CS 184/284A Final Project Report HDR+

Siming Liu
University of California, Berkeley
liusm220036@berkeley.edu

Cyrus Vachha
University of California, Berkeley
cvachha@berkeley.edu

Haohua Lyu
University of California, Berkeley
haohua@berkeley.edu

Xiao Song
University of California, Berkeley
xiaosx@berkeley.edu

CCS CONCEPTS

• **Computing methodologies → Computational photography**.

KEYWORDS

HDR+, HDR, Denoise

1 INTRODUCTION

We implemented a burst (multiple similar shots) photograph HDR pipeline for the final project based on Google's HDR+ paper [3]. The HDR+ paper proposed an image processing pipeline that combines multiple underexposure (to avoid over-saturation) raw images and generates enhanced HDR photos. Our project applies a similar HDR+ pipeline to the original dataset of HDR+ paper, with some modifications on the alignment and finishing part while maintaining the overall performance.

Smartphones have limited camera hardware and a small image sensor and lens, making computational photography essential to produce clear images. Taking photos on smartphones can result in parts of the image being over-or under-exposed given the limited range. Additionally, smartphone cameras traditionally cannot take photos in low lighting conditions and may generate noisy images. Since smartphone cameras are not able to capture the same SNR as traditional cameras, HDR+ allows for combining multiple images to generate a visually complete photo. The HDR+ algorithm allows us to create an image that has a higher amount of visual detail in more lighting conditions and remove noise at the same time. HDR images are images with a high dynamic range where the image captures a large range of luminosity from a scene.

This paper is divided into 5 Sections. In Section 1, we give a brief introduction of what is HDR+ and why we decided to choose it as our final project. In Section 2, we give an overview of the HDR+ algorithm, how we implement the HDR+ pipeline, and some of our novel implementation choice. In Section 3, we talk about the problems we encountered when working on this project and how we handle those problems. In Section 4, we show our results on multiple datasets and compare our results with the original google HDR+ paper. In Section 5, we summarize our work and provide potential future work. Our project website can be found here: https://ucberkeley-spring2022-cs284a-project.github.io/HDRplus/

2 TECHNICLE DETAILS AND IMPLEMENTATIONS

2.1 Pipeline Overview and Dependency

The HDR+ pipeline can be roughly divided into three parts: Alignment, Merging, and Finishing (including adjusting white balance, tone-mapping, etc). A full pipeline, implemented by [3], is shown in Figure 1.

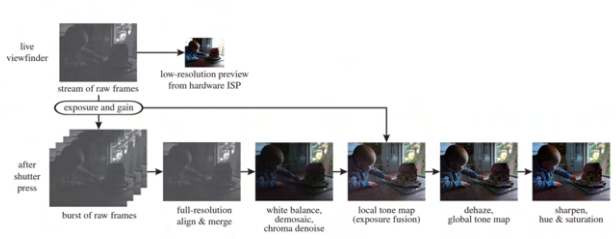


Figure 1: HDR+ pipeline. Align and Merge is represented as one stage. Finishing is represented as multiple stages. Source: [3]

The alignment part aligns each image tile between the reference image and other alternative images inside the burst. The merging part merges multiple image tiles onto the reference image, this helps to improve the SNR. The finishing part is mainly in charge of raw Bayer image processing including correction, demosaicing, etc. A more detailed description of each part and how we implemented will be mentioned below.

