PARALLEL PROPAGATION OF ERROR ON A GPU

©PHILIPP DRIEGER 2012 – NOUMENTALIA.DE – DIGITAL ARTS

Abstract By serendipity this visual computing experiment led to a visualization of colorful structures that evolve from the parallel propagation of errors on a graphics processing unit (GPU). We wanted to transfer color values in subsequent volume slices using an OpenCL kernel. Due to concurrent memory access this transfer operation was erroneous and produced interesting visual patterns in a point-based visualization. Meanwhile we fixed this "error" with another kernel but we wanted to share the initial results as a piece of visual computing art. The visual patterns reveal the structure of block-wise kernel execution on the GPU. In this short paper we briefly describe the setup of this experiment and give an explanation of the observed effects.

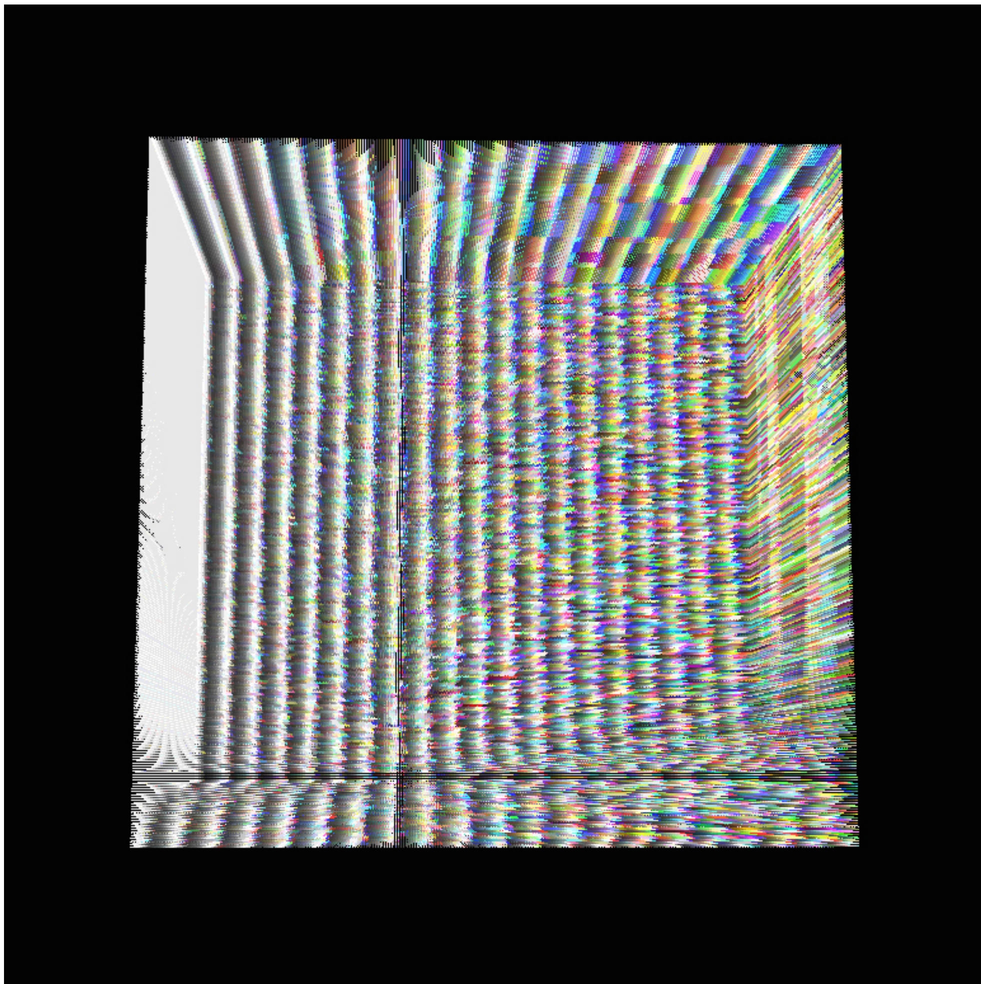

Figure 1. Visualization of the experiment. The color values are propagated from left to right through the volume. The structure of the colors illustrate the kernel execution in blocks.

1. Experimental setup

The experimental setup is based on a cubic volume represented by a set (N^3) of points in $\mathbb{R}^3$ with $N = 256$ yielding $256^3 = 16777216$ points. The cube can be interpreted as a set of N slices $(S_0, S_1, \ldots, S_N)$ of N^2 points yielding 256 slices with each containing $256^2 = 65536$ points. Each point is represented by a vector and a color value. Initially, the points are arranged in a regular grid.

