5 The Computation Pipeline

5.1 Introduction

Throughout the book up to this point, we have seen how the new features of Direct3D 11 can be used to perform extremely flexible rendering operations. However, there is yet another pipeline stage available for performing flexible computations that can be applied to a wider range of applications. This allows the GPU to be used in such diverse applications as ray tracing and physical simulations, and in some cases, it can even be used for artificial intelligence calculations. Of course, this pipeline stage is the *compute shader* stage, and it represents the implementation of a technology often referred to as *DirectCompute*. DirectCompute provides the first steps to adding a general-purpose processing paradigm to Direct3D, and supplies a more natural and flexible processing environment to the developer to harness the power of the GPU for non-rendering algorithms.

This chapter will explore this new computation pipeline, starting with the architectural details of how the compute shader pipeline stage operates. This includes details about its new threading model, memory model, synchronization system, and addressing scheme. With a clear understanding of how the compute shader functions, we will then turn our attention to how we can use it for various different computational tasks. We attempt to provide general guidelines to follow when designing an algorithm for DirectCompute. Since several thousand threads can be active on the GPU simultaneously, it is important to have a solid understanding of how to harness all of these threads to fully use the power of the GPU.

5.1.1 DirectCompute

DirectCompute introduces a new processing paradigm that attempts to make the massively parallel computational power of the GPU available for tasks outside of the normal raster-based rendering domain. Since the GPU is comprised of a large number of small processors working in parallel, it is especially well suited for computational tasks that allow for strong parallelization. In fact, this is not the first time that GPUs have been used for purposes other than rendering. An entire field of application development, named *general purpose GPU computations (GPGPU),* has sprung up, which uses existing rendering APIs to implement support for a desired algorithm.

This process has been effective, but it is not without its drawbacks. Although application developers started to use the 3D rendering API to perform calculations it wasn't originally designed for, the process of implementing an algorithm is significantly more difficult than it is when using a traditional programming language such as C++. In addition, there are certain operations which were not even possible to perform on the GPU through the rendering API, such as scattered writing to resources. The GPU manufacturers were well aware of this situation, and several new APIs have been developed to allow more flexible use of the GPU's power. APIs such as *CUDA* and *OpenCL* provide a more general processing environment than trying to use a 3D API. These solutions are an improvement over the do-it-yourself frameworks.

DirectCompute takes this improved flexibility a step further. DirectCompute is squarely embedded in the Direct3D 11 processing environment, and it shares much of the existing framework from the rendering portion of the API. Within this context, an application developer familiar with Direct3D 10 and/or Direct3D 11 already knows how to perform many of the normal interactions with the API code to get an application up and running. Furthermore, the resource models, execution paradigms, and general debugging process all leverage existing knowledge. These are all reasons that DirectCompute is an attractive technology to use. However, perhaps the single most important benefit that DirectCompute provides is that it is so close to the rest of the rendering pipeline and can easily be used directly to supply input to rendering operations. This makes the API instantly useable by a wide variety of Direct3D programmers, who can use it in their applications almost immediately. DirectCompute uses the exact same resources that are used in rendering, making interoperation between the two very simple. With general-purpose processing available with rendering, we have a very potent combination for use in a wide variety of applications.

Another great benefit of using DirectCompute is that the performance of a particular algorithm can easily scale with a user's hardware. If a user has a high-end gaming PC with two or more high-end GPUs, then an algorithm can easily provide additional complexity to a game without the need to rewrite any code. The threading model of the compute shader inherently supports parallel processing of resources, so adding additional work when more computational power is available is trivial. This is even more true for GPGPU applications, in which the user typically processes as much data as possible, as fast as possible. If an algorithm is implemented in the compute shader, it can easily scale to the current system's capabilities.

With such a wide variety of benefits, and tight integration with Direct3D 11, DirectCompute represents the first steps toward having a fully general-purpose coprocessor. Throughout the rest of this chapter, we will take a detailed look at how this technology works and how it can be used.

5.1.2 The Compute Shader Stage

The compute shader follows the same general usage concept as the other programmable shader stages. A shader program is compiled and then used to create a shader object through the device interface, which can then be loaded into the compute shader stage through the device context interface. The stage can take advantage of the same set of resources, constant buffers, and samplers that we have seen in Chapter 3 for the other rendering pipeline stages, with the additional capability to bind resources to the stage with unordered access views. However, the compute shader is fundamentally different than the other programmable pipeline stages, since it doesn't explicitly have an input or output that is passed from a previous stage or passed to the next stage. It can receive some system value semantics (which will be discussed shortly) as input arguments, but this is the only attribute -type data that can be use in a shader program. All of the remaining data input and output is performed through resources instead.

This arrangement indicates that the compute shader implements complete algorithms within a single program, as opposed to the rendering pipeline, which has the option to implement an algorithm over many different pipeline stages. Of course, the compute shader can be used to iteratively implement an algorithm in steps, but the choice of how best to design the algorithm is left to the developer. This is an interesting design, and it allows for algorithms to be composed in a different way than was possible prior to the introduction of the compute shader. We will further explore the details about how to use the compute shader from the API perspective later in the chapter.

5.2 DirectCompute Threading Model

We begin our journey through DirectCompute by examining its threading and execution model. We have already noted that the GPU is very good at processing parallel algorithms due to its large number of processing cores. With so many processors available to perform work, it is necessary to have some methodology that allows us to efficiently map a particular algorithm to run on many threads. The typical multithreading paradigm used in traditional CPU-based algorithms uses separate threads of execution, coupled with a shared memory space and manual synchronization. It is a fairly well known model and has been in use for many years on systems with multiple processors.

However, this model is not quite optimal when it is mapped onto a processing paradigm that requires thousands of threads operating simultaneously. DirectCompute uses a different type of threading and execution model, which attempts to provide a balance between generality and ease of use. As we will see later in this section, this model allows for easier mapping of threads to data elements. This provides a simple way to break a processing task down into smaller pieces and have it run on the GPU.

5.2.1 Kernel Processing

Also like the other programmable shader stages, the compute shader implements a kernel based processing system. The compute shader program itself is a function, which when executed can be considered to be a form of a processing kernel. This means that the shader program provides a kernel that will be used to process one unit of work. That unit of work will vary from algorithm to algorithm, but the currently loaded kernel is instantiated and applied to a single input set of data. In the case of the compute shader, the data set is provided through access to the appropriate resources bound to the compute shader stage.

