Image format conversion with CUDA and General Purpose GPU Programming

Shawn Anderson
Anna Mkrtchyan

I. INTRODUCTION

The purpose of this report is to compare CPU and GPU implementations of image processing tasks, particularly, format conversion. In this report, we outline the details of our parallel implementation of image colour space conversions using the CUDA interface. We also perform experiments on performance, and provide insight into some subtleties that arise in practice.

II. BACKGROUND

A. Graphics Processing Units

Graphics Processing Units(GPU) allow for massive parallelization compared to CPUs. Leveraging such parallelization has been shown to improve completion time of some computational tasks by orders of magnitude. CPU's typically contain a number of cores in the range of two to eight. The experiments in this report where completed on an Nvidia GTX 1050ti, which has 768 cores. The most idiomatic use case for the parallelization provided by GPUs is graphical processing, essentially, image manipulation.

B. CUDA

In addition to manufacturing GPUs, Nvidia supplies a proprietary application program interface (API) called CUDA, which allows programmers to transfer data to, and execute code on the GPU device. CUDA integrates with the C programming language, Nvidia also provides a compiler called NVCC, which will compile C code containing calls to the CUDA API.

CUDA utilizes the concepts of blocks and threads to organize its parallel structure. The programmer specifies the number of parallel code executions by setting the number of blocks and threads. For example, a CUDA program with N blocks, and one thread per block will run N parallel executions; a CUDA program with one block and N threads per block will execute N parallel executions; a CUDA program with N blocks and M threads per block will execute N*M parallel executions. A block can have maximum 1024 threads. A process can have $2^{31} - 1$ blocks[1].

As with any system that aims to parallelize or distribute computation, there is an overhead cost that must be paid. In the case of CUDA, the overhead can be observed in the segregation of CPU memory and GPU memory. Any data that is to be processed by the GPU must be transfered over the PCI-E bus from CPU memory to GPU memory. We investigate the extent of this overhead in our experiments.

Simon Fraser University

C. Colour Space Conversion

RGB is a raw pixel image format in which each pixel is represented by a three-tuple of values, one for each of the primary colours red, green, and blue. Pixel values range from 0 to 255. YUV is an alternative colour space encoding which is more robust to transmission errors or compression artifacts, at least in terms of human perception. The conversion from RGB to YUV, and back, involves a simple formula being applied to the image pixel-wise. Since the same computation is applied to each pixel, the task is embarrassingly parallelizable.

III. EVALUATION

All experiments are ran over two versions of the same image. Firstly, a low resolution version, with dimensions 1000 x 700, occupying roughly 2 MB. Secondly, a high resolution version, with dimensions 10000 x 7000, occupying roughly 210 MB.

A. Experiment 1: Host to Device Data Transfer

The CUDA paradigm requires transferring data from host(CPU) memory to device(GPU) memory. The first experiment that we perform is measuring how long it takes to transfer the image file over the PCI-E bus. We first copy small and large images (see Sec. III) to the device and measure transfer time of 0.53ms and 20.98ms for the small and large images respectively. We then copy images to the device and copy it back from device to host and measure the transfer times of 0.928ms and 49.12ms. Comparison of these two measurements reveals that it takes roughly the same time to copy the image from the host to the device as it takes to copy the image back from the device to the host. It is important to mentions that these experiments were performed after the initial launch of the empty kernel. We also experiment with copying images to the device without launching an initial kernel first and notice that times increase up to 106.71ms for the small image and 115.81ms for the large image. This is due to the fact that the device code should be first compiled into PTX, which happens just-in-time (JIT) manner [8]

B. Experiment 2: Various Threads and GPU Speedup

We next experiment with the various thread/block configurations for the image processing with cuda. As described in Section II-C, we consider two types of image processing tasks: RGB to YUV and YUV to RBG conversions. Our benchmark is the CPU processing times for small and large images. Specifically, we measure 31.71ms for RBG to YUV and 11.16ms for YUV to RGB for the small image. As for the large image, we measure 1610.69ms for RGB to YUV and 1015.02ms YUV to RGB conversions.

How should we choose number of blocks/threads? While designing our program, we follow the best practices and specify number of threads. Number of blocks is then estimated based on the dimensions of the problem and to our specified number of threads per block to ensure that the entire image is scanned and processed (best practices were discussed during the lab and in [6]). Motivated by the thread scheduling hardware design, we vary the number of threads per block as the multiples of 32. We also consider smaller number of threads (specifically 1, 8 and 16). Our results are shown in Figure 1. As we can see from the Fig. 1, there is

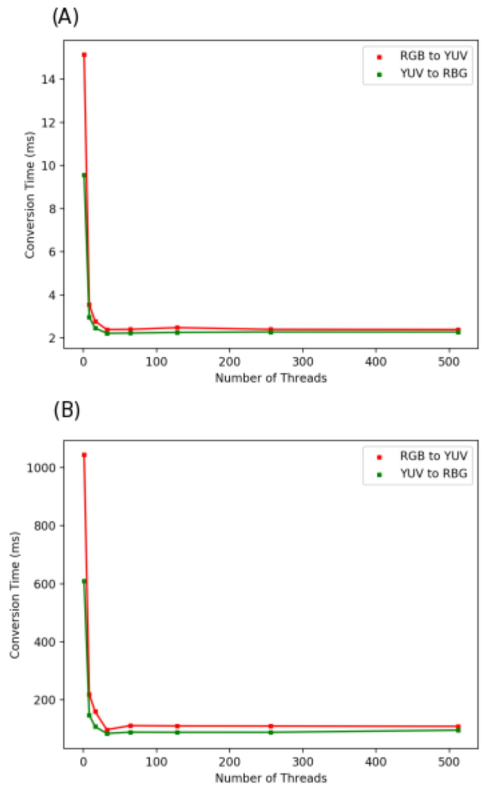

Fig. 1. Dependency of image conversion times on the number of threads for RGB2YUV and YUV2RGB conversions for (A) small image (≈ 2MB) and (B) large image (≈ 200MB). The largest conversion time for the one-thread-per-block configuration sharply drops with increasing number of threads and stabilizes at about 32 thread-per-block configuration.

