Color Correction Acceleration Using a Color Cube and OpenCL

Tenio Vachev

Faculty of Mathematics and Informatics,
St. Kl. Ohridski University of Sofia, Bulgaria
UP2 Technology, www.up2.eu
tvachev@up2.eu, tvachev@hotmail.com

Abstract. The article deals with the problem of real time color correction on modern but not dedicated video hardware, suggesting a new implementation of fast algorithm for color transformation utilizing 3D look-up tables. We focus on highly parallel nature of the proposed method and employ the GPU to perform the color calculations side-by-side. The paper is comparing the performance of the implementation of the algorithm on the CPU and on the GPU.

Keywords: secondary color correction, tetrahedral interpolation, LUT, color cube, GPU, parallel computing, OpenCL

1 Introduction

The modern hardware platforms - both central and graphics processors are SIMD (Single Instruction Multiple Data Streams). They can be targeted through a common abstraction and API. We are utilizing OpenCL to make parallel calculations on both the CPU and the GPU in order to compare the performance on them as a step towards the goal to make soft real-time color correction of HD and 4K [1] resolution images available on systems with modern graphics cards.

OpenCL (Open Computing Language) is a framework for developing and executing parallel computation across heterogeneous processors (CPUs, GPUs and other Digital Signal Processing (DSP) hardware). It is an open standard managed by the non-profit consortium the Khronos Group [5]. OpenCL supports both data-parallel and task-parallel programming models. OpenCL gives applications access to the Graphical Processing Unit for non-graphical computing. A key feature, however, is that it is designed not only as a GPU programming platform, but also as a parallel platform for programming across a range of computational devices. This fact makes it different to NVIDIA's CUDA [6], ATI's Stream [7] and Microsoft's DirectCompute which are proprietary and designed to work only on their respective hardware platforms or in the case of Microsoft coupled with an operating system and DirectX.

2 Tetrahedral Transformation

There are many interpolation algorithms that can be used to calculate a point from an arbitrary set of input data. These general interpolation algorithms give

good results, but at the same time we have to pay attention on their complexity and performance. At absence of a general way to quickly find out which of the known points lie closest to our target point, we must traverse them all before we can interpolate. Other problems are present when close to the gamut limits, and when all the nearest points are on one side of the target point.

One type of algorithm which meets our criteria is a tetrahedral transform which calculates the value at a point in the domain by using functional values at four known points in a three dimensional space. To speed up the calculations, we choose a special set of points at equal distances to interpolate from. They are arranged in rows and columns that go right up and down to the gamut limits.

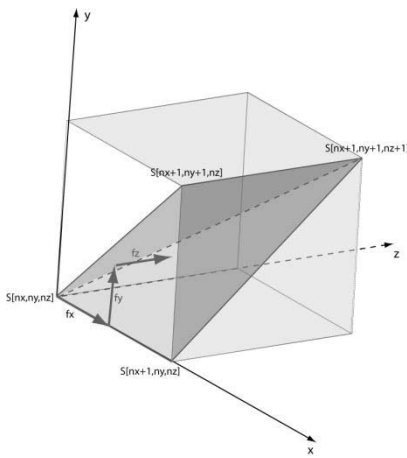

Fig. 1

This tetrahedral shape has three faces parallel to the cube edges: one parallel to each one of the axis. Each of these edges has a known value of S (the color transformation function) at each end, so we can calculate the gradient of S in the three perpendicular directions. If we have the value in the first corner, and the gradient in three perpendicular directions, it is easy to calculate the value at point (f_x, f_y, f_z). The result of the interpolation will be:

$$S_{xyz} = (1-f_x)S_{n_x n_y n_z} + (f_x - f_y)S_{(n+1)_x n_y n_z} + (f_y - f_z)S_{(n+1)_x (n+1)_y n_z} + (f_z)S_{(n+1)_x (n+1)_y (n+1)_z}$$

This formula is for the case $f_x \geq f_y \geq f_z$. We can obtain the respective expressions for the other cases by interchanging f_x, f_y and f_z.

3 Color Cube

The input image is modified by controls manipulating the hue, saturation and luminance components of the pixel color and filtering the color set we are working on. The color transformation goes in two stages - modifications are first applied on the bypass color cube, which in turn modifies the color information of the image – interpolating it.

The color correction method relies on utilizing a 3D LUT (Look-up table) [3] to modify color data in order to keep the color transformation in the input color space, which we assume RGB. Instead of going to additional color spaces as HSL [4], which are comfortable for a colorist to work with but require time consuming transformations, for each pixel we are applying color modifications to a relatively small number of colors and modifying the input data through this subset by interpolating it.

4 Color Cube Applications

- Calibration

Let's have an image Ω_1, which represents the colors in the default color cube. If we want to calibrate a device we can display Ω_1 from the device resulting in another image Ω_2. By building a color cube which is a result of the difference Ω_1 - Ω_2 and applying it to the output of the device we will prevent the color distortion (if it is not drastic), caused by the very nature of the device.

- Gamut Warning

We can issue a warning if a color is moving into an area which cannot be displayed or reproduced on print film by applying the color cube to an image and for example de -saturating the colors that are reproducible and leaving intact those that are out of range.

- Fixing Gamut Limits Specific to a Device

If we take the digital intermediate process as an example we will start by using the colorist's video monitor as a reference. We send the output of our color corrector unlimited to the monitor but we will expect to apply a gamut limit when we output to film.