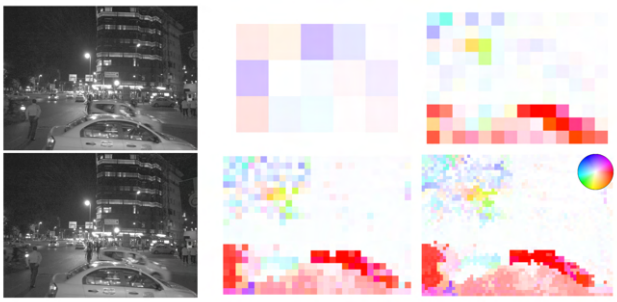
We build our project with C++ from scratch, with the help of libraries like OpenCV [1] for image processing function, LibRaw and Exiv2 for raw Bayer image processing. Our project is open-sourced and is available publicly at https://github.com/UCBerkeley-Spring2022-CS284A-Project/HDRplus

2.2 Alignment

2.2.1 Algorithm. The input of the alignment part is a burst of raw Bayer image. The output of the alignment part is a list of list of alignment pairs. Where the first list corresponds to each image in the burst. The second list corresponds to each tile in the reference image. The alignment pair contains displacement in pixel value of how tile (i,j) in the alternative image is matched with tile (i,j) in the reference image.

In the alignment part, an image pyramid of 4 levels is first created based on the input data. The image pyramid is created with a gaussian blur and box filter downsample. Tile size is chosen for each pyramid level (8x8 pixels for the coarsest level, 16x16 pixels for all other levels). The sharpest image is chosen as a reference image and all other images try to align their tiles to the reference image. In our implementation, we do not choose the sharpest image ourselves, instead, we use the sharpest image choice of the original google dataset. The alignment of tiles first starts from the coarsest level using L1/L2 loss. For each following level (of larger size), the previous level (coarser level) alignment is upsampled and used as the initial guess for alignment. Following the original paper, for each upsampled alignment at tile (i,j) , we also consider the nearest neighbor tiles in x and y dimensions (e.g. tile (i-1, j) and tile (i, j-1))

as candidate tiles and choose the upsampled alignment at tile (i,j) from the three candidate alignment based on smallest L1 distance. Image tile is choose to overlap each other tile by 50%, this ensures the merged image do not have blocking effect. Figure 2 shows the alignment at each image pyramid level where the color of each tile represents the alignment direction and magnitude.



(a) *Image pair* **(b)** *Intermediate alignment fields*

Figure 2: Alignment at each image pyramid level Source: [3]

2.2.2 Implementation. The alignment part is computation and memory intense and requires lots of optimization. Here, we briefly talk about the optimization we have done in the alignment section.

OpenMP [2] is used to parallel alignment search for each tile. We choose to use OpenMP over pthread because (1) OpenMP internally maintains a thread pool and has a lighter overhead of creating thread and (2) OpenMP's `# pragma omp parallel for` is easier to use compared with explicitly creating pthread.

When searching alignment for each reference image tile (i,j), explicit data caching is applied to the reference image tile and the alternative image's search area. Specifically, we create a `cv::Mat` that contains a copy of the reference image tile (i,j) and use this copied tile (i,j) during searching. Since the copied tile (i,j) is continues in underlying memory (whereas the tile on the original reference image does not continue for each tile row) and is placed on the cache (since we just finish the memory copy), we can reduce access to main memory by a factor of `search radius * search radius`. The same copy is also applied to the corresponding search area in the alternative image to reduce main memory access. The explicit memory caching reduce the bandwidth pressure.

Customized kernel function is used instead of the general OpenCV function for many kernel functions like nearest neighbor downsample, box filter, etc. This helps speed up the kernel size since we can tune the kernel for specific image types (i.e. `uint16_t`) and kernel size.

Template with tile size as the template parameter, `-O3` optimization flag, and `#pragma GCC unroll tile_size` is used to enable better compiler SIMD support. Having tile size (of size 8, 16) as a template parameter (i.e. `template<int tile_size>`) instead of a function argument would enable the compiler to generate SIMD function kernel since the compiler would have the tile size information when the template is instanized. Using `#pragma GCC unroll tile_size` help avoid the branching overhead inside the for loop. This kind of template trick is applied to many customize kernels including l1 distance, l2 distance, nearest neighbor downsample, box filter, etc.

