RGB to YUV Transform Concurrently on Many Core GPU

41202, 41203, 41205: Aditi Wikhe, Jagruti Agrawal, Anuraag Shankar

YUV is a color encoding system typically used as part of a color image pipeline. It encodes a color image or video taking human perception into account, allowing reduced bandwidth for chrominance components, thereby typically enabling transmission errors or compression artifacts to be more efficiently masked by the human perception than using a "direct" RGB-representation. Other color encodings have similar properties, and the main reason to implement or investigate properties of Y'UV would be for interfacing with analog or digital television or photographic equipment that conforms to certain Y'UV standards. The project has been implemented in Python and parallelism has been achieved by the PyCUDA library.

1 INTRODUCTION

The Y'UV model defines a color space in terms of one luma component (Y') and two chrominance components, called U (blue projection) and V (red projection) respectively. The Y'UV color model is used in the PAL composite color video (excluding PAL-N) standard. Previous black-and-white systems used only luma (Y') information. Color information (U and V) was added separately via a subcarrier so that a black-and-white receiver would still be able to receive and display a color picture transmission in the receiver's native black-and-white format.

Y’ stands for the luma component (the brightness) and U and V are the chrominance (color) components; luminance is denoted by Y and luma by Y’ – the prime symbols (’) denote gamma correction, with “luminance” meaning physical linear-space brightness, while "luma" is (nonlinear) perceptual brightness.

Each unique Y, U and V value comprises 8 bits, or one byte, of data. Where supported, our color camera models allow YUV transmission in 24-, 16-, and 12-bit per pixel (bpp) format. In 16 and 12 bpp formats, the U and V color values are shared between pixels, which frees bandwidth and may increase frame rate. Known as “chroma subsampling,” this technique takes into account the human eye’s greater sensitivity to variations in brightness than in color.

Our cameras support YUV444, YUV422 and YUV411 data formats.

2 PROJECT SCOPE

The scope of the terms Y'UV, YUV, YCbCr, YPbPr, etc., is sometimes ambiguous and overlapping. Historically, the terms YUV and Y'UV were used for a specific analog encoding of color information in television systems, while YCbCr was used for digital encoding of color information suited for video and still-image compression and transmission such as MPEG and JPEG. Today, the term YUV is commonly used in the computer industry to describe file-formats that are encoded using YCbCr.

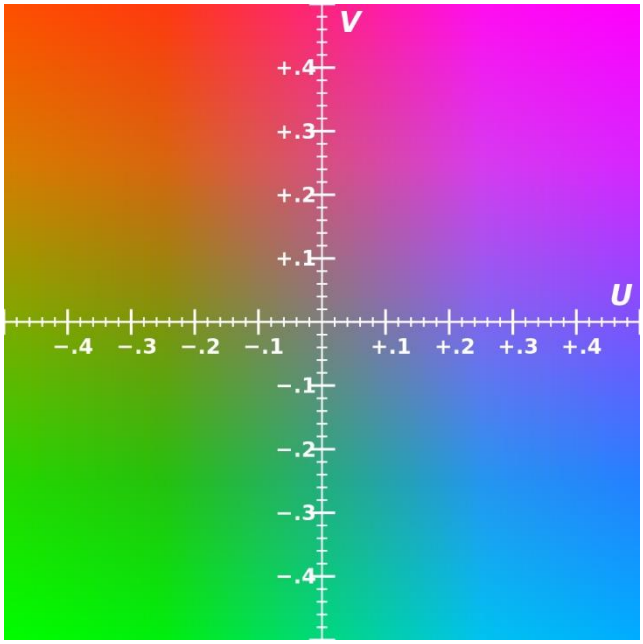

Fig. 1. UV Plane with Y' = 0.5

Less data means less time and energy used per image and therefore YUV is preferred in some embedded imaging applications. RGB image files might use unnecessary big amount of data, and even 10bit RAW format would be less data intensive if pre-processing, like white balance, is needed.

3 REQUIREMENTS

The project has been built using Python on Google Colaboratory. The following libraries were used while building the project:

1. PyCuda – pycuda
2. NumPy – numpy
3. Time – time
4. OpenCV – cv2
5. Google Colab – google.colab

The PyCUDA environment internally uses CUDA programming in C++. The CUDA/C++ code can be passed as an argument to the Python function. The configurations of the notebook are as follows:

1. Processor Name: Intel(R) Xeon(R)
2. Processor Speed: 2.20 GHz
3. Operating System: Ubuntu 18.04 64-bit
4. RAM: 12.0 GB
5. GPU: Nvidia Tesla K80.

4 ALGORITHM

The video is first split into frames. To convert the RGB image into YUV, the image must first be split into its individual channels. Then the channels are converted to Y, U and V respectively.

```
function rgb_to_yuv(r, g, b)
  y =  0.257 * r + 0.504 * g + 0.098 * b +  16
  u = -0.148 * r - 0.291 * g + 0.439 * b + 128
  v =  0.439 * r - 0.368 * g - 0.071 * b + 128
```

Algorithm 1. RGB to YUV Algorithm

Conversion can be done parallelly using CUDA functions as follows:

```
__global__ void rgb_to_yuv_cuda(float *y, float *u, float
*v, float *r, float *g, float *b)
{
  int idx = blockIdx.x * gridDim.y + blockIdx.y;
  y[idx] =  0.257 * r[idx] + 0.504 * g[idx] + 0.098 *
b[idx] +  16;
  u[idx] = -0.148 * r[idx] - 0.291 * g[idx] + 0.439 *
b[idx] + 128;
  v[idx] =  0.439 * r[idx] - 0.368 * g[idx] - 0.071 *
b[idx] + 128;
}
```

Algorithm 2. RGB to YUV Parallel Algorithm

5 RESULTS

Two sample frames have been shown here. As it can be observed, the YUV image by itself does not look appealing. But Y, U, V channels describe the image separately.

(a) (b)

Fig. 2. Sample Frame 1 (a) RGB (b) YUV

(a) (b) (c)

Fig. 3. Sample Frame 1 YUV Channels (a) Y (b) U (c) V

(a) (b)

Fig. 4. Sample Frame 2 (a) RGB (b) YUV

(a) (b) (c)

Fig. 5. Sample Frame 2 YUV Channels (a) Y (b) U (c) V

The following table shows a comparison of the time taken by the algorithms when performed sequentially on CPU vs parallelly on a GPU.

Image	CPU Time Taken	GPU Time Taken
Sample Frame 1	5.2116 secs	0.0098 secs
Sample Frame 2	5.3125 secs	0.0084 secs

6 CONCLUSION

In this project we took a video as an input and split it into frames. The individual frames were then converted into separate R, G, B channels. These channels were converted in Y, U, V respectively. This was achieved using parallelism using PyCUDA library. As observed, this program runs much faster than a sequential program. Thus GPU programming using CUDA can be really efficient when converting frames from one form to another.