Chapter No. 5
"Multithreading"
In this package, you will find:
A Biography of the author of the book
A preview chapter from the book, Chapter NO.5 "Multithreading"
A synopsis of the book’s content
Information on where to buy this book

About the Author

Pooya Eimandar was born on January 07, 1986. He graduated with a degree in Computer Science and Hardware Engineering from Shomal University and has been programming mainly in DirectX and OpenGL since 2002.

His main research interests are GPU-programming, image processing, parallel computing, and game developing.

Since 2010, he has been leading a game engine team for a company Bazipardaz, working on their latest titles for Xbox 360 and PC. You can find more information about this at http://persianengine.codeplex.com/.

For More Information:
I thank God for every moment of my life.

I would like to thank the staff at Packt Publishing, in particular Yogesh Dalvi and Amigya Khurana, and thanks a million to the technical reviewers for their valuable suggestions.

Also, I would like to thank Amir Sarabadani, Seyed Mohammad Hossein Mayboudi, and Simin Vatandoost for their valuable support while editing the book.

I would also like to thank my colleagues at Bazipardaz, and finally my family for their love and support.

Your feedback is valuable to me, so never hesitate to contact me. You can find me at http://www.Pooya-Eimandar.com
DirectX 11.1 Game Programming

In the last few years, the number of devices in which complex graphics are embedded has vastly increased. Recently, Microsoft released a new version of Windows called Windows 8. The Direct3D 11.1 API is also included with Windows 8 and provides a significant expansion in capabilities over its previous version. Microsoft showed that Direct3D 11.1 plays a key role in writing high-performance 3D Metro applications in Windows 8. To ease portability, Windows 8 introduces a new type of application called the Windows Store application, which is a great opportunity for developers to write cross-platform applications to write cross-platform applications over the Microsoft platforms.

This book will help you easily create your own framework and build your first game for Metro Style all by yourself in order to publish it on the Windows Store.

What This Book Covers

Chapter 1, Say Hello to DirectX 11.1, covers the new features of Windows 8, DirectX 11.1, and the new extension of C++ called C++/CX. This chapter also covers how to set up a framework and initialize the Direct3D device.

Chapter 2, Getting Started with HLSL, provides you with a preliminary knowledge of the new features of HLSL in DirectX 11.1 and explains how to interact with buffers in Direct3D. It also introduces the new additions of Direct2D for Windows 8.

Chapter 3, Rendering a 3D Scene, presents the details of system usages and how to use the Visual Studio Model Editor to render models. This chapter also covers how to handle inputs, cameras, and finally integrate XAML and Direct3D.

Chapter 4, Tessellation, introduces the tessellation stages. It also outlines how to use the graphics debugging feature in Visual Studio 2012.

Chapter 5, Multithreading, introduces the C++ AMP library and the Compute Shader stage and compares the performances of both.

For More Information:
Today, most computers possess multiple cores within their processors. The CPU and GPU core counts will continue to increase, and in a few years, many applications and tools will be developed to utilize these hardware improvements efficiently. In this chapter, we are going to demonstrate how to improve the framework for a parallel game engine using the new technology of Microsoft, which is called C++ Amp. We are also going to integrate our engine to use Compute Shaders and then compare their performances.

In this chapter, we will cover the following topics:

- C++ AMP
- Compute Shaders
- Compute Shader versus C++ AMP
- Post-processing

By the end of this chapter, we are going to have a multithreaded game engine. We will also learn when it is necessary to use C++ AMP and when we must use the Compute Shader.

**C++ AMP**

C++ AMP (C++ Accelerated Massive Parallelism) is a small extension library that enables heterogeneous computing, which provides developers with the GPGPU (General Purpose GPU) programming for making use of the GPU.

C++ AMP fairly looks like an extension. It is C++, not C, and is used to implement data parallelism directly in C++. It also allocates interoperability with Direct3D for accelerating your code. If the code cannot be run on the GPUs, it will fall back onto the CPUs. This means that C++ AMP is a cross-platform library.

For More Information:
Taking over the runtime errors, you need a debugger to visualize the threads and the value of each buffer; Visual Studio 2012 fully supports C++ AMP. You can run and debug your code on Windows 7, Windows 8, Windows Server 2008 R2, and Windows Server 2012. Debugging on the software emulator is only supported on Windows 8 and Windows Server 2012.

In order to have a multithreaded game engine, our framework must be redesigned to use as many processors that are available within a GPU and CPU. Some parts of our framework that will take this approach are post-processing and the real-time collision detection.

