ComputeShaderSort11

<https://github.com/walbourn/directx-sdk-samples>

This is the DirectX SDK's Direct3D 11 sample updated to use Visual Studio 2012 and the Windows SDK 8.0 without any dependencies on legacy DirectX SDK content. This sample is a Win32 desktop DirectX 11.0 application for Windows 10, Windows 8.1, Windows 8, Windows 7, and Windows Vista Service Pack 2 with the DirectX 11.0 runtime.

**This is based on the legacy DirectX SDK (June 2010) Win32 desktop sample. This is not intended for use with Windows Store apps, Windows RT, or universal Windows apps.**

# Description

This sample demonstrates the basic usage of the DirectX 11 Compute Shader 4.0 feature to implement a bitonic sort algorithm. It also highlights the considerations that must be taken to achieve good performance.

http://i1.code.msdn.s-msft.com/directcompute-basic-win32-7d5a7408/image/file/94435/1/computeshadersort11.jpg

## Bitonic Sort

[Bitonic sort](http://en.wikipedia.org/wiki/Bitonic_sorter) is a simple algorithm that works by sorting the data set into alternating ascending and descending sorted sequences. These sequences can then be combined and sorted to produce larger sequences. This is repeated until you produce one final ascending sequence for the sorted data.

## Bitonic Sort with Compute Shader

Now let's look at how to implement the bitonic sort in computer shader for a single thread group. To achieve good performance when implementing the sorting algorithm, it is important to limit the amount of memory accesses where possible. Because this algorithm has very few ALU operations and is limited by its memory accesses, we perform portions of the sort in shared memory, which is significantly faster. Unfortunately, there are two problems that must be worked around. First, there is a limited amount of group shared memory and a limited number of threads in a group. And second, in CS4.0, the group shared memory supports random access reads but it does not support random access writes. Even with these limitations, it is possible to create an efficient implementation using group shared memory.

**Step 1:** Load the group shared memory. Each thread loads one element.

    shared\_data[GI] = Data[DTid.x];

**Step 2:** Next, the threads must by synchronized to guarantee that all of the elements are loaded because the next operation will perform a random access read.

    GroupMemoryBarrierWithGroupSync();

**Step 3:** Now each thread must pick the min or max of the two elements it is comparing. The thread cannot compare and swap both elements because that would require random access writes.

    unsigned int result = ((shared\_data[GI & ~j] <= shared\_data[GI | j]) == (bool)(g\_iLevelMask & DTid.x))? shared\_data[GI ^ j] : shared\_data[GI];

**Step 4:** Again, the threads must be synchronized. This is to prevent any threads from performing the write operation before all threads have completed the read.

    GroupMemoryBarrierWithGroupSync();

**Step 5:** The min or max is now stored in group shared memory and synchronized. (The algorithm loops back to step 3 and must finish all writes before threads start reading.)

    shared\_data[GI] = result;  
    GroupMemoryBarrierWithGroupSync();

**Step 6:** With the memory sorted, the results can be stored back to the buffer.

    Data[DTid.x] = shared\_data[GI];

## Sorting More Data

The bitonic sort shader we have created works great when the data set is small enough to run with one thread group. Unfortunately, for CS4.0, this means a maximum of 512 elements, which is the largest power of 2 number of threads in a group. To solve this, we can add two additional steps to the algorithm. When we need to sort a section that is too large to be processed by a single group of threads, we transpose the entire data set. With the data transposed, larger sort steps can be performed entirely in shared memory without changing the bitonic sort algorithm. Once the large steps are completed, the data can be transposed back to complete the smaller steps of the sort.

## Transpose

Implementing a transpose in Compute Shader is simple, but making it efficient requires a little bit of care. For best memory performance, it is preferable to access memory in a nice linear and consecutive pattern. Reading a row of data from the source with multiple threads is naturally a linear memory access. However, when that row is written to the destination as a column, the writes are no longer consecutive in memory. To achieve the best performance, a square block of data is first read into group shared memory as multiple contiguous memory reads. Then the shared memory is accessed as column data so that it can be written back as multiple contiguous memory writes. This allows us to shift the burden of the nonlinear access pattern to the high-performance group shared memory.

# Dependencies

Samples typically make use of runtime HLSL compilation. Build-time compilation is recommended for all production Direct3D applications, but for experimentation and samples development runtime HLSL compilation is preferred. Therefore, the D3DCompile\*.DLL must be available in the search path when these programs are executed.

* When using the Windows 8.x SDK and targeting Windows Vista or later, you can include the D3DCompile\_46 or D3DCompile\_47 DLL side-by-side with your application copying the file from the REDIST folder.

%ProgramFiles(x86)%\Windows kits\8.0\Redist\D3D\arm, x86 or x64

%ProgramFiles(x86)%\Windows kits\8.1\Redist\D3D\arm, x86 or x64

%ProgramFiles(x86)%\Windows kits\10\Redist\D3D\arm, x86 or x64

# More Information

[Where is the DirectX SDK (2015 Edition)?](https://walbourn.github.io/where-is-the-directx-sdk-2015-edition/)

[DXUT for Win32 Desktop Update](https://walbourn.github.io/dxut-for-win32-desktop-update/)

[Games for Windows and DirectX SDK blog](https://walbourn.github.io/)