initialized with the accelerator's memory address that corresponds to the original value of dptr before the region (the host's representation of the device address). See Section 6.11 for more details on device pointers.

6.3.4 Array Sections

Pointer variables are used extensively in C and C++. The value stored in a pointer variable is the address of another variable. As we saw in the last section, in order to support a variety of systems, the OpenMP model assumes that the host and accelerator may not share the same address space. Thus, mapping a pointer variable by itself is not very useful. We want to map the pointed-to variable (the memory that the pointer references). In order to map the pointed-to variable, we need to know its size.

For C and C++, we need something in the OpenMP syntax to express the concept of mapping the pointed-to variables. This is one of the reasons that OpenMP 4.0 added *array section* syntax for array and pointer variables.[3]

[3]Array sections may also appear in the depend clause.

An array section is a subset of the elements in an array. In OpenMP array sections are restricted to a contiguous set of elements. The C and C++ array subscript syntax is extended to support an array section expression. The array section syntax `base[`*`offset`*`:`*`length`*`]` is described below:

- The *base* is a C or C++ variable name with array, pointer type, or in C++, reference to array or reference to pointer type.
- The *offset* is an non-negative integer expression that is an offset from the start of the array. The *offset* is optional and, if not specified, defaults to 0.
- The *length* is a non-negative integer expression that is the length of the array section. If the *base* variable has a type of array or reference to array, then the *length* is optional and defaults to the number of elements in the array. If the *base* variable has a type of pointer or reference to pointer, then the *length* must be specified.

The value of a pointer variable used in an array section is the address of a pointed-to array variable. The pointed-to array variable may or may not have been dynamically allocated. Even if the pointed-to variable is a single scalar variable, when it's used in an array section, it is an array of one element.

An array section is *pointer-based* when the *base* is a pointer variable. A pointer-based array section is mapped using the following steps:

1. Create a pointer variable in the accelerator's data environment.
2. Map the host's pointed-to variable to the accelerator's data environment.
3. Initialize the accelerator's pointer variable with the address of the pointed-to variable in the accelerator's address space.

In Figure 6.11, the host pointer variable `hptr` is assigned the address of a storage location in the host's data environment. The array section `hptr[0:1024]` is then mapped to the accelerator's data environment. The 1024 element array pointed to by `hptr` is mapped to the accelerator. The `hptr` pointer variable is not mapped but is private in the target region and initialized with the address of the pointed-to array. Compare this to Figure 6.8.

```
char *hptr;

hptr = malloc(1024);

// Map an array section.
#pragma omp target map(hptr[0:1024])
for (int i=0; i<N; i++)
  hptr[i] = 0;

```

Figure 6.11: **Map a pointer-based array section** – Use an array section to map pointed-to memory.

```
float *p = malloc(N);
float a[N];

// Map pointer based array section
map(p[0:N:1])
map(p[0:N])
map(p[:N])

// Map array based array section
map(A[0:N:1])
map(A[0:N])
map(A[:N])
map(A[:]) // Size is N

// Map array section with offset
map(p[32:N-32]
map(A[N/2:N/4]

```

Figure 6.12: **Array section syntax examples** – Various usage of array section syntax in C and C++.

Array sections are available in the Fortran base language. In C/C++, an array section may appear only as a list item in an OpenMP `map` or `depend` clause. The C/C++ base language was not extended to support array sections. Some examples of array sections in C/C++ are shown in Figure 6.12

```
#define BIG 256
#define N (1024*1024)
int a[N*BIG];

void F(const int c, const int d)
{
  for (int k=0; k<N*BIG; k+=N) {
    #pragma omp target map(from:a[k:N]) firstprivate(c,d)
    for (int i=0; i<N; i++) {
      a[k+i] =  k+i * (c + d);
    } // End of target
  }
}
```

Figure 6.13: **Use array section to map a subset of an array** – Map a slice of the array `a` each time through the loop.

Array sections are also useful for mapping a slice of an array. It might be that mapping a very large array exceeds the storage capacity of the accelerator's local memory. In this case, we would like to map slices of the array and then compute on each slice. Figure 6.13 shows how this can be done. The rest of the sections in this chapter describe the syntax and semantics of the device constructs and clauses.

6.4 The Target Construct

The purpose of the `target` construct is to offload the execution of code to an accelerator. The code in a target region is executed by a new initial thread. The code in Figure 6.14 is a simple hello world example that uses the `target` construct.

The OpenMP runtime function `omp_is_initial_device` returns true if the code is executing on the host device. If there are no accelerators on the system where the code is running, then the initial thread that executes the target region runs on the host device.

Since the initial thread that executes the target region can always *fall back* to the host, programs that use device constructs are portable to systems that do not have accelerators. However, in the following description of the `target` construct it is assumed that the code is running on a system with at least one accelerator.

```
#include <stdio.h>
#include <omp.h>
void hello(void)
{
  #pragma omp target
  {
    if (!omp_is_initial_device())
      printf("Hello World from accelerator\n");
    else
      printf("Hello World from host\n");
  }
}
```

Figure 6.14: **Example of a target construct** – If the initial thread is running on an accelerator, it executes the first `printf()`. Otherwise, it is running on the host device and executes the second `printf()`. Note that some implementations may not support calling `printf()` on an accelerator.

#pragma omp target *[clause[[,] clause]. . .]*   *structured block*
!$omp target *[clause[[,] clause]. . .]*   *structured block* **!$omp end target**

Figure 6.15: **Syntax of the target construct in C/C++ and Fortran** – A target region is executed by an initial thread running on an accelerator.

The `target` construct syntax in C/C++ and Fortran is given in Figure 6.15. The clauses that are available on the `target` construct are listed in Figure 6.16.

The `target` construct is a task generating construct. When a thread encounters a `target` construct a target task is generated on the host device. You can think of the target task as a task bound to the host device that wraps the execution of the target region. The target task is complete (on the host) when the target region is complete (on the accelerator). The `nowait` and `depend` clauses affect the type and asynchronous behavior of the target task.

By default, the execution of the target task is synchronous. The encountering thread cannot continue past the target construct until the target task is complete. The `nowait` clause makes the execution of the target region asynchronous. After

if *([*`target`*:] scalar-expression)*	(C/C++)
if *([*`target`*:] scalar-logical-expression)*	(Fortran)
map *([[map-type-modifier[,]] map-type:] list]*	
device *(integer-expression)*	(C/C++)
device *(scalar-integer-expression)*	(Fortran)
private *(list)*	
firstprivate *(list)*	
is_device_ptr *(list)*	
defaultmap(tofrom:scalar)	
nowait	
depend *(dependence-type: list)*	

Figure 6.16: **Clauses supported by the target construct** – The `if` and `device` clauses are discussed in Section 6.10. The `map` clause is discussed in Section 6.6.1. The `nowait` and `depend` clauses are discussed in Section 6.9. The `is_device_ptr` clause is discussed in Section 6.3.3 The `defaultmap` clause is discussed in Section 6.6.3.

generating the target task, the encountering thread does not wait for the target task to complete, but instead it can continue and execute the code after the `target` construct. Task scheduling via the `taskwait` construct or the `depend` clause may be used to synchronize with the completion of the target task. The execution model of the `target` construct is covered in detail in Section 6.2.

The target region is executed by an initial thread. Where the initial thread runs (the host or an accelerator) is determined by the *default-device-var* ICV. The `device` clause can be used to specify a device other than the default. The `if` clause is available to conditionally fall back to running the initial thread on the host.

The initial thread gets a private instance of a variable that appears in a `private` or `firstprivate` clause on a `target` construct. For a `firstprivate` clause, the private variable is initialized with the value of the original variable from the host thread that encountered the target construct. Likewise, any automatic (stack) variables that are declared in a scope contained within the target construct are private to the initial thread. Variables that appear in `map` clauses are mapped. Any assignments specified by a `map` clause occur when the target task executes. If a variable referenced in the `target` construct does not appear in a `map`, `private`, `firstprivate` or `is_device_ptr` clause, then default data-mapping rules determine if and how the variable is mapped (see Section 6.6).

C/C++ pointer variables that appear in `map` clauses as the base of an array section are private in the target region. The private variables are initialized with the value of the address of the array section's pointed-to memory in the accelerator's address space.

The code in Figure 6.17 illustrates how to use the `target` construct to offload to an accelerator the matrix times vector product example taken from [2]. By adding the `target` construct at line 6, the loop body is offloaded to an accelerator. When a host thread encounters the `target` construct at line 5, it generates an included target task, suspends the current task, and starts executing the target task. Because the target task is included, the host thread must wait for the target region, and thus the target task to complete, before continuing to execute the statement after the target construct. Scalar variables that do not appear in a `map` clause default to firstprivate, but for clarity the variables `m` and `n` are listed explicitly in a clause. Because the variables `a`, `b`, and `c` are pointers, array sections are required in the `map` clause to describe the size of the pointed-to memory. The pointer variables themselves are private in the target region.

The array sections' pointed-to memory is mapped to the accelerator's address space. If the host and accelerator do not share memory, storage is allocated in the accelerator's local memory, and the array sections are copied from the host's memory to the accelerator's local memory. The private pointer variables are assigned the address of the array section's pointed-to memory in the accelerator's address space. Because the variables `i` and `j` are declared in a scope enclosed in the target construct at line 7, they are private.

The target region is executed by an initial thread running on the default accelerator. The initial thread encounters the `parallel for` worksharing construct at lines $8-9$ and becomes the master of a new team of threads that work together to execute the for-loop at line 10. The variables `m`, `n`, `a`, `b`, and `c` which were private to the initial thread are now shared among the threads in the team. The variables `i` and `j` are private to each thread in the team.

If the host and accelerator do not share memory, then when the target region is complete, the array sections are copied back from the accelerator's local memory to the host's memory. The storage allocated in the accelerator's local memory is then released. After the target region is complete, the target task completes and the host thread starts executing again at line 17 after the `target` construct.

```c
void mxv(int m, int n, double * restrict a,
         double * restrict b, double * restrict c)
{

  #pragma omp target map(a[:n],b[:n],c[:n]) firstprivate(m,n)
  {
    int i, j;
    #pragma omp parallel for default(none) \
            shared(m,n,a,b,c) private(i,j)
    for (i=0; i<m; i++)
    {
      a[i] = b[i*n]*c[0];
      for (j=1; j<n; j++)
        a[i] += b[i*n+j]*c[j];
    } // End of parallel for
  } // End of target
}
```

Figure 6.17: **Example using the target construct to execute the matrix times vector on an accelerator** – The host thread waits for the execution of the target region to finish before it continues after the construct.

The restrictions on the usage of the `target` construct are as follows:

- A `target` construct cannot be nested inside a target region.
- A `target data`, `target update`, `target enter data`, or `target exit data` construct cannot be nested in a target region.
- `threadprivate` variables cannot be accessed in a target region.
- In C++ a virtual member function cannot be invoked on an object that was not constructed on the accelerator. The object cannot be a mapped variable.
- In Fortran, if an array section is derived from a variable that has a `POINTER` or `ALLOCATABLE` attribute, then the variable cannot be modified in the target region.

#pragma omp teams *[clause[[,] clause]. . .]*   *structured block*
!$omp teams *[clause[[,] clause]. . .]*   *structured block* **!$omp end teams**

Figure 6.18: **Syntax of the teams construct in C/C++ and Fortran** – Create a league of initial threads each in its own team.

6.5 The Target Teams Construct

Strictly speaking, `target teams` is a combined construct made up of the `target` and `teams` constructs. But since a `teams` construct may appear only nested immediately inside a `target` construct with no other intervening statements or declarations between the two constructs, the two constructs are inseparable. The `teams` construct syntax in C/C++ and Fortran is shown in Figure 6.18.

Similar to the `parallel` construct, the `target teams` construct specifies that the subsequent code block should be run in parallel. A `parallel` construct creates a team of threads, where the thread that encountered the `parallel` construct becomes the master thread. Each thread in the team executes the parallel region. The `target teams` construct starts a *league* of initial threads where each thread is in its own team. Each initial thread executes the teams region in parallel (see Section 6.2.2). One can think of the `target` construct as a `target teams` construct that creates a league with only one initial thread.