The best way to begin learning C++ AMP is to start from the beginning by creating a simple example application. Now we'll take you step by step through the process of your first C++ AMP application.

To run the C++ AMP code, a DirectX 11 feature level 11.0 or a later version is needed. Also, drivers of your graphics card must be installed for debugging.

We are going to compare the performance of C++ AMP with more traditional methods. Our example will demonstrate how to use C++ AMP to accelerate the execution of vector multiplication. Open the first project of this chapter, which is named Array, then take a look at the Test.h header, which is located in the Scenes folder of the solution.

The algorithm is simple; in the first function, we are going to execute a sequence for the process for these vectors. It does not use any parallel or threaded algorithms to reduce the computation time:

```cpp
for (UINT i = 0; i < size; i++)
{
    for (UINT j = 0; j < size; j++)
    {
        C[i] += A[i] * B[j];
    }
}
```

This way will take a long time to execute for large size vectors; let's write this function in another way using the std library:

```cpp
std::for_each(A.begin(), A.end(), [&]{float a})
{
    std::for_each(B.begin(), B.end(), [&]{float b})
    {
```

For More Information:
C[i] += a * b;
    }
    i++;
});

The std::for_each loop is very flexible and is faster than other traditional ways; comparing the results will prove this. Now let's implement this algorithm with the concurrency::parallel_for loop. It iterates over a range of indices and executes a user-supplied function per iteration in parallel.

    parallel_for(0, size, [&](int i)
    {
        parallel_for(0, size, [&](int j)
        {
            C[i] += A[i] * B[j];
        });
    });

OpenMP is available on Windows Store applications and on Visual Studio 2012. OpenMP is a set of compiler directives and callable runtime library routines that extend Fortran (and separately, C and C++) to express the shared memory parallelism. At the time of writing this book, Visual Studio 2012 supports OpenMP 2.0. If you would like to use this feature for your C++ compiler, go to the Properties window of the project, navigate to C/C++ | Language, and from the Language tab, choose the Yes option for OpenMP support. The following screenshot shows the way you can enable OpenMP:

OpenMP makes the multithreading process much easier. In OpenMP, #pragma omp parallel for divides the loop iterations between the spawned threads:

    #pragma omp parallel for
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {

For More Information:
Multithreading

\[
C[i] += A[i] \times B[j];
\]

As you can see, all those functions do the same job, but the first one does it sequentially while the others do it in parallel modes. Now let's implement this function by applying the C++ AMP:

\[
\text{array\_view<const float, 1>} \_A(\text{size}, A);
\text{array\_view<const float, 1>} \_B(\text{size}, B);
\text{array\_view<float, 1>} \_C(\text{size}, C);
\text{parallel\_for\_each}(\_A.\text{extent}, \{\text{index}\_l \text{idx}\}_\text{restrict(amp)}
\{
\text{for (int } i = 0; i < \text{size}; i++)
\{
\_C[idx] += \_A[idx] \times \_B[i];
\}
\});
\_C.\text{synchronize();}
\]

Prior to coding with C++ AMP, make sure to use the concurrency namespace, which is included within the amp.h header.

As you can see, the code was not changed much. array\_view<const float, 1> is a dimension float array that is responsible for copying the data from a vector to an accelerator. The parallel\_for\_each method provides the mechanism for iterating through the data elements and runs a function across the compute domain.

We used this function to multiply to each one of the elements in array\_view variables, and we used restrict(amp) keyword in order to use an accelerator-compatible code.

Let's see what really happens in the background. First, the CPU transfers the data from RAM to VRAM (Video Random Access Memory) by initializing the array\_view variable. Then the GPU starts to process the data by calling parallel\_for\_each in a parallel mode.

The result is stored in the \_C variable, so we need to call \_C.synchronize() to transfer the data back to the CPU. Actually, in this way, the running thread is blocked until the data transfer is completed. If you would like to prevent the blocking, you must call \_C.synchronize_async().

In the next section, we will discuss the Compute Shader concept and execute the vector multiplication on it to compare the result times.

---

For More Information:
Compute Shader

This section will be devoted to the new general purpose of the computation stage. We are going to introduce this interesting new processing concept and study the threading architecture of this stage.

Compute Shader is a programmable shader stage that can read and write to the GPU resources. As you can see in the following figure, Compute Shader is fundamentally different from the other pipeline stages since it works beside the other stages. Therefore, it does not explicitly have an input or output parameter for the previous or next stage.

