Project Report: TorchISP - A Vectorized GPU Accelerated ISP in PyTorch

Student: Kaustubh Sadekar
Instructor: Prof. Feng Liu

Abstract

This report details the implementation of "TorchISP," a vectorized, GPU-accelerated Image Signal Processor (ISP) pipeline built using PyTorch. It describes each critical stage of the ISP, from raw sensor data acquisition to the final rendered image, emphasizing the underlying mathematical principles and explaining how the results from each stage are compared against a `rawpy` baseline. The project showcases the efficiency gained by leveraging GPU acceleration for image processing tasks.

1 Introduction

Digital cameras and imaging systems capture light as raw sensor data, which is fundamentally different from the visually appealing images we perceive. An Image Signal Processor (ISP) is a crucial component in this transformation pipeline, converting the raw, unprocessed data from an image sensor into a high-quality, viewable image. This involves a complex series of algorithmic steps, each addressing specific characteristics of the sensor data and human visual perception [2]. Modern ISPs are integral to achieving desirable image attributes such as correct colors, appropriate brightness and contrast, reduced noise, and sharpened details.

Implementing a complete ISP pipeline from scratch presents significant computational challenges. Many ISP stages involve pixel-wise operations, convolutions, and transformations that can be computationally intensive, especially for high-resolution images or real-time applications. Traditional CPU-bound implementations can become bottlenecks, limiting processing speed and frame rates. This is where the power of Graphics Processing Units (GPUs) becomes invaluable. GPUs, with their highly parallel architectures, are exceptionally well-suited for vectorized operations and tensor computations, making them ideal accelerators for image processing tasks.

This project, "TorchISP," focuses on implementing a vectorized, GPU-accelerated ISP pipeline using PyTorch. PyTorch, a popular open-source machine learning framework, provides robust tensor operations and seamless GPU integration, enabling efficient and scalable implementations of complex image processing algorithms. The goal is to demonstrate the various stages of an ISP, from raw sensor data acquisition to final image rendering, while leveraging the computational power of GPUs for efficient processing. Each implemented stage is meticulously designed and validated by comparing its outputs against those generated by the `rawpy` library, which serves as a widely recognized baseline for raw image processing. This systematic approach not only verifies the correctness of our implementation but also highlights the performance benefits achieved through GPU acceleration.

2 ISP Stages Implementation

The ISP pipeline in TorchISP comprises several critical stages, each contributing to the transformation of raw sensor data into a high-quality, viewable image.

2.1 Extracting Color Filter Array (CFA) and Raw RGB

Raw Bayer-mosaiced sensor data provides only one color component per pixel, arranged in a specific Color Filter Array (CFA) pattern. To convert this sparse color information into a full-color image, a process called demosaicing is performed.

This project extracts CFA pattern information using `rawpy.raw_pattern()` and `rawpy.color_desc()` methods. For the DNG file from a Pixel 7, the pattern is `[[3, 2], [0, 1]]` and the `color_desc` is `b'RGBG'`. This indicates a GBRG arrangement of green, blue, red, and green filters across the sensor.

This information is then used by the `make_raw_gbrg` function to separate the raw sensor data into distinct color planes (G1, B, R,

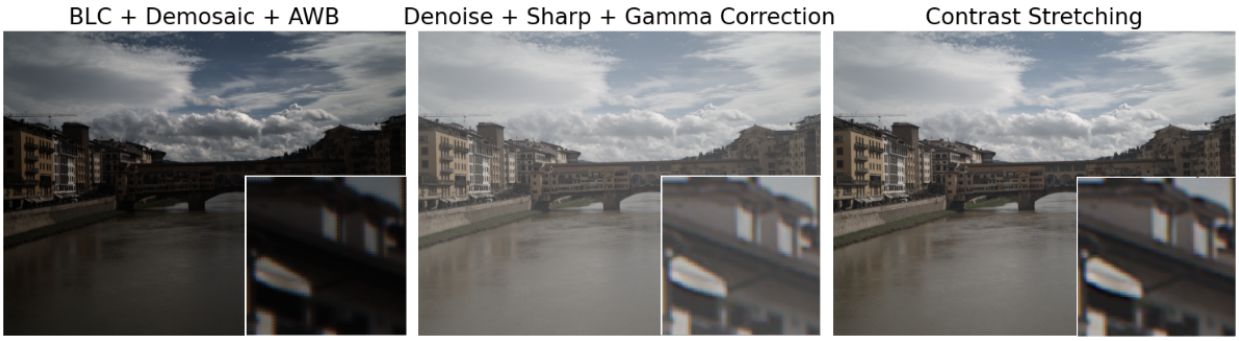

Figure 1: Major stages of TorchISP. The left image is the output after black level compensation, demosaicing, and auto white balance, the middle image is the output after applying denoising, sharpening, and gamma correction, and finally, the image on the right is the output after contrast stretching. Observe how denoising and sharpening improve the quality of edges and reduce the chromatic aberration and noise around edges.

