OpenCL Harness

Dynamic inspection of OpenCL GPU Kernel, OpenCL CPU Kernel, and Direct Compute Shader

Last updated: November 22, 2013

# Introduction

This paper is a survey of the development and inspection of GPGPU paths from a novice perspective. It centers on a VS2012 DX11 sample called OCLHarness that was written to help learn the basics of OpenCL and Direct Compute and inspect the similarities, differences, advantages, disadvantages to the different GPGPU paths. The sample was written from a Naïve perspective. Meaning that while having considerable experience with rendering engines no real experience in GPGPU programming existed when this project was created. The Harness in short is built around a simple working set where each work item is based on a 3x3 data area. It was created around a very simple Cellular Automata program which lends itself ideally to GPGPU programming. It provides a great way to play with shared buffers as well as give some graphical rewards for the programming efforts. The harness sets up 3 compute paths for the Cellular Automata life rules; OpenCL GPU kernel, OpenCL CPU kernel, and Direct Compute Shader; These paths strictly update the life rules each pass and then the current rule set is used to render the life state via a color look up for each pixel. The buffers are shared between the compute paths and you can dynamically switch between compute paths to see real time differences in performance and behavior. This sample highlights some simple observations for both functionality and performance but it by no means is meant to be a performance tuning document but rather a survey or base sample for anyone wanting to investigate this topic anew.