This provides a very simple and intuitive way to program work for the thousands of threads that the GPU is capable of operating on. Each thread will be tasked with executing one individual invocation of the kernel on a particular data element. This simple concept reduces the complexity of designing algorithms for many threads by changing the algorithm design task from "What do I make each thread do?", to "Which piece of data does each thread process?" When the kernel is the same for all threads, the task becomes finding a data model that allows the desired data set to be broken into individual data elements that can be processed in isolation, instead of manually trying to synchronize the actions of all the threads. Instead of trying to orchestrate all the different responsibilities for each individual thread, the developer can instead focus on the best way to split a problem into a series of many instances of the same problem for every thread.

5.2.2 Dispatching Work

With an understanding of what the individual threads will be executing, we can continue on to consider how the developer actually executes a batch of work with the desired number of threads. This is performed with the device context ID3D11DeviceContext::Dispatch() method, or the ID3D11DeviceContext::DispatchIndirect() method, which is similar to the indirect rendering methods we have already seen in Chapter 3. The Dispatch method is analogous to the many draw call calls from the rendering pipeline. It takes three unsigned integer parameters as input: *x, y,* and z. These three parameters indicate how many *groups* of threads we would like to "dispatch" to execute the desired processing kernel. These three parameters provide the dimensions of a three-dimensional array of thread groups that will be instantiated, and the size of each parameter can range from 1 to 64k. For example, if an application calls Dispatch (4,6,2), a total of 4*6*2 = 48 thread groups will be created. Each group of threads would be identified by a unique set of indices within the specified dispatch arguments, ranging from 0 to size-1 in each of the three dimensions.

Notice that the dispatch call defines how many *groups of threads* are instantiated, and not how many *threads* are instantiated. The number of threads instantiated is defined by specifying how many threads will be created for each thread group with a numthreads HLSL function attribute preceding the compute shader program in its source code file. As in the dispatch call, this numthreads statement also defines the size of a three dimensional array, except that this array is made up of threads instead of thread groups. The size of each of these parameters is dependent on the shader model being used, but for cs_5_0 the *x* and *y* components must be greater than or equal to 1, the *z* component must be between 1 and 64, while the total number of threads *(x*y*z)* cannot exceed 1024. The statement shown in Listing 5.1 is an example of how this numthreads statement must appear in the HLSL shader program.

```
[numthreads( 10, 10, 2 )]
// Shader kernel definition follows...
```

Listing 5.1. The number of threads per thread group declaration.

With this example, a total of 10*10*2 = 200 threads will be instantiated for each thread group. Each of these threads can also be uniquely identified by its integer indices, ranging from 0 to size-1 in each of the three dimensions. If we use the dispatch call example from above, we would have a total of 48*200 = 6,400 threads instantiated for this dispatch call.

However, with the three dimensional method of specifying and identifying these threads, we can think of their organization in a geometric way. Consider the visualization of a thread group shown in Figure 5.1.

Instead of thinking of the threads in our dispatch call as a linear list of 6,400 threads, we can instead use the thread group size to provide a spatial orientation for each thread. Since each thread has a unique identifier, it is easy to reference one particular thread within

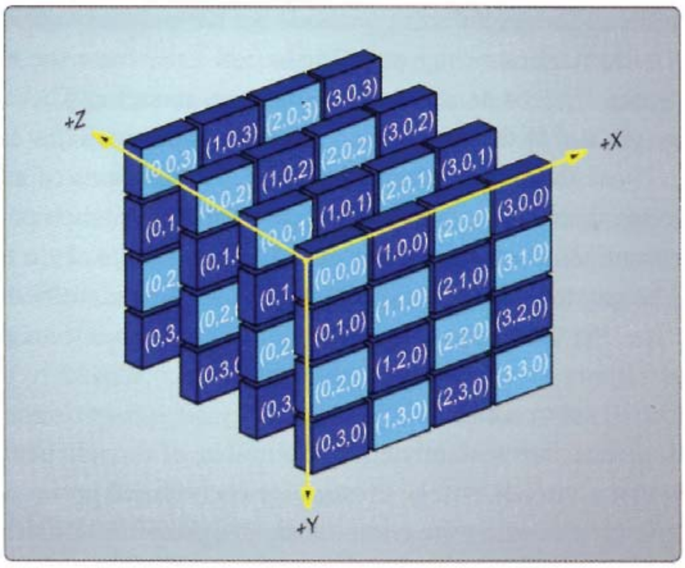

Figure 5.1. A visualization of the threads within a thread group.

the thread group. Similarly, we can visualize all of the thread groups defined by the dispatch call as a three dimensional array, as shown in Figure 5.2.

If we combine the two previous images, we can get a sense for how each individual thread is located within the complete system of threads instantiated by the dispatch call. This is shown in Figure 5.3.

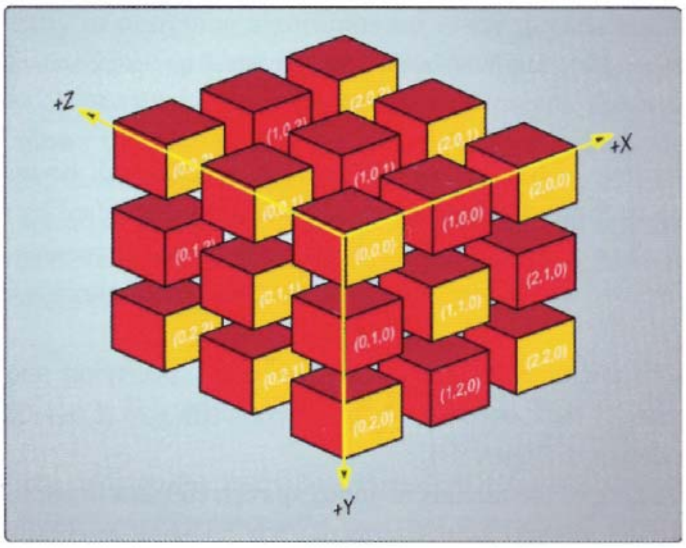

Figure 5.2. A visualization of the thread groups within a dispatch call.