G2). Subsequently, bilinear interpolation is applied to each color plane to estimate missing color values at every pixel location. For the green channel, values from both G1 and G2 planes are combined for a more accurate reconstruction. This process converts the single-channel Bayer data into a complete Raw_RGB image, enabling full-color representation for subsequent processing.

A comparison between the `rawpy` output and the TorchISP output for this stage shows a mean error (MAE) of approximately -0.0000, indicating a very close resemblance between the generated raw RGB images. The zoomed-in views further confirm the pixel-level similarity.

2.2 Black Level Compensation

Black-level compensation is a crucial step that corrects each raw pixel by subtracting the sensor's inherent dark offset. This ensures that a zero reading truly signifies no light. Without this compensation, subsequent ISP stages, such as white balance and tone mapping, would operate on biased data. This would lead to lifted shadow detail, reduced dynamic range, and a broken linear relationship between light intensity and pixel values.

The `blacl_level_comp` function implements this stage by subtracting the black level values, obtained per channel (R, G, B, G), from the corresponding pixels in the raw sensor data. Any resulting negative pixel values are clamped to zero.

The comparison with `rawpy` after black level compensation also demonstrates a mean error of approximately 0.0000, validating the accuracy of the implemented black level compensation.

2.3 Demosaicing Using CFA Information

Demosaicing is the process of converting raw sensor data, which contains only one color per pixel arranged in a Bayer CFA pattern (e.g., RGGB), into a full RGB image by interpolating the two missing color channels at each pixel.

The `mosaic_to_gbrg_stack` function is used to create a 4D stack of images with dimensions $H/2$, $W/2$, where H and W are the dimensions of the raw sensor image. This 4D data structure is then leveraged by the `demosaic_gbrg` function, which utilizes PyTorch's `F.interpolate` method to perform bilinear upsampling for interpolating the missing color values. For the GBRG CFA, the process involves extracting G1, B, R, and G2 channels, upsampling them, and then combining them to form the full RGB image. Specifically, for the green channel, a weighted average of G1 and G2 is used at appropriate pixel locations to reconstruct a more accurate green plane.

Initially, after demosaicing, the image exhibits a green color tint. This necessitates the next stage, automatic white balancing.

2.4 Auto White Balance

The demosaiced image often presents with a color cast, typically a green tint in this case. Simple Auto White Balance (AWB) algorithms, such as Gray-World, assume that the average scene reflectance is neutral, meaning the average red, green, and blue values should be equal ($R_{avg} = G_{avg} = B_{avg}$).

Given that the Bayer pattern has twice as many green readings, the average green level is computed from both green pixels. Then, the red and blue gains

Figure 2: Effect of performing auto white balance on the image. The left image shows the demosaicing results without auto white balance, and the right image shows the demosaicing results with auto white balance.

are derived using the formulas:

$$R_{gain} = G_{avg}/R_{avg}$$

$$B_{gain} = G_{avg}/B_{avg}$$

Typically, the green gain is fixed at 1.0, and the red and blue channels are scaled accordingly to align their average values with the green baseline. This is implemented in the `gbrg_stack_awb` function.

A comparison with `rawpy` after AWB reveals a small non-zero mean error (e.g., -0.0110). Debugging suggests that `rawpy` applies some smoothing, which accounts for the slight difference observed in zoomed-in crops and the scaled difference image.

2.5 Denoising Stage

The denoising stage in the TorchISP pipeline employs a simple Gaussian low-pass filter to attenuate high-frequency sensor noise.

Mathematically, for an input image $I \in \mathbb{R}^{H\times W\times 3}$, a 2D Gaussian kernel $G(i,j)$ of size $k\times k$ and standard deviation σ is constructed:

$$G(i,j) = \frac{1}{2\pi\sigma^2}\exp\left(-\frac{i^2+j^2}{2\sigma^2}\right)$$

where $i,j \in \left[-\frac{k-1}{2},\ldots,\frac{k-1}{2}\right]$. The kernel is normalized such that $\sum_{i,j} G(i,j) = 1$.

The denoised image $\tilde{I}(x,y)$ is obtained by per-channel convolution:

$$\tilde{I}(x,y) = \sum_{i=-a}^{a}\sum_{j=-a}^{a} G(i,j)I(x-i,y-j)$$

where $a = (k-1)/2$. Symmetric ("same") padding is used to maintain the image dimensions.

The standard deviation σ controls the smoothing strength, with larger values leading to stronger smoothing. The kernel size k determines the spatial support of the filter. This Gaussian averaging effectively reduces high-frequency noise while preserving larger-scale image structures. The `gaussian_kernel` function generates the kernel, and `denoise_rgb` applies the convolution.

A visual comparison using `plot_denoise_comparison_with_zoom` allows for an assessment of the denoising effect, showing the original and denoised images side-by-side with zoomed-in crops.

2.6 Image Sharpening

The image sharpening stage in TorchISP utilizes an "unsharp mask" technique. This method enhances high-frequency details by subtracting a blurred version of the image from the original and then recombining them.

