Burst photography for high dynamic range and low-light imaging on mobile cameras

C.S. Siksma (0902066), R.J.A. Surtel (0942699), R. Valkov (1537857)

Fig. 1. Reference frame (left), our result (middle), original HDR+ result (right). [2]

Abstract— Mobile phone photographic hardware has seen significant progress over the last decade. With each new generation, mobile phone cameras have been constantly improved upon, with either upgraded software or a more complex set of hardware. However, due to their limited size, they are still lacking when compared to consumer grade DSLR/mirrorless cameras. There are a few main disadvantages of mobile phone cameras. Their inability to shoot at slow shutter speeds, their fixed apertures and small sensors, which restrict the amount of light that the camera can capture. This leads to darker photos and noisier photos, especially in situations where the light is not optimal, which is exactly the problem we aim to tackle with our HDR+ implementation.

Index Terms—HDR, burst photography, low-light imaging

1 Introduction

There are multiple ways to increase image brightness when taking photos in an environment with low light, for example, increasing ISO value. However, ISO value has no effect on actual exposure, instead, it essentially brightens a photo that is already captured. While this may work in some cases, an increase in the ISO value might cause undesired side effects like an increase in noise/grain and a decrease in dynamic range. Preferably, alternative methods that affect the exposure itself are used to capture more light. Methods like increasing the exposure time, using a bigger aperture and/or using flash are proven to work but have significant drawbacks. These drawbacks include affecting the depth of field and lowering the shutter speed which are simply not desirable or achievable given the limitations of mobile phone cameras and can work against the photographer's true intentions. For this reason, the HDR+ (High Dynamic Range) method was conceived. The HDR+ method, as described by Samuel W. Hasinoff et al. [2] as a Google Research project, proposes the application of computational photography to a sequential burst of photos taken with a single press of the shutter. Each image in the burst is taken with a constant exposure time and aperture setting, the frames are then aligned and merged to produce a resulting image that surpasses the light gathering abilities of the camera given just a single frame. HDR+ also optimizes the image by reducing noise, improving colors, contrast, sharpness and removing undesired artifacts.

In this paper we will first address the HDR+ method itself in Section 2, explaining all the necessary steps and their implementations. Then, in Section 3 we will discuss the implementation details of the application and showcase our cross-platform solution that exploits our HDR+ implementation. In Section 3.1 we briefly discuss details behind the optimization of our application. In Section 4 we will reflect upon our results and compare those with the results of the original implementation. In Section 4.1 we will discuss the performance of our application. In Section 5 we conclude our paper by summarizing the results and discussing some difficulties we faced during the project. Finally, in Section 6 is the link to the demo of our application.

2 The HDR+ method

2.1 Overview

For our implementation, we take a similar approach to the one pointed out by Hasinoff et al. in the original HDR+ paper [2]. Since the original version of the HDR+ software is proprietary and the paper covers very little implementation details, we also reference an open source implementation of the project by Timothy Brooks and Suhaas Reddy [1], which is written in C++. We have chosen to use Python in our project, along with the embedded Halide programming language, which was created specifically for image processing and was used in the original implementation as mentioned in the paper [2]. Therefore, most of the code written for the HDR+ pipeline is Halide code embedded

• *Eindhoven University of Technology*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

in Python, using the Python bindings available for Halide. The steps we used follow the original pipeline from the HDR+ paper, consisting of: Align, Merge and Finish. First, an alignment is computed over a burst of raw 10 bit images. Secondly, this alignment is used to merge the frames into one raw 14 bit image. Finally, the image is processed using the finishing pipeline and saved as an 8 bit image. An overview of the HDR+ steps can be seen in Figure 2 and a complete overview of the HDR+ processing pipeline as implemented by us can be seen in Figure 3.