This technology is commonly useful for the GPGPU programming; however, there are many graphical effect techniques that can be implemented on it. Also, it is not limited to graphical applications, since some algorithms such as animation, physic, AI, and compression can be implemented with Compute Shaders.

For More Information:
In the GPGPU programming, the computation results must be transferred back from the GPU memory (VRAM) to the CPU memory (RAM). Like the preceding example, the array that was stored in RAM must wait to be synchronized with the GPU array that was stored in VRAM. Assuming that the amount of data is huge or the time of calculation might be larger than a frame time, the CPU must wait for the GPU since it has been observed that this might cause a bottleneck for your graphical application. In the Compute Shader, we typically use the output results of the computation as the input of the rendering pipeline; in short, the transformation will be done just inside the GPU.

In the next section, we will take you step-by-step through the process of your first Compute Shader application example.

**C++ AMP versus Compute Shader**

In many technologies, it seems that Compute Shader outperforms C++ AMP, but this is not a sufficient reason to throw C++ AMP away and just focus on Compute Shader to accelerate the computation. Let's start comparing the performances of these technologies; it is an efficient way to find out when it is required to use C++ AMP and when we must switch to Compute Shader:

- Never use C++ AMP when your result's data must be an input of another pipeline stage. In this case, switch to Compute Shader; this is an efficient way when you need to use the results of Compute Shader's process just inside the pipeline stages. On the other hand, C++ AMP needs to transfer the data back to the CPU; however, it cannot access the pipeline stages. Assuming that the relative memory bandwidth speed between the CPU and the GPU is 1 GB/s, the difference between the bandwidths of the GPU and VRAM and that of the CPU and RAM is quite large. In this case, the bandwidth speed is 10 GB/s for the CPU and RAM in comparison to 100 GB/s for the GPU and VRAM.

- Never use the Compute Shader if your application has to run on a machine that might not have hardware-detected DirectX 11-compatible GPU. In this case, switch to C++ AMP; AMP always supports the use of the accelerator class to check whether the target machine has the hardware with a DirectX 11 driver or not. You can also decide to use a WARP accelerator, which will run your code on the CPU, taking advantage of multiple cores. The WARP accelerator is only a Windows 8 solution and also supports the DirectCompute API.

For More Information:  
In this section, we are going to demonstrate how to use the Compute Shader to accelerate the execution of vector multiplication and finally compare the result times. Open the same project (Array) and then take a look at the ComputeShader.hlsl file that is located in the Assets/Shaders folder. The Compute Shader code is as follows:

```hlsl
StructuredBuffer<float> A;
StructuredBuffer<float> B;
RWStructuredBuffer<float> C;

[numthreads(32, 1, 1)]
void main( uint3 DTid : SV_DispatchThreadID )
{
    for(int i=0; i < 5024; ++i)
    {
        C[DTid.x] += A[DTid.x] * B[i];
    }
}
```

A basic element of the Compute Shader is a thread. The Compute Shader declares the number of threads to operate on them; these threads are divided up into thread groups. Each thread group is executed on a single multiprocessor of the GPU. The `numthreads(x, y, z)` attribute defines that each thread group has a total of `x*y*z` threads. The following image shows the visualization of threads within one thread group:

For More Information:
The preceding image shows a single thread group. This thread group has 32 threads ($4 \times 4 \times 2 = 32$), which is specified by $\text{numthreads}(4, 4, 2)$.

We can set the number of thread groups by calling

\text{D3D11DeviceContext::Dispatch}. The input parameters of this function are three unsigned integer numbers that provide the three-dimensional array of the group threads. If any of the values of the input parameters is 0, the driver's \text{Dispatch} function does nothing.

The following diagram shows the visualization of thread groups within the \text{Dispatch} call:

According to the preceding image, we have \text{Dispatch}(4, 3, 2); actually, 24 thread groups ($4 \times 3 \times 2 = 24$) will be created and each thread group has 32 threads specified by ($\text{numthreads}(4, 4, 2)$), so the total number of threads is 768 (that is, $24 \times 32 = 768$).

For More Information:

In the Shader Model 5, the maximum number of threads can be 1024 ($x*y*z \leq 1024$). The $z$ component of the `numthreads` attribute must be a value between 1 to 64, and the $x$ and $y$ components must be greater than or equal to one.