When a `parallel` construct is encountered by a league, each thread in the league becomes the master of a new team of threads. The result is a league of teams where each team has multiple threads. Each team of threads concurrently executes the parallel region. The clauses that may appear on the `teams` construct are listed in Figure 6.19. Clauses from both the `target` and `teams` constructs may appear on the `target teams` construct.

The number of teams created by a `target teams` construct is implementation defined or is specified by the `num_teams` clause. Each team is executing in its own contention group. The maximum number of threads active in a contention group is specified by the `thread_limit` clause.

The `target teams` and `parallel` constructs both fork multiple threads that execute the subsequent block of code in parallel. The `target teams` construct is

num_teams *(integer-expression)*	(C/C++)
num_teams *(scalar-integer-expression)*	(Fortran)
thread_limit *(integer-expression)*	(C/C++)
thread_limit *(scalar-integer-expression)*	(Fortran)
default(shared | **none)**	(C/C++)
default(shared | **firstprivate** | **private** | **none)**	(Fortran)
private *(list)*	
firstprivate *(list)*	
shared *(list)*	
reduction *(reduction-identifier : list)*	

Figure 6.19: **Clauses supported by the teams construct** – The `num_teams` and `thread_limits` clauses are described below.

asserting a more restricted form of parallelism than the `parallel` construct allows. The compiler can take advantage of these restrictions and be much more aggressive at exploiting parallelism. These restrictions are as follows:

- Because the teams that are started by a `target teams` construct are each in their own contention group, threads from different teams cannot synchronize with each other.
- The only OpenMP constructs that can appear in a `teams` region are the `parallel`, `distribute` and any other `parallel` or `distribute` regions arising from related constructs. These are listed here:
  - `parallel`
  - `parallel for` (C/C++)
  - `parallel do` (Fortran)
  - `parallel sections`
  - `distribute`
  - `distribute simd`
  - `distribute parallel for` (C/C++)
  - `distribute parallel do` (Fortran)
  - `distribute parallel for simd` (C/C++)
  - `distribute parallel do simd` (Fortran)

```
 1 #include <omp.h>
 2 extern void do_team_work(int, int, int, int);
 3 #pragma omp declare target(do_team_work)
 4 void f()
 5 {
 6   #pragma omp target teams
 7   {
 8     int team = omp_get_team_num();
 9     int nteams = omp_get_num_teams();
10     int tid = omp_get_thread_num(); // Always 0
11     int nthreads = omp_get_num_threads(); // Always 1
12     do_team_work(team, nteams, tid, nthreads);
13   } // End of target teams
14 }
```

Figure 6.20: **Example of the target teams construct** – Multiple initial threads execute the function `do_team_work()`.

In Figure 6.20 the `target teams` construct at line 6 creates a league of initial threads. Each initial thread is in its own team. The initial threads (and therefore the teams) are numbered from 0 to $N-1$ where N is the number of initial threads created. The number of initial threads is returned by the OpenMP runtime function `omp_get_num_teams`. Calling the `omp_get_team_num()` in a teams region returns the team number of the initial thread. Since each team is a single initial thread, the calls to `omp_get_num_threads()` at line 11 and `omp_get_thread_num()` at line 10 will always return one and zero, respectively. Each initial thread calls the function `do_team_work()` at line 12 passing in the team number, the number of teams, the thread number and the number of threads in the team. Figure 6.21 diagrams the execution of the region assuming four initial threads.

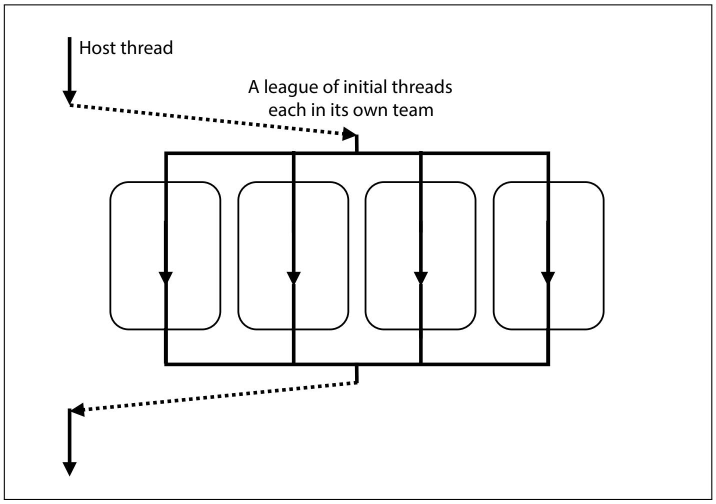

Figure 6.21: **The target teams construct creates a league of initial threads** – Each initial thread is a team of one thread. The initial threads execute the teams region in parallel.

In Figure 6.22, the `target teams` construct again creates a league of initial threads, but this time each initial thread immediately encounters the `parallel` construct at line 7. Each initial thread then becomes the master of a new team of multiple threads. The number of teams and a thread's team number are determined by the `omp_get_num_teams` and `omp_get_team_num` runtime functions, respectively. The call to the `omp_get_num_threads()` function at line 12 returns the number of threads in a team, which is 5. The call to `omp_get_thread_num` at line 11 returns the threads number in the range 0 to 4. Each thread in each team (a total of 20 threads) then calls the function `do_team_work()` passing in the team number, the number of teams, the thread number and the number of threads in the team. Figure 6.23 diagrams the execution of all the threads, assuming four teams with five threads per team. Note that if the thread calling `omp_get_team_num` is in a team that was initiated by a parallel region nested inside a teams region, the function still returns the number of the initial thread that is the ancestor of the thread.

```
void f()
{
  #pragma omp target teams num_teams(4)
  #pragma omp parallel num_threads(5)
  {
    int team = omp_get_team_num();
    int nteams = omp_get_num_teams();
    int tid = omp_get_thread_num();
    int nthreads = omp_get_num_threads();
    do_team_work(team, nteams, tid, nthreads);
  } // End of target teams
}
```

Figure 6.22: **Example of a parallel construct nested in a target teams construct** – Multiple teams of threads execute the function `do_team_work()`.

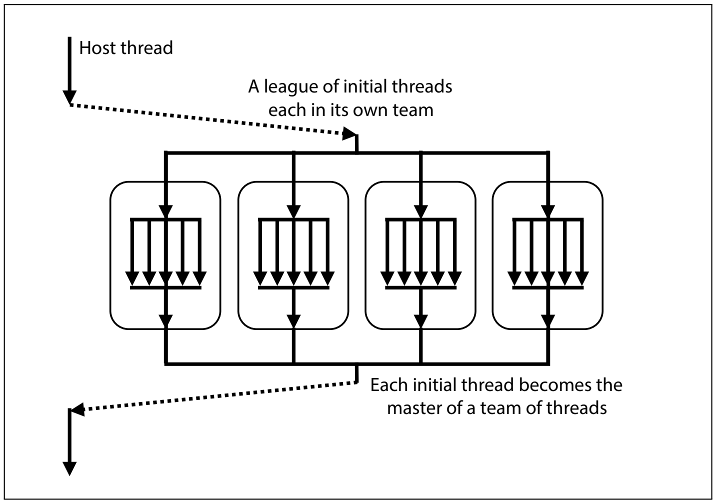
Host thread
A league of initial threads each in its own team
Each initial thread becomes the master of a team of threads

Figure 6.23: **The initial threads created by the teams construct each become the master of a new team of threads.** – Each initial thread starts execution as team of one thread. The initial threads execute the teams region in parallel and immediately encounter a parallel construct. Each initial thread then becomes the master of a new team of threads.

#pragma omp distribute *[clause[[,] clause]. . .]* *for-loops*
!$omp distribute *[clause[[,] clause]. . .]* *do-loops* **!$omp end distribute** *[clause[[,] clause]. . .]*

Figure 6.24: **Syntax of the distribute construct in C/C++ and Fortran** – Distribute loop iterations to the initial threads in a league.

private *(list)*
firstprivate *(list)*
lastprivate *(list)*
collapse *(n)*
dist_schedule *(kind[, chunk_size])*

Figure 6.25: **Clauses supported by the distribute construct** – The details for the `dist_schedule` clause are given in the text.

6.5.1 The Distribute Construct

The `target teams` construct starts a league of initial threads where each thread is in its own team. Similar to the loop construct, the `distribute` construct is a worksharing construct that distributes the iterations of a loop to the initial threads in a league. The loop iterations are divided into chunks, which are then scheduled across the initial threads in a league. The `distribute` construct syntax in C/C++ and Fortran is shown in Figure 6.24. The clauses that are available on the `distribute` construct are listed in Figure 6.25.

Variables that appear in the `private`, `firstprivate` or `lastprivate` clause are private in each initial thread. The `distribute` construct has no implicit barrier at the end of the construct. This is like having a loop construct with a `nowait` clause. The initial threads do not synchronize at a barrier at the end of the region.

The `collapse` clause has the same behavior as it does on the loop construct. It collapses the iterations of perfectly nested loops into a single iteration space. The restrictions on the format of the loop to which the construct applies are the same as those for the loop construct.

How the loop iterations are scheduled to execute across the initial threads in the league is implementation-defined, unless the `dist_schedule` clause is present. When the `dist_schedule(static)` clause is present, the loop iterations are divided

```
void saxpy(float *restrict y, float *restrict x, float a, int n)
{
  #pragma omp target teams map(y[:n]) map(to:x[:n])
  #pragma omp distribute
  for (int i=0; i<n; i+=n)
  {
      y[i] =  y[i] + a*x[i];
  }
}
```

Figure 6.26: **Example of the distribute worksharing construct** – Each initial thread created by the target teams construct executes a subset of the iterations in the loop's iteration space.

into contiguous chunks. If *chunk_size* appears in the clause, then it specifies the size of the chunks. Otherwise, each thread is assigned no more than one chunk, and the chunks are roughly equal in size.

A version of the familiar saxpy (single precision $y = a * x + y$) function is shown in Figure 6.26. The `distribute` worksharing construct distributes the iterations of the loop to the initial threads in the league started by the `target teams` construct.

What is the difference between the execution of the `for` (or `do` in Fortran) and the `distribute` worksharing constructs? The `distribute` construct has the potential for better performance because of the restrictions on where it can be used and what other OpenMP constructs can appear inside the distribute region. This enables the compiler to be more aggressive with optimizations. The `for` and `do` constructs are more versatile but may not perform as well.

The idea behind the `target teams` and `distribute` constructs is to spread the execution of a loop coarsely across hardware compute units and then more finely to the threads that execute within those compute units. What we have shown so far is how to distribute the loop iteration to the compute units.

The code in Figure 6.27 converts the saxpy loop into a doubly nested loop.

The `distribute` construct at line 5 assigns the execution of two iterations in the outer loop to a league of two initial threads. The `parallel` construct at line 8 is then encountered by each initial thread with different values for `j`. Each initial thread becomes the master thread in a team of four threads. The first team executes the first half of the loop iterations and the second team executes the other half.

```
 1 void saxpy(float *restrict y, float *restrict x, float a, int n)
 2 {
 3   // Assume n is even
 4   #pragma omp target teams map(y[:n]) map(to:x[:n]) num_teams(2)
 5   #pragma omp distribute
 6   for (int j=0; j<n; j+=n/2)
 7   {
 8     #pragma omp parallel num_threads(4)
 9     #pragma omp for
10     for (int i=j; i<n/2; i++)
11       y[i] = y[i] + a*x[i];
12   }
13 }
```

Figure 6.27: **Example of worksharing a loop across two levels of parallelism** – Use team level parallelism on the outer loop and thread level parallelism on the inner loop. Distribute the loop iterations to two teams. Each team then uses four threads to execute the iterations that are assigned to it.

The loop iterations scheduled to execute on an initial thread are then scheduled according to the `for` worksharing construct at line 9, to execute on the team of threads that the initial thread is now the master of.

In accelerated worksharing, loops are first scheduled at a coarse level to teams and then more finely to the threads in each team. We rewrote the saxpy loop in order to schedule it across two levels of parallelism: teams and threads. However, rewriting loops is tedious and is something we want to avoid.

Section 6.5.2 introduces *composite* accelerated worksharing constructs. When composite accelerated worksharing constructs are used, loop iterations are distributed across multiple levels of parallelism without having to rewrite the loop as we did in Figure 6.27.