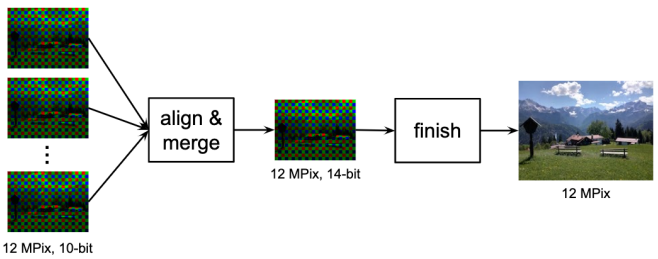

Fig. 2. Overview of the main HDR+ steps.

2.2 Align

In the alignment phase, we mostly follow the approach given by the original HDR+ paper. However, we use the first frame of the input burst as a reference frame. Furthermore, we minimize L1 residuals (least absolute deviations) between the reference frame and alternate frames, whereas the original implementation used a mix of L1 and L2 residuals (least squared deviations). Moreover, the original implementation also used a sub-pixel alignment method, allowing for alignment at a finer granularity. Apart from these details, our alignment method follows the same general structure. The alternate frames are hierarchically aligned to the reference frame using a coarse to fine alignment on a Gaussian pyramid of four levels. Each level is downsampled by a factor of four, and also converted to grayscale. We produce an alignment on 16 x 16 tiles for each level of the pyramid, with a search region of 4 pixels, using the alignments from the coarser scale as an initial guess. This alignment method reduces blur and ghosting, however, as will be explained in Section 4, it is not quite perfect, and our implementation does not match the original implementation in terms of performance. Simplified pseudocode for the align function can be found in Algorithm 1.

2.3 Merge

Merging is the second step of the HDR+ pipeline that takes the previously computed alignment to merge the frames into one. It yields overall noise reduction by combining information from multiple aligned frames. Merging also contributes to smoothing out certain imperfections, like improving colour accuracy and reducing unwanted motion blur. We attempt to achieve this through combining the aligned tiles by weighting different frames based on the similarity between the reference tile and alternative tiles, again minimizing L1 distances. Distances greater than some threshold are discarded, since the time between frames is at most 33 ms, extreme distances are likely caused by alignment failures. Simplified pseudocode for this merge function can be found in Algorithm 2. In the spatial domain, we use spatial denoising while attempting to smoothly blend overlapping tiles. Here we follow the approach of the original HDR+ implementation, using a modified raised cosine window to blend the overlapping tiles.

2.4 Finish

The finishing step of the pipeline consists of a series of techniques that aim to improve the overall quality of the merged image, this follows a similar trend to how raw files are typically processed into jpg/png files. We follow the methodology of the original HDR+ paper where possible, however, we did not implement the following techniques described in the original paper due to time constraints: lens shading

Algorithm 1: Align

```
input : frames
/* Downsample all frames and convert to grayscale */
frames = downsample_to_grayscale(frames)

/* Choose the first frame as reference */
ref_frame = frames[0]
frames = frames[1:]

/* Initialize 4-level gaussian pyramid */
ref_gauss_pyramid = pyramid(ref_frame)

alignments = []
for frame in frames do
    gauss_pyramid = pyramid(frame)

    /* Loop from coarse to fine */
    for ref_level, level in ref_gauss_pyramid, gauss_pyramid do
        level_alignment = []

        /* Search region is a [-4, 8], [-4, 8] region
           surrounding the tile.  For the function D(u,
           v), see the original HDR+ paper.  */
        tile_alignment = (u,v) such that D(u,v) =
          min([D(u,v) for (u,v) in search_region])

        level_alignment.append(tile_alignment)

        if level is the finest level then
            alignments.append(level_alignment)
        end
    end
end

return alignments
```

Algorithm 2: Merge

```
input : frames, alignments
/* Choose the first frame as reference */
ref_frame = frames[0]
frames = frames[1:]

image = ref_frame
total_weight = 1

for frame in frames do
    /* d(frame1, frame2) calculates the L1 distance
       between aligned frames */
    dist = d(ref_frame, alignment[frame])
    weight = 1/dist

    image += frame · weight
    total_weight += weight
end
image /= total_weight
return image
```