2.3 Merging

2.3.1 Algorithm. The merging part of the HDR+ pipeline takes in the burst sequence and the alignment generated in the previous step to create a merged image with proper denoising. This is mainly accomplished in four steps:

1) Noise Level Estimation. We evaluate the noise level in the reference image so that we will be able to distinguish differences between different frames from base noise of the image. We follow the Google HDR+ paper [3] and assume that the noise can be evaluated with a single value for each tile; specifically, the root mean square (RMS) of the tile. Once the RMS is obtained, we evaluate the noise variance using noise parameters evaluated from the ISO, white level, and black level of the image. Mathematically, the noise variance σ^2 of tile T_0 is calculated by $\sigma^2 = \lambda_s \rho(T_0) + \lambda_r$, where $\rho(T_0)$ is the RMS of T_0, and λ_s and λ_r are constants representing the shot noise and read noise, respectively.

2) Fourier Pairwise Temporal Denoising. With the alignment calculated, we use Fourier transformation to perform temporal denoising across the burst sequence. For a reference tile, we acquire the aligned tiles on the alternate images and find their Fourier transforms in the frequency domain. We then linearly interpolate them with a shrinkage operator that is similar to a Wiener filter. Mathematically, we have $T_n(\omega) = \mathrm{DFT}(T_n)$ for $n = 0$ to N. Then, the merged tile in the frequency domain is

$$\tilde{T}_0(\omega) = \frac{1}{N} \sum_{n=0}^{N-1} (1 - A_n(\omega)) T_n(\omega) + A_n(\omega) T_0(\omega),$$

where, for a scaling factor $k = \frac{2n^2}{4^2}$ [3] and a tuning temporal factor τ,

$$A_n = \frac{|T_n(\omega) - T_0(\omega)|^2}{|T_n(\omega) - T_0(\omega)|^2 + k\tau\sigma^2}.$$

3) Spatial Denoising. Since we are in the frequency domain, it is convenient to perform post-processing and filter out higher frequency noise as well. We assume that in the previous step, all images in the burst sequence are aligned; as a result, we divide the noise variance σ^2 by N. We then apply a similar Wiener filter:

$$\hat{T}_0 = \frac{|\tilde{T}_0(\omega)|^2}{|\tilde{T}_0(\omega)|^2 + |\omega| * \frac{ks}{2N}\sigma^2},$$

where s is another tuning spacial factor. We are now finished with the frequency domain and revert the image by finding the inverse DFT of the merged tile.

4) Merge Overlapping Tiles. As mentioned in Section 2.2.1, tiles are overlapped to avoid visible blocking. Hence, a raised cosine window is applied in both dimensions to make sure tiles are superimposed with correct values; higher values on the center and lower values on the side. Mathematically, for each pixel value p at position x, y of a tile with length n, the new value p' is

$$p' = (\frac{1}{2} - \frac{1}{2}\cos\frac{2\pi}{n}(x + \frac{1}{2}))(\frac{1}{2} - \frac{1}{2}\cos\frac{2\pi}{n}(y + \frac{1}{2})).$$

Finally, values of overlapping tiles are added together to create the processed merge image, which has a higher SNR.

2.3.2 Implementation. We largely follow the above algorithm for implementation, using OpenCV's `cv::Mat` class. Since we are taking bayer images as input, four values on the bayer images represent one pixel. Hence, we manually split the image into four channels (R, G1, G2, B) and use single channels as the input for the above algorithm, merging them channel-wise. When possible, we use batch operations on the whole array (`cv::Mat`) to avoid iterations. We then put channel values back to a `cv::Mat` representing a new bayer image.