6.5.2 Combined and Composite Accelerated Worksharing Constructs

Recall that combined constructs are short-hand notation for specifying the individual constructs in which one construct is immediately nested inside another. For example, the `parallel for` combined construct is equivalent to a `parallel` construct with a `for` construct nested immediately inside the `parallel` construct. The combined construct has the same execution behavior as the two separate constructs. However, in some instances, depending on the compiler, the combined constructs may achieve better performance than the individual constructs.

With some exceptions, the clauses that may appear on a combined construct are any of the clauses that may appear on the individual constructs that make up the combined construct. There are many new combined constructs involving the device constructs. They are presented in this section in two groups.

The first group is the *combined target* constructs. The constructs in this group combine the `target` construct with other constructs. The second group is the *combined target teams* constructs. They combine the `target teams` construct with new worksharing constructs. This second group is discussed at the end of this section after the new worksharing constructs are presented.

The syntax for the combined target constructs in C/C++ and Fortran are shown in Figure 6.28. The combined target constructs that include a `parallel` directive create a team of threads where the initial thread is the master of the team. The target simd region is executed by an initial thread that uses SIMD parallelism to execute the iterations of the subsequent loop.

A composite construct is different than a combined construct. Composite constructs combine multiple constructs, but the combination has execution behavior that is different from when the constructs are specified separately.

The `distribute parallel for` construct is a composite accelerated worksharing construct that distributes the iterations of a loop across two levels of parallelism. Each initial thread in the league that encounters the construct becomes the master thread of a team. The iterations of a loop are first distributed to the master threads. The subset of loop iterations assigned to the master thread are then again distributed to the threads in the team.

The code in Figure 6.29 shows how the `distribute parallel for` accelerated worksharing construct is used to distribute the iterations of the saxpy loop to teams and then to the threads in those teams.

#pragma omp target parallel *[clause[[,] clause]. . .]*   *structured block*
#pragma omp target parallel for *[clause[[,] clause]. . .]*   *for-loops*
#pragma omp target parallel for simd *[clause[[,] clause]. . .]*   *for-loops*
#pragma omp target simd *[clause[[,] clause]. . .]*   *for-loops*

!$omp target parallel *[clause[[,] clause]. . .]*   *structured block* **!$omp end target parallel** *[clause[[,] clause]. . .]*
!$omp target parallel do *[clause[[,] clause]. . .]*   *do-loops* **!$omp end target parallel do** *[clause[[,] clause]. . .]*
!$omp target parallel do simd *[clause[[,] clause]. . .]*   *do-loops* **!$omp end target parallel do simd** *[clause[[,] clause]. . .]*
!$omp target simd *[clause[[,] clause]. . .]*   *do-loops* **!$omp end target simd** *[clause[[,] clause]. . .]*

Figure 6.28: **Syntax of the combined target constructs in C/C++ and Fortran** – Constructs combining `target` with other constructs. A `copyin` clause may not appear on any of the combined target constructs.

Another way to look at this type of construct is to consider the nested version of the C/C++ saxpy loop from Figure 6.27. We had to rewrite the loop to distribute its iterations across two levels of parallelism.

The `distribute parallel for` (or `distribute parallel do` in Fortran) construct tells the compiler to create the second level of parallelism and to distribute loop iterations across the two levels of parallelism. So, now we don't have to rewrite loops!

The first level of parallelism is created by a `target teams` construct. When the resulting league of initial threads encounters the `distribute parallel loop` construct, the following steps occur:

1. By the `distribute` part of the construct, each initial thread is assigned loop iterations according to the `distribute` construct's scheduling algorithm.

```
void saxpy(float *restrict y, float *restrict x, float a, int n)
{
  #pragma omp target teams map(y[:n]) map(to:x[:n])
  #pragma omp distribute parallel for
  for (int i=0; i<n; i++)
    y[i] = y[i] + a*x[i];
}
```

Figure 6.29: **Example of the distribute parallel loop accelerated worksharing construct** – Create multiple thread teams executing in parallel. Distribute loop iterations to the teams and then to the threads in each team.

2. By the `parallel` part of the construct, each initial thread becomes the master thread of a thread team. This creates the second level of parallelism. Now multiple teams of threads are executing in parallel.

3. By the `for` part of the construct, the subset of iterations assigned to each initial thread (the master thread) are then distributed across the threads in each team.

The composite accelerated worksharing constructs and their syntax in C/C++ and Fortran are shown in Figure 6.30. With a few exceptions, all clauses that may appear on the individual directives that make up the construct, may appear on the composite construct.

The `distribute simd` construct distributes loop iterations across two levels of parallelism. Loop iterations are assigned to the initial threads in a league according to the `distribute` constructs scheduling algorithm. Each initial thread then uses SIMD parallelism to execute the loop iterations assigned to it.

The `distribute parallel for simd` (or `distribute parallel do simd` in Fortran) construct distributes loop iterations across three levels of parallelism. Loop iterations are first assigned to the initial threads in each team. Each initial thread becomes the master of a new team of threads. The loop iterations assigned to an initial thread are then distributed to the threads in the master thread's team. Each thread then uses SIMD parallelism to execute the iterations assigned to it.

The composite accelerated worksharing constructs may be combined with the `target teams` construct. As mentioned at the beginning of this section, these are

#pragma omp distribute parallel for *[clause[[,] clause]...]*
    for-loops

#pragma omp distribute simd *[clause[[,] clause]...]*
    for-loops

#pragma omp distribute parallel for simd *[clause[[,] clause]...]*
    for-loops

!$omp distribute parallel do *[clause[[,] clause]...]*
    do-loops
!$omp end distribute parallel do *[clause[[,] clause]...]*

!$omp distribute simd *[clause[[,] clause]...]*
    do-loops
!$omp end distribute simd *[clause[[,] clause]...]*

!$omp distribute parallel do simd *[clause[[,] clause]...]*
    do-loops
!$omp end distribute parallel do simd *[clause[[,] clause]...]*

Figure 6.30: **Syntax of the composite accelerated worksharing constructs in C/C++ and Fortran** – Distribute loop iterations across multiple levels of parallelism: teams, threads and SIMD lanes.

called combined target teams constructs. The combined target teams constructs and their syntax in C/C++ and Fortran are shown in Figure 6.31.

Because the `target teams` construct is a combined construct, the `target` construct may be separated out of a combined target teams construct (see Section 6.5). For example, a `target teams distribute` construct may be separated into a `target` construct with an immediately nested `teams distribute` construct. Typically, this is simply a syntax preference, but there are some instances when `target` must be a separate construct. This can occur when a variable must be `private` in the teams region and mapped in the target region.

A variable cannot appear in both a `map` clause and a data-sharing attribute clause on the same construct. For example, in Figure 6.32 the variable `sum` appears in a `reduction` clause, and therefore, cannot also appear in a `map` clause. Because `sum` is not mapped, its reduced value is lost after the `target teams` region completes.[4]

The solution, as shown in Figure 6.33, is to use a separate `target` construct that explicitly maps the variable `sum`. Each initial thread that executes the teams region

[4]The same problem can occur when using a `reduction` clause on the target combined constructs.

#pragma omp target teams distribute *[clause[[,] clause]. . .]*   *for-loops*
#pragma omp target teams distribute parallel for *[clause[[,] clause]. . .]*   *for-loops*
#pragma omp target teams distribute simd *[clause[[,] clause]. . .]*   *for-loops*
#pragma omp target teams distribute parallel for simd *[clause[[,]. . .*   *for-loops*

!$omp target teams distribute *[clause[[,] clause]. . .]*   *do-loops* **!$omp end target teams distribute**
!$omp target teams distribute parallel do *[clause[[,] clause]. . .]*   *do-loops* **!$omp end target teams distribute parallel do**
!$omp target teams distribute simd *[clause[[,] clause]. . .]*   *do-loops* **!$omp end target teams distribute simd**
!$omp target teams distribute parallel do simd *[clause[[,] clause]. . .]*   *do-loops* **!$omp end target teams distribute parallel do simd**

Figure 6.31: **Syntax of the combined target teams constructs in C/C++ and Fortran** – Constructs that combine `target teams` and accelerated worksharing constructs.

gets a private instance of `sum`. Once the teams region is complete, the mapped `sum` variable contains the reduced value. In the map-exit phase for the target region, the reduced value is assigned to the host's original `sum` variable.

6.6 Data Mapping Clauses

Recall from Section 6.3.2 that an accelerator has a *device data environment*, which contains the set of all variables that are available to the threads executing on that accelerator. When an *original variable* in the host's data environment is mapped to an accelerator, a *corresponding variable* is allocated in the accelerator's device data environment. During the execution of a program, the set of corresponding

```
int dotp(int *restrict a, int *restrict b, int n)
{
  int sum = 0;

  #pragma omp target teams distribute map(to:a[:n],b[:n]) \
                                       reduction(+:sum)
  for (int i=0; i<n; i++)
    sum += a[i] * b[i];

  return sum;  // Sum is always 0
}
```

Figure 6.32: **A variable cannot appear in both map and reduction clauses on the same construct** – The `reduction` clause is associated with the `teams` directive. The variable `sum` is not mapped, and therefore, the reduced value of `sum` is lost after the region.

```
int dotp(int *restrict a, int *restrict b, int n)
{
  int sum = 0;

  #pragma omp target map(sum) map(to:a[:n],b[:n])
  #pragma omp teams distribute reduction(+:sum)
  for (int i=0; i<n; i++)
    sum += a[i] * b[i];

  return sum;
}
```

Figure 6.33: **Use a separate target construct to map reduction variables** – The variable `sum` is private in the teams region, but now mapped in the target region.

variables in an accelerator's device data environment will change as variables are mapped and unmapped from it.

Depending on the memory architecture of the heterogeneous system, the original and corresponding variables may or may not share the same storage location. Because of this, a user must consider these two aspects of mapped variables:

- Because the original and corresponding variables *may* share the same storage location, a mapped variable should be thought of like a shared variable. This means that if either the original or the corresponding variable is written to by a thread, synchronization and memory consistency operations are required to avoid data races.
- Because the original and corresponding variables *may not* share the same storage location, copy operations might be required to make the original and corresponding variables consistent. These copy operations can be costly in regards to performance and should be avoided if possible.

If a variable is accessed in a target region, but the variable does not appear as a list item in a `map` clause on the construct then there are default rules to determine if the variable is mapped or private. These rules and the `defaultmap` clause are covered in Section 6.6.3. Structure members may appear as list items in a `map` clause but with some limitations that are described in Section 6.6.2. Section 6.6.4 shows how to access previously mapped memory using pointer-based array sections with a length of zero.

6.6.1 The Map Clause

Variable names, array sections, and structure elements may appear as list items in a `map` clause. An optional *map-type* and `always` *map-type-modifier* control how the list items are mapped. The syntax of the `map` clause is shown in Figure 6.34. If no *map-type* is specified, the default is `tofrom`.

There are three phases that occur when mapping a variable in a `target` region:

1. The *map-enter* phase occurs on entry to the `target` region when the variable is mapped to the accelerator.
2. The *compute* phase occurs when, during the execution of the `target` region, threads executing on the accelerator access the mapped variable.
3. The *map-exit* phase occurs on exit from a `target` region when the variable is unmapped from the accelerator.

The map-enter and map-exit phases manage the storage allocation and copy operations for a mapped variable. In the map-enter phase storage is allocated for the variable in the accelerator's address space, and then the value of host's original variable is copied to the accelerator's corresponding variable. In the map-exit phase, the value of the accelerator's corresponding variable is copied to the host's original variable, and then the storage for the corresponding variable in the accelerator's address space is released.

In Figure 6.35, the map-enter phase occurs on entry to the target region at line 3. Storage is allocated for the three corresponding arrays `a`, `b` and `t` in the accelerator's address space. The values of the host's original `a`, `b` and `t` array variables are then copied to the accelerator's corresponding array variables.

The map-exit occurs on exit from the target region at line 14. The values of the accelerator's corresponding array variables are copied back to the host's original array variables, and the storage for the three variables in the accelerator's address space is then released.

Notice that, in the compute phase, the arrays `a` and `t` are only written to and that array `b` is only read from. Let's assume that `t` is a temporary variable, and that the values written to it are never used after the target region. It is apparent then that the copies for `a` and `c` that occur during the map-enter phase are not needed. Further, the copies that occur during the map-exit phase for `b` and `c` are not needed.