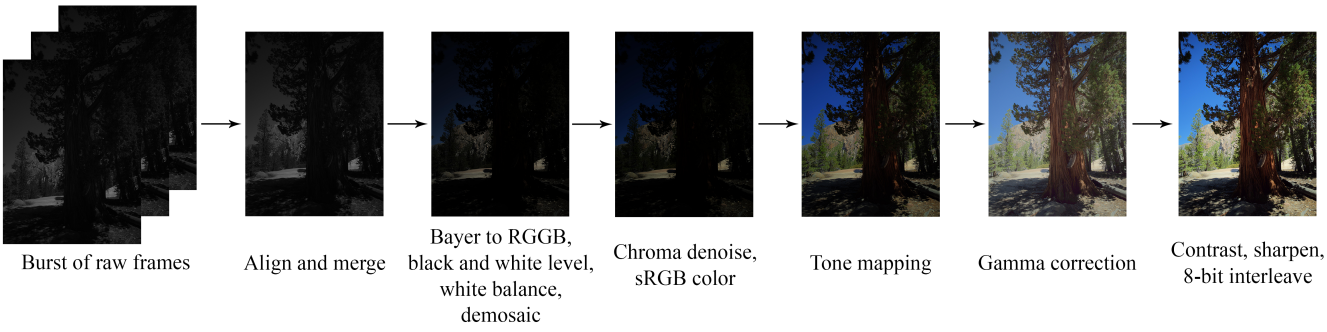

Fig. 3. Our HDR+ pipeline and its post-processing steps.

correction, dehazing, chromatic aberration correction, hue-specific color adjustments, and dithering. Our finish pipeline consists of the following steps and can also be seen with an example in Figure 3.

- Shift Bayer to RGGB: shifting the Bayer image channels to RGGB channels using the CFA pattern.
- Black-level subtraction: offset deduction from all pixels, which leads to pixels that receive no light becoming zero.
- White balancing: scaling the RGGB channels with the aim to map grays in the scene to grays in the image.
- Demosaicing: conversion from Bayer raw to a linear RGB with a full resolution.
- Chroma denoising: reduction of red and green splotches, particularly in low lit images that have dark areas
- sRGB color correction: converting the color values to a linear sRGB color profile
- Tone mapping: reducing the tonal values within an image, like the contrast between highlights and shadows while preserving local contrast and making it more suitable to be viewed on a digital screen.
- Gamma correction: applying a gamma correction to the overall image.
- Contrast adjustment: increasing the overall contrast.
- Sharpening: increasing the overall image sharpness by using the difference of Gaussian unsharp masking.
- 8bit interleaving: converting the image to 8 bit and interleaving the color channels.

Combining these processing steps into the finishing function results in Algorithm 3. The algorithms of the individual processing steps have not been included as separate pseudo-code algorithms due to their size and complexity.

3 Implementation Details

Now that the HDR+ method and its implementation have been discussed, we will consider the application that we created for it. Originally, the program is supposed to run on mobile devices, embedded into the camera app of the Pixel 2 and 3 phones. Rather than going down the same path and creating a CLI (Command Line Interface) application, we created a desktop application with a GUI (Graphical User Interface) that is cross platform and simple to use. This allows users to leverage the HDR+ method for any kind of raw image bursts, be it from mobile devices or DSLRs.