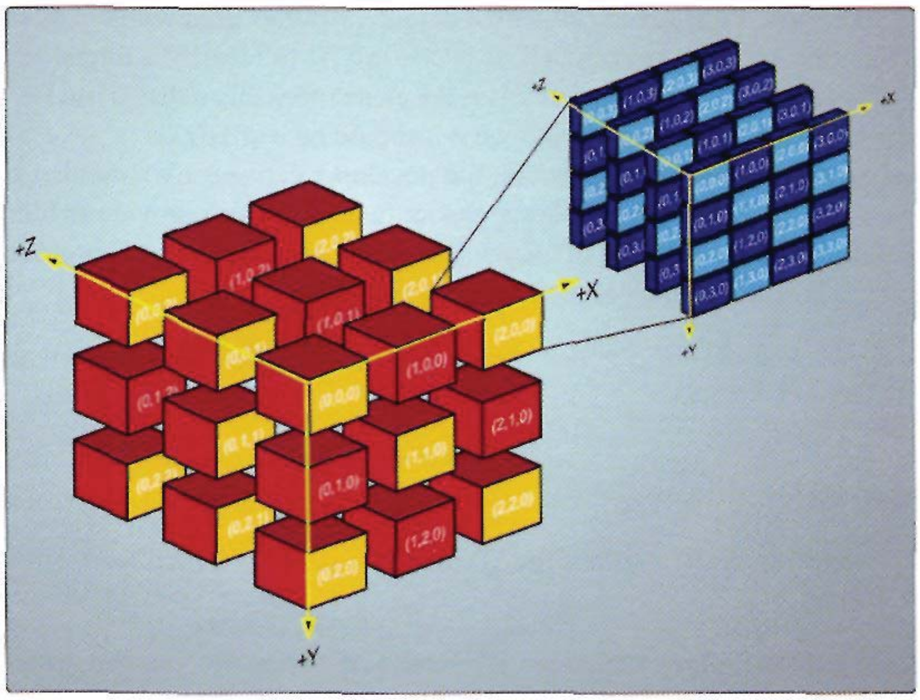

Figure 5.3. A visualization of individual threads within a dispatch call.

From this geometric interpretation of the thread locations, we can also consider an overall unique identifier for each of the threads from within the complete dispatch call. This identifier could essentially locate the thread with an *X*, *Y*, and *Z* coordinate within the three-dimensional grid shown in Figure 5.3. As described before, each of these threads will be used *to execute one instance of our processing* kernel.

5.2.3 Thread Addressing System

At this point, we have an understanding of how the compute shader instantiates threads, and subsequently, of how many instances of our processing kernel will be executed for a given dispatch size. From the example dispatch call in the previous section, we would have 6,400 threads/kernel instances being executed. But how do all of these individual threads know what data to operate on? If they are all instantiations of the same kernel program, then the desired data cannot be manually specified in the shader program. There must be a different mechanism for informing each thread about which portion of a data set it should be processing.

The geometric interpretation of the threading structure of a dispatch call holds the key to understanding how this information is provided. This geometric organization of the threading system was an intentional design decision, which allows for a very clear and concise thread addressing system. As we saw in the previous section, it is trivial to identify each of the threads of a dispatch call with a unique identifier from within a thread group, or from within the entire dispatch call. It is also trivial to identify a thread group from within a dispatch call. It turns out that this is the exact mechanism that is used in the computer shader to inform each thread about what it should be working on.

When authoring the shader program, the developer can specify a number of system value semantics as input parameters for the shader function, which provide an identifier for that particular invocation. The list of available system value semantics is provided below, with a short explanation of each of their meanings.

SV_GroupID: Defines the 3D identifier (uint3) for which thread group within a dispatch a thread belongs to.

SV_GroupThreadID: Defines the 3D identifier (uint3) for a thread within its own thread group.

SV_DispatchThreadID: Defines the 3D identifier (uint3) for a thread within the entire dispatch.

SV_GroupIndex: Defines a flattened 1D index (uint) for which thread group a thread belongs to.

Once each thread invocation has this identifier information, the developer can use that identifier to determine what portion of the input data set should be processed by each thread. To see the utility of such a processing scheme, it is beneficial to consider an example. One simple example would be a compute shader that doubles the value of every element within a Buffer<float> resource. If the buffer contains 2,000 elements, then we could define a thread group size of [20,1,1], which is dispatched with [100,1,1] thread groups. Listing 5.2 shows what the compute shader would look like for this example.

```
Buffer<float>     InputBuf  :  register(  t0  );
RWBuffer<float>   OutputBuf  :  register(  u0  );

// Group size
#define size_x 20
#define size_y 1

// Declare one thread for each texel of the input texture.
[numthreads(size_x, size_y, 1)]

void CSMAIN( uint3 DispatchThreadID : SV_DispatchThreadID )
{
    float Value = InputBuf.Load( DispatchThreadID.x );

    OutputBuf[DispatchThreadID.x] = 2.0f * Value;
}
```

Listing 5.2. A sample compute shader for doubling the contents of a buffer resource.