The `map` clause's *map-type* is used to optimize the copies that occur during the map-enter and map-exit phases. On many heterogeneous systems, it is costly to copy variables between the host and an accelerator. The *map-type* is used to disable these copies as shown in Figure 6.36.

`map` *([[map-type-modifier[,]] map-type:] list)*

where the optional *map-type* is one of:
`alloc`
`to`
`from`
`tofrom`
`release`
`delete`

where the optional *map-type-modifier* is: `always`

Figure 6.34: **Syntax of the map clause in C/C++ and Fortran** – The map clause controls how variables are mapped.

In Figure 6.37 the code from Figure 6.35 is updated to use explicit map-types. The map-enter phase occurs on entry to the target region at line 4 and storage is allocated for the three corresponding arrays `a`, `b` and `t` in the accelerator's address space. Only the value of the host's original `b` array variable is copied to the accelerator's corresponding `b` array variable. The corresponding `a` and `t` array variables are left uninitialized. The map-exit phase occurs on exit from the target region at line 15. Only the value of the accelerator's corresponding `a` array variable is copied

```
 1 #include <stdlib.h>
 2 void func(float a[1024], float b[1024], int t[1024])
 3 {
 4   #pragma omp target map(a, b, t) // Map-enter
 5   {
 6     int i;
 7
 8     for (i=0; i<1024; i++)
 9       t[i] = rand()%1024;
10
11     for (i=0; i<1024; i++)
12       a[i] =  b[t[i]];
13
14   } // End of target, map-exit
15 }
```

Figure 6.35: **Example of the map clause** – Copies occur for the arrays `a`, `b`, and `t` at the entry to and exit from the target region.

`map-type`	Perform map-enter copies	Perform map-exit copies
`alloc`	No	No
`to`	Yes	No
`from`	No	Yes
`tofrom`	Yes	Yes
`release`	–	No
`delete`	–	No

Figure 6.36: **Map-type effect on mapping variables** – The default *map-type* is `tofrom`. The `release` and `delete` *map-types* apply only to the map-exit phase and can only appear in a `map` clause on a `target exit data` construct (See Section 6.8.3).

```c
#include <stdlib.h>
void func(float a[1024], float b[1024], int t[1024])
{
  #pragma omp target map(from:a) map(to:b) \
                     map(alloc:t) // Map-enter
  {
    int i;

    for (i=0; i<1024; i++)
      t[i] = rand()%1024;

    for (i=0; i<1024; i++)
      a[i] =  b[t[i]];

  } // End of target, map-exit
}
```

Figure 6.37: **Example of the map clause with map-types** – Eliminate superfluous copies by using map-types.

to the host's original `a` array variable. The storage in the accelerator's address space for all three variables is then released. Using map-types, unnecessary copy operations have been eliminated.

On entry to a `target`, `target data`, or `target enter data` construct, a map-enter phase occurs. Likewise, on exit from a `target`, `target data`, or `target exit data` construct, a map-exit phase occurs. The `target data`, `target enter data`, and `target exit data` data mapping constructs only map variables and do not execute any code on an accelerator (see Section 6.8).

What happens when a construct maps a variable, but that variable has already been mapped by a `target enter data` construct or by an enclosing `target data` construct? There can be only one instance of a corresponding variable in an accelerator's device data environment. A reference count is associated with each corresponding variable. When a variable is *present* in an accelerator's device data environment, its corresponding variable's reference count is greater than or equal to one. The map-enter and map-exit phases increment or decrement the corresponding variable's reference count. The point of the reference count is to keep track of the number of times a variable has been mapped and to only remove it from the

```
 1 #include <stdlib.h>
 2 #include <stdio.h>
 3 void func(float a[1024], float b[1024], int t[1024])
 4 {
 5   #pragma omp target data map(from:a) map(to:b) \
 6                           map(alloc:t) // Map-enter
 7   {
 8     #pragma omp target map(always,from:t) // Map-enter
 9     for (int i=0; i<1024; i++) {
10       t[i] = rand()%1024;
11     } // Map-exit
12
13     for (int i=0; i<1024; i++)
14        printf("t[%d]=%d\n", i, t[i]);
15
16     #pragma omp target map(a,b,t) // Map-enter
17     for (int i=0; i<1024; i++) {
18       a[i] =  b[t[i]];
19     } // Map-exit
20
21   } // End of target data, map-exit
22 }
```

Figure 6.38: **Example of a variable appearing in nested map clauses** – There is only one instance of a variable in an accelerator's address space.

accelerator device data environment after it has been unmapped the same number times.

In the code in Figure 6.38 `target data` construct at line 5 maps the variables `a`, `b` and `t` according to their respective map-types. However, the variables `a`, `b` and `t` are mapped again by the enclosed `target` constructs at lines 8 and 16.

The `target data` construct's map-enter phase at line 5 allocates storage in the accelerator's address space for the variables `a`, `b` and `t`. Only the corresponding variable `a` is assigned the value of the host's original `a` variable. The `b` and `t` corresponding variables are uninitialized.

The target construct's map-enter phase at line 8 does not allocate storage for the variable `t`, because `t` is already present in the accelerator's device data environment.

The `always` *map-type-modifier* combined with the `from` *map-type* forces a copy of `t` from the accelerator to the host during the map-exit phase at the end of the target region at line 11.

The target construct's map-enter phase at line 16 does not allocate storage or copy the variables `a`, `b` and `t` to the accelerator, because they are already present in the accelerator's device data environment. Likewise, the map-exit phase at line 19 does not copy the variables back to the host or release the storage on the accelerator. In this case, it is as if the effects of the `map` clause are ignored.

Finally, at line 20 the map-exit phase for the `target data` construct copies the value of the `b` from the accelerator and then releases the storage for all three variables.

At the start of the map-enter phase, if a corresponding variable's reference count is greater than or equal to one, then no new storage is allocated, and the value of the original variable is not copied to the corresponding variable. The only effect is that the reference count is incremented.

Likewise, at the start of a map-exit phase, if a corresponding variable's reference count is greater than one, then the value of the corresponding variable is not copied back to the original variable, and its storage is not released. The only effect is that the reference count is decremented.

The `always` *map-type-modifier* asserts that the map-enter and map-exit copies should occur regardless of the reference count. It provides a way to force a copy to occur.

The steps associated with the map-enter and map-exit phases are updated to incorporate the reference count, *map-type*, and *map-type-modifier* as shown below:

The map-enter phase:

1. If a corresponding variable is not present in the accelerator's device data environment then:
   - Allocate storage in the accelerator's address space for the corresponding variable, and initialize its reference count to zero.
2. Increment the corresponding variable's reference count by one.
3. If the `to` or `tofrom` map-type is specified then:
   - If the corresponding variable's reference count is one or the `always` *map-type-modifier* is specified, then assign the value of the host's original variable to the accelerator's corresponding variable.

The map-exit phase:

- If a corresponding variable is present in the accelerator's device data environment then:

  1. If the `from` or `tofrom` *map-type* is specified then:
     - If the corresponding variable's reference count is one or the `always` *map-type-modifier* is specified, then assign the value of the accelerator's corresponding variable to the host's original variable.
  2. Decrement the corresponding variable's reference count by one.
  3. If the `delete` *map-type* is specified and the corresponding variable's reference count is not infinite, then set it to zero.
  4. If the corresponding variable's reference count is zero, then release the storage for the corresponding variable in the accelerator's address space.

Although the `alloc` and `release` *map-types* do not appear explicitly above, their effect is to either decrement or increment the reference count. The `delete` *map-type* sets the reference count to zero. It provides a way to force the removal of a variable from an accelerator's device data environment. Globally mapped variables and device memory associated to host memory by the `omp_target_associate_ptr()` function have an infinite reference count and cannot be removed. Both the `delete` and `release` *map-types* may appear only in `map` clauses on the `target exit data` construct (see Section 6.8.3).

6.6.2 Mapping Structure Members

Similar to how an array section can map a subset of the elements in an array, individual structure members can appear in `map` clauses in order to map a subset of the members in a structure variable. The restrictions on mapping structure members are as follows:

- Structure members must explicitly appear in `map` clauses, otherwise a reference to a structure member in a `target` construct implicitly maps the whole structure.

- To map a subset of the members in a structure variable, all structure members in the subset must appear in a `map` clause(s) on the same construct.

- If a subset of members of a structure variable are mapped, only the structure members in the subset can be referenced in a `target` region.
- When a subset of members of a structure variable are mapped, the subset may not be increased by additionally mapping other members of the structure variable.
- C/C++ structure members with type pointer may appear as the base of an array section, but only the rightmost structure member can specify an array section. For example, `S.a[:100]` is legal syntax, but `S.b[:100].z` is not.

To illustrate these concepts, some simple use cases are presented in Figure 6.39. A structure type `Stype` is declared at line 1. Lines 3 – 4 use the `declare target` construct to declare that `f1()`, `f2()`, and `f3()` are mapped functions and may be called from a target region (see Section 6.7).

For the `target` construct at line 8, the whole structure variable S is mapped even though only the members `S.x` and `S.y` are referenced in the construct. The structure members `S.x` and `S.y` appear explicitly in a `map` clause on the `target` construct at line 11, and now only these two members are mapped instead of the whole structure variable `S`. The `map` clause on the `target` construct at line 14 has an array section where `S.p` is used as the base of the array section.

On line 17, only `S.x` is mapped, but the call at line 18 refers to the address of `S`. This is allowed as long as the code inside the function `f3()` accesses only `S.x`. In the last example that spans lines 20 – 24 the structure member `S.x` is mapped by the `target data` construct at line 20. The `target` construct then maps `S.y`. This is an error because it tries to change the subset of mapped structure members (the fourth restriction above). This may be remedied by adding `S.y` to the `map` clause on line 20.

6.6.3 The Defaultmap Clause and Data-mapping Attributes

If and how a variable is mapped is determined by the variable's data-mapping attributes. The rules for determining the data-mapping attributes for a variable are either explicitly or implicitly determined.

The data-mapping attributes for variables that appear in clauses or are declared in a scope inside the `target` construct are explicitly determined according to the following rules:

```
typedef struct { int x, y, size, *p; } Stype;
extern Stype S;
extern int f1(int,int), f2(int*,int), f3(Stype *);
#pragma omp declare target to(f1, f2, f3)

void foo1(Stype S)
{
   #pragma omp target
   f1(S.x, S.y);

   #pragma omp target map(S.x, S.y)
   f1(S.x, S.y);

   #pragma omp target map(S, S.p[:S.size])
   f2(S.p, S.size);

   #pragma omp target map(S.x)
   f3(&S);

   #pragma omp target data map(S.x)
   {
      #pragma omp target map(S.y) // Error
      f1(S.x, S.y);
   }
}
```

Figure 6.39: **Example of mapping structure members** – Structure members may appear in map clauses and array sections with some restrictions.

- A variable that appears in a `map` clause is mapped according to the *map-type*.
- A variable that appears in a `private`, `firstprivate`, or `is_device_ptr` clause is private.
- A variable that is declared in a scope inside the `target` construct is private.

The data-mapping attributes for all other variables referenced in a `target` construct are implicitly determined according to the following rules:

- If the variable is not a scalar variable, the variable is mapped with a *map-type* of `tofrom`.
- If the `target` construct does not have a `defaultmap(tofrom:scalar)` clause, then a *scalar* variable is `firstprivate`.
- If the `target` construct has a `defaultmap(tofrom:scalar)` clause, then a *scalar* variable is mapped with a *map-type* of `tofrom`.
- A pointer variable that appears in a pointer based array section in C/C++ is `private`.

In general, variables with implicitly determined data-mapping attributes are treated as if they had appeared in a `map` clause with a *map-type* of `tofrom`. The exception is scalar and C/C++ pointer variables. Scalar variables have a base language built-in type (for example, `int` or `float` in C/C++).[5]

The performance overhead of a mapped variable can be higher than a `private` variable. A mapped variable requires a presence check to see if the variable is already present in the accelerator's device data environment. Variables that are `private` do not require this check. The overhead of performing the presence check is more pronounced for the smaller variables, such as scalars. For these reasons, the implicit data-sharing attribute for scalar variables referenced in a `target` construct is `firstprivate`. However, the `defaultmap(tofrom:scalar)` clause is provided to change the implicit data-mapping attribute for these variables to mapped with a *map-type* of `tofrom`.