# OCLHarness

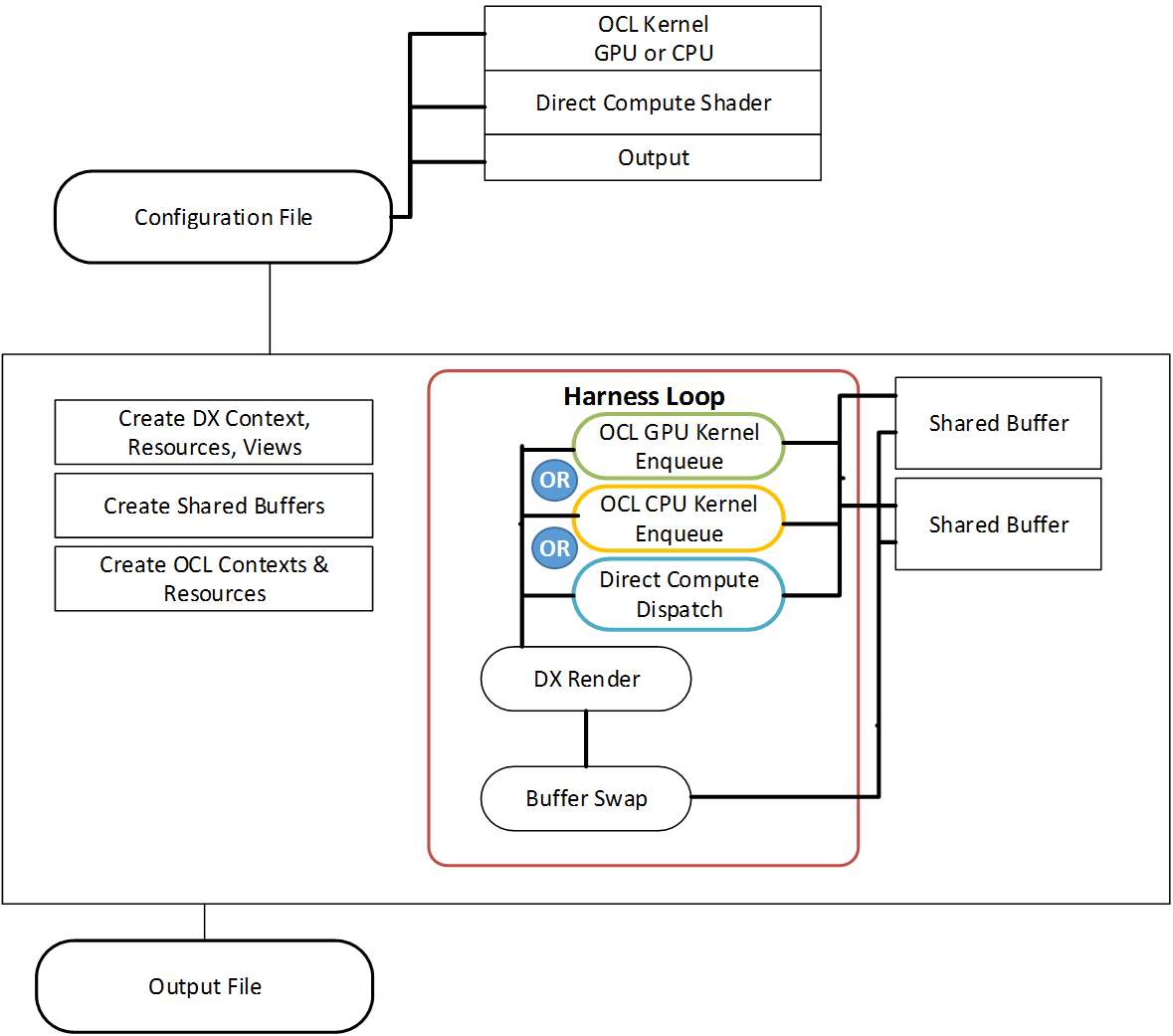
## About the program

The program is a DX11 based program that uses Compute Shader 5.0, Open CL 1.2 with the extensions for DX11 buffer sharing. There is a rendering thread called OCLHarness that loops continually.

* Choose compute path for life rule buffer update

One of:

* + Direct Compute
    - Dispatch compute shader
  + OpenCL GPU
    - Enqueue OpenCL Kernel
  + OpenCL CPU
    - Enqueue OpenCL Kernel
* Render Current Life State
  + Vertex Shader, Pixel Shader
    - Draw using buffer info
* Swap buffers



Setup for the program consists of creating a DX11 context, DX11 resources, Open CL context, Open CL resources. One of the keys to the resource creation is the shared buffers needed to for the life rules. This is a set of 2 buffers that will contain the current and next life states. The buffers will be swapped at the end of each render cycle. Simple Structured Buffers were used in this case that were globally accessible to the OCL kernels and Direct Compute shader. This exercise is by no means comprehensive into the types of buffer sharing available. This seemed to be the straightest forward for this exercise but the use of image buffers could have been used as well. The Direct Compute path defines a StructuredBuffer as well as a RWStructuredBuffer that are mapped to the two created buffers that have associated ShaderResourceViews and UnorderedAccessViews respectively. The OpenCL Kernel path passes in the two buffers as global variables and are set up using the "clCreateFromD3D11BufferKHR" extensions. The pixel shader also uses the Shader Resource view of the buffers to read the state values from the StructuredBuffer and use a color lookup based on the state value to assign a color to each pixel. The vertex buffer is uninteresting and merely processes a simple quad (two triangles) that expands the display resolution.

The files of interest here would be:

OCLHarness.cpp, CellularAutomata.cl, ComputeShader.csh, PixelShader.psh

Functions of interest initial inspection would be:

OCLHarness.cpp: CreateDXSharedBufferResources(), OCLHarness\_Init(), OCLHarness()

Points of special interest in setting up;

Use D3D11\_RESOURCE\_MISC\_SHARED in the misc flags of the buffer description when creating the shared buffer. This definitely provides a performance benefit by allowing the buffer to be shared rather than copied internally.

Use CL\_MEM\_USE\_HOST\_PTR with clCreateFromD3D11BufferKHRFunc when creating the OCL view to the buffer. This also allows reading between the host and gpu without having to copy the buffer back and forth for reading.

When calling clEnqueueNDRangeKernel you can set the local work size variable to NULL and it will automagically create a good work size dimension. Note however this only works if you are not using shared local memory in the kernel where the dimensions have to be known ahead of time. More on this later.