Algorithm 3: Finish

input : $image, w, h, black, white, white\_balance\_val, compression, gain, contrast\_strength, cfa\_pattern, ccm$

```
/* Declare constants */
denoise_passes = 1
sharpen_strength = 2

/* Begin finishing pipeline */
bayer_shifted = shift_bayer_to_rggb(imgs, cfa_pattern)

black_white_level_output =
 black_white_level(bayer_shifted, black, white)

white_balance_output =
 white_balance(black_white_level_output, w, h, white_balance_val)


demosaic_output = demosaic(white_balance_output, w, h)

chroma_denoised_output =
 chroma_denoise(demosaic_output, w, h, denoise_passes)

srgb_output = srgb(chroma_denoised_output, ccm)

tone_map_output =
 tone_map(srgb_output, w, h, compression, gain)

gamma_correct_output = gamma_correct(tone_map_output)

contrast_output =
 contrast(gamma_correct_output, contrast_strength, black)

sharpen_output =
 sharpen(contrast_output, sharpen_strength)

/* End finishing pipeline, convert to 8 bit */
u8bit_interleave_output =
 u8bit_interleave(sharpen_output)

return u8bit_interleave_output
```

The user interface of the application was created using the Python library Kivy. The library enabled us to create a file explorer interface and to show and dynamically update images, both of which are crucial to our application. Furthermore, since Python (and Kivy) is cross-platform it allows us to run our application on different operating systems (tested on MacOS and Ubuntu).

Once the application is started, the user will see the interface as shown in Figure 4. In the center are two image placeholders, which will show the reference frame from the input burst and the output of our processing pipeline once a burst of images has been chosen and processed. On the right side of the interface there are three sliders. These sliders allow the user to choose custom values for the compression and gain, used in tone mapping, as well as the strength of the contrast that is applied. The sliders default to values 3.8, 1.1, and 1.0 respectively, which should produce good results in most cases. At the bottom of the interface there are two buttons, one labeled ”Select burst”, and another labeled ”Process”. When the ”Select burst” button is pressed, a file explorer window will pop up, where the user can choose a set of burst images by selecting one of the frames in the burst, and then clicking the ”Select” button. This file explorer interface is shown in Figure 5. When the ”Select” button is pressed, the path to the selected burst will be saved internally. The user can then press the ”Process” button. If this button is pressed before a burst is selected, an error will shown, as seen in Figure 6. This same interface is shown with different error messages for any other errors encountered during execution.

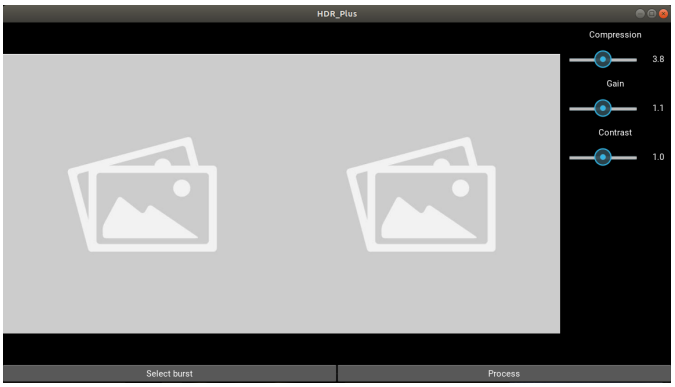

Fig. 4. The GUI of the application before a burst is chosen.

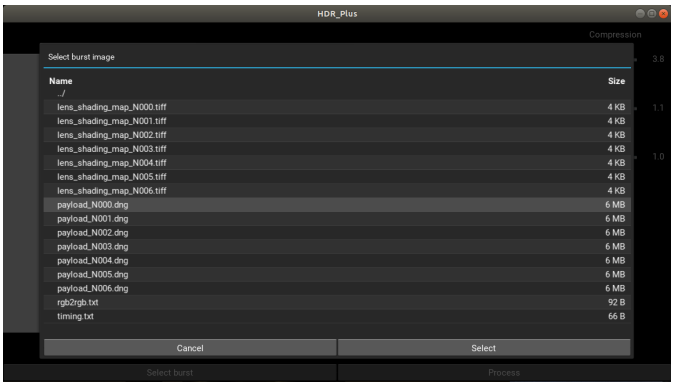

Fig. 5. The file explorer interface for burst selection.

