SeparableFilter11

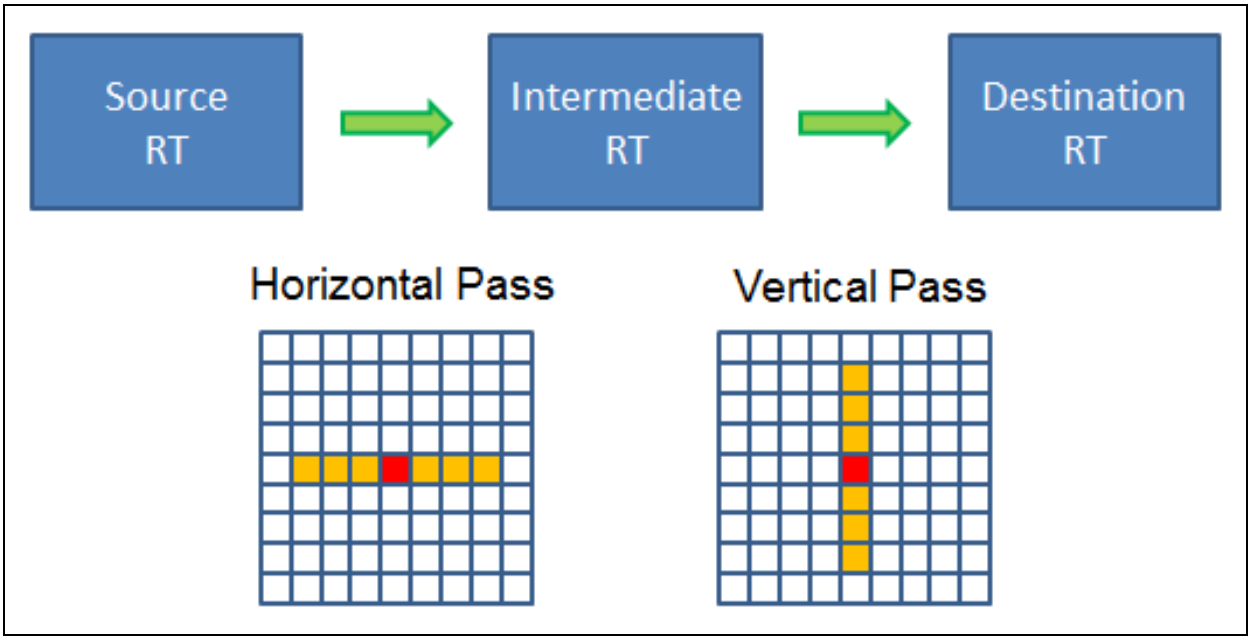
AMD Developer Relations

Overview

This sample, presents a technique for achieving highly optimized user defined separable filters. It utilizes Direct3D 11 APIs and hardware to make use of DirectCompute11 to greatly accelerate this common post processing technique. Included in the sample are implementations for the classic Gaussian filter, and also a fairly simple bilateral filter, but the shader and source code have been setup to allow the user to add their own different filters, with the minimum of fuss. The application implements both a Compute Shader and Pixel Shader path, so that the performance gains achieved through using DirectCompute11can be measured.

Pixel Shader Path

The pixel shader path implements the classical solution to this problem, with the pipeline stages looking like this:



Let's take a look at the HLSL function for the horizontal Pixel Shader pass:

```
//--------------------------------------------------------------------------------------
// Pixel shader implementing the horizontal pass of a separable filter
//--------------------------------------------------------------------------------------
PS_Output PSFilterX( PS_RenderQuadInput I ) : SV_TARGET
{
    PS_Output O = (PS_Output)0;
    RAWDataItem RDI[1];
    int iPixel, iIteration;
    KernelData KD[1];

    // Load the center sample(s)
    int2 i2KernelCenter = int2( g_f4OutputSize.xy * I.f2TexCoord );
    RDI[0] = Sample( int2( i2KernelCenter.x, i2KernelCenter.y ), float2( 0.0f, 0.0f ) )

    // Macro defines what happens at the kernel center
    KERNEL_CENTER( KD, iPixel, 1, O, RDI )

    i2KernelCenter.x -= KERNEL_RADIUS;

    // First half of the kernel
    [unroll]
    for( iIteration = 0; iIteration < KERNEL_RADIUS; iIteration += STEP_SIZE )
    {
        // Load the sample(s) for this iteration
        RDI[0] = Sample( int2( i2KernelCenter.x + iIteration, i2KernelCenter.y ), float2( 0.5f, 0.0f ) );

        // Macro defines what happens for each kernel iteration
        KERNEL_ITERATION( iIteration, KD, iPixel, 1, O, RDI )
    }

    // Second half of the kernel
    [unroll]
    for( iIteration = KERNEL_RADIUS + 1; iIteration < KERNEL_DIAMETER; iIteration += STEP_SIZE )
    {
        // Load the sample(s) for this iteration
        RDI[0] = Sample( int2( i2KernelCenter.x + iIteration, i2KernelCenter.y ), float2( 0.5f, 0.0f ) );

        // Macro defines what happens for each kernel iteration
        KERNEL_ITERATION( iIteration, KD, iPixel, 1, O, RDI )
    }

    // Macros define final weighting
    KERNEL_FINAL_WEIGHT( KD, iPixel, 1, O )

    return O;
}
```