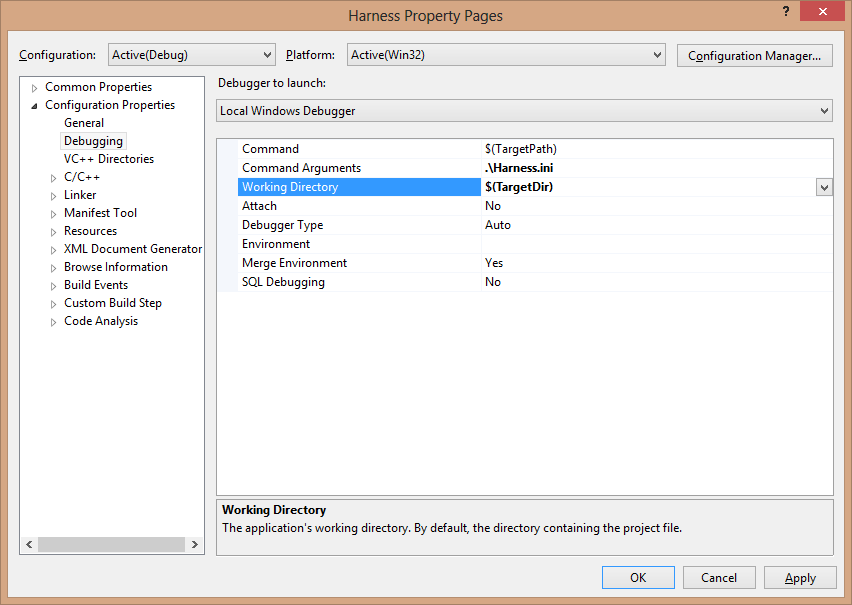
## Setting up and running the program

The OCLHarness program is a Visual Studio 2012 solution that supports x86 and x64 platforms with both release and debug configurations. It requires the DirectX SDK or Win8 SDK and the Intel OpenCL SDK.

OCLHarness takes a configuration file for input that allows for some variation in running the program:

A typical command line: C:\OCLHarness\bin\OCLHarnessX64.exe .\Harness.ini

In setting up the solution working directory and command line arguments are user specific and have to be set up. Note the working directory in this case is $(TargetDir) and the command line argument is “.\Harness.ini”



The Configuration File:

[INITVALS]

RESOLUTIONWIDTH=1600

RESOLUTIONHEIGHT=900

WINDOWEDMODE= 1

OCLKERNEL ="../src/CellularAutomata.cl"

VERTEXSHADER ="../src/VertexShader.vsh"

PIXELSHADER ="../src/PixelShader.psh"

COMPUTESHADER ="../src/ComputeShader.csh"

ENTRYNAME ="CellularAutomataHS"

//OpenCL kernel Build Rules

CPUBUILDRULE ="-cl-fast-relaxed-math"

GPUBUILDRULE ="-cl-fast-relaxed-math"

GROUPSIZEX = 160

GROUPSIZEY = 3

NUM\_GROUPSX = 10

NUM\_GROUPSY = 30

// Set to which compute path to start with

// use input of “G”, “C”, “D” changes the set path to

// OCL\_GPU, OCL\_CPU, DX\_CS respectively

OCL\_GPU = 1

OCL\_CPU = 0

DX\_CS = 0

This should give you the preliminary setup for the project so you can run the program and get started.

There is an output file that is created by default called wickedweasle.csv. It has frame rate, resolution, and compute path dimensions for any compute path that was run.

# Resolution Setup or “OMG” this just got hard.

Part of the configuration in the previous section outlines setting the resolution. It would be nice if you could just change the display resolution however when changing the resolution you immediately jump ahead to some rather important details that isn’t necessarily easy to digest at this stage. It requires some understanding of some fundamental GPGPU programing structure and how work item or threads depending on your semantics and work groups are set up and used. In this instance it’s because our shared buffers are the same dimensions as the display resolution. Apologies for jumping in head first but it should be an exciting time in the deep end of the pool. This document and program covers both OpenCL and Direct Compute. A mild effort is given to tie the semantics of both compute paths together so it makes a bit more sense. For the Working Set of compute tasks OpenCL uses the terms Work Group, Work item where Direct Compute uses Thread Group and Thread for the same concept. There are three basic ways to call on the GPU to do special programming. Aside from setup there is the traditional “Draw” call for graphics rendering but for GPGPU computing you either have “Dispatch” for direct compute or “Enqueue” for OpenCL kernels. Note: Enqueue is also called for the CPU OCL kernels.