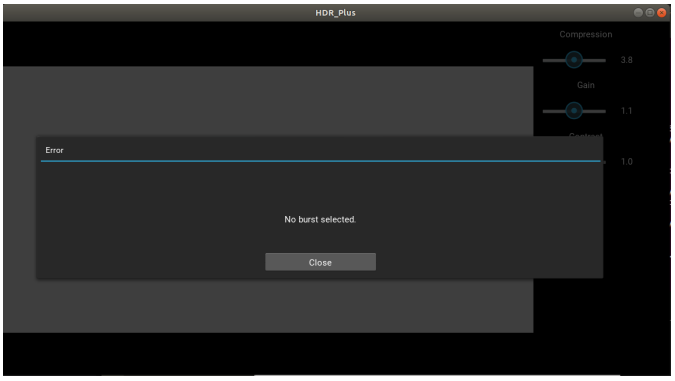

Fig. 6. The error reporting popup window. This particular error is shown when the user presses the ”Process” button before selecting a set of input images.

When the ”Process” button is pressed, all frames corresponding to the selected burst will be automatically loaded into memory using the Python library RawPy. Once the entire burst is loaded, the input frames will be passed to the HDR+ pipeline, which is explained in-depth in Section 2. While the images are being processed, a progress bar is shown so the user knows approximately how long it will take. This progress bar can be seen Figure 7. Once the input has been fully processed, the finished output image will be saved to disk and shown on the right side of the interface, next to a reference frame from the input burst, as shown in Figure 8.

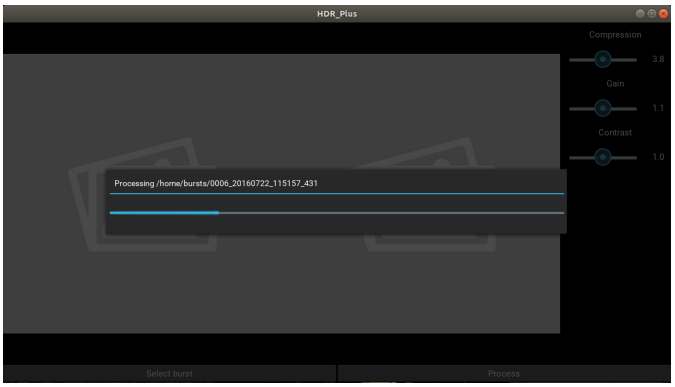

Fig. 7. The progress bar that is shown while the input images are being processed.

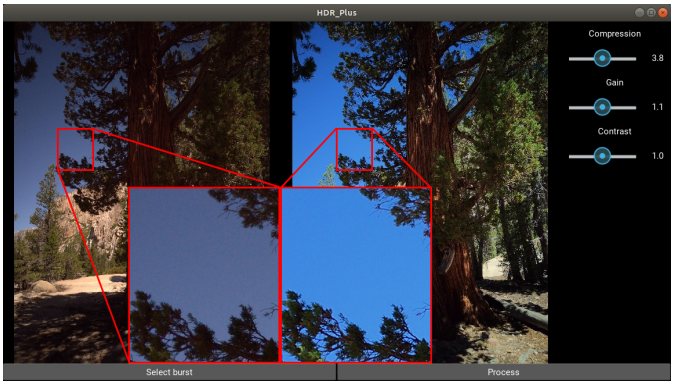

Fig. 8. The reference frame from the input burst (left), and the processed output image (right). Note the reduction in noise that can be seen in the zoomed in section of the sky.

3.1 Optimization

The Python bindings for Halide compile into well optimized C++ code by design, as Halide was specifically made for image processing. Halide allows us to split the algorithm (what is computed) and the schedule (where and when it is computed) into different statements. This makes it easy to build pipelines and leverage optimizations such as parallel processing across multiple cores and SIMD/vector operations which are typically very hard to program. Halide allows us to simply schedule functions using .PARALLEL() and .VECTORIZE() functions to have the task run on multiple cores to use SIMD/vector operations respectively. All the steps in our implementation utilize this functionality which results in a very fast processing pipeline especially well suited for mobile devices with limited processing power.