a dramatic decrease in processing times as the number of threads increases, but up to a point. For both large and small images, we notice no improvements once number of threads per block reaches 32. We continue to increase the number of threads up to 1024 (maximum allowed number of threads per block) and 1025. With 1024 threads per block, images are processed normally, but once number of threads is set to 1025, we notice that images the output image is corrupted. Thus, our experiments confirm that the maximum number of threads that can be used per block is indeed 1024.

In Figure 2 we compare the optimal GPU runtime for each image conversion task with the corresponding CPU runtime.

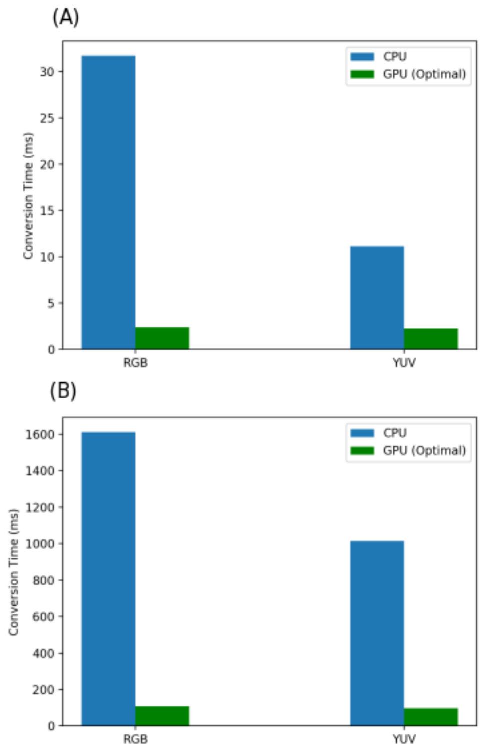

Fig. 2. Speedup for RGB to YUV and YUV to RGB conversions after switching from CPU to GPU for (A) small image (≈ 2MB) and (B) large image (≈ 200MB). Optimal GPU time (i.e. after speedup plateaus) is used for comparison.

We consider the optimal GPU runtime as the one which does not improve further with the increasing number of the threads per block. For both large and small images the speedup is incredible, between 5- to 10-fold (speedup more prominent for RGB to YUV conversion than YUV to RGB conversion).

C. Error Detection: Differences Between CPU and GPU Floating Point Algebra

There is a subtle difference between images processed by CPU and GPU. Specifically, during the RGB to YUV conversion, we notice that GPU-processed images have some of their pixel values differ by 1 from the corresponding CPU-processed images. This is due to the difference in how different devices, such as CPU and GPU, handle the rounding of the floating points [7]. Generally, such level of discrepancy between two outputs is expected and acceptable. We keep track of the maximum pixel difference for the RGB to YUV conversion and found that the difference is of a 1 pixel at most. While not a problem for a single conversion, this difference propagates further during the next stage of the calculations (YUV to RGB conversion). Indeed, we notice the maximum difference in pixels between two images after YUV to RGB conversion being as high as 4 pixels (still small value).

IV. CONCLUSIONS

In this work we experimented with the "Hello Cuda World!" program, the conversion of image from one format to another, which is a problems that can be highly parallelized. We captured several important details about working with cuda which summarize below. Firstly, we saw the implications of the just-in-time compile on the first kernel runtime. Thus, for reliable time measurements, one needs to launch an empty kernel first. Secondly, we noticed that copying image from the host to device and copying image from the device to host takes about the same time. Next, we notice that even without optimizing GPU processing parameters (number of blocks/threads) we get a noticeable speedup over CPU processing times. We also notice that as we increase number of threads per block, we get a significant improvement in the GPU performance. This is due to the fact that threads share resources (memory etc), which can increase the performance. Once we increase number of threads to 32, we do not see any further improvements. We believe this depends on the problem size, and the fact that kernels request threads in chunks of 32 (warps), so the optimal load is reached once the number of threads reach 32 per block. Finally, we notice the discrepancies between how CPU and GPU handle floating points. For our task, we noticed difference up to a pixel which had to do with the way floats are rounded up/down to integers. While not essential for this particular problem, this is something that one needs to keep in mind.

REFERENCES

[1] CUDA Wikipedia
[2] Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication
[3] Introduction to GPU Computing and CUDA Programming: A Case Study on FOlD
[4] From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming
[5] NVIDIA CUDA Programming Guide
[6] https://devblogs.nvidia.com/easy-introduction-cuda-c-and-c/
[7] Precision & Performance:Floating Point and IEEE 754 Compliance for NVIDIA GPUs https://developer.nvidia.com/sites/default/files/akamai/cuda/files/NVIDIA-CUDA-Floating-Point.pdf
[8] http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#just-in-time-compilation