Basically we can start from the two ends of the working set and then work in. On one end the complete number of pseudo identical tasks to compute or said another way the range of compute tasks is a way to describe the problem and at the other is the individual task or threads themselves to be executed. An OCL kernel or DC Shader is a program that is written from the perspective of the current lowest level (work item or thread) (OCL or DC) to be executed. The program executes once for each workitem/thread in the working set. In our case for this program we have compute program that is executed by each thread or work item that works on its own 3x3 array of data. If we look at the other end we have the complete collection of tasks and in our case our collection spans one work item or thread per pixel. So the Working Set range or N-Dimensional Range is described by the width and height of the display resolution. Since GPGPU programming works on multiple work items/Threads at one time we need to define how to break up the working set into Work Groups / Thread Groups. Groups are defined in dimensions just like the working set is. While you have a specific number of work items in each group you need to think of the work group in terms of its dimensions. This is where things get tricky. For Direct Compute you must know ahead of time what the dimensions are for the thread groups. For OpenCL you don’t have to know anything but the overall working set dimensions which is in our case the WindowWidth and WindowHeight. However if you don’t specify the work group/ thread group dimensions then you cannot use shared local memory. Shared local memory is only accessible on Group Boundaries and can only be declared with immediate scalar values so they can’t be dynamically declared based on a changing buffer dimension. So that means in order to use shared local memory you must know the work group dimensions which means you also need to know limits of work group sizes for the architecture you are targeting. For Direct Compute you must declare the Thread Group dimensions in the Compute Shader Entry point and the number of Workgroups / thread groups in each dimension for the dispatch call. Let’s see if we can make some sense of it.

Direct Compute Dimensions

In our case for the 1600 x 900

#define THREADS\_X 160

#define THREADS\_Y 3

StructuredBuffer<CABufType> caBuffOld;

RWStructuredBuffer<CABufType> caBuffNew;

[numthreads(THREADS\_X,THREADS\_Y,1)]

void CellularAutomataHS( uint3 get\_group\_id : SV\_GroupID,

uint3 get\_global\_id : SV\_DispatchThreadID,

uint3 get\_local\_id : SV\_GroupThreadID,

uint GI : SV\_GroupIndex )

g\_pd3dImmediateContext->Dispatch( 10, 300, 1 );

Note: ThreadsX \* GroupsX = 1600 and ThreadsY\*GroupsY=900 our display dimensions

So our ThreadGroup Size is 160x3 or 480 Threads times the number of workgroups 10x300 or 3000 gives 480\*3000 = 1,440,000 work items which is the same as the number of pixels 1600\*900 also = 1,440,000

For OCL kernel the entry point is:

\_\_kernel

void CellularAutomataHS(

\_\_global CA\_STATE\* caBuffOld,

\_\_global CA\_STATE\* caBuffNew,

OCL for the GPU the dimensions are the same as the Direct Compute. In this case it was set up automatically based on the global work size of [1600,900] and the local work size set to NULL. It could have been explicitly set.

const size\_t globalWorkSize[2] = { g\_WindowWidth, g\_WindowHeight };

const size\_t localWorkSize[2] = { 160, 3 };

note: for OCL\_CPU this resolution with the localWorkSize set to NULL the localWorkSize would be {800,1} with a number of groups {2,900}

cl\_event ev;