If you have ever written a Gaussian blur filter, or indeed any other separable filter, then this function will look familiar to you. The key thing to note is that all of the text highlighted in red, are in fact macros supplied by the specific filter being implemented. Therefore it is possible to create a wildly different filter without having to touch the nuts and bolts of how the two pass effect actually works.

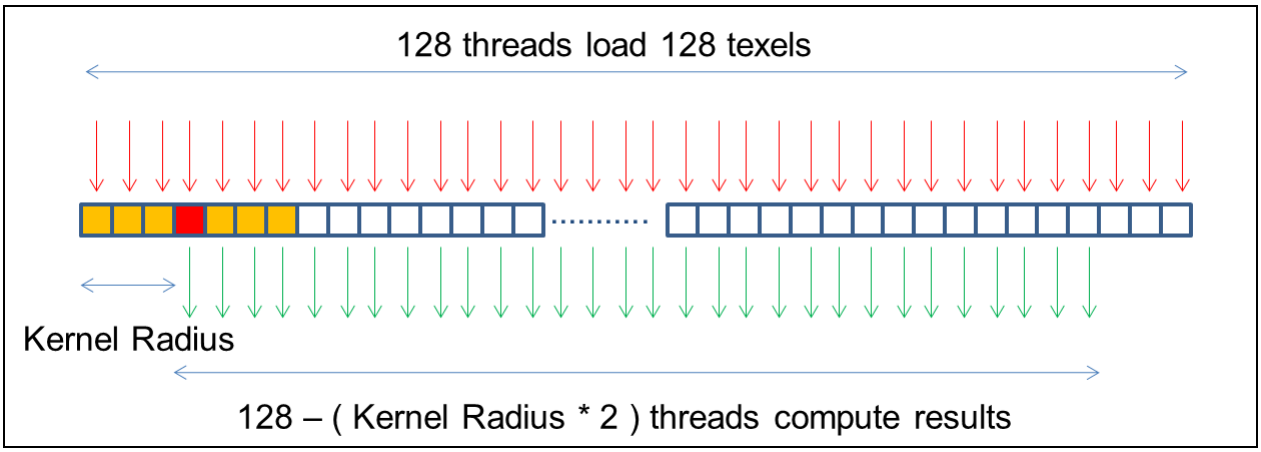
As already mentioned this sample comes with 2 built in filters, a classical Gaussian filter, defined in GaussianFilter.hlsl, and also an example of a simple bilateral filter defined in BilateralFilter.hlsl. These two HLSL files implement their own versions of the macros used above, so taking a look at their contents should mean that creating your own filter is self-explanatory. The really good news is that once you have defined your own macros, and have the Pixel Shader versions working, then the Compute Shader versions will work too, as they use the very same macros.

Compute Shader Path

As mentioned above the logic of how the filter works, is defined by the macros specific to each filter, and are shared between the Pixel and Compute Shader paths. However that is where the similarity ends, as the nuts and bolts of the Compute shader implementation are rather different.

One of the chief advantages offered by the Compute Shader is the ability to group HW threads together and grant access to Thread Group Shared Memory (TGSM). So in essence we can define a bunch of threads, to work on a specific region of the input resources and compute results to the same region of the output resource, but utilize TGSM to cache texture reads and computations.