In the `saxpy()` function shown in Figure 6.40, the scalar variables `a` and `n` are `firstprivate`. Because they are used in array sections, the pointer variables `y` and `x` are `firstprivate` and initialized with the address of the corresponding array section in the accelerator's device data environment (see Section 6.3.4).[6] Each initial thread created by the `target teams` construct at line 3 gets a private instance of `a`, `n`, `y` and `x`.

Notice that if a variable is `firstprivate`, then any changes to that variable in the target region will not be reflected back to the original variable. This can result in unexpected results. For example, in Figure 6.41 the variable `sum` is `firstprivate`. After the target region completes at line 8, the computed value of `sum` is lost, and the

[5]The precise definition of a scalar variable is determined by the base language.

[6]The original pointer value is discarded when the host and accelerator do not share the same address space.

```
void saxpy(float *restrict y, float *restrict x, float a, int n)
{
  #pragma omp target teams map(y[:n]) map(to:x[:n])
  {
    #pragma omp distribute
    for (int i=0; i<n; i++) {
        y[i] =  y[i] + a*x[i];
    } // End of distribute
  } // End of target teams
}
```

Figure 6.40: **Example of default data-mapping attribute rules** – The pointer variables `x` and `y` are `private`. The scalar variables `a` and `n` are `firstprivate`.

```
float dotp(float *restrict y, float *restrict x, int n)
{
  float sum = 0.0;
  #pragma omp target map(y[:n]) map(to:x[:n])
  {
    for (int i=0; i<n; i++)
      sum += y[i] * x[i];
  } // End of target

  return sum;
}
```

Figure 6.41: **Example of problems with implicit firstprivate variables** – Because the variable `sum` is `firstprivate`, the computed value of `sum` is lost at the end of the target region.

`dotp()` function always returns 0.0. To correct the problem with the `sum` variable, either place it in a `map` clause or add a `defaultmap(tofrom:scalar)` clause to the `target` construct. It is recommended to not rely on implicit behavior and instead list variables explicitly in `map`, `private`, or `firstprivate` clauses as appropriate.

6.6.4 Pointers and Zero-length Array Sections

In Section 6.3.4, we showed how array sections are used to map memory that a pointer variable points to. Recall that an array section has a length. If the length of an array section is zero, then it is a *zero-length array section*.

A pointer-based zero-length array section has a special meaning in a `map` clause on a `target` construct. The pointer variable is `private` in the target region. If the value of the original pointer is an address that is already mapped, then the corresponding `private` pointer variable in the `target` region is assigned the corresponding device address. If the value of the original pointer is an address that is not mapped, then the corresponding `private` pointer variable is assigned NULL.

A pointer-based zero-length array section is convenient when a pointer variable is used in a `target` construct, and the memory it points to was mapped (for example, by an enclosing target data region) before the `target` construct is encountered. The code that uses the pointer variable in the target region might not refer to the length of the pointed-to memory.

Pointer variables referenced in a `target` construct that have an implicit data-mapping attribute are treated as if they had appeared in a `map` clause as the pointer variable in a zero-length array section with a *map-type* of `tofrom`. Some simple examples using zero-length array section are shown in Figure 6.42.

The `map` clause on the `target data` construct on line 11 maps the array `A` and the dynamically allocated memory that `p` points to. The pointer variable `p` is not mapped. Because it appears in an array section, the pointer variable `p` is `private` in the `target` construct starting at line 13. The `private` variable `p` is assigned the corresponding device address of the memory allocated at line 8.

For the `target` construct at line 16, the pointer variable `q` is implicitly treated as if it appeared in a `map` clause in a zero-length array section. At line 17, `q` is a private pointer variable that is pointing at the device address of the mapped array `A`. If you explicitly map a pointer variable as shown in line 19, then you are mapping the value of a host address to the accelerator. The mapped pointer variable does not contain a valid accelerator memory address. The call to `f1()` will possibly de-reference the invalid address stored in `p`.

```
#include <stdlib.h>
void f()
{
   char *p, *q, A[128];
   extern void f1(char *);
   #pragma omp declare target to(f1)

   p = malloc(1024);
   q = A;

   #pragma omp target data map(p[:1024], A)
   {
     #pragma omp target map(p[:0])
     f1(p);

     #pragma omp target // Implicit map(q[:0])
     f1(q);

     #pragma omp target map(p)
     f1(p); // Error
   }
   free(p);
}
```

Figure 6.42: **Example of C/C++ pointers as zero-length array sections** – Pointer variables are implicitly treated as pointer-based zero-length array sections in target regions.

6.7 The Declare Target Directive

The `declare target` construct is used for both functions and variables. If a function is called from a target region, then the name of the function must appear in a `declare target` directive. The `declare target` directive is used to map global variables to an accelerator's device data environment for the whole execution of the program. The `declare target` construct syntax in C/C++ and Fortran is shown in Figure 6.43 along with its supported clauses.

The various syntactical forms of the `declare target` directive result in an *extended-list* of variable and function names, or in the case of the `link` clause,

#pragma omp declare target
declarations-definitions-seq
#pragma omp end declare target
or
#pragma omp declare target*(extended-list)*
or
#pragma omp declare target *clause[[[,] clause] ...]*

!$omp declare target*(extended-list)*
or
!$omp declare target *[clause[[,] clause]...]*

where clause is:
to *(extended-list)*
or
link *(list)*

Figure 6.43: **Syntax of the declare target construct in C/C++ and Fortran** – The various syntactical forms of the directive result in a *extended-list* of variable and function names.

a list of variable names. In describing the functionality, we will refer to a function or variable name that appears in a `declare target` directive.

In C/C++, the `declare target` and `end declare target` directives provide a convenient means to create an *extended-list* of the names of variables and functions that are declared between the two directives, where the variable declarations are at file or namespace scope and the function declarations are at file, namespace, or class scope. An *extended-list* of function and variable names may appear as list items on the directive or in clauses. The `to` clause accepts an *extended-list* of variable and function names. The `link` clause accepts only a list of variable names. In Fortran, the `declare target` directive without clauses or an *extended-list* may appear in the interface specification for a subroutine, function, program or module.

If a function name appears in a `declare target` directive in the same translation unit as the definition of the function, then it is a *mapped function.* A mapped function has a corresponding accelerator-specific version of the function. A function name must appear in a `declare target` directive before the function is called from a `target` construct or another mapped function.

A variable whose name appears in the `declare target` directive must have static storage duration. In C/C++, these are variables that are declared at file, namespace, static-block, or static-class scope. In Fortran, these are named variables and named common blocks.

A variable name that appears in a `declare target` directive in the same translation unit where it is defined is *globally mapped*. If the variable is referenced in a mapped function or device construct, it must appear in a `declare target` directive in the same translation unit as the function or construct.

A variable that is globally mapped is created and initialized in an accelerator's device data environment before a program begins execution. Globally mapped variables have an infinite reference count and are never removed from an accelerator's device data environment. They are permanently mapped for the execution of the whole program. Some examples of the `declare target` directive are shown in Figure 6.44.

The first form of the `declare target` directive on line 3 declares that the variables `Lastpos` and `Buf` are globally mapped. The original `Lastpos` variable on the host and an accelerator's corresponding version of the variable are initialized to 0. The variable `Buf` is declared but not defined in the example. `Buf` must appear in a `declare target` directive in the place where `Buf` is defined.

The `declare target` directive on line 6 uses the `to` clause to declare that `F()` is a mapped function. The prototype of the function `F()` is declared, but `F()` is not defined in the example. The function name `F` must appear in a `declare target` directive in the place (some other file) where the definition of `F()` occurs.

The `declare target` on line 8 and the `declare end target` on line 20 declare that variable `State` and `search()` and `find_state()` are mapped functions. The host's original variable `State` and an accelerator's corresponding version of the variable are initialized to -1. Like `F`, the function name `search` must appear in a `declare target` directive where it is defined. The function `find_state()` is both declared and defined between the pair of directives, and an accelerator-specific version of it is generated. The call to `find_state()` appears inside a `target` construct at lines $24 - 25$. When an initial thread executes the target region, the accelerator-specific version of the `find_state()` function is invoked.

A variable that is globally mapped will reserve memory in an accelerator's device data environment for the whole program. The `link` clause provides another way to access a global variable in a mapped function without having to globally map the variable. Note that function names cannot appear in a `link` clause. The `link` clause declares that a variable is *globally linked*. A reference to a globally linked variable

```
int Lastpos = 0;
extern char Buf[128];
#pragma omp declare target(Lastpos, Buf)

extern int F(int, int);
#pragma omp declare target to(F)

#pragma omp declare target
int State = -1;
extern int search(char *);

void find_state(char c)
{
  int pos;
  Buf[Lastpos] = c;
  pos = search(Buf);
  Lastpos = pos;
  State = F(Lastpos, pos);
}
#pragma omp end declare target

void process_input(char c)
{
  #pragma omp target firstprivate(c)
  find_state(c);
}
```

Figure 6.44: **Example of the declare target directive** – Various forms of the directive all have the same effect.

can appear in a mapped function with the restriction that before the function is called from a target region, the variable will have been mapped by a `map` clause.

An example of `link` clause is shown in Figure 6.45. On line 2 the variable `Vector` is globally linked by the `declare target` directive, which declares that the global variable `Vector` will be mapped before the `compute()` function is executed on the accelerator. The `map` clause on the `target` construct at line 15 maps `Vector`. The call to `compute()` in the target region can then find the mapped `Vector` variable.

```
float Vector[1024];
#pragma omp declare target link(Vector)

#pragma omp declare target
extern float F(float , float);
int compute(float a)
{
   for (int i=0; i<1024; i++)
      Vector[i] = F(Vector[i], a);
}
#pragma omp end declare target

int update_vector(float a)
{
  #pragma omp target map(Vector) firstprivate(a)
  compute(a);
}
```

Figure 6.45: **Example of the link clause on a declare target directive** – Variables appearing in the `link` clause are *globally linked*. They must be mapped before they are referenced in a mapped function.

6.8 The Data-mapping Constructs

This section describes a group of constructs that map variables and manage the consistency of mapped variables between the host and an accelerator. A map-enter and map-exit phase occurs at the entry and exit of each target region for variables that are mapped in the region. Redundantly mapping a variable in multiple target regions can be detrimental to performance, especially when the host and the accelerator do not share memory. The `target data` construct described in Section 6.8.1 maps variables across a region of code that encloses multiple target regions. Variables are mapped once and then used in many enclosed target regions. The `target enter data` and `target exit data` construct are not associated with a specific code region and perform only the map-enter or map-exit phase, respectively. These two unstructured constructs are described in Section 6.8.3. Once a variable is mapped, the `target update` construct described in Section 6.8.2 can be used to make its value consistent between the host and the accelerator where it is mapped.

#pragma omp target data *[clause[[,] clause]. . .]* *structured block*
!$omp target data *[clause[[,] clause]. . .]* *structured block* **!$omp end target data**

Figure 6.46: **Syntax of the target data construct in C/C++ and Fortran** – Map variables to a device for the extent of the region.

if *([*`target data`*:] scalar-expression)*	(C/C++)
if *([*`target data`*:] scalar-logical-expression)*	(Fortran)
map *([[map-type-modifier[,]] map-type:] list]*	
device *(integer-expression)*	(C/C++)
device *(scalar-integer-expression)*	(Fortran)
use_device_ptr *(list)*	

Figure 6.47: **Clauses supported by the target data construct** – The `if` and `device` clauses are discussed in Section 6.10. The `map` clause is discussed in Section 6.6.1. The `use_device_ptr` clause is discussed in Section 6.11.2.

6.8.1 The Target Data Construct

The `target data` construct maps variables to a device data environment for the extent of the target data region. In other words, when a thread executing on the host device encounters a `target data` construct, variables appearing in `map` clauses on the construct are mapped according to the map-enter phase. However, unlike the `target` construct, the host thread continues executing the code inside the target data region. Once the host thread encounters the end of the target data region, the map-exit phase occurs for the variables that appeared in the `map` clauses on the construct.

In summary, a `target data` construct is like a `target` construct minus the code executing on the accelerator. It only maps variables. The `target data` construct syntax in C/C++ and Fortran is shown in Figure 6.46. Clauses that can appear on the `target data` construct are shown in Figure 6.47.