In Google's original implementation, Hasinoff et al. have direct control over the noise parameters λ_s and λ_r through the camera's API [3]. However, the information is not retained in the dataset, making it difficult to evaluate the noise level. We follow Monod et al.'s approach [4] and evaluate these parameters by comparing the ISO of input image with baseline parameters at ISO 100. Mathematically, we denote the actual input as $\frac{\text{ISO}}{100}x$, a value scaled with the ISO ratio. Given baseline parameters at ISO 100 $\lambda_{s,100}$ and $\lambda_{r,100}$, we have:

$$\begin{aligned}\sigma^2(\frac{\text{ISO}}{100}x) &= (\frac{\text{ISO}}{100})^2\sigma^2(x)\\ &= (\frac{\text{ISO}}{100})^2(\lambda_{s,100}x + \lambda_{r,100})\\ &= \frac{\text{ISO}}{100}\lambda_{s,100}(\frac{\text{ISO}}{100}x) + (\frac{\text{ISO}}{100})^2\lambda_{r,100}\\ &= \lambda_{s,\text{ISO}}(\frac{\text{ISO}}{100}x) + \lambda_{r,\text{ISO}}.\end{aligned}$$

This approach allows us to acquire the noise parameters without direct access to the camera capturing the images; however, an estimate of $\lambda_{s,100}$ and $\lambda_{r,100}$ is still needed. After running experiments, we decide to follow Monod et al. [4] and use 3.24×10^{-4} and 4.3×10^{-6} respectively; the resulting images are visually satisfactory, with balanced blurriness, denoising, and exposure. We experiment on other tuning parameters like the temporal factor τ and the spatial factor s as well; we end up using $\tau = 75$ and $s = 0.1$ for better subjective image quality.

2.4 Finishing

2.4.1 Algorithm. The finishing part aims to mimic an ISP that performs correction, demosaicking and tone mapping operations on the merged image. But a key difference is, in order to reduce the contrast between highlights and shadows while preserving local contrast, we adapt exposure fusion in local tone mapping step using synthetic exposure which applies gain and gamma correction to the extracted grayscale image. Fig3 is the pipeline of the finishing part.

2.4.2 Implementation. As we can see from the pipeline, after receiving the merged bayer image from the merging part, the finishing part performs the following operations in order to get the final output image.

1) Basic Post Processing. We perform black level subtraction, white balance, demosaicking and color correction on the merged bayer image in this step. Note that we perform this post processing whenever we read in a raw bayer image.

2) HDR Local Tone Mapping. We first transform the processed merged image into a grayscale image by averaging on the RGB channel. Then we perform synthetic exposure on the grayscale

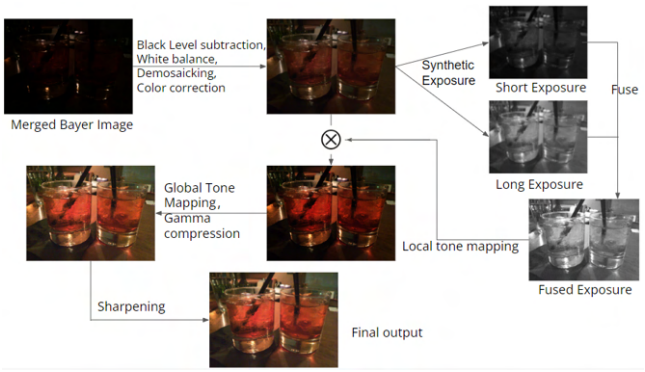


Figure 3: The pipeline of the finishing part[3]

image to get a short exposure image and a long exposure image and fuse them into a fused exposure image. Eventually, we use the fused exposure to modulate the processed merged image. Here we only use the two images for exposure fusion in order to reduce computation and memory consumption. The hyperparameter for calculating the gain in each exposure can be modified according to the hardware and environment to get a better result. We also optimized some functions like getting the grayscale image out of the RGB channels using OpenMP to yield a higher speed.

3) Global Tone Mapping. In this step, we generally enhance the contrast of the merged image by applying an S-shaped curve function to each channel. Here we use the following sin function for contrast enhancement:

$$y = \min(1, \max(0, x - \alpha\sin(2\pi x)))$$

where α is a hyperparameter that we need to choose carefully according to the real situation.