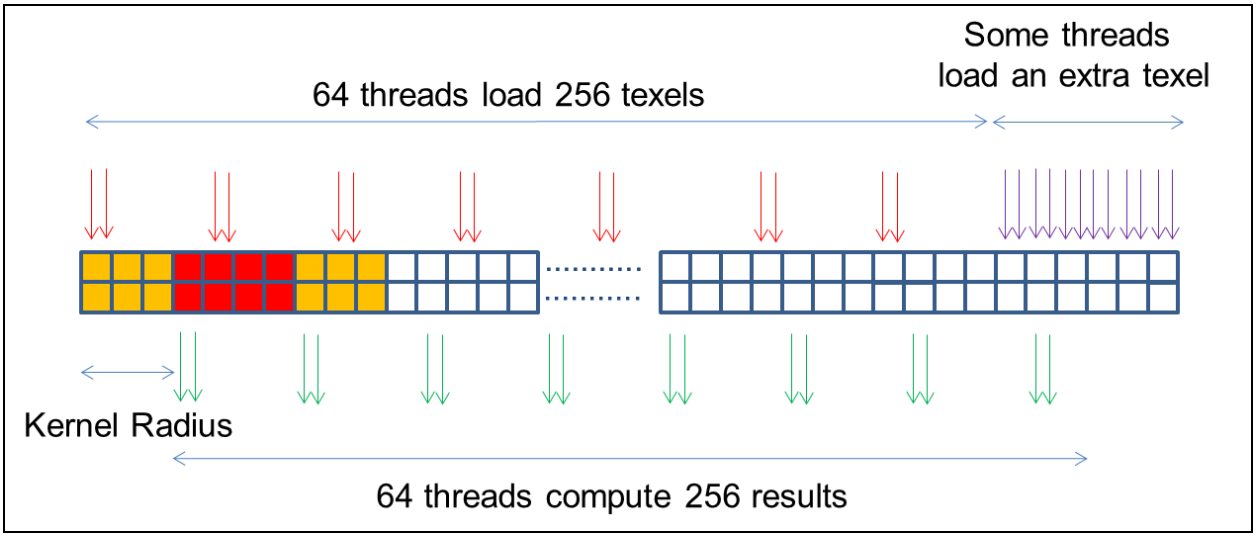
In the development of this Compute Shader kernel we went through several different implementations before reaching what we consider to be optimal. The first kernel looked like this:



Here you can see that we defined a thread group of size 128, and loaded 128 horizontal (or vertical for the vertical pass) texels from the input resource, storing them to TGSM. Then we have to ensure that all threads are synced at a barrier, before continuing to perform the maths of the filter by performing reads from TGSM. In this way we are able to drastically reduce the number of texture operations required to perform the filter. This performance win increases as the size of the kernel increases.

As already mentioned the above kernel was the very first attempt at optimizing this problem using DirectCompute11, and not too surprisingly it was actually quite far from optimal. One of the obvious inefficiencies is that some threads in the group are redundant after the barrier, which is a waste of HW threads on the GPU.

Let's take a look at the final and optimal kernel:



As you can see the threading strategy is rather different, and I'll cover here the reasoning behind setting things up like this:

- Instead of loading just one horizontal (or vertical) line of texels from the input resource, we actually load two horizontal lines, one above the other. This is of benefit because the loaded texels are more texture cache friendly.
- To ensure that all threads in the group have useful work after the barrier, we need to make some of the threads load the extra texels required by the filter kernel size.
- You can see that we have a ratio of 1 thread to 4 computed results, and this leads to a large increase in performance for a couple of reasons:
    - Since 1 thread computes 4 results, it can cache reads from TGSM on General Purpose Registers (GPRs), therefore we are able to roughly quarter the number of reads from TGSM.
    - It just so happens that doing 4 things at the same time fits very neatly on to AMD's Very Large Instruction Word (VLIW) HW. But conversely it doesn't hurt other scalar architectures.
- Lastly something that is not visible in the diagram above is that we have implemented compression of input texels, such that they can be stored as 8, 16, and 32 bits per channel. This again saves on the TGSM required, and the number of reads needed to execute the filter.

Approximate Filters

In addition to supporting different kernel radii, and TGSM (or LDS = Local Data Store) precision, the sample also supports something we call approximate filtering. What this technique does is effectively halve the cost of the filter by halving the number of texture samples required. It does this by offsetting the sample locations to be between two adjacent texels and using the bilinear filter HW to perform the sample.

In effect you can think of this as a pre-filtering stage, and it means that you can also halve the maths computations required to produce the final result. In general the results obtained are very good, and for many purposes indistinguishable from the full filter implementation.

This functionality is built into both the Pixel and Compute Shader implementations, and can be triggered through a macro supplied at compile time.