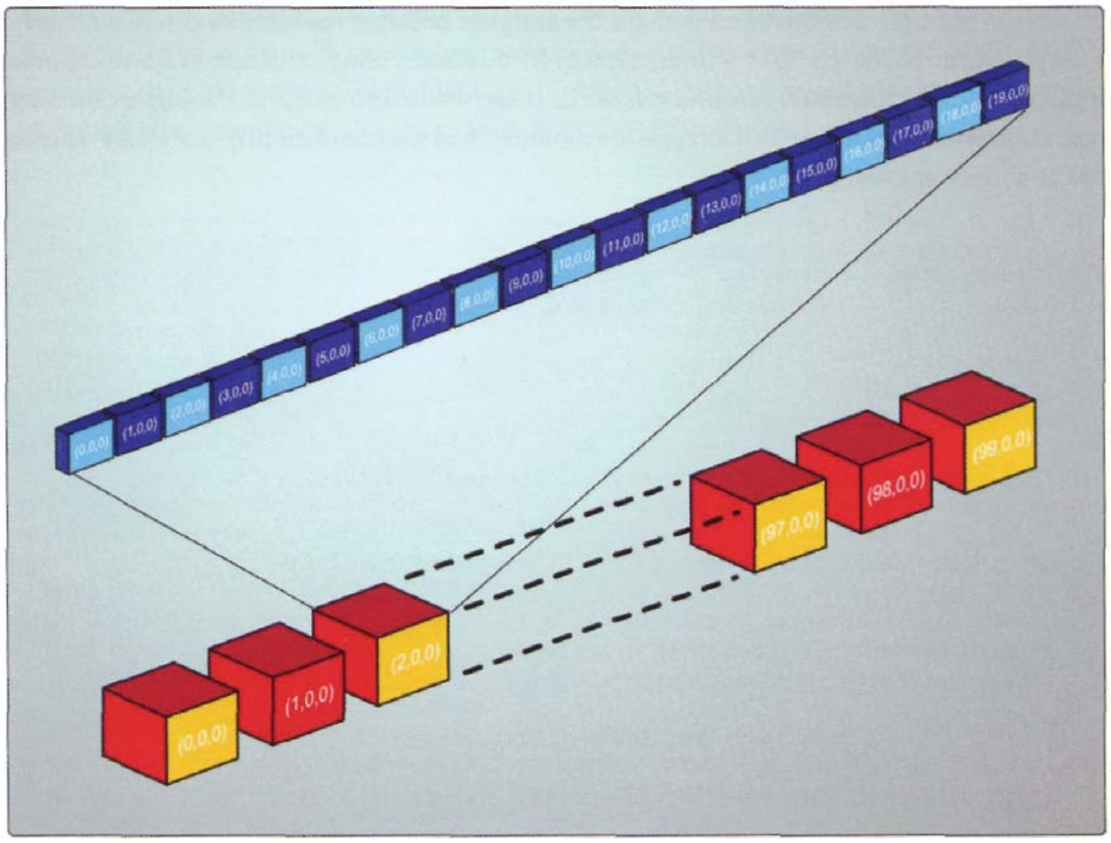

Figure 5.4. The threading structure for the sample compute shader program.

Notice that in the code listing, the size of the thread group is declared with the numthreads function attribute. This is followed by the function definition that represents the processing kernel for our compute shader. In the arguments to this function, we specify which system values we would like to receive to use for our addressing scheme. Since there is some redundancy in the system values (there is more than one way to identify each thread), we have some freedom in choosing which ones would be the most efficient or advantageous to use. Tn this case, we can simply use the SV_DispatchThreadID and use its *X*-component to provide a linear index into our buffer resource. This thread addressing structure can be visualized as shown in Figure 5.4.

With SV_DispatchThreadID, we can directly use its *X*-component as the index when reading from the buffer resource. The buffer can be accessed with the normal square bracket array syntax. We would bind the resource to the compute shader stage with an unordered access view (UAV), which allows each thread to read the float value, double it, and then write it back to the same resource. The dispatch thread ID provides a trivially simple way to have each thread access the elements of a resource, only leaving the developer to determine how large of a dispatch call to make.

This example demonstrates perhaps the simplest possible resource structure with only a single linear buffer of float values. However, this addressing structure is quite capable of handling more complex resources as well. If we wanted to perform the same doubling operation on all elements of a Texture2D resource, then we could modify our HLSL source file as shown in Listing 5.3.

```
Texture2D<float>      InputTex  :  register(  t0  );
RWTexture2D<float>   OutputTex  :  register(  u0  );

//  Group  size
#define  size_x  20
#define  size_y  20

//  Declare  one  thread  for  each  texel  of  the  input  texture.
[numthreads(size_x,  size_y,  1)]

void  CSMAIN(  uint3  DispatchThreadID  :  SV_DispatchThreadID  )
{
    int3 texturelocation = int3(  0,0,  0 );
    texturelocation.x = DispatchThreadID.x;
    texturelocation.y = DispatchThreadID.y;

    float Value = InputTex.Load( texturelocation );

    OutputTex[DispatchThreadID.xy]  = 2.0f * Value;
}
```

Listing S.3. A sample compute shader for doubling the contents of a 2D texture resource.

Here we can see that we have changed the number of threads per thread group, as well as the type of resource that will serve as our input and output storage. There is no change in the system value semantics used for addressing remains, since the dispatch thread ID can also handle two-dimensional identifiers. If our input texture resource was 640x480, then to process it, we would bind it to the compute shader with a UAV and call Dispatch with a size of [32,24,1]. After the dispatch call was completed, the resource's data elements would all have doubled values. This same concept would also be simple to extend to a three-dimensional case, as well.

Up to this point, we have only considered direct resource mappings where the thread addressing structure maps directly to the shape of the resource we are working with. However, one of the benefits of allowing the developer to directly perform the resource accesses in the shader program is that it is also possible to create customized resource schemes. For example, if we wanted to store a four-dimensional data set in a resource, there are no native four dimensional resource types. However, we could easily use a linear buffer resource and then manually implement the data access patterns. If we wanted a [10,10,10,10] resource, we would create a buffer resource with 10*10*10*10 = 10,000 elements. Then we would modify our numthreads statement to use a size of [10,10,10], and use a dispatch call of size [10,1,1]. Listing 5.4 demonstrates how the shader would be modified to calculate the index to lookup in the buffer for each element.

```
Buffer<float>     InputBuf  :  register(  t0  );
RWBuffer<float>   OutputBuf  :  register(  u0  );

// Group size
#define size_x 10
#define size_y 10
#define size_z 10
#define size_w 10

// Declare one thread for each texel of the input texture.
[numthreads(size_x, size_y, size_z)]

void CSMAIN( uint3 DispatchThreadID : SV_DispatchThreadID, uint3 GroupID :
SV_GroupID )
{

    int index = DispatchThreadID.x +
                DipsatchThreadlD.y * size_x +
                DipsatchThreadlD.z * size_x * size_y +
                GroupID.x          * size_x * size_y * size_z +


    float Value = InputBuf .Load( index );

    OutputBuf[index] = 2.9f * Value;
}
```

Listing 5.4. A sample compute shader for doubling the contents of a custom 4D resource.