4 RESULTS

To assess our results we compare some example inputs and outputs of our application to those of the original implementation. Figure 9 shows the reference frame of a set of input frames. Figure 10 shows the corresponding output of our HDR+ implementation. And finally, Figure 11 shows the final result of the original HDR+ pipeline by the research team at Google.

Fig. 9. Reference frame of the input burst, processed by the Python library RawPy.

Fig. 10. Output image from our HDR+ pipeline.

Fig. 11. Output image from the original HDR+ pipeline. [2]

When comparing our result to the reference frame, we can see that noise in the image has been reduced significantly. As the input frames are merged, more detail has been retrieved from the scene, the image is brighter and especially in dark areas our output image is much clearer than the reference frame. Look for instance at the trees and the sky in the top left corner of the image, as shown in Figure 18. The reduction in noise can be seen especially well when looking at the bright sky. The finishing pipeline also does a great job by increasing sharpness and contrast and by leveling out the highlights and shadows. The resulting image is far more visually pleasing and much more in line with what the photographer intended to capture. The result shows us how programmatically combining multiple frames yields a result that seems to capture more light which far surpasses the natural capabilities of the camera.

Three more input and output examples can be seen in Figures 12 through 17. The flower in Figure 13 has much more vibrant colors and more details can be seen in the leaves and center of the flower. The picture of a church in Figure 15 shows recovered detail in the pavement and colors in the sky. The image in 17 shows how well noise is removed and the colors are restored by adjusting the white-balance and tone mapping.

Fig. 12. Flower: reference frame of the input burst, processed by the Python library RawPy.

Fig. 13. Flower: output image from our HDR+ pipeline.

Fig. 14. Church: reference frame of the input burst, processed by the Python library RawPy.

Fig. 15. Church: output image from our HDR+ pipeline.

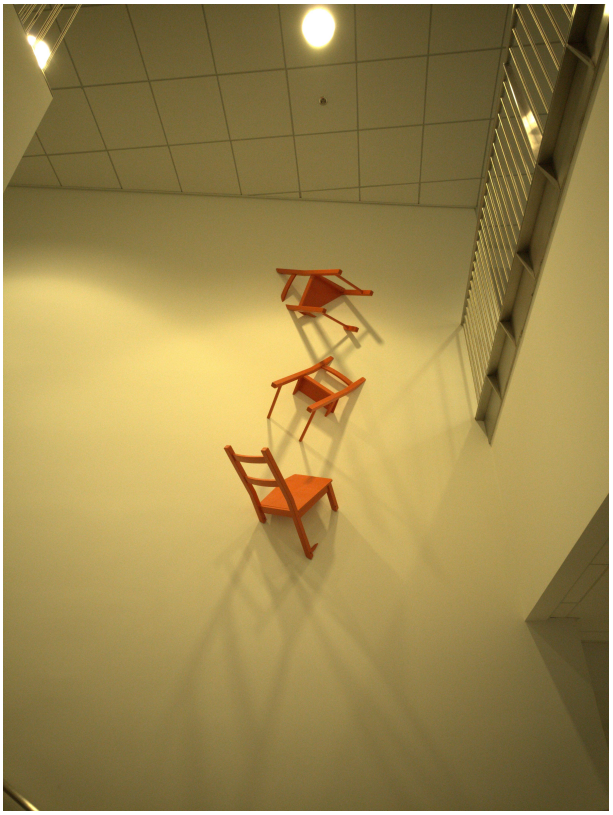

Fig. 16. Chairs: reference frame of the input burst, processed by the Python library RawPy.