The semantics that are attached to a Compute Shader input or output convey the following information:

- **SV_DispatchThreadID**: This semantic refers to the 3D identifier within the entire Compute Shader thread stage. This ID is uniquely an integer and indices to the specific thread (indexing by $x$, $y$, and $z$ components).
- **SV_GroupID**: This semantic refers to the ID of a thread group within a dispatch.
- **SV_GroupThreadID**: This semantic indicates to the ID of the thread within its own thread group.
- **SV_GroupIndex**: This semantic refers to a flattened index of a thread within a thread group.

As you can see, the algorithm of our Compute Shader code is the same as the algorithm that was used in the C++ AMP code. $A$ and $B$ are the input arrays, and the results of the calculation will be stored in the $C$ array. The input array is defined as a structured buffer, which is simply an array of elements; this buffer can be implemented in the same manner as a simple buffer, except for some properties of its description. Navigate to Graphics/Shaders of the project, open the `StructuredBuffer.h` header file, and see the definition of the `Load` method:

```cpp
D3D11_BUFFER_DESC bufferDesc;
bufferDesc.ByteWidth = (size of element) * (number of elements);
bufferDesc.BindFlags = D3D11_BIND_SHADER_RESOURCE;
bufferDesc.StructureByteStride = (size of element);
bufferDesc.MiscFlags = D3D11_RESOURCE_MISC_BUFFER_STRUCTURED;
```

The $C$ array behaves like the structured array, except that it also provides the ability of writing. The only difference in creating this buffer is `BindFlags`, which must be set to `D3D11_BIND_UNORDERED_ACCESS`.

The structured buffer can be bound as an **SRV** (shader resource view); it is needed to create a shader resource view object by calling the `ID3D11Device::CreateShaderResourceView` function.

```cpp
D3D11_SHADER_RESOURCE_VIEW_DESC srvDesc;
srvDesc.Format = DXGI_FORMAT_UNKNOWN;
srvDesc.ViewDimension = D3D11_SRV_DIMENSION_BUFFEREX;
```

For More Information:

Multithreading

```c
srvDesc.BufferEx.FirstElement = 0;
srvDesc.BufferEx.Flags = 0;
srvDesc.BufferEx.NumElements = number of elements;
```

By calling the `ID3D11Device::CreateUnorderedAccessView` function, we can create a UAV in order to bind it to `RWStructuredBuffer`. In DirectX 11.1, unlike the previous version, a large number of UAVs (Unordered Access Views) can be created, and these UAVs can be used in all pipeline stages:

```c
D3D11_UNORDERED_ACCESS_VIEW_DESC uavDesc;
uavDesc.Format = DXGI_FORMAT_UNKNOWN;
uavDesc.ViewDimension = D3D11_UAV_DIMENSION_BUFFER;
uavDesc.Buffer.FirstElement = 0;
uavDesc.Buffer.Flags = 0;
uavDesc.Buffer.NumElements = number of elements;
```

As mentioned earlier, in the GPGPU computing, we might often get the result from VRAM back to RAM. We need to create a system buffer that can be accessed from the CPU with the `D3D11_USAGE_STAGING` usage flag:

```c
D3D11_BUFFER_DESC bufferDesc;
bufferDesc.Usage = D3D11_USAGE_STAGING;
bufferDesc.BindFlags = 0;
bufferDesc.CPUAccessFlags = D3D11_CPU_ACCESS_READ;
```

Within the `Synchronize` method, the `ID3D11DeviceContext::Map` function is used to copy the results of the Compute Shader from the GPU memory to the system memory; also, the `ID3D11DeviceContext::Unmap` function must be called when the system memory is done updating.

In order to optimize the performance of your application, avoid copying the resources from the GPU to the CPU for each frame; this might decrease the frames per second of your graphical application.

The following screenshot shows the results for the array with size of 5024 float numbers for each array on the target machine with the following features:

- CPU: Intel® Core™ i5-2450M CPU @ 2.50 GHz 2.49GHz
- RAM: 6.00 GB
- System type: 64-bit operating system
- Graphics card: GeForce GT 525M
- Dedicated video memory: 1024 MB DDR3

For More Information:
The total time of execution with C++ AMP is less than 99 milliseconds, while the total time of the Compute Shader is around 50 milliseconds, which seems amazing. Try it on your own machine; build and run the project and compare the result time of the Compute Shader with the result time of C++ AMP.

Post-processing
A post-processor takes in an image and runs some techniques on that image, such as blurring, blooming, negative, and so on. Performing the post-processing for rendering a scene is required to increase the quality of the output presented to the user.

