GPU Programming - Colour Space Conversion

Alexandre A. S Lopes ID: 301307001
Jay Maity ID: -301302461
Simon Fraser University
CMPT 886 - Assignment 2

I. Introduction

YUV is a color space typically used in a color image pipeline. An image pipeline or video pipeline is the set of components commonly used between an image source and an image renderer [4]. It turns out in the context of image processing that we often need to transform images from RGB colour space and YUV colour space. Graphics processing units (GPUs) seems to be more suitable for this type of processing than the CPUs because they were created to accelerate the computation of the algorithms by using SIMD architectures to execute a single instruction for multiple data[1].

II. Objective

The goal of this report is to describe experiments and results, for RGB to YUV and YUV to RGB conversions in CPUs and GPUs. The objectives can be described as the following

1. Time required to copy data from for an image from main memory to GPU memory and vice versa
2. Loss of image data in conversion using GPU than that of CPU
3. Discussion on experiments with different parameters of GPU block and threads in order to find the GPU parameters that optimize the performance of transformation.

III. Methodology

We have performed experiments with two images first image was a ppm image of dimensions 1000x700 and file size 2.1 MB. Second image is 10000x7000 in dimension with a size of 201 MB. In this report we will use the nomenclature '**Image 1**' and '**image 2**' to refer to first and second images respectively.
All the experiments are executed on intel CPU with 16 cores of 2400 MHz and with NVIDIA TITAN X architecture PASCAL GPU. In order to measure the time lapses, we are using the timer created by "`sdkCreateTimer()`" function present in CUDA library. we are also considering the whole time spent in the conversion including the time to transfer the image to device memory and to transfer the results back from device to host.
After each transformation we are verifying whether the output image is correct and with small conversion losses.

A. CPU Conversion

The CPU conversions were performed in only one thread, the operations performed basically consists in multiplication and additions, each RGB and YUV channels are separately calculated.

B. GPU Conversion

Prior executing GPU tests we have called same conversion function once to avoid incorrect measurements and to warm up GPU. The reason behind it is , during the experiments, we found out that the first call to a kernel function takes more time than consecutives ones. It is because first call is needed to load our program and CUDA library in the GPU. The operations performed to convert the images are the same to those used in the CPU conversion model described before.

IV. Copying variables

Before GPU conversion starts working, we need to copy image data from main memory to GPU memory. It is taking on an average 0.5 microsecond. But copying data for the first time is taking more time (1.33 microseconds) than that of next iterations. We also have noticed that it is gradually decreasing as iteration progress. The possible reason behind this could be hardware level caching for same data. The results are same for copying GPU to main memory. First time 2.92 microseconds and on an average 0.7 microsecond.
We have also measured GPU memory allocation time by cudaMalloc and have similar kind of progression in terms of timing compare to GPU to Main memory.

Data Transfer between main Memory and GPU Memory

	Main > GPU	GPU> Main	Space Allocation(GPU)
Average	0.46ms	0.75ms	3.24ms
Max	1.33ms	2.92ms	5.25ms

V. Conversion loss

Conversion of each pixel is not same while we are using GPU compare to CPU conversion. We have added a tolerance of 2 for the absolute value of the difference and it seems to conform to all pixels.