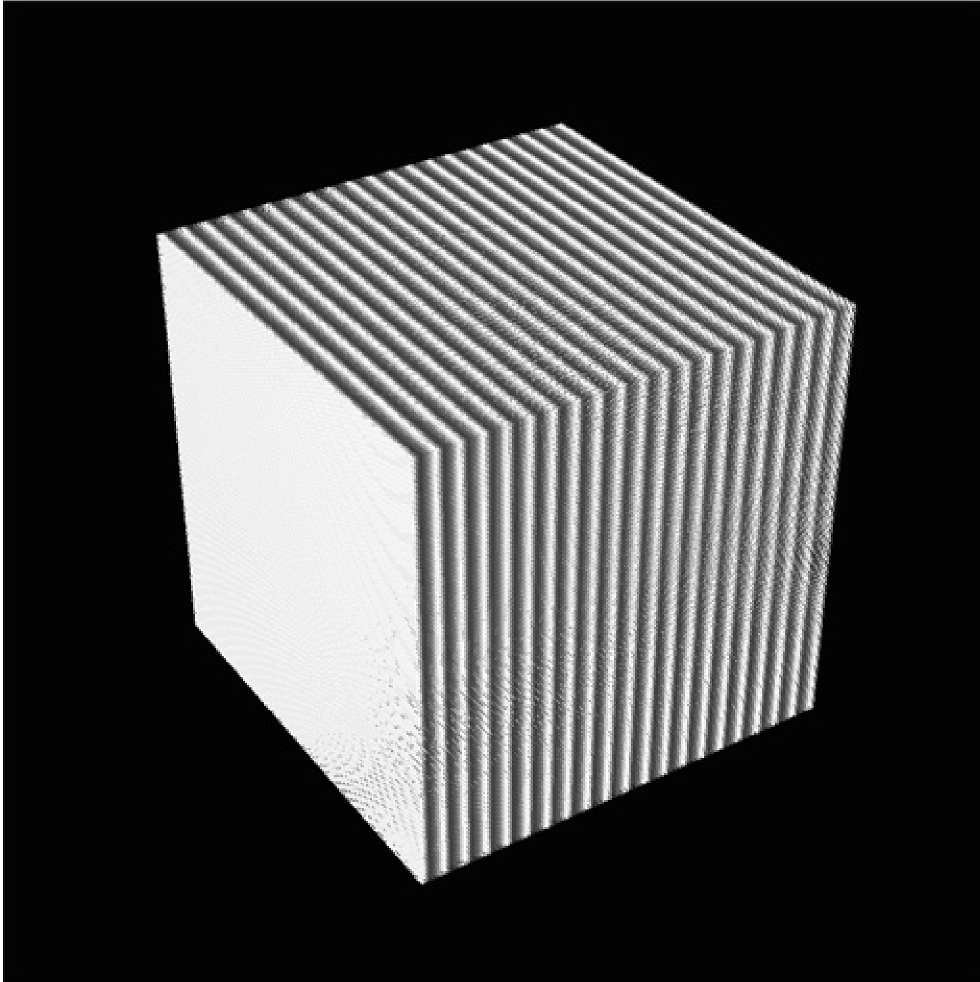

Figure 2. Correct version of the kernel to propagate the color values.

It is intended to copy the points' color values from one slice S_{n-1} to the next slice S_n by the use of a simple kernel program. As the kernel program is executed in parallel on a GPU all points in a slice are processed in parallel. All color values of the first slice S_0 are bound to an simple sinus function oscillating between black and white. After each update of S_0 the kernel is being executed over the given volume in order to copy one slice to the next, passing the color values from one slice to the next. In each timestep all slices should carry the values generated by the oscillator reaching from black to white over the grayscale, looking like a "zebra cube" as shown in figure 2. Running this setup in a real-time 3D environment, we expect to observe this effect as described. One side of the cube is oscillating from black to white as expected, but surprisingly the color values of succeeding slices get more and more distorted as shown in figure 1. Looking at the last slice S_n colorful

structures appear after they changed from slice to slice after passing the cube. Examining the structure of the colors could lead to the visual anticipation of an explanation of this effect.