The process involves the following steps:

1. **Gaussian Blur**: First, a blurred version of the input image I is computed, denoted as $B(x,y)$. This is achieved by convolving I with a Gaussian kernel G, as described in the denoising section.
2. **High-frequency Component**: The high-frequency details $H(x,y)$ are extracted by subtracting the blurred image from the original: $H(x,y) = I(x,y) - B(x,y)$.
3. **Unsharp Combination**: The sharpened image $S(x,y)$ is then generated by combining the original image and the high-frequency component using a blending factor $\alpha \in [0,1]$: $S(x,y) = (1-\alpha)I(x,y) + \alpha H(x,y)$. Here, $\alpha = 0$ leaves the image unchanged, while $\alpha = 1$ yields a pure high-pass filtered image.
4. **Normalization**: Finally, the resulting sharpened image is rescaled to span the range $[0,1]$ using min-max normalization.

The `sharpen_rgb` function implements this process. The `plot_three_images` utility is used to visually compare the original, denoised, and sharpened images, allowing for an assessment of the sharpening effect.

2.7 Gamma Correction and Tone Mapping

This stage is implemented based on the approach described in "DynamicISP: Dynamically Controlled Image Signal Processor for Image Recognition" [1]. It addresses the non-linear perception of brightness by human vision and the conversion from linear sensor data to display-ready formats.

Given an input image $x : \Omega \to [0, 1]$, we first compute a basic power-law correction:

$$x_1 = x^{1/p_{\gamma 1}}, \quad p_{\gamma 1} > 0$$

where $p_{\gamma 1}$ controls the midtone contrast. Following this, a soft highlight roll-off is applied:

$$\text{out}(x) \;=\; x_1 \;\times\; \frac{1 - (1 - p_{\gamma 2})\, x_1}{1 - (1 - p_{\gamma 2})\, p_k^{1/p_{\gamma 1}}},$$

where $p_{\gamma 2} \in [0, 1]$, $p_k \in (0, 1)$, $p_{\gamma 2}$ sets the knee strength (where 0 implies no roll-off and 1 implies maximum roll-off), and p_k is the pivot level where highlights begin to compress. A small epsilon is internally added to the denominator for numerical stability. This mapping effectively boosts midtones with the power-law term while gently compressing highlights, ensuring the output values remain within the $[0, 1]$ range.

The `gamma_tone_map` function implements this piecewise gamma curve, utilizing the parameters `p_gamma1`, `p_gamma2`, and `p_k` to control the tonal response and achieve a desired visual appearance.

2.8 Contrast Stretching Stage: Linear Gain and Clipping

In our ISP pipeline, contrast stretching is implemented as a simple linear transform followed by non-negative clipping. This stage, implemented by the `contrast_stretch` function, aims to improve image contrast by expanding the range of intensity values.

Given an input image $I : \Omega \to \mathbb{R}$, we first apply a gain and offset:

$$S(x, y) \;=\; a\, I(x, y) \;+\; b,$$

where $a > 0$ controls the contrast (slope) and b adjusts the brightness. To prevent negative intensities, we then apply a ReLU (rectified linear unit) operation:

$$\tilde{S}(x, y) \;=\; \max\big(0,\; S(x, y)\big).$$

This ensures all output values are in $[0, \infty)$. If a bounded range (e.g., $[0, 1]$) is required, a further upper-clamp or normalization step can follow this operation.

3 Speed-up Comparison

The performance of TorchISP was evaluated against `rawpy.postprocess()`. The execution times for each method are summarized below, along with the calculated speed-up factors relative to `rawpy`.

Table 1: Execution Time and Speed-up Comparison

Method	**Time (s)**	**Speed-up**
`rawpy.postprocess()`	1.14	1.00×
Torch-ISP on CPU	5.77	0.20×
Torch-ISP on GPU	0.049	23.27×

As shown in Table 1, Torch-ISP running on the GPU achieves a significant speed-up of approximately 23.27 times compared to `rawpy.postprocess()`. While Torch-ISP on the CPU is slower than `rawpy`, the GPU acceleration demonstrates the clear advantages of vectorized processing in PyTorch for image signal processing tasks.

4 Conclusion

The TorchISP project successfully implements key stages of an Image Signal Processor pipeline in PyTorch, leveraging GPU acceleration for efficiency. Each stage, including CFA extraction and Raw RGB generation, black level compensation, demosaicing, auto white balance, denoising, sharpening, gamma correction/tone mapping, and contrast stretching, has been carefully developed and validated. The comparisons with `rawpy` demonstrate the accuracy and effectiveness of the implemented algorithms, with minor differences attributable to varying internal implementations (e.g., smoothing in `rawpy`). This project serves as a robust foundation for further exploration and development of advanced ISP techniques in PyTorch.

References

[1] Yoshimura, Masakazu, et al. "Dynamicisp: dynamically controlled image signal processor for image recognition." Proceedings of the IEEE/CVF International Conference on Computer Vision. 2023.

[2] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*. 3rd ed. Prentice Hall, 2008.

[3] Eklund, Anders, et al. "Medical image processing on the GPU–Past, present and future." Medical image analysis 17.8 (2013): 1073-1094.