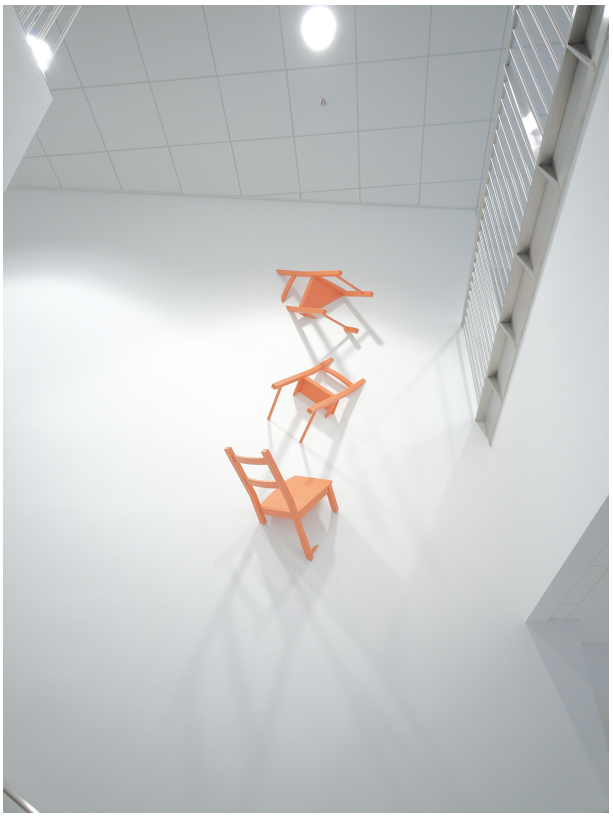

Fig. 17. Chairs: output image from our HDR+ pipeline.

In general we find that images after being processed by our HDR+ implementation are more visually pleasing than they were before. Especially in low light situations the HDR+ method has very promising results that better reflect the artistic intentions of the original photographer.

However, when comparing our result to that of Google's research team, there are some notable differences. Take for example the reference frame of Figure 9, we can see that the trees in the processed result of our implementation are more blurry compared to those of the original implementation. This is likely due to slight differentiations in the alignment and merging of the input frames compared to the original HDR+ implementation. Their result also appears have no vignetting and greater dynamic range. This is probably (partly) due to the omitted steps in the finishing pipeline and could potentially be resolved by adding these steps, as lens correction also addresses vignetting for example, but we leave this for future work.

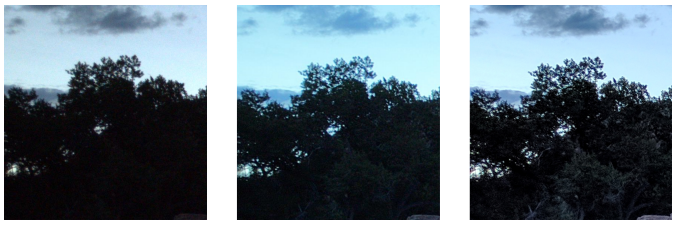

Fig. 18. Zoomed view of the top-left area of the reference frame Figure 9 (left), our output Figure 10 (middle) and output original implementation Figure 11 (right).

Another flaw in our burst processing is observed particularly well in high-motion burst sequence, such as in Figure 19. Although this issue does not occur in Google's own implementation, our pipeline mostly does not respond well to input bursts with a lot of movement between the separate burst frames. The differences between individual frames stems from both camera shake and movement in the scene. When these frames are not merged properly this results in the so called "ghosting" effect, which causes figures that were in motion during the duration of the burst to appear slightly transparent and faded. We suspect the flaw likely stems from a misalignment issue, as it also can be observed in some bursts where there less movement in between frames such as in Figure 20, although not as severe. This is not so much a problem for general use, but can be a problem where extreme detail matters, such as when photographing scenes that contain written text, or when the image contains heavy-motion such as a moving cyclist.

Fig. 19. Note the ghosting artifacts from all the movement on the busy street.