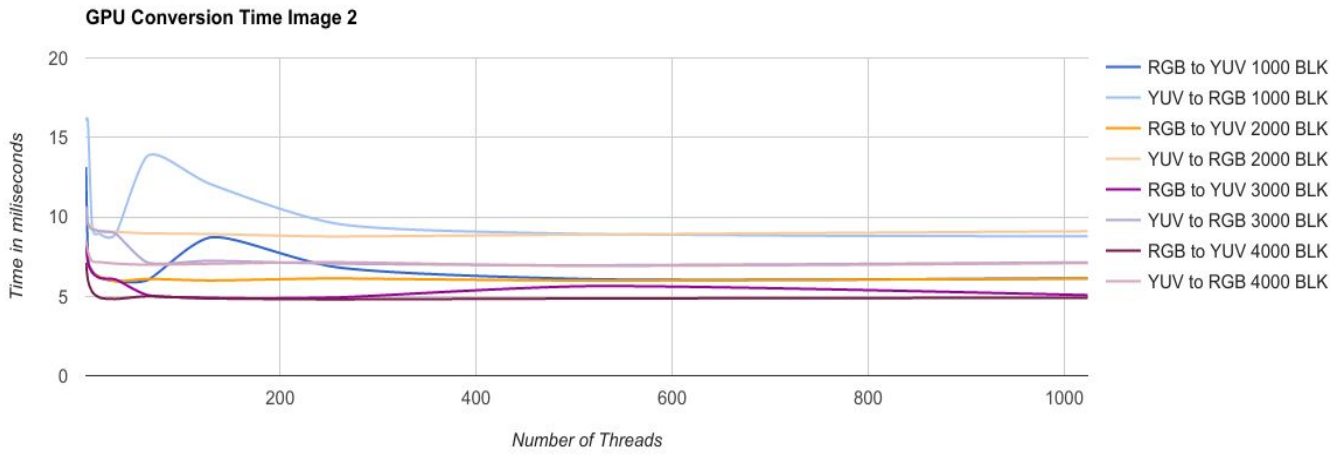


Figure 1

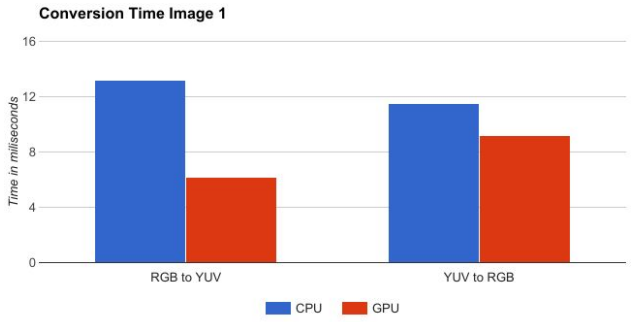
VI. Threads and Blocks

Threads and blocks played a vital role for the performance of the program. In addition to that while working with small images

`index = blockIdx * blockDim.x + threadIdx.x`

was enough to supply index number for each pixel but as soon as we switch to large image we have to consider `blockIdy for more space`

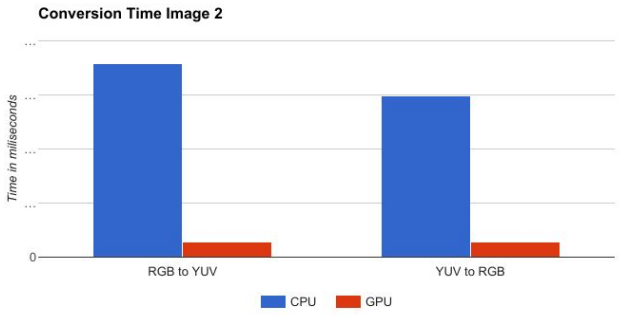
```
nt blockId=blockIdx.y * gridDim.x + blockIdx.x
int index = blockId * blockDim.x + threadIdx.x
```



VII. Results

Varying thread and blocks we got different results for conversion performance. we varied GPU threads per blocks from 2 to 1024 for 1000, 2000, 3000 and 4000 blocks. The range of thread was chosen because "deviceQuery" command shows the maximum number of threads per block supported by this device is 1024.

The table below is the performance results for small and large image conversion.

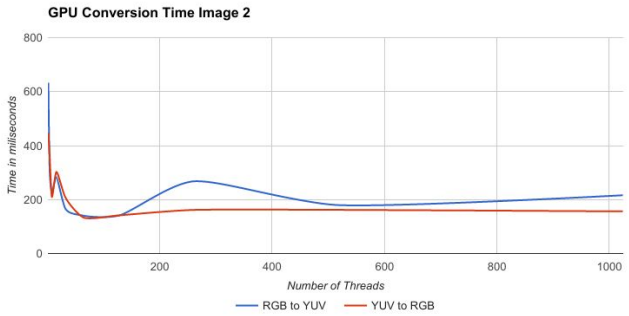


Best Conversion Times CPU and GPU

Conversion	CPU	GPU
RGB to YUV-Image 1	13.2 ms	6.15 ms
YUV to RGB-Image 1	11.5 ms	9.158 ms
RGB to YUV-Image 2	1790.13 ms	138.1 ms
YUV to RGB-Image 2	1493.44 ms	136.514 ms

By comparing Image 1 and Image 2 in the charts below it can be seen GPU is more effective when large conversion object is considered.

Now, if we vary no of thread in a fixed block performance gain is achieved until thread 32 after that it's almost same.



Performance gain after increasing block size above 2000 is same Figure 1 shows it.

VII. Conclusion

- The GPU conversion times are faster than CPU processing times specially for big images.
- The GPU I/O overhead was huge we believe that for some small images it can make the GPU performance worse than CPU.
- Choosing the correct number of GPU threads affected the results and as per our experiments generally more threads leads to a better performance.

References

[1] Silberschatz, Galvin, Gagne. *Operating System Concepts Essentials.*
[2] Andrew S. Tanenbaum. Herbert BOS. *Modern Operating Systems.*
[3] https://www.open-mpi.org/projects/hwloc
[4] https://en.wikipedia.org/wiki/Color_image_pipeline