Here we simply change our resource access code to use each thread's SV_GroupID system value as the fourth-dimension index, and then calculate a linear address for that particular element's location in the buffer resource. The important consideration here is that how a resource is accessed does not need to be simple—it is quite possible to calculate arbitrary memory locations within a resource, which provides significant freedom for the developer to implement the desired access patterns.

5.2.4 Thread Execution Patterns

Throughout this section, we have seen how the threading model of the compute shader functions from the developer's perspective. However, it is important to understand that an implementation may or may not execute the the threading commands of the developer precisely how they are declared. For example, when a thread group is declared, it can have up to 1024 threads. From a programmatic point of view, all of these threads execute simultaneously. However, from a hardware perspective they may not all execute in parallel. Specifically, if a particular GPU doesn't have 1024 processing cores, then it is impossible for a complete thread group to be executed simultaneously.

Instead, the threads are executed in a manner that ensures that they behave as if they were operating at the same time. For example, whenever a point in the shader program requires a synchronization of all of the threads (synchronization is covered more later in this chapter), then each subgroup of threads will be executed to the synchronization point, and then are swapped out so that another subgroup can be executed to the same point. Only after all of the threads in a thread group have completed up to this synchronization point can they continue on. This method of operation can have some performance implications if there is excessive synchronization points in the compute shader, but how much of an impact will depend on the GPU hardware that it is executing on. As the number of processing cores continues to increase, this will be come less and less of a performance issue.

5.3 DirectCompute Memory Model

The overall compute shader execution model provides a great deal of flexibility for instantiating a suitable number of threads to execute the desired processing kernel on the elements of a resource. It is easy to map a complete resource to a given number of threads and perform some computation on each of its data elements. With this execution model in mind, we will now turn our attention to what can be done within the compute shader itself. We will investigate some of the unique features of the compute shader memory model that give developers even more flexibility in deciding how to implement an algorithm.

5.3.1 Register-Based Memory

The compute shader runs on the same programmable processing hardware as the other programmable shader stages. This means that it is based on the same general processing paradigm and also implements the common shader core. It uses a register-based processing concept similar to that for the other pipeline stages, with the exception that the computation pipeline only consists of a single stage. The set of registers that the computer shader supports is quite similar to that of the other programmable stages, and supports input attribute registers (v#), texture registers (t#), constant buffer registers (cb#), unordered registers (u#), and temporary registers (r#, x#). Since all shader programming is performed in HLSL, these registers are not directly used and are not directly visible to the developer. However, the shader programs are eventually compiled into an assembly form before they can be used to create shader objects for use in the pipeline. In addition, it is also possible to obtain an assembly listing of a compiled shader (using the fxc.exe tool), which does allow for inspection of the low-level details of a program.

Because of this, it is worth understanding some of the register functionality. The basic architecture of the common shader core has been covered in Chapter 3, so please review this section for reference during this discussion. In this section, we are the most interested in the temporary registers. These registers can be used to hold intermediate calculations during execution of a shader program. They are only accessible to the thread that is currently executing, and are typically extremely fast registers. Up to 4096 temporary registers (r# and x# combined) are specified in the common shader core, which seems like a fairly large amount for the small scale shader programs. However, while the hardware implementation must be able to handle 4096 registers, it doesn't necessarily need to provide each processing core its own set of registers. Many architectures share a pool of registers among many processing cores, meaning that the overall available number of registers can in reality be less than 4096. While this level of detail is hidden from the developer, the upper limit of 4096 registers is a good indication of how much data can be stored in these registers during the execution of a shader program.

The temporary registers used in this processor architecture can be considered the first class of memory that the compute shader can use. Due to its access speed, this storage memory mechanism is the first choice for data that must be kept available within the shader program. Selection and allocation of these registers are performed automatically by the compiler, so in general, once data has been loaded into the shader core, it will use temporary registers as much as possible.

5.3.2 Device Memory

While the register-based memory is very fast, it is finite in size. In addition, the desired data must be loaded into the shader core before the registers can be used, and after a thread has completed its shader program, the contents of these registers are reset for the next shader program. Of course, we need to have data that persists between executions of a shader program, and we also need much larger possible storage areas. This need is filled by the memory resources that we have already seen many times throughout the book. In the compute shader context, these resources are commonly referred to as *device memory resources* to distinguish them from the other available types of memory.

Direct3D 11 provides a significant array of resource types that can be used for read-only, write-only, or read/write access. The compute shader can use shader resource views and unordered access views to access device memory resources. These two resource views allow read and read/write access, respectively. In addition, the compute shader is also able to utilize constant buffers in the same manner we have seen in the rendering pipeline. Constant buffers provide read-only access to the data stored in them.

Among all the different types of resources that these access mechanisms can attach to, there are literally gigabytes of storage available for shader programs to use. However, to accommodate this large amount of memory, it must be stored in memory located outside of the GPU itself. It is currently not feasible to store such large amounts of data within a processor, so there is generally off-board memory modules used. This memory is normally accessed with a very high bandwidth connection, but there is also a relatively high latency between the time a value is requested and when it is returned. Because of this, device memory resources are considerably slower than register-based memory. While an unordered access view can be used to implement the same operations in device memory as in register-based memory, there would be significant performance penalties when frequent read and write operations are performed.

Another consideration for device memory resources is that access to these resources is provided to all threads that are executing the current shader program. This means that if there are 6400 threads (as we saw in our initial `dispatch` example), each of those threads can read or write to any location within a resource through an unordered access view. Naturally, this requires manual synchronization of access to the resource, either by using atomic operations or by defining an access paradigm that can adequately ensure that threads will not overwrite each other's desired data ranges.

5.3.3 Group Shared Memory

As discussed in Chapter 3, "The Rendering Pipeline," all of the programmable shader stages in the rendering pipeline are kernel based. Each instance of the rendering pipeline kernels is performed in complete isolation from one another. The compute shader breaks free from this paradigm and allows the use of a shared memory area that can be accessed simultaneously by more than one thread. In fact, every thread in a complete thread group is allowed to access the same memory area. This gives the shared memory its name—*group shared memory (GSM).* The group shared memory is limited to 32 KB for each thread group, but it is accessible to all of the threads in the thread group. It is intended to reside on the GPU processor die, which allows for much faster access than the device memory resources.