In this section, we are going to introduce an approach that will show you where to use both C++ AMP and Compute Shader to implement some post-processing techniques in our framework.

Implementing post-processing using C++ AMP
Open the Post Processing project from the source code. Open the Quad.cpp file from the Graphics/Models folder and find the definition of the Load method of this class; we will try to apply a post-process on the texture of the quad.
We need to create an empty texture on which we can write our processed result pixels. This resulting texture will have the same height and width as the original texture, and also it is required to create an SRV from this texture. Let's call this SRV `processedResourceView`. The following code shows the texture description of our texture:

```cpp
D3D11_TEXTURE2D_DESC desc = {0};
desc.Height = height;
desc.Width = width;
desc.MipLevels = 1;
desc.ArraySize = 1;
desc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
desc.SampleDesc.Count = 1;
desc.SampleDesc.Quality = 0;
desc.Usage = D3D11_USAGE_DEFAULT;
desc.BindFlags = D3D11_BIND_SHADER_RESOURCE |
                 D3D11_BIND_UNORDERED_ACCESS;
desc.CPUAccessFlags = 0;
desc.MiscFlags = 0;
```

Set one to `mipLevels`, because C++ AMP can only access the first MIP map level of the texture. The `D3D11_BIND_SHADERRESOURCE` bind flag is set to allow reading to the texture, and the `D3D11_BIND_UNORDERED_ACCESS` flag is set to allow writing to the texture.

C++ AMP supports `texture<T, N>` in the `concurrency::graphics::direct3d` namespace, which is defined in the `amp_graphics.h` file, to map the texture in C++ AMP to an exact DXGI format.

It is still impossible to both read and write to this type of `texture<T, N>`; therefore, C++ AMP provides the `write_only_texture_view<T, N>` type to allow writing to the texture.

By calling the `make_texture()` method, we can create a texture associated with our `texture2D` resource; the return value is stored in `amp_textureView`:

```cpp
auto writeTexture = make_texture<unorm4, 2>(
  accViewObj, texture.Get());
amp_textureView =
  unique_ptr<writeonly_texture_view<unorm4, 2>>(
    new writeonly_texture_view<unorm4, 2>(writeTexture));
```

For More Information:

The main section of the post-processing in C++ AMP occurs in the AMPPostProcess method. In this method, first we specify which post-processing technique is required, then we create an input texture associated with our texture2D resource:

```cpp
auto input_tex = make_texture<unorm4, 2>(
    accViewObj, texture2D->Texture);
auto processedView = *this->amp_textureView.get();
```

The `accViewObj` is an `extern` object that is a type of `accelerator_view`. We have created an accelerator view from the D3D device interface pointer by the `concurrency::direct3d::create_accelerator_view` function right after creating the Direct3D device in the `Game.cpp` file.

Now send the input texture captured by the value to the `parallel_for_each` function. The following is the body of your C++ AMP code. First add the `fast_math` namespace to save some typing and implement the negative, which simplifies inverting each pixel:

```cpp
parallel_for_each(input_tex.accelerator_view,
    processedView.extent,
    [=, &input_tex](index<2> idx) restrict(amp) {
        using namespace fast_math;

        auto Negative = [=] (unorm4 _in) restrict(amp) -> unorm4 {
            auto rgb = static_cast<unorm3>(1) - _in.rgb;
            return unorm4( rgb.r, rgb.g, rgb.b, 1.0f);
        };
        auto color = input_tex[idx];
        if (hasNegative)
        {
            color = Negative(input_tex[idx]);
        }
}
```

In the following technique, we are going to make each pixel lighter using the array of `weights`:

```cpp
if (hasLighter)
{
    const float weights[3] = { 0.05f, 0.1f, 0.2f };
    for(int i = 0; i < 3; i++)
    {
        color.rgb += color.rgb * unorm3(weights[i]);
    }
}
```
Finally, we have a technique that applies a separate mathematical calculation on the color of odd and even pixels:

```cpp
if (hasOddEven)
{
    const float _CONST0 = 0.5f;
    const float _CONST1 = 10.0f;

    if (idx[0] % 2 != 0) // if is Odd
    {
        color.rgb += unorm3(sin(color.r * _CONST1) * 
                            _CONST0, sin(color.g * _CONST1) * _CONST0, 
                            sin(color.b * _CONST1) * _CONST0);
    }
    else
    {
        color.rgb += unorm3(cos(color.r * _CONST1) * 
                            _CONST0, cos(color.g * _CONST1) * _CONST0, 
                            cos(color.b * _CONST1) * _CONST0);
    }
}
```