4) Gamma compression. This step plays an important role in turning a dark image into a more visually appealing image by shifting pixel values from linear to nonlinear sRGB space. We use the sRGB transform function defined by IEC 61966-2-1.

5) Sharpening. We first apply Gaussian blur on the image 3 times with different Gaussian kernels to get 3 blurred images $\{I^G_{\sigma_n}|n = 1, 2, 3\}$, where σ_n is the standard deviation of Gaussian kernel n. Then we subtract the original image with 3 blurred image to get 3 low-contrast images $\{I^L_{\sigma_n}|n = 1, 2, 3\}$:

$$I^L_{\sigma_n}(x, y) = |I^G_{\sigma_n}(x, y) - I(x, y)|$$

Then we use the low-contrast image as mask and merge with the original image to get the final output according to the following rule:

$$I_{sharpen} = \frac{1}{3}\sum_{n=1}^{3}(I^n_{sharpen})$$

Where

$$I^n_{sharpen}(x, y) = \begin{cases} I(x, y) + \alpha_n I^L_{\sigma_n}(x, y), & I_{\sigma_n} > t_n \\ I(x, y), & \text{otherwise} \end{cases}$$

Where t_n and α_n are hyper parameters we set to control the sharpness. During our experiments, we found that $\sigma_n \in \{1, 2, 4\}$, $\alpha \in \{1, 0.5, 0.5\}$ and $t_n \in \{0.02, 0.04, 0.06\}$ can give us a good result.

But it might depend on the hardware or environment we use and users might need to adjust these parameters in order to get similar results.

3 PROBLEM ENCOUNTERED

3.1 Debugging numerical bug

One major problem we encountered is how to debug numerical bugs. Unlike logical bugs that may cause the segmentation fault and can be easily debugged by gdb. Numerical bugs, specifically how the tiles are aligned, can be hard to catch. We can only tell from the final result that something is wrong with the alignment, but we can't really tell using gdb which steps inside the alignment is wrong. We found that print out the intermediate output of each step and comparing the intermediate output with hand computed results is more helpful than gdb.

3.2 OpenCV 3.2 Memory Leak

One segmentation fault we have encountered is a memory leak related to OpenCV 3.2. Initially, we're using ubuntu 18.x, and OpenCV 3.2 is installed by default through `apt install libopencv-dev`. We found that only the top part of the final image is shown correctly, while all the other image parts are random noise. We try to debug this issue by printing out the intermediate image value and finally realized this might be related to how OpenCV uses reference count and maintain a memory pool for `cv::Mat`. We solve this problem by manually install OpenCV 4.x with CMake.

3.3 Varying Output On Different Environments

During testing, we observe that the final output varies when generated on different environments. We have tested two environments, both on Ubuntu 20.04; all dependencies have matching versions, and all parameters are set to be identical. Nonetheless, difference is visually noticeable, as illustrated in Figure 4. While we are unable to determine the root cause, our speculation is that system architecture plays an important role here, as Environment 1 is on a native Linux machine and Environment 2 is run inside Windows Subsystem for Linux (WSL). Kernel adaptions for Windows may lead to altered numerical results, thus leading to slightly different visuals. We use final images from Environment 1 to present our work in this report.

(a) Final Output on Environment 1

(b) Final Output on Environment 2

Figure 4: Varying outputs on different environments.

4 RESULTS AND ANALYSIS

4.1 Compare single image with merged image

We perform an ablation study to help us better visualize the effect of denoising. In one experiment, we just use a single image and directly go through the finishing part. In the other experiment, we input a group of burst images and go through the whole pipeline, including alignment, merging and finishing. Fig 5 shows the results for this comparison.

As is shown in the visual results, the finishing part can generally give a better exposure on the original image and enhance the contrast to a more visually appealing level but can't deal with noise. Comparing column (b) and (c), we can clearly tell that alignment and merging play an important role in noise reduction.