The group shared memory is declared in the global scope of the compute shader with a special storage scope identifier called groupshared. This memory area can be declared either as an array of basic data types, or as structures of more complex data arrangements. Once a thread group is instantiated, the group shared memory is available to all of its threads simultaneously. Since the entire group shared memory is available to all threads in the group, the compute shader program must determine how the threads will interact with and use the memory, and hence it also must synchronize access to that memory. This will depend on the algorithm being implemented, but it typically involves using the thread addressing mechanisms described in the previous sections to ensure that access to the GSM is performed safely, without the need for atomic functions.

While group shared memory provides a fast and efficient means for threads within a thread group to share information, it does come with some limitations. Since it is limited to 32 KB for each thread group, in any situation where a larger shared pool of memory is needed it is not sufficient. In addition, sharing information is confined to within a single thread group. If an algorithm calls for a large number of threads to all have access to the shared memory pool, group shared memory is not an option.

The group shared memory joins the register-based memory of the programmable shader cores and the larger resource-based memory that can be bound to the pipeline. These three types of memory provide a variety of access speeds and available sizes, which can be used in different situations that match their abilities. Register-based memory is the fastest to access, but it has the smallest amount of memory available. Device memory resources provide gigantic available memory sizes, but also exhibit the slowest access times. Group shared memory strikes a balance between the other two. It is faster to access than resource memory and provides a larger available memory size than register-based memory.

This is a major increase in flexibility for the compute shader. With a memory area that is accessible to multiple threads, it is possible for threads to share information with one another. It also increases the potential for improved efficiency of a thread group as a whole. For example, texture accesses needed by more than one thread could be loaded by one thread and then shared with all other threads in the thread group. This would effectively lower the overall number of texture accesses for the thread group, and would thus improve the overall efficiency of the algorithm. In this way GSM can be used as a customized memory cache that can be directly controlled by the shader program. Sharing between threads is not limited to simple texture accesses, either. It is also possible to share intermediate calculations between threads, which would otherwise need to be performed individually. We will see examples of both of these optimizations in the second half of this book.

5.4 Thread Synchronization

With a large number of threads operating simultaneously, and with the ability for threads to interact with one another through either the group shared memory or through unordered access views of resources, there is clearly a need to be able to synchronize memory access between threads. As with traditional multithreaded programming, where many threads can read and write the same memory locations, there is a potential for memory corruption due to read-after-write hazards (additional details about multithreading programming on the

CPU can be found in Chapter 7, "Multithreaded Rendering"). How can such a massive number of threads be efficiently synchronized without losing the performance that the GPU's parallelism provides? Fortunately, several different mechanisms are available for synchronizing the threads of a thread group. We will explore each of these possibilities in the following sections.

5.4.1 Memory Barriers

We will first look at the highest-level synchronization techniques, referred to as *memory barriers.* HLSL provides a number of intrinsic functions that can be used to synchronize memory accesses across all threads in a thread group. It is important to note that this is an access mechanism that synchronizes only the threads within a thread group, and not across an entire dispatch. These functions have two properties that differentiate them from one another. The first is the class of memory that the threads are synchronizing across when the function is called. It is possible to synchronize access to the group shared memory, device memory, or both. The second property specifies whether all of the threads in a given thread group are synchronized to the same point within their execution. These two properties provide a range of different synchronization behaviors for the developer to choose from. The different versions of these intrinsic functions are listed in Table 5.1 below.

Without Group Synchronization	With Group Synchronization
GroupMemoryBarrier()	GroupMemoryBarrierWithGroupSync()
DeviceMemoryBarrier()	DeviceMemoryBarrierWithGroupSync()
AllMemoryBarrier()	AllMemoryBarrierWithGroupSync()

Table 5.1. Intrinsic Functions: without and with group synchronization.

Each of these functions will block a thread from continuing until that function's particular conditions have been met. The first function, GroupMemoryBarrier(), blocks a thread's execution until all writes to the group shared memory from all threads in a thread group have been completed. This is used to ensure that when threads share data with one another in the group shared memory that the desired values have had a chance to be written into the group shared memory before being read by other threads. There is an important distinction here between the shader core executing a write instruction, and that instruction actually being carried out by the GPU's memory system and being written to memory, where it would then be available again to other threads. Depending on the hardware implementation, there can be a variable amount of time between writing a value and when it actually ends up at its destination. By performing a blocking operation until these writes

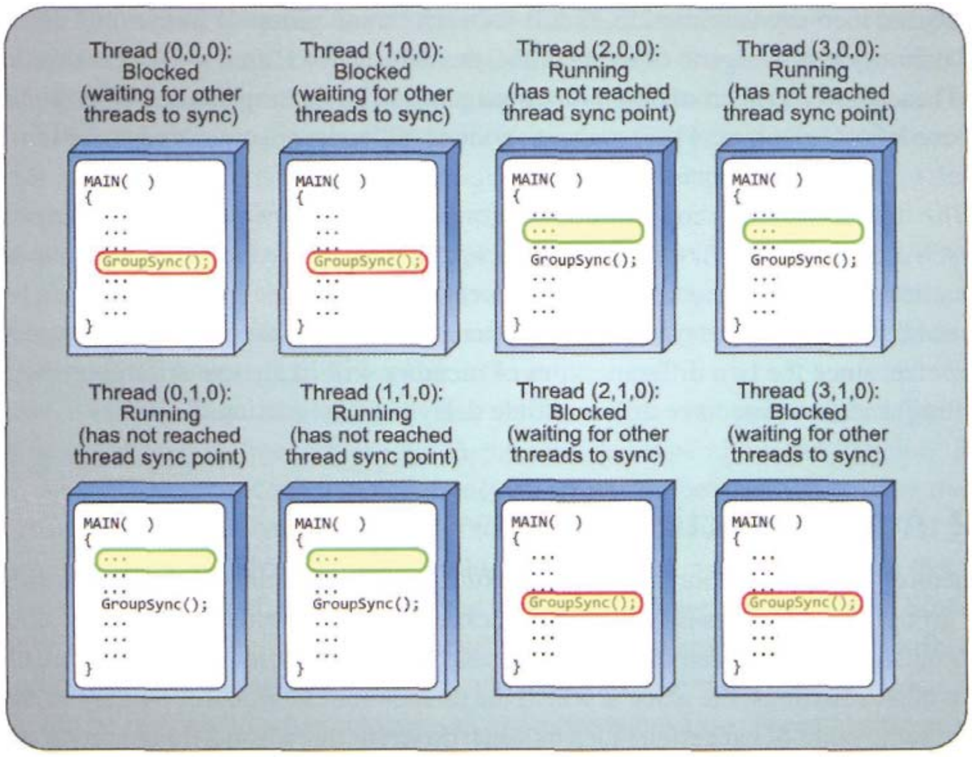
Thread (0,0,0): Blocked (waiting for other threads to sync)
Thread (1,0,0): Blocked (waiting for other threads to sync)
Thread (2,0,0): Running (has not reached thread sync point)
Thread (3,0,0): Running (has not reached thread sync point)
Thread (0,1,0): Running (has not reached thread sync point)
Thread (1,1,0): Running (has not reached thread sync point)
Thread (2,1,0): Blocked (waiting for other threads to sync)
Thread (3,1,0): Blocked (waiting for other threads to sync)
MAIN( )
GroupSync();