2. Explanation of effects

In this experiment the kernel is running in parallel over the given set of points to propagate values from one slice to another. The execution of the kernel on the used GPU (NVIDIA GeForce GTX 580) follows the SIMT paradigm (Single Instruction, Multiple Threads). The kernel code is executed in warps which are groups of 32 threads running in parallel. Kernel1 (see section 3.4) has a race condition that leads to erroneous results due to concurrent operations. A thread can propagate the value from one slice to the next one prior to the previous slice being computed. As every 32 threads operate in one lock-step the race conditions will always occur the same for every warp. By repeatedly running the kernel the errors spread from one slice to another. As a result, the complete volume is filled with changing errors that lead to the described effect.

As a side effect, the colorful structures reflects the SIMT execution and therefore may illustrate the operation of a GPU. The colored structures clearly show rectangular blocks of the same color and thus indicate blocks of the same thread execution. Each colored block represents a warp in the volume. Interestingly, this typical structure of kernel execution on a GPU reveal from the propagation of the errors and their coloring.

3. Code snippets

The following code snippets have been used to achieve the described effects. Both kernels operate on the given set of variables and use a common index function to access points in the volume:

3.1. Code snippet: given variables.

```
// size of the cube:
VBOobject.NX = VBOobject.NY = VBOobject.NZ = 256;

// oscillator value:
float[] params = (float3)((Math.Sin(runner) + 1.0f) * 0.5f);
```

3.2. Code snippet: host-side kernel execution call.

```
// call for executing kernel1:
LayoutCQ.Execute(ComputationKernel, null, new long[3]
{ (long)VBOobject.NX, (long)VBOobject.NY, (long)VBOobject.NZ },
null, null);

// call for executing kernel2:
LayoutCQ.Execute(ComputationKernel, null, new long[2]
{ (long)VBOobject.NX, (long)VBOobject.NZ },
null, null);
```

3.3. Code snippet: index function used in both kernels.

```
int index(int ix, int iy, int iz, int nx, int ny) {
    return (iz*nx*ny + iy*nx + ix);
}
```

3.4. Code snippet 1: ”erroneous” kernel (correspond to figure 1).

```
__kernel void kernel1(
__global float * vec,// shared opengl vertex positions
__global uchar * col,// shared opengl color values
__global int * sizes,// size of the cube (256^3)
__global float * params // oscillator values
)
    {
        int ix = get_global_id(0);
        int iy = get_global_id(1);
        int iz = get_global_id(2);
        int i = index(ix, iy, iz, sizes[0], sizes[1]);
        if(iy<sizes[1]-1) {
            int i1 = index(ix, iy+1, iz, sizes[0], sizes[1]);
            col[i*3+0] = col[i1*3+0];
            col[i*3+1] = col[i1*3+1];
            col[i*3+2] = col[i1*3+2];
        }
        else {
            col[i*3+0] = (uchar)(params[0]*255.0f);
            col[i*3+1] = (uchar)(params[1]*255.0f);
            col[i*3+2] = (uchar)(params[2]*255.0f);
        }
    }
```

3.5. Code snippet 2: ”correct” kernel (correspond to figure 2).

```
__kernel void kernel2(
__global float * vec,// shared opengl vertex positions
__global uchar * col,// shared opengl color values
__global int * sizes,// size of the cube (256^3)
__global float * params // oscillator values
)
    {
        int ix = get_global_id(0);
        int iz = get_global_id(1);
        for(int iy = 0; iy<sizes[1]; iy++) {
            int i = index(ix, iy, iz, sizes[0], sizes[1]);
            if(iy<sizes[1]-1) {
                int i1 = index(ix, iy+1, iz, sizes[0], sizes[1]);
                col[i*3+0] = col[i1*3+0];
                col[i*3+1] = col[i1*3+1];
                col[i*3+2] = col[i1*3+2];
            }
            else {
                col[i*3+0] = (uchar)(params[0]*255.0f);
                col[i*3+1] = (uchar)(params[1]*255.0f);
                col[i*3+2] = (uchar)(params[2]*255.0f);
            }
        }
    }
```

4. Conclusion

We described the setup of this visual computing experiment and gave an explanation of the effects which occurred by serendipity as a result of erroneous operations. The visualization of the propagation of the error revealed the typical block-wise structure of kernel executions on GPUs. As a side effect, we can watch the GPU working to produce a volume of millions of errors by propagation and recombination. You can watch this experiment running in real time on YouTube: http://www.youtube.com/watch?v=15yMLU-jmyg

5. Acknowledgements

We thank Dr. Justin Luitjens (NVIDIA Cooperation) for technical discussions and his profound explanations of the described effects.

This experiment has been contributed to http://www.visualcompute.com/ – a space for visual computing arts.

For further information visit http://www.noumentalia.de/

©Philipp Drieger 2012