4.2 Compare with results from reference papers

We also compared our results with google and gopro's results. Fig 6 shows the results from Google HDR+, GoPro's python HDR+ and our HDR+ respectively. We can see that Google's results are generally darker than GoPro's and our results. Their processed images usually have a classical style, which kind of gives people a feeling of solemnity. GoPro's results are generally brighter and have a more energetic feeling. Unlike their styles, our HDR+ seems to pay more attention to details. For example, the textures in the fire, color variance in green grass and the ice cubes in the glasses.

We believe that these differences can mainly be due to the choice of hyperparameters like standard deviation of Gaussian kernels, gains for long/short exposures, the reference images and so on. It doesn't necessarily mean that our HDR+ is somewhat better than Google's and GoPro's. As an old saying goes, there are a thousand Hamlets in a thousand people's eyes. Thus, different people might prefer different results generated by these three HDR implementations.

4.3 Manually adding noise

To test the robustness of our HDR pipeline, we add salt-pepper noise onto the original burst images. Fig 7 (a) and (b) are the original view and zoom-in view of the results we get respectively. The first row shows the output of the finishing part using a single image with noise as input. The second row shows the output of our whole pipeline using a group of burst images as input. We can clearly see that our HDR pipeline can still generate a denoised output given a high noise level. For example, the grass in the bottom right corner of the third image is hard to tell when only using a single image but it's still quite visible when using our merged denoised image. Thus, we believe our HDR+ pipeline is robust towards noise.

5 CONCLUSION

This HDR pipeline overcomes limitations in smartphone camera hardware by combining multiple underexposed raw images to generate enhanced HDR photos with increased SNR. Our pipeline modifies the alignment and finishing part while optimizing performance. During this project we learned about the intricacies of lower level image processing and computational photography. While implementing, we encountered lots of numerical related bugs, and we found the easiest way to debug them is to print them out, since gdb

(a) single image (minimum processed) (b) single image + finishing (c) merged image + finishing

Figure 5: Comparison between minimum processed single reference image, single reference image after finishing part and a group of burst images go through our whole HDR+ pipeline

does not really help a lot here. There are a lot of detailed implementation decisions not mentioned in the original Google paper, and we needed to figure out by ourselves which is a more rational decision. Our results are visually comparable with the results from Google and GoPro and achieve the denoising. Future improvements could include implementing bracketing, as followed up by Google, which would alter the merging step by combining additional images with different exposures. We could also improve our pipeline to run more effectively on mobile hardware.

REFERENCES

[1] G. Bradski. 2000. The OpenCV Library. *Dr. Dobb's Journal of Software Tools* (2000).

[2] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. 2001. *Parallel programming in OpenMP*. Morgan kaufmann.

[3] Samuel W. Hasinoff, Dillon Sharlet, Ryan Geiss, Andrew Adams, Jonathan T. Barron, Florian Kainz, Jiawen Chen, and Marc Levoy. 2016. Burst Photography for High Dynamic Range and Low-Light Imaging on Mobile Cameras. *ACM Trans. Graph.* 35, 6, Article 192 (nov 2016), 12 pages. https://doi.org/10.1145/2980179.2980254

[4] Antoine Monod, Julie Delon, and Thomas Veit. 2021. An Analysis and Implementation of the HDR+ Burst Denoising Method. *Image Processing On Line* 11 (2021), 142–169. https://doi.org/10.5201/ipol.2021.336.

(a) Google's HDR+ results (b) GoPro's HDR+ results (c) Our HDR+ results

Figure 6: Comparison between the results of Google's HDR+, GoPro's python HDR+ and our HDR+ implementation

(a) Original view of the results in adding noise experiments

(b) Zoom-in view of the results in (a)

Figure 7: Results after adding salt-pepper noise to input images. In each subfigure: Row a) single image + noise + finishing; Row b) burst images + noise + alignment + merging + finishing