Figure 5.5. The synchronization of multiple threads within a thread group with a memory barrier with group synch.

are guaranteed to have completed, the developer can be certain there won't be any issues with read-after-write errors.

The sister function for GroupMemoryBarrier() is GroupMemoryBarrierWithGroup Sync(). This function blocks a thread from continuing until all group shared memory writes are completed, but it also blocks execution until all of the threads in the group have advanced to this function. This allows the developer to be certain that all threads have had a chance to execute to this function call before other threads advance farther. This is a useful mechanism to utilize when each thread in a thread group is loading a set of data into the group shared memory for the other threads to use. It is clear that we wouldn't want any thread to advance until all of the group shared memory has been loaded, making this the perfect synchronization method. Figure 5.5 demonstrates how this thread management is performed.

The second pair of synchronization functions performs similar operations, except that they operate over the *Device* memory pool. This means that all pending memory write operations that write to a resource through an unordered access view can be synchronized before continuing execution of the shader program. This can be very useful for synchronizing larger amounts of memory, which would require the use of device resources since the group shared memory is limited to 32 KB for each thread group. If the desired size of the shared memory is too large to fit in the GSM, the data can be stored in a larger resource instead. These *Device* synchronization functions provide a synching method when a program uses these larger resources. There are two versions of the device memory barrier functions, one with group synchronization and one without it.

The final pair of synchronization functions essentially performs both of the previous types of synchronization. They are useful when there is a mixture of both group shared memory and resource memory that will be accessed by multiple threads and must be synchronized before use. These types of mixed memory scenarios are particularly important to synchronize, since the two different types of memory will likely use different subsystems for writing, and therefore have different time delays for completing the writes.

5.4.2 Atomic Functions

The memory barrier functions are very useful for synchronizing all of the threads in a thread group. However, this is not always necessary or desirable. There are many situations when smaller scale synchronizations are needed, perhaps among only a few threads at a time. In other situations, the place at which the threads should synchronize may or may not be at the same point of execution (such as when different threads in a thread group perform heterogeneous tasks). In these cases, the memory barrier functions are not an appropriate synchronization method.

Instead, Shader model 5 has introduced a number of new atomic functions that can provide more fine-grained synchronization between threads. These functions are guaranteed to be executed in the order that they are programmed, and they can thus be used from one thread, and the result of the function will be propagated to any other threads trying to access the same destination. The following list specifies all of the available functions.

- `InterlockedAddQ`
- `InterlockedMin()`
- `InterlockedMaxQ`
- `InterlockedAndQ`
- `InterlockedOr()`

  `InterlockedXorQ`
- `InterlockedCompareStore()`

  `InterlockedCompareExchange()`
- `InterlockedExchange()`

Like the memory barrier functions, these atomic functions can be used on group shared memory as well as resource memory, which allows for a wide range of potential uses. Each function performs an operation that can be used to turn the contents of either a group shared memory location or a device resource location into a synchronization primitive. For example, if a compute shader program wants to keep a count of the number of threads that encounter a particular data value, then the total count can be initialized to zero, and each thread can perform an InterlockedAdd() function on either a GSM location (for the fastest access speed) or a resource (which persists between dispatch calls). These atomic-style functions ensure that the total count will be incremented properly without any overwriting of the intermediate values by different threads.

Since each of these functions provides a different type of operation, developers have a significant amount of freedom to implement a desired type of synchronization. For example, the InterlockedCompareExchange() function can be used to compare the value of a destination to a reference value, and if the two match, a third argument could be written to the destination. This functionality allows for the implementation of data that can be "checked-out" by a thread and later checked back in for use by another thread. Since these functions are very low level, they can be applied to a particular situation very flexibly. Each function has its unique input requirements and operations, so the Direct3D 11 documentation should be referenced when selecting an appropriate function. These functions are also available to the pixel shader stage, allowing it to also synchronize across resources (since it doesn't have a group shared memory, it can only use the device resources for thread-to-thread communication).

5.4.3 Implicit Synchronization

The final form of synchronization that we will discuss is actually not an explicit part of the compute shader functionality. We will refer to this as *implicit synchronization,* which occurs when an algorithm is designed to access its data set in such a way that there are no potential interactions between threads. This is the preferred method of synchronization—when no synchronization is needed! Each of the other two methods mentioned above is effective and useful, but both come at some cost to performance. If an algorithm can access a memory resource in an explicit and orchestrated manner, then no additional functions are needed, and no extra thread context switching is needed either.

For example, in our simplified example program from earlier in the chapter, we started by reading a value from a resource, then doubled it, and then stored it back to the resource. In this case, no communication was needed from thread to thread, and hence no synchronization was needed. Each thread can operate completely independently from one another. Another example of this type of algorithm is the creation of a particle system, where the particle state is stored in a structured buffer and accessed with an append/

consume structured buffer resource object. In this case, each thread would read one particle's data using the consume intrinsic function. Since the thread doesn't know which particle it is getting, it is completely independent of any other particle's data and can hence execute independently of the other threads. After the particle is updated, it can be added back into an append structured buffer with the append intrinsic function. There is no need to synchronize between threads, and hence the individual GPU processing elements can execute without managing any extra interthread communication. This type of particle system is explored further in Chapter 12, "Simulations."