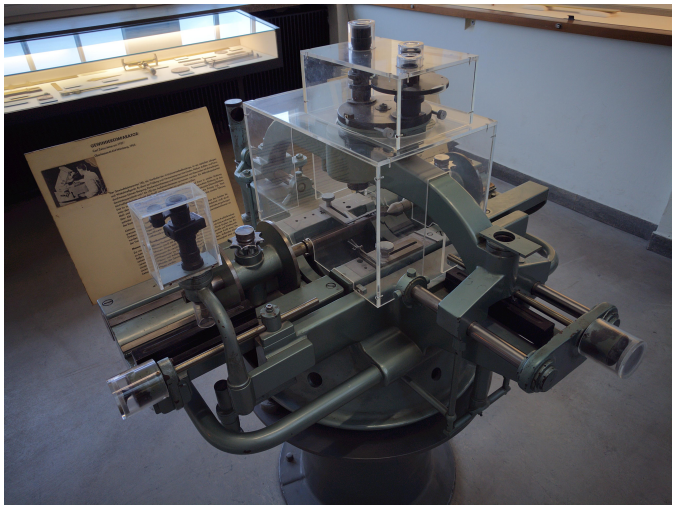

Fig. 20. Note the ghosting artifacts from misalignment on the text in the left side of the image.

4.1 Performance

As stated in section 3.1, the Python bindings for Halide compile into well optimized C++ code. Disk I/O is responsible some of the running time. The tested images have a resolution of roughly 3000 by 4000 pixels. Loading the raw files into image buffers from an SSD, there is about 350 ms overhead per frame, from an HDD this is increased to about 725 ms. On mobile devices, the images are retrieved from memory which would result in much faster loading times. Excluding the loading time, on a system with a 3,5 GHz Quad-Core 64-bit architecture our implementation of the full HDR+ pipeline takes 12723 ms. On a system with a 3,6 GHz Octa-Core 64-bit architecture our implementation takes 9277 ms, which is significantly faster. However, the original paper describes running times of up to 4 seconds on a 2 GHz Octa-Core 64-bit architecture which we were not able to achieve [2]. These differences in processing times are likely due to the huge amount of optimizations done by the team at Google, such as a custom Fast Fourier Transform, and better parallelism for a particular architecture. In general, we think about 12-13 seconds for the full HDR+ pipeline to run is definitely acceptable and quite usable given the amount of data and operations necessary.

5 Conclusion

The HDR+ method is a technique that is still utilized even in the most modern devices. The great results it yields, combined with the fact that it comes without any serious trade-offs makes it a viable tool for artificially improving the light gathering capabilities of mobile phone cameras. Our Python HDR+ implementation, albeit with minor issues, manages to deliver satisfying results. The output images of our pipeline have significantly improved characteristics overall. The original images go through the processes of alignment, merging, and finishing, coming out much better looking in almost all cases. The process was made more accessible by our cross-platform GUI, which allows for the selection and loading of image bursts into our HDR+ pipeline. All our tests were conducted with images from the original HDR+ data set. Our implementation is quite well optimized and results in a very usable running time. Sadly we will never know what the exact differences are between the original implementation and ours, because the software is proprietary. Still, we can confidently say that our HDR+ implementation yields results that greatly improve upon a single frame of the input in a timely manner, and therefore we are satisfied with our result.

6 Demo

The demo video of our application can be found here: `https://www.dropbox.com/s/rx863qx6o3t4slg/demo.mp4?dl=1`.

REFERENCES

[1] Tim Brooks and Suhaas Reddy. Hdr+ burst processing pipeline. 2017.

[2] Samuel W. Hasinoff, Dillon Sharlet, Ryan Geiss, Andrew Adams, Jonathan T. Barron, Florian Kainz, Jiawen Chen, and Marc Levoy. Burst photography for high dynamic range and low-light imaging on mobile cameras. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)*, 35(6), 2016.