clEnqueueNDRangeKernel( g\_cl\_gpu\_cmdQueue, // valid device command queue

g\_cl\_gpu\_kernelCA, // compiled kernel

2, // work dimensions

NULL, // global work offset

globalWorkSize, // global work size

localWorkSize, // local work size

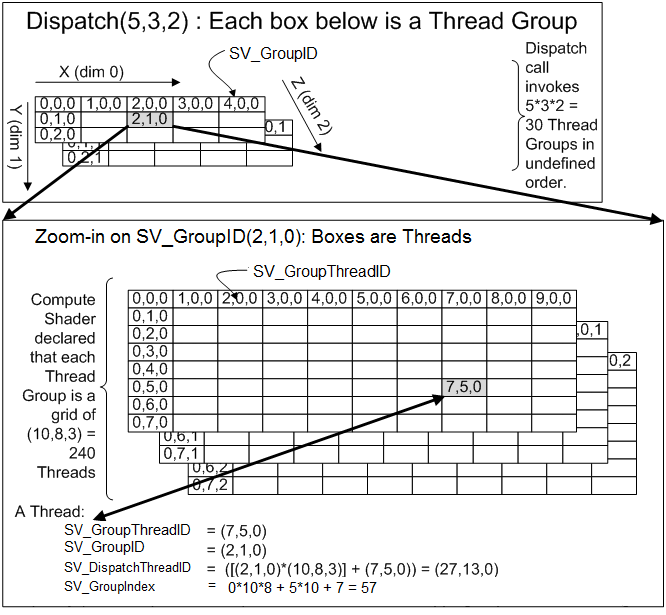
//NULL, // local work size

0, // number of events to wait for

NULL, // list of events to wait for

&ev); // this event

clWaitForEvents(1, &ev);

To borrow from the MSDN library the same diagram that’s been hijacked in a lot of publications, the dispatch diagram. 

What does all this mean? When we go to change resolution we need to be aware of the dimensional information in both the dispatch and enqueue calls as well as the thread dimensions in the compute shader. If you are using shared local memory then the dimensions need to be included and adjusted accordingly in the OCL kernel as well. In the case of OCLHarness that means if you change the resolution you need to accommodate for adjusted dimensions for the work groups and number of work groups.

By example let’s try a different resolution. 1366 x 768. What do we need to do?

1. Change the resolution in the Harness.ini file.
2. Set the OCL\_GPU configuration variable to 1 (true) and OCL\_GPU and DX\_DC variable to 0 (false)
3. Set a break point just past the buffer read in the OCL\_GPU path of the OCLHarness() loop.
4. Make sure that the local work size variable in the enqueue call is set to NULL
5. Look at the dimension values returned in myDims

In this case we get

* + groupSizeX 2
  + groupSizeY 256
  + groupNumX 683
  + groupNumY 3
  + worksetSizeX 1366
  + worksetSizeY 768

1. Just note for learning sake that 2x683 = 1366 our width and 256\*3=768 our height.
2. Reset the program and make some additional changes.
3. In the config (harness.ini) file change the group size and group number variables to reflect the newly retrieved values.
4. The dispatch call will now reflect g\_pd3dImmediateContext->Dispatch( 683, 3, 1 ); if you break on it.
5. In the Compute Shader file change the ThreadX to 2 and the ThreadY to 256
6. You should now be able to clear the breakpoints and fly.

# Observations

# Cellular Automata Algorithm

Cellular Automata or life rules is simply a sequential state machine where by the next state is defined by some rules imposed on the present state. Also known as life rules. Common CA themes are defined by an array of cells where by the next state of each cell is defined by applying some rule to the surrounding cells. i.e. if cell 1, 7, 8 are alive change to alive otherwise my state is dead. The rules are endless. In this case my CA life rule is to have a choice of 16 states [0..15] for each cell. The next state is the product of all non zero adjoining cells adding 1 and taking the modulus of 16. I then display the current state by assigning each of the 16 states to a unique color in a color pallet.



The guts of the kernel is the iteration over the rules with a \*= for each valid non zero cell. “the product for all adjoining cells”. Height Compensation “hComp” and Stride Compensation “sComp” are variables to determine offsets for the edge cases of the buffer for wrapping data.

The Lookup is just a variation of a single dimensional array treated as a two dimensional array with the form offset into buffer = currentRow \* rowLength + current row position “Y\*Stride + offsetX”

uint hComp[3] = { heightTComp, 1, heightBComp };

uint sComp[3] = { strideLComp, 0, strideRComp };

uint maskDimX = 3;

uint maskDimY = 3;

uint mask[3][3] =

{

{ 1, 1, 1 }, { 1, 0, 1 }, { 1, 1, 1 }

};

for ( int h = 0; h < maskDimY; h++)

{

for (int s = 0; s < maskDimX; s++)

{

if( mask[s][h] != 0 )

{

if( caBuffOld[

buffIdx + ((Stride \* hComp[h]) \* ( h-1 )) + ( sComp[s] + s-1 )

].state != 0 )

{

product \*= caBuffOld[

buffIdx + ((Stride \* hComp[h]) \* ( h-1 )) + ( sComp[s] + s-1)

].state;

}

}

}