5.5 Algorithm Design

Throughout this chapter, we have learned about the various capabilities of the compute shader. Some of the concepts we have seen are quite similar to the other programmable shader stages, but some are quite different from what we have seen before. Indeed, the concept of having a raw-computation-based shader stage is a completely new idea that has been added in Direct3D 11. With the introduction of so much new functionality, it can be somewhat difficult to approach a completely new algorithm and decide what tools to use to implement it. This section aims to provide some general design guidelines that can be applied when developing an algorithm. Of course, there is no perfect methodology to design an algorithm, so these guidelines should be taken as suggested starting points that can be built on for a particular scenario.

5.5.1 Parallelism

The first area we will consider is how to maximize the parallelism of an algorithm. The whole reason that the compute shader has been added to Direct3D 11 is to allow developers to harness all of the CPU's available parallel processing power. We have alluded to this throughout the chapter, but it should be an explicit design goal when developing an algorithm to run in the compute shader. The data to be processed should be organized in such a way that it can be processed with a minimal amount of memory access and computation, which will result in a generally faster algorithm. If the problem can be broken down into smaller, coherent parts, then the compute shader should be a good candidate for performing the calculations.

Minimize Synchronization

As discussed in the previous section, there are many ways to synchronize data between threads. Group shared memory, device resources, atomic functions, and memory barriers all provide different varieties of synchronization techniques. However, these synchronization techniques all introduce some overhead during the execution of the compute shader. If it is possible to perform the same calculations without synchronization, the algorithm will run faster. It is often easier to design an algorithm with synchronization than without, but unless the synchronization methods are used to increase efficiency, their use may be detrimental to performance.

Sharing Between Threads

The next point may seem contradictory to our previous comments, but in some cases, explicitly designing synchronization into an algorithm can result in improved performance. When memory bandwidth or the computational load can be reduced by sharing data between threads, it is certainly possible to speed up an algorithm's execution time by synchronizing data across multiple threads. The key is to determine when it is appropriate to do so.

Share loaded memory. One of the primary uses for compute shader programs is in image processing algorithms. Because image-like resources are accessed, it is very natural to map the compute shader onto an image-processing basis. This also happens to be one of the areas that can take advantage of the group shared memory to improve performance. Depending on the algorithm being implemented, image processing is typically bound by the memory accesses it performs. However, if multiple pixels within a thread group can use the same sampled values, then it is quite possible to load a small number of values into the GSM in each thread, followed by using a memory barrier with group synchronization that can be accessed by all of the threads from that point on. This effectively reduces the number of device memory accesses that each thread needs to perform and moves the desired data into the GSM, which is in general faster to access.

However, care must be taken with this approach as well. There is some latency involved with writing to the GSM and then reading from it. If the reduced device memory bandwidth does not offset the costs of accessing the GSM, then this technique could actually hurt performance. This topic becomes even less clear when texture caches are taken into consideration. It is quite possible that the built in texture caches are already performing sufficient data caching, making it difficult to predict which technique will be faster. This can also vary by GPU manufacturer, complicating matters even further. A good suggestion is to write your algorithms in a way that lets you quickly test them in both scenarios, and the higher-performing method can be chosen appropriately. An even better approach is to allow your algorithms to test the current platform and dynamically decide which technique to use.

Share long calculation results. Just as loaded device memory contents can be cached, threads can also share calculated values in the GSM. This is also difficult to predict if it is faster to share calculations, or to just perform them independently in each thread. Modern

GPUs are typically able to perform many arithmetic logic unit (ALU) operations in the time it takes to fetch content from device memory, and there is a somewhat smaller difference when reading from the GSM. Once again, a best practice would be to develop an algorithm so it can either independently calculate the desired values, or share them through the GSM, and then either profile the performance or dynamically decide which version to use.

5.5.2 Choose Appropriate Resource Types

Another very important consideration is the selection of the resource type that will be used. Depending on the data being processed, one resource type may provide a better overall algorithm design than others. The resource type will dictate how the resource is accessed by a thread group, and will also dictate the actual thread group shape and sizes.

Memory Access Patterns

In some cases, the best resource type to choose is fairly obvious. For example, image processing algorithms typically work with a two-dimensional texture, since that is the format that images are stored and used in. However, other algorithms may not be as clearly defined. For example, when implementing a GPGPU algorithm, the data set can usually be manipulated into whichever resource type makes the most sense. One of the most important considerations in this regard is *how* the data must be accessed. Earlier, we mentioned a particle system using append/consume buffers. Since the algorithm doesn't care about the order the particles are processed in, it can use a buffer resource that allows the append/ consume functionality. Alternatively, if there are data sets that are not directly accessed, but are rather spatially sampled, texture resources would be a much better choice. There are also intrinsic functions, such as the `gather` function, that can return multiple values from texture resources, but that can't be used on buffer resources. The resource type should be chosen so that the algorithm can take advantage of all of the available built-in hardware and software functionality.

Thread Group and Dispatch Size

Once a resource type has been selected, an appropriate threading pattern needs to be chosen. In reality, this will probably be decided in conjunction with the resource type to select the best method of accessing the needed data. The dimensions of a thread group will dictate the thread addressing system that can be used to access a resource, both for reading input data and for eventually writing output data. In addition, the total size of the resource must be covered by the dispatch dimensions, which instantiates thread groups. Therefore, these two dimensions can be chosen simultaneously to provide the simplest and most efficient method of accessing a resource.

This can be a tricky decision, and it can take some trial and error to find the best technique for a given situation. In general, these two sets of dimensions should be chosen so that a thread group contains threads that will be accessing coherent memory locations and/or can share intermediate calculations with each other. Then the dispatch size can be used to ensure that the complete resource is processed accordingly.

Mixing Computation with Rendering

The final design consideration we will look at regards the use of a resource for performing some rendering. If the compute shader is used to process a data set, and the end results of the processing will be used in a rendering pass, the output resource must be chosen such that it will allow for the most efficient possible access. This will be a balancing act between choosing a resource type and layout that allow for efficient calculations in the compute shader, while also allowing for efficient use of the output data in rendering operations.