- Secondary Color Correction

Secondary correction brings about alterations in luminance, saturation and hue. The main objective of secondary controls is to adjust values within a narrow range while having a minimum effect on the remainder of the color spectrum and thus the rest of the image [5]. Color cubes provide this and which is a plus – the nature of the interpolation results in gradual transitions between colors. We can theorize that splitting the color channels in real time can add another dimension to the movies e.g. make it 3D but more research should be done in this new direction.

5 OpenCL Implementation

A kernel implementing tetrahedral interpolation was developed which was run on the CPU and GPU. The kernel applies a color cube to an image by using tetrahedral interpolation. The same algorithm was implemented using C# on .Net Framework 4.0 for reference purposes.

The kernel consists of three functions:

```
#define ColorDepth 65535
#define NSteps  17

__kernel void colorcube(__constant uint4* inputImage,__constant uint4*
cube,__global uint4* outputImage) {…}
int sub(int x){return x/(ColorDepth/NSteps);}
float fract (int x){
                        int tmp = x/(ColorDepth/NSteps);
                        return ((float) x/(ColorDepth/NSteps) - tmp);}
```

The input image, color cube and the output frame are in form uint4 which consists of four unsigned integer values.

The sub function returns in which sub-cube the color lies in respect to the mesh of the cube and the fract shows the relative position of the color in the sub-cube itself. These are run for every color channel separately – R,G and B.

6 Performance Results

The PC test setup is:

Processor: Intel® Core™ i7 CPU Q 720 @ 1.60GHz
RAM: 12 GB DDR 3 in dual channel - running at 1333 MHz
Graphics Card: ATI Mobility Radeon 5870 RAM:1 GB Clock: 700MHz
OpenCL: OpenCL 1.0 ATI-Stream-v2.1 (145) Compute Units: 10
Graphics Driver: CAL 1.4.675 supports OpenCL

The test program was run for 1 and 1000 iterations of the kernel in order to remove the reading and writing times of the data over PCI Express and to get a notion on the time necessary for the calculations only. We also believe that the kernel could be optimized a lot because it did not utilize the parallelism enough and was heavily branched.

The results of the performance tests are shown in Table 1. (all times are in seconds and per iteration):

Table 1

Iterations	CPU (OpenCL)	GPU (OpenCL)	CPU (C#)
1	0.0460824	0.01465500	0.213
1000	0.0243477	0.00485557	

The testing image was 1024x1024 with 48 bits per pixel which allows for 16 bit color channels.

The times spent per pixel for CPU and GPU respectively are 1.16898E-08 and 2.33125E-09 which lead us to conclude that with this kernel we can apply a color cube to a 4K (4096 x 4096) image for *0.19612228* sec. on the CPU and for *0.039111927* sec. on the GPU.

The results reinforce the belief that we can achieve great performance improvements utilizing the graphics processor. We can optimize the kernel which will allow us to increase the image size, the color depth or both.

The real world scenarios are two:

- The colorist works on an image and modifies it – in this case the image stays the same the look-up table changes
- There is a feed of frames that needs to be modified in real-time (not rendered)

In the first scenario the image stays in the GPU memory and we can send only the color cube to the GPU in order to be modified. The real time requirements are none existent – only the perception of the operator is involved – we need it to happen “fast enough”.

The calibration and secondary color correction applications however require operations to be done in real time.

In the second scenario we have a constant stream of frames each coupled with a color cube. We need to be able to send the image and the look-up table, apply transformations and read back the output in time which is at most 25 times per second for PAL/SECAM and 29.97 for NTSC. At present we are able to do this for HD images.

7 Conclusion

We can achieve 48bpp (bits per pixel) full HD image color correction on a mainstream graphics card in soft real time.

The results show at least five times increase in performance utilizing the graphics processor compared to running on the CPU and are highly encouraging towards the goal to achieve "soft" real-time application of a color cube to an image. By real-time we mean speeds of below 40 milliseconds per 4K image with three 14-16 bits per channel.

This would mean that we can utilize the top end but mainstream graphics cards for doing professional color grading and thus make accessible more features that belong to dedicated professional video editing hardware. Calibration of input and output devices is also possible within a tighter budget.

There are areas that can be improved in the current solution. First and foremost the kernel could be optimized resulting in performance improvement that will give us time to utilize more than one cube applied per image. A sequence of color cubes applied in a cascading manner will give us much more flexibility in real world applications.

Acknowledgements. The paper is supported by Grant 162/2010 from Sofia University Research Fund.

References

1. Wikipedia. 4K Resolution. [Online]. http://en.wikipedia.org/wiki/4K_resolution
2. Pandora International Limited. [Online]. http://pogle.pandora-int.com/
3. Wikipedia. 3D Look Up Tables. [Online]. http://en.wikipedia.org/wiki/3D_LUT
4. Philippe COLANTONI. Color Spaces. [Online]. http://www.couleur.org/index.php?page=transformations
5. Khronos Group. [Online]. http://www.khronos.org
6. NVIDIA. CUDA Zone. [Online]. http://www.nvidia.com/object/cuda_home_new.html
7. AMD. ATI Stream Technology. [Online]. http://www.amd.com/us/products/technologies/stream-technology