Shader Compilation

The sample compiles over 500 shader variants to support all of the combinations made possible through the GUI. For this reason we have used a shader cache, otherwise the sample would take a very long time to compile. By default the shader cache will simply use pre-cached shaders. You can change the behavior of the shader cache by passing one of the following flags to the method:

```
AMD::ShaderCache::GenerateShaders( CREATE_TYPE CreateType )

CREATE_TYPE_FORCE_COMPILE   // Clean the cache, and compile all
CREATE_TYPE_COMPILE_CHANGES // Only compile shaders that have changed
CREATE_TYPE_USE_CACHED      // Use cached shaders
```

Obviously you may have your own solution for shader management, and may only need a handful of shaders for the particular application you're working on. Here is an example of the compile switches and macros passed to fxc.exe to compile a specific Pixel and Compute Shader:

Pixel Shader performing the horizontal pass, of a full precision Gaussian filter, of radius 14:

/T ps_5_0
/E PSFilterX Source\GaussianFilter.hlsl
/Fo Object\PSFilterX.obj
/D KERNEL_RADIUS=14
/D USE_APPROXIMATE_FILTER=0

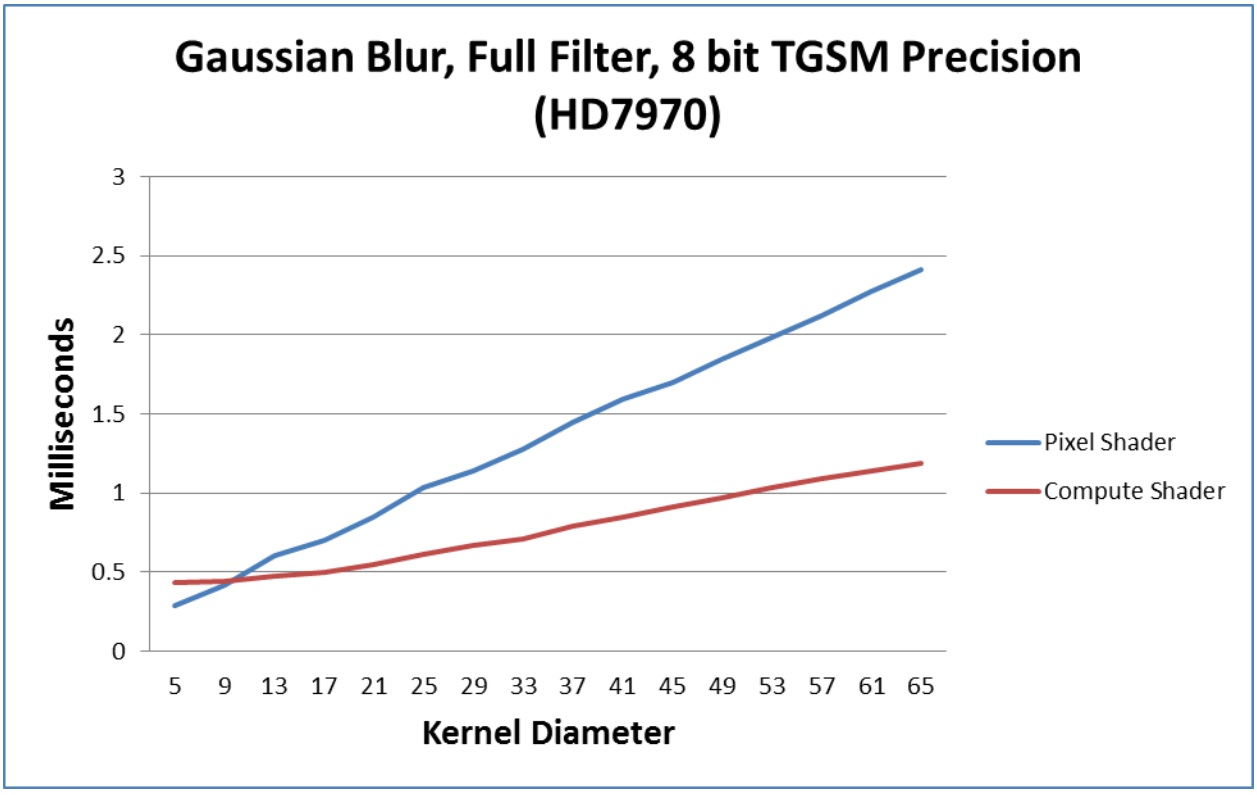
Compute Shader performing the vertical pass, of an approximate precision bilateral filter, of radius 6, using LDS precision of 16 bits per channel:

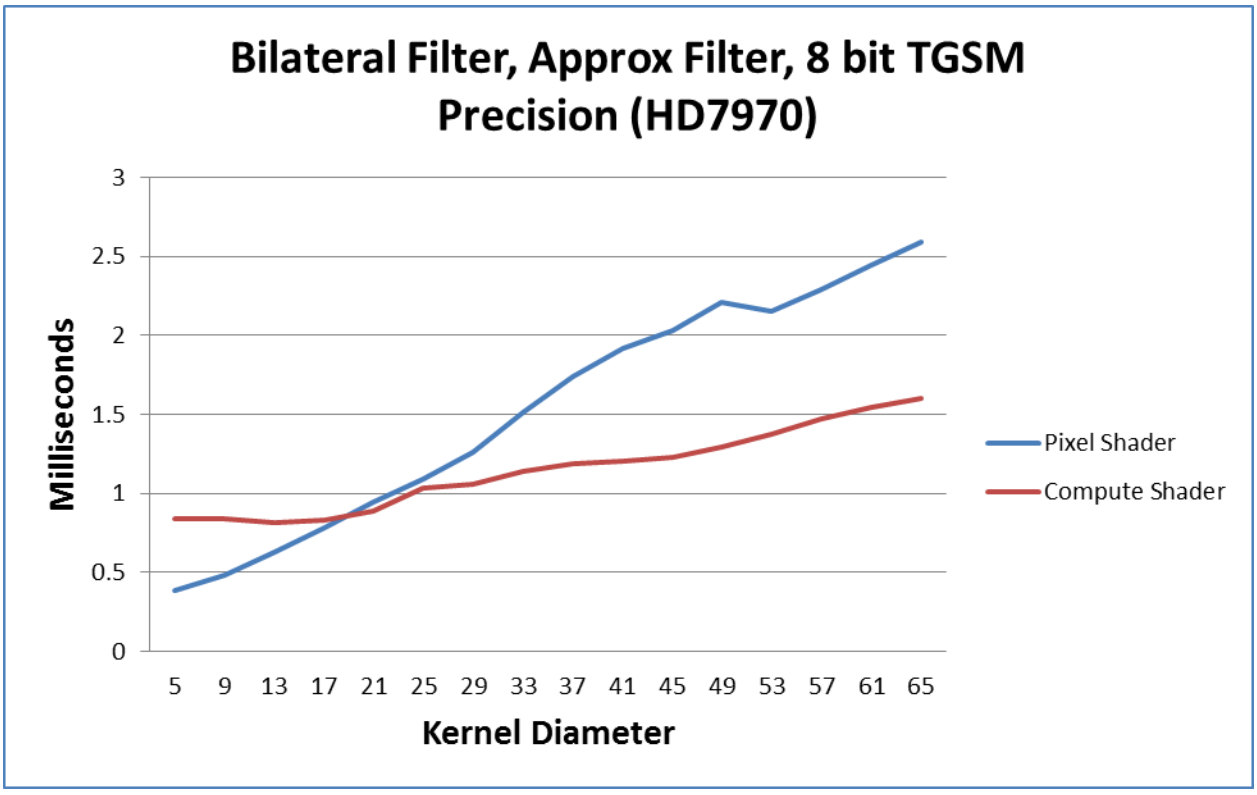
```
/T cs_5_0
/E CSFilterY Source\BilateralFilter.hlsl
/Fo Object\CSFilterY.obj
/D USE_COMPUTE_SHADER=1
/D KERNEL_RADIUS=6
/D USE_APPROXIMATE_FILTER=1
/D LDS_PRECISION=16
```

Rendering Source Code

The rendering source code that actually submits the appropriate calls to Direct3D for the Pixel and Compute Shader paths, has been encapsulated in the files SeparableFilter.h and SeparableFilter.cpp. It is intended that it should be relatively easy to cut and paste this code or take it wholesale for use in another application.

Performance





Call to Action

If you currently make use of 2 pass separable filters we would strongly recommend that you consider trying your filter out using the macros described here, and measuring the performance gain to be had from DirectCompute11. In many cases a 2x increase can be achieved with larger kernel sizes – but why not test it out on your machine…

Advanced Micro Devices
One AMD Place
P.O. Box 3453
Sunnyvale, CA 94088-3453

http://www.amd.com
http://developer.amd.com