After modifying the pixels, we need to save the result to the processed texture view:

```cpp```
processedView.set(idx, color);
```cpp```

Finally, in the Render method, we set the processed SRV to the Pixel Shader:

```cpp```
shader->BindSRV(ShaderType::PixelShader, 0, 1, 
processedResourceView.GetAddressOf());
```cpp```

We need to unbind the SRV from the Pixel Shader and clean the pipeline; this will happen when EndApply is called:

```cpp```
ID3D11ShaderResourceView * NullSRV = nullptr;
GDevice.d3dContext->PSSetShaderResources(0, 1, &NullSRV);
```cpp```

Within this section, we implemented some post-processing techniques using the C++ AMP library. In the next section, we are going to implement post-processing using the Compute Shader.
Implementing post-processing using Compute Shader

In this section, we are going to implement the post-processing techniques described in the previous section with the help of the Compute Shader.

Open the ComputeShader.hlsl file from the Assets/Shaders folder. The code is given as follows:

```hlsl
#define N 256
[numthreads(N, 1, 1)]
void main(uint3 DTid : SV_DispatchThreadID)
{
    float4 color = T[DTid.xy];
    if (PixelState.x == 1)
    {
        //Apply negative
    }
    if (PixelState.y == 1)
    {
        //Apply High Lighter
    }
    if (PixelState.z == 1)
    {
        //Change odd and even pixels
    }
    RWT[DTid.xy] = color;
}
```

The implementation of post-processing techniques using the Compute Shader is similar to the C++ AMP implementation, so we will leave them and focus on the main parts of this shader code. In the first line of the code, we accessed the current pixel of the T texture with an integer index to the specific running thread. The T variable was declared as a Texture2D type.

When the resultant color is modified, we need to store it to a UAV type object. We declared this object as a RWTexture2D<float4> type, which is a readable and writeable resource unlike the T variable, which is declared as a Texture2D type, and can only be a read-only texture.

For More Information:
Multithreading

Now look at the CSPostProcess method from the Qaud.cpp file. First, we set which post-process is required to be applied by setting PixelState:

```cpp
ObjectInforVars.Const.PixelState.x = negative ? 1.0f : 0.0f;
ObjectInforVars.Const.PixelState.y = highLighter ? 1.0f : 0.0f;
ObjectInforVars.Const.PixelState.z = oddEven ? 1.0f : 0.0f;
ObjectInforVars.Update();
shader->SetConstantBuffer(0, 1, ObjectInforVars.Buffer);
```

Actually, to bind `RWTexture2D<float4>`, an SRV and a UAV are needed. As mentioned before, the resource can be accessed by the Compute Shader for the input by creating an SRV to the texture; we bind it to the Compute Shader by calling the `BindSRV` method of the shader class:

```cpp
shader->BindSRV(ShaderType::ComputeShader, 1, 1, rwTexture->SRV);
```

The resource can be accessed by the Compute Shader for the output parameter. By creating a UAV to the texture, we can also bind it by calling `BindUAV`:

```cpp
shader->BindUAV(0, 1, rwTexture->UAV);
```

In each draw frame, it is more efficient to perform the Compute Shader before performing the render.

Now it's time to set the number of group threads and call the `Dispatch` method to execute the commands in the Compute Shader. Make sure to unbind the SRV and UAV when the Compute Shader is done updating, because a resource cannot be both an output and input at the same time:

```cpp
UINT groupThreadsX = static_cast<UINT>(
    ceilf(texture2D->Width / 256.0f));
shader->Dispatch(groupThreadsX, texture2D->Height, 1);
shader->UnbindSRV(ShaderType::ComputeShader, 1, 1);
shader->UnbindUAV(0, 1);
```

Finally, the result must be set to the Pixel Shader:

```cpp
shader->BindSRV(ShaderType::PixelShader, 0, 1, rwTexture->SRV);
```

For More Information:
Summary
In this chapter, we learned what C++ AMP is and how to use it for accelerating the application. We also introduced the Compute Shader, a programmable shader stage that can read and write to the GPU resources. Both technologies provide a high-speed and general-purpose computing that takes advantage of a large number of parallel processors on the GPU.

We also compared the performances of both technologies in the vector multiplications and image-processing techniques and found out when we must use either of these technologies in our applications.
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.