On entry to and exit from a target data region, map-entry and map-exit phases occur, respectively, for the variables that appear in `map` clauses on the construct. An original variable can have only one corresponding variable in an accelerator's device data environment (See Section 6.6.1). When a variable is mapped by a `target data`

```
 1 #define N (1024*1024)
 2 double A[N], B[N];
 3 extern double F(double * restrict);
 4
 5 void G(double c, double d)
 6 {
 7   double e;
 8   #pragma omp target data map(B)
 9   {
10     #pragma omp target map(B) map(always,from:A) \
11                        firstprivate(c,d)
12     for (int i=0; i<N; i++)
13       A[i] =  B[i] * c + d;
14
15     e = F(A);
16
17     #pragma omp target map(B) firstprivate(e)
18     for (int i=0; i<N; i++)
19       B[i] =  B[i] / e;
20
21   } // End of target data
22 }
```

Figure 6.48: **Example of a target data construct** – The array variable `B` is mapped once to an accelerator across two target regions.

construct, all device constructs enclosed in the target data region that also map that variable will find a corresponding variable present in the accelerator's device data environment and use it. An example of a `target data` construct that encloses two `target` constructs is shown in Figure 6.48.

The variable `B` is mapped by the `target data` construct at line 8, which encloses the two `target` constructs at lines 10 and 17. Storage is allocated in the accelerator's device data environment for the corresponding `B` variable, assigned the value of the host's original `B` variable, and its reference count is initialized to one.

The host thread that encounters the `target data` construct executes the code inside the target data region. It encounters the `target` construct at line 10, which maps `B`. During its map-enter phase, `B` is found to already be present in the accelerator's device data environment and the reference count for `B` is incremented to two. When the target region completes and during the map-exit phase, the reference count for `B` is decremented back to one. `B` is not removed from the accelerator's device data environment.

After the target region spanning lines 10 − 13 has completed, the host thread continues executing the code in the target data region calling the function `F()` at line 15 and then encountering the second `target` construct at line 17. The reference count for the variable `B` is again incremented at the entry to and decremented at the exit from the target region. Because it still has a reference count greater-than zero, `B` remains mapped.

Finally, the host thread continues execution after the second target region is complete and then completes the target region at line 21. During the map-exit phase for the variable `B`, its reference count is decremented to zero, the value of the accelerator's corresponding `B` variable is copied to the host's original `B` variable, and the storage for `B` on the accelerator is released.

The variable `A` may be mapped by a `target data` or `target enter data` construct before the function `G()` is called, and so it would be present in the accelerator's device data environment with a non-zero reference count. However, the updated value of `A` is always needed for the call to `F()` at line 15. The `always` *map-type* in the `map` clause on line 10 ensures that, regardless of the reference count for `A`, after the associated target region completes, the value of the variable `A` on the host and the accelerator is always consistent (the same).

6.8.2 The Target Update Construct

If a variable is mapped by the `target data` or `target enter data` constructs or by the `declare target` directive, there are instances when a host thread might need to access the mapped variable. The `target update` construct makes the value of a mapped variable the same on the host as on an accelerator. It either assigns the value of the host's original variable to the accelerator's corresponding variable, or it assigns the value of the corresponding variable to the original variable. It makes the original and corresponding variables consistent. The `target update` construct syntax in C/C++ and Fortran is given in Figure 6.49. The clauses that are available on the `target update` construct are shown in Figure 6.50.

#pragma omp target update *[clause[[,] clause]. . .]*
!$omp target update *[clause[[,] clause]. . .]*

Figure 6.49: **Syntax of the target update construct in C/C++ and Fortran** – Make the value of a mapped variable consistent between the host and an accelerator.

if *([*`target update`*:] scalar-expression)*	(C/C++)
if *([*`target update`*:] scalar-logical-expression)*	(Fortran)
device *(integer-expression)*	(C/C++)
device *(scalar-integer-expression)*	(Fortran)
nowait	
depend *(dependence-type: list)*	
to *(list)*	
from *(list)*	

Figure 6.50: **Clauses supported by the target update construct** – The `if` and `device` clauses are discussed in Section 6.10. The `map` clause is discussed in Section 6.6.1. The `nowait` and `depend` clauses are discussed in Section 6.9. The `to` and `from` clauses are described below.

An example using the `target update` construct is shown in Figure 6.51. At line 4 the array variable `B` is globally mapped by the `declare target` directive (see Section 6.7). The `target update` construct at line 8 uses the `from` clause to assign the first and last `l` elements of `B` from the accelerator's corresponding `B` variable to the same elements in the host's original `B` variable. It makes the array elements consistent between the host and the accelerator. In this case, it is getting the values from the accelerator's version of `B`.

Let us assume that the function `update_boundary()` at line 9 reads and then writes only the `l` first and last elements of `B`. It updates the boundaries of `B`. When the host writes to the elements in `B`, those elements are no longer consistent between the host and the accelerator. The `target update` construct at line 10 then uses the `to` clause to assign the first and last `l` elements of `B` from the host to the sames elements in the accelerator's `B` variable. It makes the array elements consistent, but this time it is using the values from the host's version of `B`.

```
#define N (1024*1024)
extern void update_boundary(double *, int, int);
double B[N];
#pragma omp declare target(B)

void G(double *restrict B, double e, int n, int l)
{
  #pragma omp target update from(B[0:l],B[n-1-l:l])
  update_boundary(B, n, l);
  #pragma omp target update to(B[0:l],B[n-1-l:l])

  #pragma omp target
  for (int i=0; i<n; i++)
    B[i] =  B[i] / e;
}
```

Figure 6.51: **Example of the target update construct** – The array variable `B` is globally mapped. The target update construct is used make elements at the start and the end the array `B` consistent between the host and the accelerator.

Notice that the `target update` construct is executed by a host thread. The construct cannot appear in a target region. This example used a variable that was globally-mapped, but any mapped variable may appear in a `to` or `from` clause (for example, one that was mapped by an enclosing target data region).

6.8.3 The Target Enter and Exit Data Constructs

The `target data` construct applies to a subsequent structured block. There is a map-enter phase that occurs on entry to and a map-exit phase that occurs on exit from a target data region. However, sometimes the way we want to map variables does not fit a structured block model. We want to map variables in an unstructured way. The target enter and exit data constructs provide this capability. These are standalone constructs that are not associated with a statement or structured block of code. When a host thread encounters the `target enter data` construct, a map-enter phase occurs for variables that appear in `map` clauses on the construct. Similarly, a map-exit phase occurs for variables in `map` clauses on the `target exit data` construct when it is encountered.

#pragma omp target enter data *[clause[[,] clause]. . .]*
#pragma omp target exit data *[clause[[,] clause]. . .]*
!$omp target enter data *[clause[[,] clause]. . .]*
!$omp target exit data *[clause[[,] clause]. . .]*

Figure 6.52: **Syntax of the target enter and exit data constructs in C/C++ and Fortran** – Standalone constructs for mapping variables to and from an accelerator's device data environment.

if *([*`target enter data`*:] scalar-expression)*	(C/C++)
if *([*`target exit data`*:] scalar-expression)*	(C/C++)
if *([*`target enter data`*:] scalar-logical-expression)*	(Fortran)
if *([*`target exit data`*:] scalar-logical-expression)*	(Fortran)
map *([[map-type-modifier[,]] map-type:] list]*	
device *(integer-expression)*	(C/C++)
device *(scalar-integer-expression)*	(Fortran)
nowait	
depend *(dependence-type: list)*	

Figure 6.53: **Clauses supported by the target enter and exit data constructs** – The `if` and `device` clauses are discussed in Section 6.10. The `map` clause is discussed in Section 6.6.1. The `nowait` and `depend` clauses are discussed in Section 6.9.

The syntax for the `target enter data` and `target exit data` constructs in C/C++ and Fortran is shown in Figure 6.52. The clauses that are available on the constructs are shown in Figure 6.53.

The *map-types* that may appear in a `map` clause on a `target enter data` construct are restricted to `alloc` and `to`. The *map-types* that may appear in a `map` clause on a `target exit data` construct are restricted to `release`, `from`, and `delete`. An explicit *map-type* must be specified in `map` clauses on these constructs. A `tofrom` *map-type* is not legal in a `map` clause on either construct.

The `delete` *map-type* may appear only in a `map` clause on a `target exit data` construct. It sets the reference count of a corresponding variable to zero and removes it from an accelerator. Its purpose is to force the removal of a variable from an accelerator's device data environment no matter what its reference count is.

A C++ class declaration is shown in Figure 6.54 with two member functions `allocate()` and `release()`. At line 9, the `target enter data` construct executes a map-enter phase for a pointer-based array section. The pointer variable in the

```
 1 class myArray {
 2   int length;
 3   double *ptr;
 4
 5   void allocate(int l) {
 6     double *p = new double[l];
 7     ptr = p;
 8     length = l;
 9     #pragma omp target enter data map(alloc:p[0:length])
10   }
11
12   void release() {
13     double *p = ptr;
14     #pragma omp target exit data map(release:p[0:length])
15     delete[] p;
16     ptr = 0;
17     length = 0;
18   }
19 };
```

Figure 6.54: **C++ Example of the target enter and exit data constructs** – The `allocate()` method will execute a map-enter phase for the dynamically allocated memory pointed to by `p`. The `release()` method will execute the corresponding map-exit phase.

array section is `p`, and it contains the address of the memory that was dynamically allocated at line 6. The `alloc` *map-type* indicates that the corresponding memory for the array section on the accelerator is not initialized with any value.

At line 16, in the `release()` member function, the `target exit data` construct executes a map-exit phase for a pointer-based array section. It is expected that the class member `ptr` is still pointing at the memory allocated in the `allocate()` member function. The locally scoped pointer variable `p` is assigned the value of `ptr` and then used in the pointer-based array section. The `release` *map-type* in the `map` clause frees the corresponding storage for the array section in the accelerator's device data environment. The corresponding array section is not copied back to the original host memory.

```
1 void G(char S[128], int v)
2 {
3   extern int mutate(char *s, int);
4   while (!v) {
5     #pragma omp target enter data map(to:S)
6     v = mutate(S, v);
7   }
8   #pragma omp target exit data map(delete:S)
9 }
```

Figure 6.55: **Example of the delete map-type** – Regardless of its reference count, remove `S` from an accelerator's device data environment.

Notice that the array sections in both `map` clauses are examples of pointer-based array sections where only the pointed-to memory is mapped and not the pointer variable itself. The locally scoped `p` variables are required because the C++ `this->ptr` is an lvalue expression, not a pointer variable. The variable that appears in an array section must be an array name or a pointer variable.

An example using the `delete` *map-type* is shown in Figure 6.55. Line 5 is executed zero or more times depending on the value of the variable `v`. Each time a host thread executes the `target enter data` construct, a map-enter phase occurs for `S`. The first map-enter phase allocates storage, initializes the corresponding variable `S` with the value of the host's original `S` variable, and sets the corresponding variable's reference count to one. Each subsequent map-enter phase increments the reference count. The `map` clause on the `target exit data` construct on line 8 uses the `delete` *map-type* to set the reference count of `S` to zero. The variable `S` is then removed from the accelerator's device data environment. The value of `S` on the host after line 9 is undefined since the `delete` *map-type* does not copy the corresponding variable back to the original host variable.

When a variable appears in a `map` clause on a `target exit data` construct and that variable is not present in the accelerator's device data environment, then it is ignored by the construct. For example, in the case where the variable `v` is 0 on entry to the function, the `target exit data` construct at line 8 does nothing.

The `target enter data` and `target exit data` constructs have another potential benefit. Using the `nowait` clause, the constructs may execute as deferrable tasks. This enables the possibility of asynchronously executing map-enter and map-exit data transfers while the host thread continues executing other tasks in parallel.

6.9 The Nowait Clause on Device Constructs

To execute code in parallel on both the host and an accelerator the host thread must not be blocked waiting for a device construct to finish. Section 6.2.4 discussed the target task, which is an explicit task that is generated by a `target`, `target enter data`, `target exit data` or `target update` construct. Because these constructs generate a task, the parallel execution features of OpenMP tasking, covered in detail in Chapter 3, are now available to the device constructs.

By default the target task is included and executed immediately. The host thread that generated the target task cannot continue executing the code after the device construct until the target task completes. The `nowait` clause changes the target task into a deferrable task. This means that after generating the target task, the host thread may immediately continue executing the code after the device construct. An explicit parallel region is not required to execute a host task in parallel with a target task. A single host thread may execute a host task, while in parallel, the device construct is executed in by an accelerator.

The code example in Figure 6.56 uses the `nowait` clause to allow the host thread to continue past the `target` construct at lines $7-9$. A target task is generated that encloses the execution of the target region. A thread on the accelerator executes the target region. In parallel, the host thread executes the loop at lines $11-12$. It then encounters the `taskwait` construct at line 13 and waits there until the target task generated at line 7 is complete. The host thread then executes the remainder of the function.

Similar to the `task` construct, the `depend` clause may be used to express target task dependences. If a `depend` clause appears on a device construct then the generated target task cannot be scheduled to execute until the dependences in the clause are satisfied.

A target task's `private` and `firstprivate` variables are created and initialized when the task is generated. However, map-enter phase assignments for mapped variables occur when the target task executes, and map-exit phase assignments occur when the task completes.

```
extern int max(int,int);
#pragma omp declare target(max)
void F(char *v, short *restrict s, int n)
{
  int i;

  #pragma omp target nowait map(v[0:n])
  for (i=0; i<n; i++)
    v[i] = max(v[i],0);

  for (i=0; i<n; i++)
    s[i] = max(s[i],0);
  #pragma omp taskwait

  for (i=0; i<n; i++)
    s[i] = s[i] - v[i];
}
```

Figure 6.56: **Example using the nowait clause** – Execute the target region on an accelerator in parallel with the code executing on the host.

Tasks may be used to overlap computation with data transfers between the host and accelerator. There can be dependences between target tasks and other tasks generated by the `task` constructs. In Figure 6.57, the `nowait` and `depend` clauses are used to execute target tasks in parallel with other tasks.

At line 7 a parallel region is started with two threads. Because of the `single` construct, one host thread executes the region and generates a sequence of tasks. In the following discussion, the tasks are labeled (*taskname*), when the are generated.

A deferrable target task ($t0$) is generated for the `target enter data` construct at line 10. When it executes, the $t0$ task performs a map-enter for the memory pointed to by `a`. A host task ($h0$) is generated that encloses the call to the function `h0()` at line 14. The $h0$ host task and the $t0$ target task may execute in parallel. A deferrable target task ($t1$) is generated that encloses the call to the function `t1()` at line 18. Because of the task dependences expressed in the `depend` clauses at line 17, the $t1$ target task cannot be scheduled to execute until the $t0$ and $h0$ tasks have completed. A host task ($h1$) is generated that encloses the call to the function `h1()` at line 21. The $h1$ task cannot be scheduled to execute until the dependence

```
 1 extern void h0(int*, int);
 2 extern void h1(int*, int);
 3 extern void t1(int*, int*, int);
 4 #pragma omp declare target(t1)
 5 void F(int *a, int *b, int n)
 6 {
 7   #pragma omp parallel num_threads(2)
 8   #pragma omp single
 9   {
10   #pragma omp target enter data map(to:a[:n]) \
11               nowait depend(out:a[:n]) // t0
12
13   #pragma omp task depend(out:b[:n])
14   h0(b, n);
15
16   #pragma omp target map(to:b[:n]) \
17               nowait depend(in:b[:n]) depend(inout:a[:n])
18   t1(a, b, n);
19
20   #pragma omp task depend(in:b[:n])
21   h1(b,n);
22
23   #pragma omp target exit data map(from:a[:n]) \
24               depend(in:a[:n]) // t2
25   }
26 }
```

Figure 6.57: **Example using the nowait and depend clauses** – Use the `depend` and `nowait` clauses to execute target tasks in parallel with other host tasks.

on `b[:n]` is satisfied, which occurs when the $h0$ task completes. Notice that the $t1$ and $h1$ tasks may execute in parallel and that the $h1$ task may start before $t1$.

An included target task ($t2$) is generated for the `target exit data` construct at line 23, and when it executes, it performs a map-exit phase for the memory pointed to by `a`. A `nowait` clause does not appear on the construct, but there is a `depend` clause. The target task $t2$ cannot be scheduled to execute until the dependence on `a[:n]` is satisfied by the completion of the $t1$ task. The host task (and thus the host thread) that generated $t2$ cannot be resume execution until $t2$ completes. Finally, there is a task synchronization point at the implicit barrier at line 25.

6.10 Selecting a Device

OpenMP provides support for multiple accelerators. Devices are enumerated such that each one has a unique device number. The actual number for a specific accelerator is implementation-defined. A device number for an accelerator must be non-negative and less than the value returned by the `omp_get_num_devices()` runtime function.

There is currently no support in OpenMP to determine the characteristics of a particular accelerator device. The only distinction made between devices is that there is a host device where the program begins execution and an optional set of accelerator devices.

You can find the device number for the host device using the `omp_get_initial_device` routine. The host device number has some odd restrictions. If it is not greater than or equal to zero and less than the value returned by the `omp_get_num_devices` runtime function, then it may be used only in certain device memory runtime functions (see Section 6.12).

Programs using the OpenMP device constructs should be portable, to systems that do not have accelerators. OpenMP supports *host fall back*, which is the concept that a target region can always be executed by the host device. If you take an OpenMP program that contains device constructs and compile and run that program on a system without accelerators, then the device constructs are executed on the host device.

6.10.1 The Default Device and the Device Clause

The device number determines to which device a construct applies. The `device` clause is used to specify a device number. When there is no `device` clause, the device number for a `target`, `target data`, `target update`, `target enter data` or `target exit data` construct is the *default-device-var* ICV. There may be only one `device` clause on a construct, and the expression in the clause must be non-negative. If the device number specified for a device construct does not correspond to any devices in the system where the program is running, then the construct falls back to the host device.

```
#include <omp.h>
extern int a[1024]; int Work(int *, int, int);
#pragma omp declare target to(a, Work)

void F()
{
  int defdev = omp_get_default_device();
  int numdev = omp_get_num_devices();

  for (int i=0; i<numdev; i++) {
    omp_set_default_device(i);
    #pragma omp target update to(a) nowait
  }
  omp_set_default_device(defdev);
  #pragma omp taskwait

  for (int i=0; i<numdev; i++) {
    #pragma omp target device(i) nowait
    Work(a,i,numdev);
  }

  if (numdev == 0) Work(a,0,1);
  #pragma omp taskwait
}
```

Figure 6.58: **Example of the device clause and related runtime functions** – The variable `a` is updated with the host's value on all devices and then the function `Work()` is executed by all devices.

When a program begins execution, the *default-device-var* ICV is initialized to the value of the `OMP_DEFAULT_DEVICE` environment variable. If the environment variable is not set, then the *default-device-var* is implementation-defined. During program execution you may determine the default device using the `omp_get_default_device` function or change the default device using the `omp_set_default_device` function.

An example using the `device` clause and related runtime functions is shown in Figure 6.58. The example first copies the value of the array `a` from the host to all of the accelerators and then starts a target region on all devices. Tasks are used to execute the operations in parallel.

The assignments on lines 7 − 8 use runtime functions to find the current default device and the number of devices. Each time through the loop spanning lines 10 − 13, the default device number is changed. The `target update` construct makes the variable `a` consistent between the host and the default device using the value of `a` from the host. Because of the `nowait` clause, the target update executes as a deferrable target task (see Section 6.9). The host thread does not wait for the target update region to complete.

The default device is restored at line 14. The host thread waits at the `taskwait` at line 15 until all of the tasks started at line 12 have finished. In the loop spanning lines 17 − 20, the `device` clause is used to select a specific device where the target region executes. Again, because of the `nowait` clause a deferrable target task is generated that encloses the target region. The host thread executing the loop does not wait for the target region to complete before continuing. The conditional call to the function `Work()` at line 22 is there just in case `omp_get_num_devices()` returns 0. The `taskwait` at line 23 ensures that all of the tasks started at line 18 have completed before returning from the function.

6.10.2 The If Clause on Device Constructs

An `if` clause may appear on a `target`, `target data`, `target update`, `target enter data` or `target exit data` construct.

Except for the `target update` construct, when the expression in an `if` clause evaluates to *false*, then host fall back occurs and:

- The device for the construct is the host.
- The execution of the region occurs on the host device.
- Variables appearing in `map` clauses are mapped to the host's device data environment.
- If a `device` clause appears on the construct, it is ignored.

When the `if` clause expression evaluates to *false* on a `target update` construct, then assignments resulting from the construct do not occur. The `if` clause is useful for setting a threshold on the amount of computation in a target region versus the

```
#define MB (1024*1024)
extern void Work(float*, int);
#pragma omp declare target(Work)
void F(float * restrict a, int n)
{
   #pragma omp target if(n > MB) map(a[:n])
   Work(a,n);
}
```

Figure 6.59: **Example of an if clause on the target construct** – If `n` is greater than a threshold, execute the target region on the default accelerator. Otherwise, execute the region on the host device.

expected overhead of launching the target region on an accelerator device. A simple code example using the `if` clause is shown in Figure 6.59.

Be careful when using the `if` clause on constructs that map variables or effect the value of mapped variables. For example, if an `if` clause evaluates to false on a `target update` construct and a `target` construct is dependent on the execution of the target update, then a program error may occur.

6.11 The Device Pointer Clauses

The clauses described in this section are used to refer to device pointers. Recall from Section 6.3.3 that a device pointer is a pointer variable on the host whose value is an object that represents an address in device memory.

Assignments to a device pointer are restricted to values that arise from the following cases:

- The return value of the `omp_target_alloc()` function.
- The address of a variable referenced in the lexical scope of a `target data` construct when that variable appeared in a `use_device_ptr` clause on the construct.
- The return value of an implementation-defined device memory allocation function.[7]

[7]For example, CUDA's `cudaAlloc`[19] or OpenCL's `BufferAllocate`[12] functions.

```
 1 #include <omp.h>
 2 void *init(int n, int dev)
 3 {
 4   char *dptr = omp_target_alloc(dev, n);
 5
 6   #pragma omp target is_device_ptr(dptr) device(dev)
 7   for (int i=0; i<n; i++)
 8     dptr[i] = i;
 9
10   return (void*)dptr;
11 }
```

Figure 6.60: **Example of the is_device_ptr clause** – The device pointer variable `dptr` must appear in the `is_device_ptr` clause to de-reference it in the target region.

The operations on a device pointer variable on the host are restricted as follows:

- A device pointer variable cannot be de-referenced.
- Pointer arithmetic cannot be performed on a device pointer variable.

6.11.1 The Is_device_ptr Clause

The purpose of the `is_device_ptr` clause is to provide a way for a device pointer to be accessed in a target region. The `is_device_ptr` clause accepts a list of device pointer variable names and may only appear on a `target` construct.

When a device pointer appears in an `is_device_ptr` clause on a `target` construct, the corresponding variable in the region is private. On entry to the target region, the corresponding private variable is initialized with the device's representation of the address stored in the original device pointer.

A simple example of a `target` construct with an `is_device_ptr` clause is shown in Figure 6.60. In line 4, a device address on the `dev` device is assigned to `dptr`. The device pointer `dptr` is listed in the `is_device_ptr` clause at line 6 indicating that `dptr` is private in the target region. On entry to the target region, the private instance of `dptr` is initialized with the `dev` device's representation of the device address that corresponds to the variable's original value.

```
 1 int A[1024];
 2 #pragma omp declare target to(A)
 3 extern int AccelFunc(void *);
 4
 5 int Func()
 6 {
 7   int err;
 8   #pragma omp target data map(err) use_device_ptr(A)
 9   err = AccelFunc(A); // Requires device address of A
10   return err;
11 }
```

Figure 6.61: **Example of the use_device_ptr clause** – Replace the reference to the host address of `A` in the lexical scope of the `target data` construct with the device address of `A`.

6.11.2 The Use_device_ptr Clause

The purpose of the `use_device_pointer` clause is to provide a way to refer to the device address of a mapped variable, so that it may be passed to a function as a device pointer argument. Some vendors implement optimized functions that expect device pointers as arguments. These optimized functions, called by a host thread, offload their execution to an accelerator, either using a `target` construct with an `is_device_ptr` clause or some other vendor-specific mechanism.

The `use_device_ptr` clause accepts a list of variable names. The clause valid only on a `target data` construct. The clause applies to the lexical scope of its associated `target data` construct. In the construct, references to variables that appear in a `use_device_ptr` clause must be to the address of the variable. In the lexical scope of the `target data` construct, a reference to the address of a variable that appears in a `use_device_ptr` clause is replaced with the corresponding device address.

In Figure 6.61, the variable `A` is globally mapped by the `declare target` construct at line 2. Because of the `use_device_ptr` clause on the `target data` construct at line 8, the reference to the address of `A` in the call to `AccelFunc()` at line 9 is replaced with the device address of `A`.

6.12 Device Memory Functions

OpenMP provides a set of runtime functions for creating, copying, and mapping dynamically allocated device memory. They use device pointers (see Section 6.3.3 on page 265) and provide advanced users with capabilities to share complex data structures across devices. These device memory functions are described in detail in Section 2.3.4 starting on page 70. In this section, some simple examples using the functions are presented and discussed.

Copy a Linked List To Device Memory The example code in Figure 6.62 uses the `omp_target_memcpy()` function to copy a linked list from the host to the default device. Storage for the linked list is allocated in the default device's memory using the `omp_target_alloc()` function.

After counting the number of elements in the list, the `omp_target_alloc()` function is called at line 14 to allocate memory for the linked list in the accelerator's address space. The variable `dst` is assigned the device pointer returned by `omp_-target_alloc()`. In lines $16 - 19$, each list item is copied from the host to the device using the `omp_target_memcpy()` function. The destination device is the default device, and the source device is the host.

A host thread cannot perform pointer arithmetic on an accelerator's device pointer. Fortunately, the `omp_target_mempcy()` function has offset arguments for the source and destination addresses. The destination accelerator address is calculated by adding the `i*sizeof(item_t)` offset expression to the `dst` device pointer.

Finally, in lines $21 - 26$, the `next` pointers in the copied list items are initialized by running a short target region on the destination device. The `is_device_ptr` clause indicates that `dst` is a device pointer in the target region.

Associate Host Memory with Device Memory Host memory may be associated with device memory using the `omp_target_associate_ptr()` function. The function performs a map-enter phase for host memory without allocating associated storage in device memory. Instead, the device address of the associated storage is passed as an argument to the function. The `omp_target_disassociate_ptr()` runtime function performs a map-exit phase for host memory, but does not free the associated storage in device memory.

```
#include <omp.h>
#include <stdlib.h>
typedef struct item {struct item *next; int v; } item_t;
void *copy_list2dev(item_t *list)
{
  int i, count=0;
  int dev  = omp_get_default_device();
  int host = omp_get_initial_device();
  item_t *src = NULL, *dst = NULL;

  if (list == NULL) return NULL;
  for (src=list; src; src=src->next)
    count++;
  dst = omp_target_alloc(count*sizeof(item_t), dev);

  for (src=list, i=0; src; src=src->next, i++)
    omp_target_memcpy(dst, src, sizeof(item_t),
                      i*sizeof(item_t), 0,
                      dev, host);

  #pragma omp target is_device_ptr(dst)
  {
    for (i=0; i<count-1; i++)
      dst[i].next = &dst[i+1];
    dst[i].next = NULL;
  }
  return (void*)dst;
}
```

Figure 6.62: **Copy a linked list to device memory** – Copy a linked list from the host to dynamically allocated device memory.

Figure 6.63 shows an example that uses these functions to stream sections of a large buffer stored in host memory through a smaller buffer allocated in device memory. At line 5, the pointer variable `devptr` is assigned the device address of a `chunk`-sized buffer of type float allocated in device memory. The pointer variable `a` points to a buffer of size `n` float elements. For simplicity, assume that $n\%chunk = 0$ and $n >= chunk$. The loop starting on line 7, iteratively maps a `chunk`-sized section

```
 1 #include <omp.h>
 2 void stream(float *restrict a, int n, int chunk, int dev)
 3 {
 4   int size = sizeof(float)*chunk;
 5   float *devptr = omp_target_alloc(size, dev);
 6
 7   for (int i=0; i<n; i+=chunk)
 8   {
 9      omp_target_associate_ptr(&a[i], devptr, size, 0, dev);
10
11      #pragma omp target map(always,tofrom:a[i:chunk]) device(dev)
12      for (int j=i; j<i+chunk; j++)
13         a[j] = 1/(1+a[j]);
14
15      omp_target_disassociate_ptr(&a[i], dev);
16   }
17   omp_target_free(devptr, dev);
18 }
```

Figure 6.63: **Map host memory to dynamically allocated device memory** – Iteratively associate a smaller device memory buffer with a section of a larger `a` buffer.

of the `a` buffer to the `devptr` buffer and executes a target region that operates on the section. The call to `omp_target_associate_ptr()` at line 9 performs a map-entry phase for the section, adding it to the device data environment. The associated storage for the section is the device memory pointed to by `devptr`.

The section appears in the `map` clause as the array section `a[i:chunk]` in order to specify its offset and size in the associated target region at lines 11 – 13. Because the section has a non-zero reference count, the `always` *map-type-modifier* is required to force the map-entry phase assignment of the section from the host to the device. Likewise, during the map-exit phase, the `always` *map-type-modifier* ensures that the assignment of the section from the device to the host occurs. Note that, instead of the `always` *map-type-modifier*, `target update to(a[i:chunk])` and `target update from(a[i:chunk])` constructs may be used before and after the target region.

The call to `omp_target_disassociate()` at line 15 performs a map-exit phase for the section and removes it from the device data environment but does not

```
 1 #include <stdio.h>
 2 #include <omp.h>
 3 int copy_2d(void *dst, void *src, int dst_dev, int src_dev,
 4             int sz, int vol_sz, int offset)
 5 {
 6    const int num_dims = 2;
 7    const int vol_dims[2] = {vol_sz, vol_sz};
 8    const int dst_dims[2] = {sz, sz};
 9    const int src_dims[2] = {sz, sz};
10    const int dst_offset[2] = {offset, offset};
11    const int src_offset[2] = {0, 0};
12
13    return omp_target_memcpy_rect(dst, src, sizeof(char),
14          num_dims,
15          vol_dims,
16          dst_offset, src_offset,
17          dst_dims, src_dims,
18          dst_dev, src_dev);
19 }
```

Figure 6.64: **Copy a sub-matrix from a source matrix to a destination matrix - part 1** – Two-dimensional square matrices are assumed. Copy a sub-matrix from `src[0][0]` to `dst[offset][offset]`.

free the associated device memory. Finally, after the loop completes, the call to `omp_target_free()` at line 17 frees the `devptr` buffer.

Copy a Sub-matrix Between Two Matrices An example program is presented below that demonstrates the `omp_target_memcpy_rect` function. The program copies a $4x4$ sub-matrix between two $8x8$ matrices. The source matrix is allocated in the device memory of the default accelerator device, and the destination matrix is in host memory. The first part of the program is shown in Figure 6.64.

The `copy_2d()` function assumes that the matrices and the sub-matrix have two dimensions and are square. The arrays initialized at lines $6-9$ define the dimensions of the matrices and the sub-matrix. The offset arrays initialized at lines $10-11$ are indexes into the matrices and, along with the `dst` and `src` pointer variables, are used to determine the starting source and destination addresses. The call to the

```
21 #define N 8
22 void main()
23 {
24   int dst_dev = omp_get_initial_device();
25   int src_dev = omp_get_default_device();
26   unsigned char DST[N][N];
27   unsigned char (*SRC)[N] = omp_target_alloc(N*N, src_dev);
28
29   #pragma omp target is_device_ptr(SRC) nowait
30   for (int i=0; i<N; i++)
31     for (int j=0; j<N; j++) SRC[i][j] = 1;
32
33   for (int i=0; i<N; i++)
34     for (int j=0; j<N; j++) DST[i][j] = 0;
35
36   #pragma taskwait
37   copy_2d(DST, SRC, dst_dev, src_dev, N, 4, 2);
38
39   omp_target_free(SRC, src_dev);
40
41   for (int i=0; i<N; i++) {
42     for (int j=0; j<N; j++) printf("%d" , DST[i][j]);
43     printf("\n");
44   }
45 }
```

Figure 6.65: **Copy a sub-matrix from a source matrix to a destination matrix - part 2** – Allocate and initialize an $8x8$ `SRC` matrix on an accelerator and fill it with 1. Initialize an $8x8$ `DST` matrix on the host and fill it with 0. Copy a $4x4$ sub-matrix from `SRC[0][0]` to `DST[2][2]`.

`omp_target_memcpy_rect()` function at lines $13 - 18$ copies a `vol_sz` by `vol_sz` sub-matrix from the source device address starting at `src[0][0]` to the destination host address starting at `dst[offset][offset]`. The second part of the program is shown in Figure 6.65. The `SRC` matrix is dynamically allocated in the default device's memory by the call at line 27 to the `omp_target_alloc()` function. The elements in the $8x8$ `SRC` matrix are initialized to 1 in the target region, and the elements in the $8x8$ `DST` matrix are initialized to 0 on the host device.

```
00000000
00000000
00111100
00111100
00111100
00111100
00000000
00000000
```

Figure 6.66: **Example output from the sub-matrix copy program** – This is the output from the program in Figure 6.64 and Figure 6.65. A $4x4$ sub-matrix from the `SRC` matrix was copied into the center of the `DST` matrix.

The `is_device_ptr` clause is necessary on the `target` construct at line 29, because `SRC` is a device pointer. Due to the `nowait` clause, the execution of the target region is enclosed in a deferrable target task, and the initialization of the `SRC` and `DST` matrices may execute in parallel. The host thread cannot continue past the `taskwait` construct until the target task has completed. After this point, both matrices have been initialized.

The call to the function `copy_2d()` at line 37 copies a $4x4$ sub-matrix from the device address starting at `SRC[0][0]` to the host address starting at `DST[2][2]`. The call to `omp_target_free` at line 39 frees the memory allocated for the `SRC` matrix. Lines $41-43$ print the `DST` matrix after the copy has occurred. The output of the program is shown in Figure 6.66.

6.13 Concluding Remarks

This chapter has covered how to use the OpenMP device constructs to program heterogeneous systems composed of a host device and one or more accelerator devices. The execution model is host-centric; the program starts on the host device, and during its execution, target regions may be executed by threads running on accelerator devices. Because the host and accelerator memories may be in different address spaces, there are restrictions on how variables are shared across devices.

The device constructs fall into two general categories: 1) constructs that are used to manage sharing variables between devices, and 2) constructs that are used to select on which device code executes. Following are guidelines to consider when using the device constructs.

There is overhead involved with executing a target region on an accelerator. Consider that there should be enough computation in the target region to justify the overhead.

To share a variable among threads executing on different devices, the variable is mapped. Very often the key to getting good performance using the device constructs is correlated with the strategy used for mapping variables.

The overhead from mapping variables is especially a concern for those heterogeneous systems where the host and accelerator do not share memory. Minimize the potential for copying data between the host and accelerator devices. Use a *map-type* in `map` clauses to eliminate unnecessary copies. Use the data-mapping constructs to map variables in a way that encloses target regions where the variables are used.

Except for the `target data` construct, the device constructs generate tasks. Use the asynchronous features of tasks so that host and accelerator code regions may execute in parallel. Use tasks to hide the overhead of mapping variables by overlapping data mapping constructs with computation.

Consider the dual-nature of mapped variables. Treat mapped variables like shared variables when determining how two threads on different devices can safely access a variable when it is mapped.

Do not assume that mapping a variable will always update its value with an assignment. If it was already mapped by a target data-mapping construct, then the map simply increments its reference count If an assignment is always required, then use the `always` *map-type* or the `target update` construct.

Consider carefully whether a scalar variable is `firstprivate` or mapped in a target region. The best way to be sure is to always list variables in explicit clauses. Be careful when mapping pointer variables, and use array sections to ensure that the pointed-to memory is mapped. The pointer variable in an array section is not mapped, and it is private in a target region. Remember that if a pointer variable does not appear in a clause on a `target` construct, then it is implicitly treated as if it had appeared in one as the base of a zero-length array section.

When using the accelerated worksharing constructs, first start with the `target teams distribute` construct, and then as needed, gradually add the more prescriptive worksharing constructs and clauses that fix the number of teams, threads, and SIMD lanes. See how far you can get with letting the compiler make these determinations.